DELHI TECHNOLOGICAL UNIVERSITY

DEPARTMENT OF APPLIED MATHEMATICS

MIDTERM EVALUATION COMPONENT PROJECT REPORT

RESEARCH PAPER REVIEWS

OPERATING SYSTEM (MC-301)

Submitted To: Prof. Trasha Gupta

Nandika Arora

2k19/MC/084

Nitya Mittal

2k19/MC/089

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to several individuals and organizations for supporting us. First, we wish to express our sincere gratitude to our teacher, Prof. Trasha Gupta, for her enthusiasm, patience, insightful comments, helpful information, practical advice and unceasing ideas that have helped us tremendously at all times in the project. Her immense knowledge, profound experience and professional expertise has enabled us to complete this work successfully.

We also wish to express our sincere thanks to the Delhi Technological University for accepting us into the undergraduate program and providing us the opportunity to do this wonderful project.

Sincerely,

Nitya Mittal (2K19/MC/089)

Nandika Arora (2K19/MC/084).

TABLE OF CONTENTS

PAPER 1

Title

Survey of Memory Management Techniques for HPC and Cloud Computing

I. INTRODUCTION

Overview:

In this review, we look at the issues of memory management in HPC and Cloud Computing, as well as several memory management systems and memory optimisation strategies.

Challenges Faced:

Cloud computing tries to serve applications with fewer hardware resources, whereas HPC settings must maximize interconnect performance. As a result, both scalability and excellent performance are required. This HPC study is aimed at achieving that equilibrium.

Proposed Methodology:

The importance of memory optimization in enabling future Exascale, the breadth of memory management in HPC and Cloud, and unresolved problems and future research paths in memory management are all explored in this paper.

II. BACKGROUND AND MOTIVATION

Small and medium-sized businesses can now use cloud HPC as well. The classic HPC scenario, in which batch jobs are run on separate nodes, is being replaced by a new scenario (cloud HPC), in which several applications with varying degrees of QoS coexist. Financial analytics, online video transcoding, and medical imaging are examples of new applications in time-critical applications. As a result, rather than the conventional necessity to maximize resource utilization while minimizing power consumption, a new goal of predictable performance has emerged. Exascale performance per watt is a goal that many people want to reach. As a result, we want to see quick advancements in high-performance computing hardware, software, and applications.

III. SCOPE OF MEMORY MANAGEMENT

Traditional HPC systems are built so that all nodes can run at full speed at the same time. Because the CPU and RAM are so tightly coupled, they are scaled accordingly. Memory bandwidth, based on common processor-memory link technologies (memory utilized at a faster rate than processor capabilities), will limit heterogeneous systems,

increasing both price and power consumption. As a result, current high-performance computing faces additional obstacles.

A. Towards Exascale Computing

Increasing the number of cores has traditionally sufficed, but in heterogeneous systems, higher utilization is essential. However, in order to assure efficient resource sharing and sophisticated computing resources, this complicates programming paradigms and languages. Memory management was seen as a serious barrier for exascale research applications due to the deeper, more complicated memory structures and smaller capacities. The use of dynamic, latency-tolerant techniques was a potential solution to deal with memory complexity challenges.

The main challenges faced are:

On-Chip Memory: It cuts down on data travel that isn't necessary. It's crucial to find solutions that can coexist with traditional, automatically maintained caches.

Using a Global Address: Message passing is likely to be used for inter-node communication in exascale systems, but it is not feasible for interprocessor communication. Global memory addressing is preferred over cache coherence for regulating fine-grained computation and data movement on-chip.

Because of the hierarchy's NUMA structure and the several types of memory, memory management is essential for determining the most efficient technique for using the various memory types. Memory management libraries were suggested as the most likely solution.

B. HPC in the Cloud

Another challenging study area in HPC is moving HPC to the cloud. Despite the fact that HPC systems are becoming more heterogeneous, cloud platforms still use homogeneous architectures. However, there are a number of drawbacks to this strategy.

Batch processing demands greater computing resources than Cloud. Resource optimization for HPC systems that takes into consideration their specific properties. In cloud computing, an imbalance of memory usages in each node is a major memory concern. The trade-off between substantially increasing memory capacity requirements for memory-intensive applications and free memory on dispersed nodes is addressed by rightsizing memory allocation and exposing a global memory bank to all computers.

C. Memory Management Functionality

Memory management systems deal with a variety of issues, including choosing the best memory for the processing units they're given, enabling concurrent, thread-safe

memory allocation and deallocation while avoiding fragmentation, translating virtual to physical addresses and vice versa, and optimizing runtime performance.

The OS oversees memory allocation in Clouds when it is executing on the virtual machine. In HPC systems, the OS or a platform-dependent library can take control of memory allocation. If enough free memory is available, the memory manager may consider relocating operating kernel data to more appropriate memory modules.

IV. MEMORY MANAGEMENT TECHNIQUES IN HPC

For many modern HPC systems, OS (Linux) based MM (dynamic memory allocation) is a conventional technology. However, due to the increasing complexity of HPC system architecture, additional OS capability is required. NUMA architectures can allocate a distinct memory to perform computational operations and provide a unification of the usage of multiple kinds of memory devices to cope with the limited number of memory accesses (which causes performance degradation).

An overview of the memory management strategies in HPC that were surveyed:

A. Access Patterns:

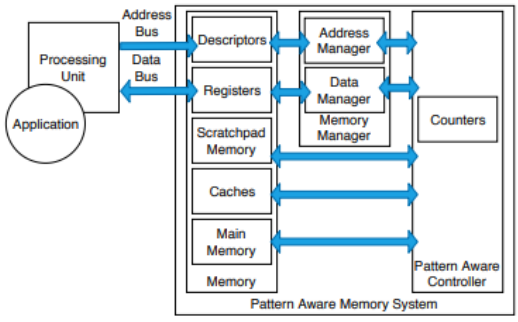


FIGURE 2. General scheme of the pattern-aware memory systems.

It addresses the limitation of the data read and write latency.

Pattern Aware Memory System (PAMS) : local mm system, accelerates both static and dynamic data structures and their access patterns based on the information provided by pre-compiled pattern descriptors.

Sharing-Aware Memory Management Unit (SAMMU) : It analyses the memory access behaviour on the hardware level.

Also provides information to the OS to perform an online thread- and data-mapping to increase the locality, thus improving performance.

B. Partitions:

On a consolidated platform, HPC applications use a specialized lightweight memory management framework that is specifically built for HPC workloads.

C. Migration:

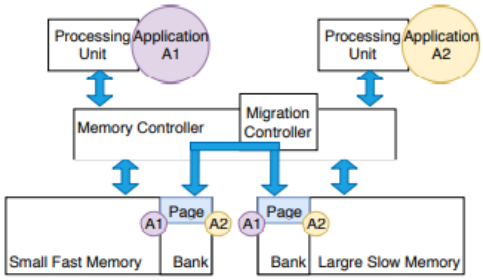


FIGURE 3. General scheme of the memory migration.

The basic motive for the memory migration is that accessing a memory page on the original node incurs remote memory access if it is not transferred along with a job that is migrated to another node. Page migration, on the other hand, is a costly operation that can result in additional memory access overhead due to page granularity.

D. Memory Locality:

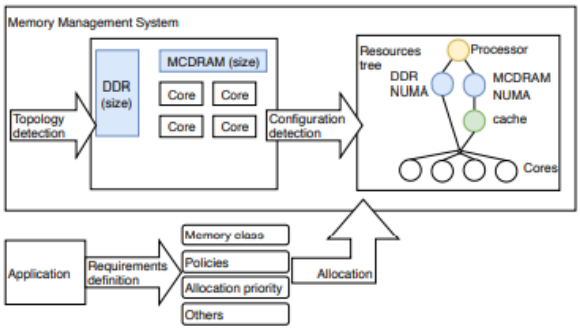
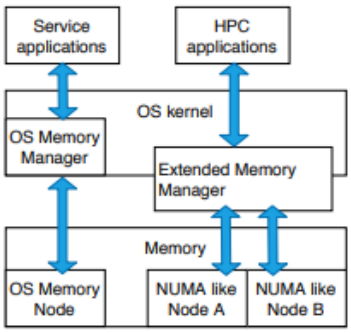


FIGURE 4. Overview of the memory locality in HPC.

The explicit control of physical memory locality has a substantial impact on an application's performance. According to current research, using memory locality automatically is still not as effective as having explicit instructions from the program developer.

E. Policies



Overview of the memory partitioning in HPC.

Locality-based policies, on average, outperform memory-balanced strategies in terms of performance. In addition to locality-based solutions, recent research have advocated that when performing memory mapping, other criteria be considered, such as balance and reliability, to avoid overloading and ensure fault tolerance.

F. High Bandwidth Memory

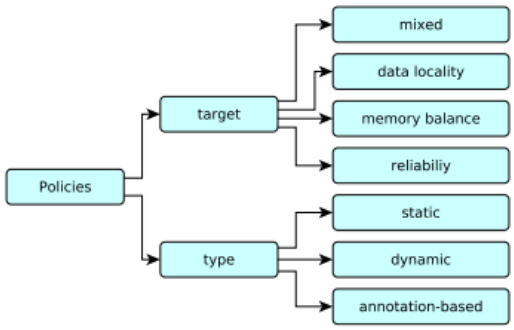
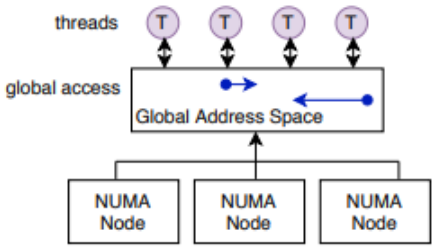


FIGURE 6. Taxonomy of the proposed policies.

High-bandwidth memory (HBM) is a novel memory technology that improves the performance of bandwidth-constrained applications significantly. However, there is room for improvement in terms of execution time.

G. Global Address Space



General scheme of the global address space.

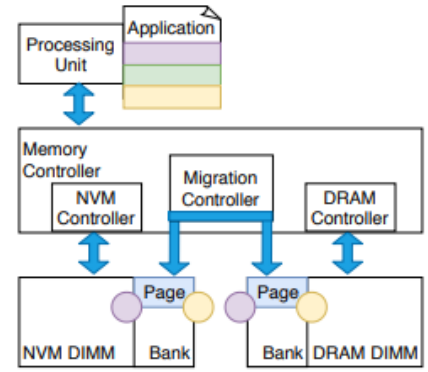
Partitioned GAS (PGAS) is a non-uniform global address space in which a global address's representation statically encodes the memory's physical address.

Active GAS (AGAS) was created to dynamically balance load and data during execution in a message-driven runtime system.

H. Allocation

Resource allocation and scheduling approaches are employed to identify a specific memory unit for the task at hand. The current research direction in memory allocation is to characterize memory access behavior online and then perform page migrations to improve it.

I. Hybrid Memories



General scheme of the hybrid memory.

For large-memory applications, hybrid memories that combine DRAM and non-volatile memory (NVM) to increase primary memory capacity are utilized.

J. Programming Models

The quickest way to accomplish the Exascale architecture's intended bandwidth is to use programmer-driven memory management. The main goal of optimizing HPC program models is to provide optimal performance for all applications in all processes with no user involvement.

V. MEMORY MANAGEMENT TECHNIQUES IN CLOUD COMPUTING

The complexity of computational multi-parametric models, as well as the expansion of scientific procedures and datasets, is forcing us to employ Cloud solutions. For the most efficient utilization of the novel workload characteristics and resources, MM adaptation is required. VMs and storage spaces are the primary resources given by clouds, in contrast to traditional HPC infrastructures. Each job must be scheduled to an appropriate VM in order to run an HPC application in the Cloud.

Some of the solutions include scheduling, resource and user allocation, memory allocation, memory aggregation and disaggregation, memory pages management, shared memory, cache, and VM placement.

VI. OPEN PROBLEMS

The differences in memory optimisation strategies in HPC and Cloud systems are due to the range of workloads served. The workload in Cloud Computing is more variable, therefore performance and cost efficiency go hand in hand.

As a result, memory optimization strategies differ in the following ways:

- **Attention:** Because of the hardware modernization of memory devices and caches, HPC pays special attention to acceleration. In Clouds, on the other hand, the focus is on resource supply and schedule optimization.

- **Using Statistical Information:** The goal of research in the HPC arena is to look into the application's memory access patterns. Instead, analysis and prediction of workload or resource utilization are common in Cloud Computing.
- **Memory migration:** Because of memory heterogeneity and locality, memory migration is becoming increasingly relevant in HPC. Memory migration in Clouds considers memory similarity in addition to memory locality to limit the amount of data transferred. However, because memory migration is a very costly procedure, a cost estimate is always required.
- **Cache:** In addition to improving standard cache management, a hybrid memory, that is, a memory that can be configured as a cache or a fat memory depending on the requirements, has gained relevance in HPC. Per-VM cache partitioning is the subject of the majority of cloud cache research.

A. Memory Hardware

Traditional memory improvement approaches focus on raising the memory clock frequency or storage bus bandwidth at the hardware level. The hybrid memory cube and high bandwidth memory, in particular, are likely to be widely used in future HPC systems. Along with new hardware, current software support is required to meet memory management issues and complications.

B. Resources Provisioning And Scheduling

Most schedulers nowadays don't take NUMA architecture or universal memory binding into account. The following groupings describe the degree of memory participation in provisioning and scheduling techniques:

- **memory utilisation** : Memory, CPU, and other resources are treated as separate resources, and the bin packing problem solver is applied to each resource separately.;
- **memory power model** : The resource parameters are taken into account by a series of power models for CPU, memory, and other components, and then all power models are combined to produce optimal or near-optimal solutions.
- **memory-processor interaction** : Memory is defined not only by its physical features, but also by how it interacts with the processor or group of processors or accelerators in order to run a given program. The first or second strategy is used by the majority of the approaches surveyed.

C. Memory Access Patterns

MM examines memory access patterns and applies the findings to thread and memory mapping to improve performance. It can be downloaded (precompiled) or accessed

online (runtime). Because of the workload consistency, this strategy is recommended for HPC. This method, on the other hand, can be used to examine memory-intensive scientific workloads in Clouds.

D. Workload Prediction

In Clouds, rather than analyzing each application, it is usual to forecast workload or future resource utilization. This method aids in the efficient and/or energy-conscious placement of virtual machines on physical nodes. It may be extended to the HPC domain by taking into account the trade-off between forecast accuracy and computational complexity, just as it can be done with memory access patterns.

E. Memory Migration

Despite the fact that migration is commonly utilized in both HPC and Cloud contexts, research in this area focuses on determining the necessity for migration, selecting a destination for transferred data, and enhancing migration speed due to the high cost of migration.

VII. CONCLUSION

The need to cut energy use is becoming more widely recognized. In the design of new HPC solutions, power consumption and energy efficiency are critical considerations. Because the memory system is one of the most energy-intensive components in today's computing environment, it's important to focus on energy-efficient memory management in both traditional HPC systems and Cloud solutions. Due to a concomitant increase in execution time, a naive reduction in energy usage does not always translate to energy efficiency.

New hardware innovations, such as HBM, NVM, RRAM, and SSD, necessitate extra support in memory management software. New policy features can be applied to memory in addition to hardware advancements. This study examines the most up-to-date state-of-the-art memory management approaches for HPC and Cloud Computing.

PAPER 2

Title

A Performance-Stable NUMA Management Scheme for Linux-based HPC Systems

I. INTRODUCTION

Overview:

Linux is becoming the de-facto standard operating system for today's high-performance computing (HPC) systems because it can meet the requirements of many HPC systems for a rich operating system (OS).

We address the OS noise created by Linux in NUMA architecture in this research and present Stable-NUMA, a new performance-stable NUMA management strategy. Stable-NUMA combines two-level thread clustering, state-based page placement, and selective page profiling to improve performance stability.

Challenges Faced:

The application's performance stability is a critical feature in HPC systems. However, program performance varies from run to run. OS noise is one of the barriers that needs to be addressed because it drastically affects application scalability and performance. Timer interruptions, followed by process preemption, CPU power management, and CPU scheduling, are all causes of OS noise.

For greater performance in the NUMA architecture, enhancing the locality of memory access is critical. These techniques, on the other hand, have the drawback of being architecture-dependent. The Linux kernel features a NUMA-aware functionality called Auto-NUMA that runs in the background, classifies thread memory access, and migrates threads and/or pages to enhance memory access locality to handle this feature of NUMA architecture.

Auto-NUMA, however, has two drawbacks in terms of performance stability:

- the runtime profiling of the memory access incurs high overheads and increases run-to-run variations in the application performance
- the ping-pong migration of threads and pages between NUMA nodes is caused by conflicts in the decision policy between the CPU load balancer and Auto-NUMA. These overheads act as OS noise, degrading application performance stability.

Proposed Solution:

Stable-NUMA is a performance-stable NUMA-aware thread and page allocation strategy for Linux-based HPC systems. The technique divides pages into three groups and uses distinct memory access profiling and page placement strategies for each group. These policies have been meticulously crafted to eliminate unwanted profiling and reduce profiling overheads.

Threads and pages are correctly placed on NUMA nodes based on the gathered memory access patterns, minimizing the distance between the pages and the threads accessing them.

To avoid a decision conflict between the CPU load balancer and our scheme, NUMA node load balancing is separated from the CPU load balancer and handled alone by our scheme.

As a result, our system maps threads to NUMA nodes, with the CPU load balancer balancing only the loads of cores within a NUMA node. As a result, wasteful ping-pong migration of threads and pages across NUMA nodes is reduced, resulting in faster execution and more stable performance.

Solution Details and Results Observed:

The proposed system is implemented in the Linux kernel and tested using a variety of high-performance computing benchmark applications. The scheme's performance is evaluated on two server configurations: one with two or four NUMA nodes, and four with four NUMA nodes each.

When compared to vanilla Linux and Auto-NUMA, the technique minimizes the run-to-run volatility in workload execution time by up to 74% in single-server testing.

II. BACKGROUND AND MOTIVATION

Performance instability is a concern with Linux-based HPC systems due to the NUMA architecture. The performance stability is affected by the run-to-run difference in the local remote memory access ratio.

A node-local or first touch memory policy in Linux allocates a memory page at the NUMA node where the thread is operating. The local-remote memory access ratio would be stable if the thread stayed on the same node. The CPU load balancer in Linux, on the other hand, can relocate threads across NUMA nodes, changing the local-remote memory access ratio.

Performance Implications of Auto-NUMA in Linux

Auto-NUMA is a feature that analyses threads' memory access habits on a regular basis and dynamically repositions threads and pages to reduce the space between them. When a thread transfers to another NUMA node, its access pages migrate to that node;

similarly, if a thread accesses the majority of its pages on a remote NUMA node, it migrates to that node. As a result, Auto-NUMA tries to maximize the ratio of local memory accesses.

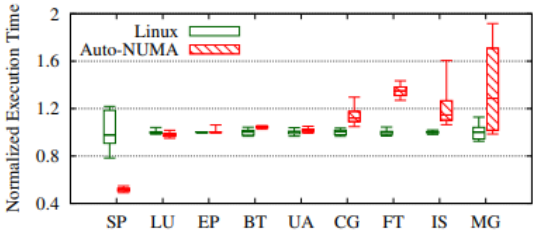


FIGURE 3. A box plot of the normalized execution time. The box plot shows the minimum, first quantile, median, third quantile, and maximum values in 20 runs of each workload.

Auto-NUMA, however, increases the performance instability of HPC workloads. Figure 3 shows the normalized execution time of the NPB workloads running on Linux without Auto-NUMA (denoted as Linux) and with AutoNUMA (denoted as Auto-NUMA) on Server A. Only the SP workload, as shown in the diagram, benefits from NUMA-aware management in terms of performance.

The performance instability issue arises because the AutoNUMA balancing presents the following three problems:

1) Memory access profiling overhead

- Memory accessed by threads exploiting the page fault is profiled by Auto-NUMA. For two reasons, Auto-profiling NUMA's overheads might reduce the performance of HPC applications. Page fault management, for starters, is a sort of OS noise. Page faults have a negative impact on application performance stability when they occur frequently.
- Second, Auto-NUMA applies the same page unmapping frequency to all of the application's pages. When memory access is unstable, however, Auto-NUMA increases the frequency of page unmapping in order to detect threads and pages that need to be relocated quickly, resulting in more page faults.

2) Unnecessary thread migration

- Threads that share pages are grouped together by Auto-NUMA. The total ratio of local memory access between threads in the group is greatest when all threads are on the node, therefore each group chooses a favorite node. Regardless of the number of actual NUMA nodes in a system or the number of threads in a group, AutoNUMA assigns just one node. As a result, Auto-NUMA tries to move a thread to a preferred node, which may clash with the CPU load balancer's choice under Linux. As a result of this dispute, a huge number of unwanted thread migrations occur.

3) Unnecessary page migration

- Private pages for a thread migrate to a memory node where the thread is operating during memory profiling. This page migration policy is successful in boosting the thread's local memory access ratio.
- The Auto-NUMA policy pulls a thread's private pages when it migrates to another memory node. This happens every time a thread moves to a different memory node. Page migration, on the other hand, may reduce the hit rate of the translation lookaside buffer.

III. RELATED WORKS

A. TRENDS OF LINUX AS AN OS IN HPC SYSTEMS

In a lightweight kernel, a large number of functions are removed except for those that are absolutely necessary. However, this method only provides a portion of the POSIX APIs. Furthermore, because the lightweight kernel is built from the ground up, there is the additional burden of creating a new device driver to handle the new device. To compensate for these flaws, the multi-kernel architecture was devised, in which a lightweight kernel runs alongside a full-weight kernel with all of the kernel's functionality. However, due to resource management characteristics, the multikernel has limits in offering full Linux compatibility.

B. THREAD AND PAGE MANAGEMENT IN NUMA ARCHITECTURE

1) Profiling based on hardware counter

The use of hardware counters concerning the cache, memory controller, and/or CPU interface is one way for performing thread and page placement profiling. Carrefour was created by Dashti et al., and it collects hardware activity data such as memory controller imbalance, local access ratio, and page access type (read/write). Carrefour evaluates whether to co-locate, interleave, or replicate for page placement in NUMA architecture based on this information.

In the NUMA architecture, Lepers et al. devised a thread and page placement mechanism that works by calculating metrics that represent the amount of CPU-to-CPU and CPU-to-memory communication.

These approaches emphasize the benchmarks' faster execution times, but they mostly rely on the architecture-dependent counter to acquire profiling data.

2) Profiling based on operating system

Another method is the utilization of page fault of the operating system. Diener et al. designed a kernel memory affinity framework (kMAF) to automatically manage thread and page placement by utilizing a page fault mechanism. To improve profiling accuracy,

kMAF generates periodic extra page faults after page faults for the demand page mechanism in a virtual memory system. With these page fault mechanisms, kMAF updates its affinity table that determines thread and page placement to maximize local memory access.

Page faults on the same page table for multiple threads to trace the memory access pattern, according to Gennaro et al., do not achieve high profiling accuracy. This is because if one thread masks on a page due to a page fault, profiling of other threads' memory access to the same page is impossible until the next page fault. To solve this problem, the authors introduced multiple page tables allocated to each thread to profile page access patterns of all threads. A page fault, on the other hand, is a typical OS noise.

IV. Stable-NUMA

Our technique, like Auto-NUMA, profiles the pattern of memory access in order to place the thread and page on the NUMA node with the best memory locality.

Stable-NUMA, on the other hand, was specifically engineered to minimize memory profiling overheads in order to reduce performance fluctuation. Furthermore, Stable-NUMA was created to eliminate policy conflicts between Auto-NUMA and the CPU load balancer.

A. NUMA-AWARE PAGE PLACEMENT

Any hardware-specific feature has no effect on Stable-memory NUMA's profiling mechanism. A NUMA-hinting fault can reveal which thread is accessing which page on which memory node. When a NUMA hinting fault occurs, Stable-NUMA records the thread that accesses a page and the CPU that accesses the page to trace the memory access pattern. Stable-NUMA just needs to inspect and record two bits of information from the thread that generates the NUMA-hinting fault, therefore the performance overhead is minimal.

Subsequently, when a NUMA-hinting fault occurs again on that page, whether the page is shared is determined based on the history information recorded on the page and the current NUMA-hinting fault information. When a NUMA-hinting fault occurs, using the history and current information, we can identify the NUMA-hinting fault type: private fault or shared fault.

- **Thread-Private:** This type of page is accessed by only one thread. Three consecutive hinting faults owing to one thread (one unclassified fault + two private faults) make a page thread-private. When a page becomes thread-private, the page is immediately migrated to the memory node on which the thread is running.

- **Node-Private:** This type of page is shared by multiple threads running on the same memory node. Two consecutive hinting faults owing to the threads in the same memory node (two shared faults from the same node) make the page node-private.
- **System-Shared:** This type of page is accessed by multiple threads running on two or more memory nodes. Two consecutive NUMA-hinting faults owing to threads on different memory nodes (two shared faults from different nodes) make a page system-shared. The scheme records a system-shared page count for each memory node. When the minimum count is less than three-fourth of the maximum count, the shared pages on the node with the maximum count are migrated to the node with the minimum count.
- **Three In-Transitions:** Each of the above three states has its own corresponding in-transition state. By using the second chance technique, StableNUMA reduces the probability of misjudging the page state.the page can return to the last state at the accidental page state change by remembering the last page state.

To improve performance stability, it's critical to reduce the number of NUMA-hinting failures. To control the frequency of profiling on each page, the approach uses a per-page period value. Different profiling frequencies should be used for different sorts of pages because an application can have many types of pages. Each page in Stable-NUMA has a page unmapping bypass counter that defines when the page should be unmapped.

B. PAGE SHARING-BASED THREAD CLUSTERING

The scheme collects the statistics of the page sharing between threads and places threads across memory nodes based on the page sharing statistics.

For managing and placing threads on physical NUMA nodes, Stable-NUMA uses a two-level thread clustering mechanism. To begin, a root group is formed by all threads that share the same address space. Stable-NUMA has sub-groups in a root group, each of which is mapped to a NUMA node. In a root group, a thread can only belong to one sub-group. When a thread joins a root group for the first time, it joins a sub-group mapped to the NUMA node where it is running. It should be noted that when a thread joins a root group, the CPU load balance no longer migrates the thread-crossing NUMA nodes. Instead, Stable-NUMA handles the thread's inter-NUMA migration.

Stable-NUMA employs two tables for threads in the root group: thread-node and thread-thread, each of which gathers memory access information. The thread-node table first gathers the number of pages accessed by a thread in each memory node. As a result, the number of rows in the threadnode table equals the number of threads in the

root group, and the number of columns equals the number of NUMA nodes. The associated entry is updated whenever a NUMA hinting fault occurs.

Second, the thread-thread table keeps track of how many pages a pair of threads in the root group share. As a result, the thread-thread database is the same size as the root group's threads in terms of rows and columns. If the hinting fault is a shared fault, an entry value in the table is updated whenever a NUMA-hinting fault happens, with the column being the thread ID from the previous access information in the page and the row being the current thread ID causing the current NUMA-hinting problem.

V. EVALUATION

TABLE 1. Server Configurations for Evaluation

Server A (2-socket)	
Model Name	Dell R740
Processor	Intel Xeon Gold 5118 2.3GHz (12 cores per socket)
L1 TLB	64 entries, 4-way
L2 TLB	1536 entries, 12-way
Memory	196GB DDR4-2666, 6 channels
NIC	Mellanox Infiniband ConnectX-3 40GB
OS & Kernel	CentOS 7.5 & Kernel 5.0 (Auto-NUMA) Ubuntu 12.04 & Kernel 3.8 (kMAF)
MPI Library	OpenMPI v4.0.3
Server B (4-socket)	
Model Name	SuperServer 2049U-TR4
Processor	Intel Xeon Gold 6138 2.3GHz (20 cores per socket)
L1 TLB	64 entries, 4-way
L2 TLB	1536 entries, 12-way
Memory	512GB DDR4-2666, 4 channels
OS & Kernel	CentOS 7.5 & Kernel 5.0 (Auto-NUMA) Ubuntu 12.04 & Kernel 3.8 (kMAF)

Table 1 lists the server configurations utilized in the evaluation. Each experiment was carried out a total of 20 times. Because this instability can damage the ability of HPC systems and the reliability of performance measures in a variety of ways, we did not eliminate any outliers from the experiment data. As a result, we included outliers in the experiment findings because they demonstrate performance anomalies in the HPC system.

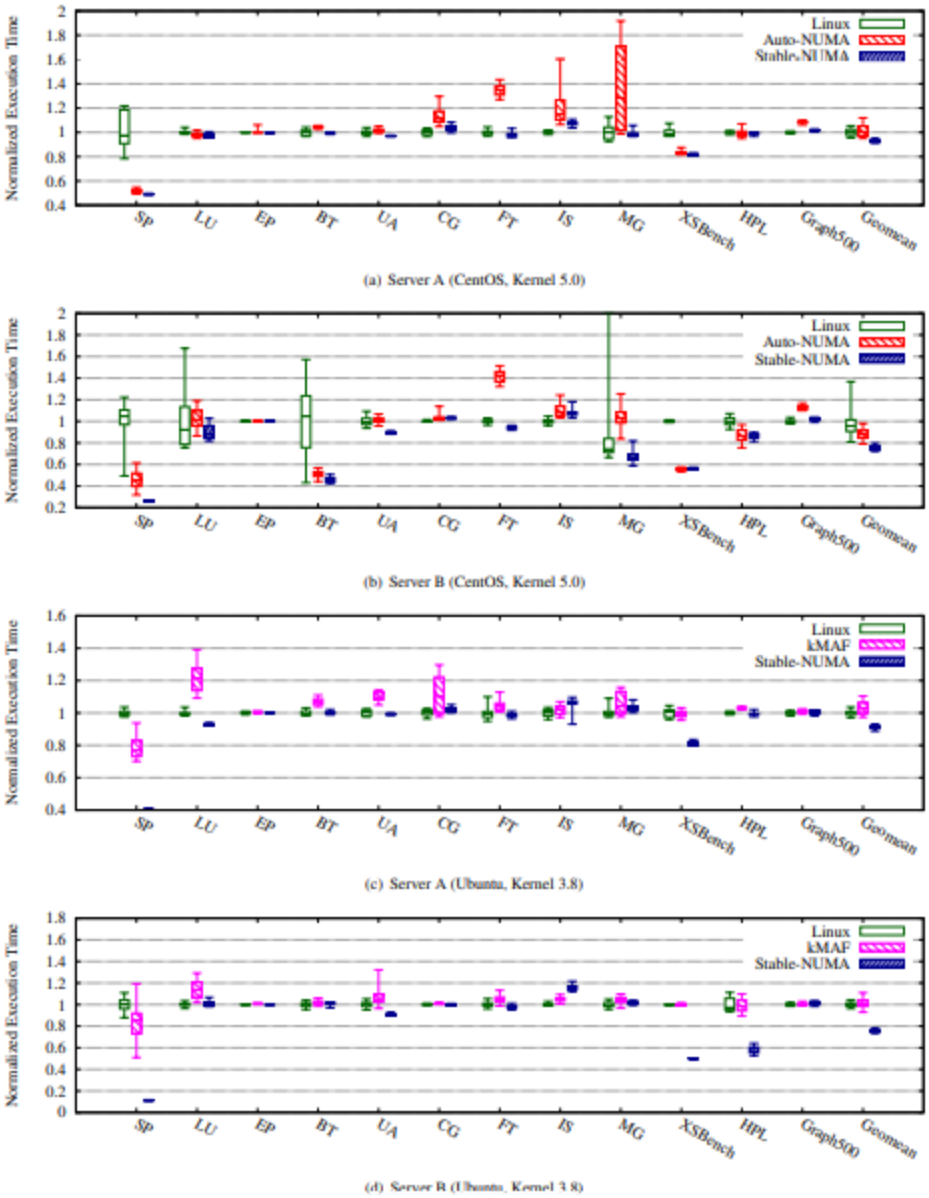


FIGURE 9. Normalized execution time while running on a single server.

Figure 9 shows the workload execution time and distribution, which are both normalized to the Linux average. Figures 9(a) and 9(b) indicate that StableNUMA outperforms Linux and Auto-NUMA for the vast majority of workloads.

The time it takes for all of the threads to reach a reduction point is delayed by the execution time variance across them, lengthening the overall execution time. In an environment with a large number of CPU cores, this delayed execution time is increased. As a result, the execution time on Server B is improved more than on Server A in Stable-NUMA.

Figures 9(c) and 9(d) illustrate that, like Linux and kMAF, Stable-NUMA increases performance stability for most workloads. Stable-NUMA had equivalent or superior

local memory access ratios for all workloads. In the studies, the lower number of page faults had a significant impact on performance stability and average execution time.

- For the vast majority of workloads, stable-NUMA reduces average execution time.
- For both workloads, the influence of thread and page placement by Stable-NUMA is modest.
- For both workloads, the influence of thread and page placement by Stable-NUMA is modest.
- In terms of average execution time and performance stability, Stable-NUMA outperforms Linux and Auto-NUMA as the number of nodes participating in the calculation increases.
- The amount of induced page faults is effectively limited by Stable-NUMA.

In conclusion, the tests show that the proposed system successfully detects thread memory access patterns with little overhead and puts threads and pages in the best possible order to maximize the local memory access ratio. As a result, the proposed approach greatly improves performance stability, resulting in a faster execution time. The performance gain using Stable-NUMA is predicted to rise as the number of processing entities grows.

VI. CONCLUSION

For Linux-based HPC systems, this study proposes a performance-stable NUMA management approach. Because of the ping-pong migration of threads and memory pages, the conflict of thread migration between the CPU load balancer and NUMA-aware feature can worsen performance. Because of run-to-run differences in thread and page migrations, this conflict further affects the performance variability of programs. As a result, to address this issue, the scheme detaches inter-node CPU load balancing from the CPU load balancer and aligns it with our NUMA-aware page and thread placement.

Memory access profiling data is carefully collected with the goal of minimizing the performance impact on applications. The data is used to divide threads into groups, with the number of groups matching the number of physical NUMA nodes. Threads are also carefully clustered to reduce the amount of times they access distant memory. The evaluation findings showed that, using the aforementioned approaches, Stable-NUMA significantly reduces performance fluctuation when compared to the Linux kernel with and without the NUMA-aware feature.

PAPER 3

Title

Lightweight Memory Management for High Performance Applications in Consolidated Environment

I. INTRODUCTION

Overview:

Many recent HPC systems use Linux-based OS/Rs as their primary operating system. They provide tangible benefits in terms of use and maintainability, as well as acceptable performance in general.

We notice a proposal in this research to bridge the gap between LWKs and commodity OS/Rs by providing a lightweight memory subsystem for HPC applications in a commodity OS/R where executing several workloads at once is usual.

By circumventing Linux's memory management layer, the solution of HPMMAP gives lightweight memory performance to HPC applications in a transparent manner. HPC applications use HPMMAP to achieve consistent performance while competing workloads are executed on the same local computing nodes, as seen in HPC clusters and "in-situ" workload deployments. When not in use, the technique is dynamically modifiable at runtime and consumes no resources. By this, we show that HPMMAP can decrease variance and reduce application runtime by up to 50 percent when executing a co-located competing commodity workload.

Challenges Faced:

A commodity OS/R architecture will be ill-suited to give appropriate performance for HPC-class applications as HPC systems continue to condense local workloads. This is due to the fact that commodity systems, particularly Linux, are designed to meet a set of design goals that are incompatible with the needs of HPC applications. Commodity systems, in particular, are almost always designed to maximize resource utilization, maintain fairness, and, most importantly, gracefully decrease in the face of rising demand. These goals frequently conflict with those of HPC systems, which are defined by requiring constant performance under extreme loads over extended periods of time.

Proposed Solution:

A variety of alternative architectures based on a lightweight approach have been created as a result of the drawbacks of commodity OS/Rs.

To enable successful consolidation for an HPC capable environment, it is critical to provide both the performance characteristics required by traditional HPC applications, as well as the additional functionality required by both commodity programs and in-situ analysis/visualization workloads.

The following contributions have been made as a result of this work:

Several weaknesses in the Linux memory management architecture are discovered, and the impact on HPC application performance is examined. The HPMMAP architecture is introduced and discussed, and how it may give LWK-like memory performance on a commodity OS/R is demonstrated. On a single node and at scale, HPMMAP can boost performance by up to 50% on a variety of commonly used HPC benchmarks from the Mantevo1 and ASC Sequoia2 suites.

II. HPMMAP

HPMMAP is based on the memory management design philosophy of the LWK OS/R architectures. HPMMAP enables the dynamic partitioning and management of a node's physical memory in a separate and isolated resource management layer. As a result, HPC programs running on a consolidated platform can take use of a specialized lightweight memory management architecture designed specifically for HPC workloads. As a result, HPMMAP not only removes the overheads associated with Linux's commodity memory management design, but it also allows HPC programs to be segregated from other programs.

HPMMAP is merely a kernel module, thus it requires no changes to the commodity OS/R or the applications themselves, bringing the benefits of a lightweight memory management stack to HPC programs in a completely transparent manner.

HPMMAP's architecture is based on modern Linux kernels' ability to selectively disable hardware resources. HPMMAP installs a lightweight subsystem that can run alongside any commodity subsystems that the OS already utilizes, rather than replacing or supplementing any existing memory management technologies.

A. Implementation:

- HPMMAP is a Linux kernel module that can be loaded into a running kernel without having to recompile the kernel or change the system settings.
- The Kitten LWK has a big influence on HPMMAP's memory management layers.
- Using a Linux configuration item called Memory Hot Remove, HPMMAP may pull memory offline from Linux and enforce its own management procedures on it.
- HPMMAP borrows from Kitten once more by using Kitten's buddy allocator to manage offline memory.

Memory offlining aids HPMMAP's process separation by preventing memory contention in commodity memory areas from spilling into HPMMAP memory regions. Furthermore, because offlined memory is available in big contiguous blocks (at least 128 MB, but frequently much more), HPMMAP can always allocate large pages, never having to resort to a smaller page size.

B. Memory Management:

Repurposing upper-level page table directories that were previously mapped by Linux for areas of the process address space that HPMMAP does not map is one technique to manage HPMMAP memory regions. However, Linux's support for a large number of commodity-type features complicates memory management algorithms and makes exchanging page table entries difficult, if not impossible, without causing race conditions and/or defects.

HPMMAP modifies the page table internally rather than attempting to share upper-level page table entries with Linux-mapped regions, relying on the fact that 64-bit processes typically only use a small fraction of their virtual address spaces. After HPMMAP adds the VMA to Linux's per-process task struct, Linux no longer recognizes the region as free and refuses to access the page table entries corresponding to it.

C. Application Interface:

By interposing system calls, HPMMAP can intercept any communication between a process and the kernel that originates in user space. By determining which system calls will allocate or potentially change the address space and redirecting to its own internal versions, HPMMAP may provide lightweight management transparently with regard to both the kernel and the process. HPMMAP must interpose a significantly less number of system calls than the total number of system calls that require virtual memory access.

HPMMAP would require internal versions of the read and write system functions since they both access a process' virtual address space and must validate the correctness of the address regions being sent by the process. The vast majority of system calls that require address verification and/or only access existing memory mappings are wrapped in a sequence of procedures that rely on the validity of hardware page tables rather than the availability of kernel-specific data structures (VMAs). As a result, when performing read/write and other basic I/O operations, Linux is concerned that any supplied user virtual addresses are valid in terms of the hardware state.

D. Device Driver Interface:

Modifications to address spaces that rely on VMAs and other kernel data structures are frequently required for file systems and devices that use direct I/O techniques.

For file systems and devices that use direct I/O techniques, modifications to address spaces that rely on VMAs and other kernel data structures are typically required. Fortunately, the interface required to invoke the Linux memory subsystem to conduct these tasks is confined to only a few helper functions. These interfaces are contained in the get_user_pages and remap_pfn_range functions, which are used to pin a portion of a process' virtual address space in RAM and to map a range of page frames to a specified virtual address range. Enabling these functionalities for HPMMAP address regions is our approach for allowing HPMMAP to be used for these types of direct I/O operations.

Both get_user_pages and remap_pfn_range have a KProbe at the position so that control is moved to the internal HPMMAP version of the function when these functions are called. The PID and virtual address region provided as function parameters are then checked by HPMMAP to see if the requested region has been mapped. If it has, the internal HPMMAP function is invoked; if it hasn't, HPMMAP returns and Linux processes the function call as usual.

III. LINUX MEMORY MANAGEMENT

In the past, Linux has taken a cautious approach to memory management, focused on the needs of commodity systems. While tools to aid HPC-class apps have been added, they are intended to be utilized as secondary components that either operate in the background (Transparent Huge Pages, THP) or require deliberate user configuration (HugeTLBfs). Both of these options provide larger page memory mappings (2/4 MB) than the default (4 KB), which increases speed by lowering page table walks and TLB strain.

While previous solutions provide performance benefits, particularly for HPC applications, they sometimes exhibit undesired behaviors, especially when the system is under a lot of stress.

The following are a few of these concerns:

A. General Linux Design Issues:

- Processes cannot be protected from the effects of memory contention even when they are mapped by large pages.
- Process address space organization is designed for modest page allocations, which often prevent big page mappings, due to alignment problems and permission conflicts.
- Page faults that result in large (2 MB) page allocations take over 350,000 more cycles to complete with THP than page faults that result in tiny (4 KB) page allocations.

- Demonstrates that HugeTLBfs is affected in the same way. While the ability to assign large pages is unaffected by the additional workload, faults handled by tiny pages see a 475,000-cycle increase in page fault treatment time when there is a competing workload.

B. Transparent Huge Pages Limitations:

- Merge operations are initiated by OS-level heuristics that are unaffected by application requirements and can occur at any point during the process execution. Because merge procedures are mutually exclusive with other address space activities, all page faults must block until the merging is done.
- Memory pinning causes large pages to be split into smaller pages, resulting in huge page "splitting."

C. HugeTLBfs Limitations:

- Processes suffer a high frequency of page faults despite the availability of preallocated memory pools.
- Page fault handling has a lot of overhead when there is a lot of memory strain from other workloads.
- Process stacks cannot be mapped on large pages.
- Memory pinning is typically not feasible for short page parts due to a lack of conventional memory.
- Even though HugeTLBfs uses its own preallocated memory pools and can essentially guarantee the presence of available physical memory, it still requires a substantial amount of page faults to back an application's address space.
- Competing workloads have an impact on the page fault handler for HugeTLBfs-supported applications. On a lightly loaded system, HugeTLBfs has a minimal overhead, but the presence of competing workloads has an immediate impact on all applications.

IV. COMPARATIVE ANALYSIS

We aim to:

- To measure HPMMAP's efficacy, we employed Transparent Huge Pages and HugeTLBfs to compare its performance to that of a commodity Linux system.
- To see how well HPC apps perform alongside commodity workloads on the same hardware. It is split into two sections:
- To look into the effects of resource contention on the performance of HPC applications.
- A system with high-speed Infiniband communication between the nodes was used for the second scaling test.

- For each trial, we used the same system configuration, changing only the memory manager that supported the workloads.

A. Single Node Experiments:

This experiment was developed to carefully measure the effects of memory management on HPC application performance when the node is simultaneously operating a commodity application workload because there is no network in use.

Results: In commodity profile A for HPC benchmarks, HPMMAP beats THP and HugeTLBfs, reducing runtime by an average of 15% compared to THP and 9% compared to HugeTLBfs across all benchmarks. Furthermore, the HPC benchmarks show significant consistency gains, with runtime variance falling by a large margin.

HPMMAP boosts performance by 16 percent over THP and 36 percent over HugeTLBfs on average, and the runtime variance is significantly reduced, as in the previous studies.

B. Multi-Node Scaling Experiments:

As previously indicated, we designed studies using both standard commodity-type network connections (Gigabit Ethernet) and high-speed supercomputing-class interconnections (Infiniband).

Results (Ethernet Multi-Node): The HugeTLBfs and HPMMAP environments clearly outperform THP in the HPCCG and miniFE benchmarks, owing to the presence of pre-allocated memory pools that can conduct huge page mappings simultaneously with respect to application allocation requests. For LAMMPS, on the other hand, each memory management performs similarly.

The commodity workload creates convergence across the different memory managers for HPCCG and LAMMPS, but there is a noticeable divergence for miniFE since HPMMAP provides greater performance. Different overheads experienced on separate nodes are unlikely to be propagated through the network in HPCCG and LAMMPS, but communications over the network itself may become the bottleneck to performance. MiniFE, on the other hand, is unlikely to be network-bound, therefore individual node overheads will remain considerable as the application scales.

Results(Infiniband Multi-Node): The results of HPCCG and miniFE are identical to those of the isolated configuration in the Ethernet experiment: HPMMAP and HugeTLBfs outperform THP. Even without competing workloads, it is evident that HPMMAP provides a better environment for LAMMPS to scale than any Linux memory manager in this experiment.

This result demonstrates that, even at small scales, the reduction in runtime variance often reported by HPMMAP in single node tests begins to translate to higher performance. While the miniFE findings from the Ethernet experiment are relatively

comparable, both HPCCG and LAMMPS exhibit a divergence, indicating that HPMMAP provides the best scalability in the face of local-node competition.

As LAMMPS grows in popularity, the performance difference between HPMMAP and both Linux memory managers continues to expand. While the HPCCG results aren't as remarkable, the performance disparity is as wide as 128 ranks, indicating that HPMMAP has a higher chance of scaling than either HugeTLBfs or THP.

V. CONCLUSION

In this study, we looked at the proposed High-Performance Memory Mapping and Allocation Platform and saw how the lightweight memory management it provides can achieve degrees of isolation that are often unattainable in commodity OS/Rs. We demonstrated that programs using HPMMAP experience up to a 50% reduction in runtime and execute in a substantially less variable environment in the face of competing commodity workloads.

PAPER 4

Title

The Effect of Memory-Management Policies on System Reliability

I. INTRODUCTION

Overview:

The impact of memory-management rules on memory dependability is investigated in this research. The experiment was conducted on a variety of memory sizes, and two solutions were presented as a result of the findings.

The analysis of how reliability is affected by the memory space allocated is presented. This can be used to figure out what the link is between memory allocation and dependability. Depending on the relative time it takes to process a fmt level memory loss, different consequences are measured:

- Inverse Performance-Reliability Relationship
- Dual Reliability-Performance Optimums
- Related Performance-Reliability

Due to the very small memory, the page length is very short, which greatly reduces the overall reliability. Two techniques are recommended to remove these long periods:

- an algorithm, selective scrubbing
- The addition of very small amounts of duplexed memory can also lead to important reliability improvements.

II. BACKGROUND AND MOTIVATION

The structure and performance of memory management policies have traditionally been researched. The impact of memory-management rules on memory dependability is proposed as a new research subject in this paper. The behavior of the system under real workloads is difficult to forecast and analyze, while component-level analysis is widely understood.

III. METHODOLOGY

The topic is first investigated on the basis of a virtual memory structure. The performance tradeoff is based on the overall amount of memory allocated to the process being modified. The major goal of this work is to measure the impact of memory allocation on reliability, as measured by the time it takes to resolve a primary memory loss. This research also demonstrates that a memory-management policy combined with natural program activity can improve memory dependability in a similar way as scrubbing.

Acronyms

CISC	complex instruction-set computer
ECC	error correcting code
LRU	least recently used
RIF	reliability improvement factor
RISC	reduced instruction-set computer
SEC-DED	single-error correcting, double-error detecting.

Notation

c	number of blocks assigned to a process
$R_b(t)$, $R_p(t)$	reliability of [block, page] at time t
$R_j(k)$	reliability of complete address trace with k residencies
$R_{db}(t)$	reliability of a duplexed block at time t
$U_b(t)$, $U_p(i,t)$	unreliability of [block, page of i blocks] at time t
$U_j(k)$	unreliability of process with k residencies
$U_{db}(t)$	unreliability of a duplexed block at time t
λ	bit failure rate
d_i	duration of page-residency i
K	total page faults produced by a particular policy
D	delay for buffer miss
T	total references in address trace
R_x	reliability at memory size x
R_M	reliability at maximum memory size
RIF_x	reliability improvement factor for memory size x
Σ	total time of all completed residencies
N	total prior page faults
F_i	initial loading time of page i
$\bar{\Sigma}$	average duration of all completed residencies
p_i	$\Pr\{i$ bits have failed at time $t\}$.

IV. MEMORY RELIABILITY BASED ON BLOCK RESIDENCIES

A. General Framework

We can partition memory into blocks and analyze each block over subintervals of usage as governed by the memory management policy. First issue is to understand the fundamental effect the memory policy has on these subintervals of blocks.

Standard Procedure:

A process' memory is made up of a predetermined quantity of storage (c blocks) that is handled using an LRU policy. When a block of data is addressed but not located in the main memory, it is transferred from secondary storage to the main memory. A block is assumed to be error-free when it is first loaded into memory. The block remains in place until it is replaced in order to create room for an incoming block. If the block has been modified, the full contents must be read; nevertheless, transferring a block from memory to secondary storage has the side effect of removing all accumulated 1-bit faults in the block. The memory transfer to secondary storage is assumed to be done via an ECC (error correcting code) decoder. The trustworthiness of each block is determined by the length of time it has been present (i.e. the time between the initial loading and the exit of the block).

TABLE 1
$c=3$
[produces 9 residencies of average duration 4.67 units]

Block	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
A	×	–	–								×	–	–	–	–
B		×	–	–								×	–	×	×
C			×	–	–	×	–	×	–	–			×	–	–
D				×	–	–	×	–	×	–	–				
E					×	–	–	–	–						
F										×	–	–			

TABLE 2
$c=2$
[produces 11 residencies of average duration 2.64 units]

Block	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
A	×	–									×	–			
B		×	–									×	–	×	×
C			×	–		×	–	×	–				×	–	–
D				×	–		×	–	×	–					
E					×	–									
F										×	–				

In the above 2 tables, comparison of residencies of the same references produced by two memory sizes is shown. The ‘ x ’ indicates a reference and a ‘ - ’ indicates the block remains resident. We notice that the smaller memory produces smaller durations. However, it has a poorer performance due to increased number of residencies.

B. Challenges Faced:

Important Metrics:

Lifetime Function: Plots the mean number of references between page faults as a function of the memory space allocation. The best memory size is found at the knee of the lifetime curve.

space-time function: space-time product is the cumulative memory held over all references and page faults.

Relation: the knee of the lifetime curve is close to the minimum of the space time product

Individual durations do not directly affect the reliability due to the exponential assumption; rather, the sum of the durations of all block residencies is the key measure. The goal in this scenario would be to decrease the space-time product, which would increase reliability. This is not the case, however, with ECC memory. Because numerous faults in a word are required to produce a failure in these memories, the length and distribution of each residency is critical.

As a result, the space-time product is not the best metric to use when looking for optimal reliability.

C. Reliability Analysis:

The assumptions taken are : a block consists of 8 bytes, a (72, 64) code (viz, 72 total bits for 64 data bits) is used for every block of 8 bytes, the code is SEC-DED, 'h' is the constant bit-failure rate, a page consists of 4096 bytes. Up(512,t) is the unreliability of the complete page for duration t. The unreliability of the process:

$$U_j(0) = 0$$

$$U_j(i) = U_j(i-1) + U_p(512,d_i) - U_j(i-1) \cdot U_p(512,d_i), i \geq 1$$

The unreliability of the complete process in which K faults occur during execution is:

$$U_j(K).$$

D. Solution Details and Results Observed:

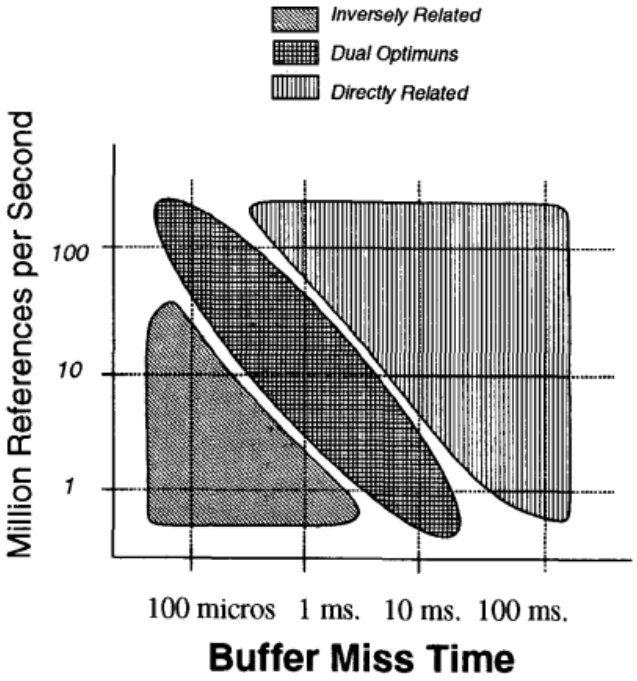
A program address trace through a virtual memory simulation with a fixed space LRU policy was used to collect data. The experiment was repeated with different memory sizes, and statistics for page defects and page residencies were recorded. Metric used: RIF measure - used to compare different reliabilities with the largest memory size as the basis.

The relationship between performance (memory size) and reliability can be seen as follows. Three basic classes of results, based on D (delay to load a block into memory):

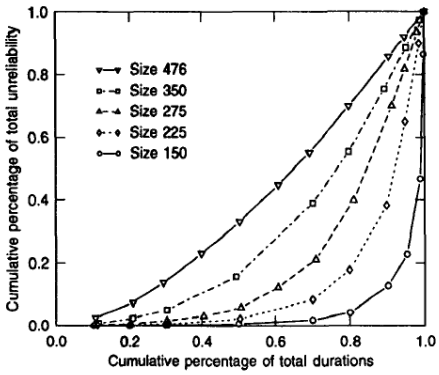
Directly related performance-reliability: As the performance decreases, the dependability improves. This improvement comes from wiping the blocks out of memory more quickly. This happens when D is tiny.

Dual performance-reliability optimums: The maximum value of reliability corresponds to the space-time minimum. This happens at a memory size that isn't at either end of the c scale. For D levels around l04, this happens.

Inversely related performance-reliability: As the performance decreases, the dependability improves. This improvement comes from wiping the blocks out of memory more quickly. This happens when D is tiny.



The behavior of page residency is investigated in depth in order to better understand the impact of memory management on reliability. The correlation between the distribution of page residency durations and overall reliability.



b. Contribution to Unreliability

We deduce that for a memory capacity of 150, just a small part of the residencies contribute significantly to the reliability. If the very small fraction of extended durations is discovered and broken down into many residencies, the potential for far bigger benefits arises. As D grows, the reliability of small memory sizes improves.

Approach for Improved Reliability:

We determined in the previous section that a major portion of the unreliability is attributable to a small percentage of pages having extremely long durations.

Breaking the long durations:

Identifying the appropriate pages and either forcing their removal from memory or refreshing the ECC. An algorithm called selective scrubbing is devised:

$$\Sigma = \Sigma + t - F_i,\ N = N + 1,$$

$$\bar{\Sigma} = \Sigma/N,\ F_i = t.$$

Notation

i	frame number selected for replacement at a page fault at time t
α	a constant parameter of the algorithm.

At periodic intervals all active pages are checked for the condition

$$(t - F_i) > \alpha \cdot \bar{\Sigma},$$

if found, the page is refreshed.

Place them in memory with far more consistency than a conventional SEC-DEC capability:

A paging mechanism known as compiler directed (CD) can identify the pages because the compiler inserts operating system directives to optimize memory management. We hope that a suitable technique has been created to store these long-duration pages in highly trustworthy memory. The following are the outcomes:

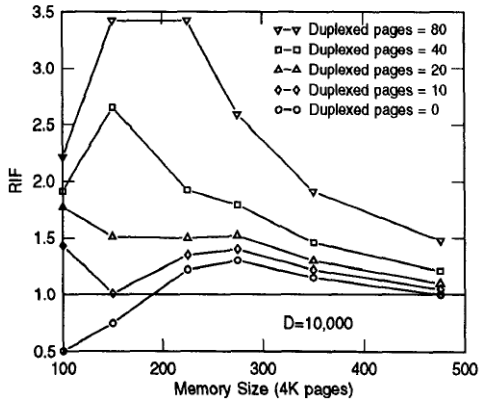


Figure 7. RIF with Highly Reliable Memory [$D = 10^4$]

The duplexed memory begins to remove the major contributors to unreliability for extremely small memories, and the various RIFs converge.

REFERENCES

- A. Pupykina and G. Agosta, "Survey of Memory Management Techniques for HPC and Cloud Computing," in IEEE Access, vol. 7, pp. 167351-167373, 2019, doi: 10.1109/ACCESS.2019.2954169.
- J. Song, M. Ahn, G. Lee, E. Seo and J. Jeong, "A Performance-Stable NUMA Management Scheme for Linux-Based HPC Systems," in IEEE Access, vol. 9, pp. 52987-53002, 2021, doi: 10.1109/ACCESS.2021.3069991.
- B. Kocoloski and J. Lange, "Lightweight Memory Management for High Performance Applications in Consolidated Environments," in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 2, pp. 468-480, 1 Feb. 2016, doi: 10.1109/TPDS.2015.2397452.
- N. S. Bowen and D. K. Pradhan, "The effect of memory-management policies on system reliability," in IEEE Transactions on Reliability, vol. 42, no. 3, pp. 375-383, Sept. 1993, doi: 10.1109/24.257820.