CONTINUOUS ONLINE MEMORY DIAGNOSTIC

by

Musfiq Niaz Rahman

Bachelor of Science, Bangladesh University of Eng and Tech, 2005

Master of Science, University of Pittsburgh, 2013

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences in partial

fulfillment

of the requirements for the degree of

Doctor of Philosophy

in Computer Science

University of Pittsburgh

2016

UNIVERSITY OF PITTSBURGH

KENNETH P. DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Musfiq Niaz Rahman

It was defended on

December 8th 2016

and approved by

Dr. Bruce R. Childers, Dietrich School of Arts and Sciences

Dr. Rami Melhem, Dietrich School of Arts and Sciences

Dr. Wonsun Ahn, Dietrich School of Arts and Sciences

Dr. Kartik Mohanram, Swanson School of Engineering

Dissertation Director: Dr. Bruce R. Childers, Dietrich School of Arts and Sciences

CONTINUOUS ONLINE MEMORY DIAGNOSTIC

Musfiq Niaz Rahman, PhD

University of Pittsburgh, 2016

Today's computers have gigabytes of main memory due to improved DRAM density. As density increases, smaller bit cells become more susceptible to errors. With an increase in error susceptibility, the need for memory resiliency also increases. Self-testing of memory health can proactively check for errors to improve resiliency. Developing a memory diagnostic is challenging due to requirements for transparency, scalability and low performance overheads. In my thesis, I developed a software-only self-test to continuously test memory. I present the challenges and the design for two approaches, called COMeT and Asteroid, that are built on a common software framework for memory diagnostic and target chip multiprocessors. COMeT tests memory health simultaneously with single-threaded and multi-threaded application execution in anticipation of memory allocation requests. The approach guarantees that memory is tested within a fixed time interval to limit exposure to lurking errors. On the SPEC CPU2006 and the PARSEC benchmarks, COMeT has a low 4% average performance overhead.

Despite the promising results, COMeT showed poor scalability on multi-programmed workload environment with high memory pressure. I developed another novel approach, Asteroid, which can adapt at runtime to workload behavior and resource availability to maximize test quality while reducing performance overhead. Asteroid is designed to support control policies to dynamically configure a diagnostic. Asteroid is seamlessly integrated with a hierarchical memory allocator in modern operating systems and is optimized to achieve higher memory test speed than COMeT. Using an adaptive policy, in a 16-core server, Asteroid has modest overhead of 1% to 4%

for workloads with low to high memory demand. For these workloads, Asteroid's adaptive policy shows good error coverage and can thoroughly test memory. Thorough evaluation of my techniques provides experimental justification that a transparent and online software-based strategy for memory diagnostic can be achievable by utilizing over-provisioned system resources.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

I like to dedicate my dissertation to my parents (Mizanur Rahman and Zeenat Jahan Begum), my wife (Meher Nigar) and my son (Shayan Rahman). Without their encouragement and support, none of this would be possible. I like to thank my committee for their valuable feedback to mature the work done in my dissertation. I am thankful to the Computer Science Department for all the help I received during my research. Finally, I am very thankful to my advisor (Dr. Bruce R. Childers) for his enormous support and feedback throughout my research. My dissertation is based upon work supported by the National Science Foundation under grant numbers CCF-1422331 and CNS-1012070.

1.0 INTRODUCTION

Due to a good balance of cost, power, performance and density, DRAM has long been used for main memory. With technology scaling, however, evidence is mounting that reliable operation is becoming a significant challenge for DRAM. Microsoft, Google, AMD and others have found indications that transient and permanent multi-bit errors are prevalent in DRAM [8, 38, 50, 66, 71]. At increasingly small device scales and operating voltages, it is probable that these errors will become even more common at technology node sizes below 22*nm*. Hence, there is a growing need for new resiliency techniques to mitigate memory errors.

A *memory diagnostic* is a resiliency technique that can play a valuable role in error mitigation for memory: A diagnostic exercises memory under varying scenarios to expose marginal locations. A diagnostic's strength is it can exercise memory in *many* ways to expose errors that only manifest themselves in specific conditions. Memory diagnostics are complementary to error mitigation; a diagnostic provides information to guide mitigation. They can be used to assist in online repair and recovery, forestall memory replacement, and direct error correction. For example, page retirement is used in Solaris and Linux to implement error avoidance of "bad" memory locations. It has also been proposed for managing failed memory pages in emerging technologies, such as phase-change memory [22].

When an error is discovered, the physical page frame containing the error is quarantined to avoid future use. A diagnostic could indicate the pages to retire as a triage measure to hold off on replacing a DRAM DIMM until an operator can access the machine. Similarly, diagnostic results can guide error correction. For example, ArchShield is a strong error correction scheme for future memory technologies with high rates of transient and permanent errors [49]. It uses a memory

diagnostic to build a fault map of locations that need extra redundancy for repair. Error correction (ECC) can also guide the diagnostic: When ECC is triggered with a relatively high frequency (i.e., to correct a bit error with a single-error-correction, double-error-detection, SECDED, code) for some region of memory, a diagnostic can be run to more thoroughly test the memory, particularly to check whether the memory is actively failing (i.e., possibility of multi-bit errors that cannot be corrected).

While memory diagnostics extract useful information, they tend to consume significant resources (e.g., bandwidth) and require unfettered access to physical memory. A diagnostic performs write, read and compare operations with test bit patterns in multiple sweeps through the memory. The process is inherently memory bound and time consuming, particularly for the most advanced diagnostics that apply several bit patterns in many sweeps. Consequently, most memory diagnostics are done offline when the system is not actively serving a workload, such as during boot-up [4] or periods of low utilization (e.g., overnight) [65]. However, with server consolidation keeping machines busier and the growing error rates, there is a need for *online* diagnostics that operate while the system does actual work. Of course, a memory failure can happen at any moment, and an online diagnostic will be more responsive than an offline one.

Contemporary computing systems are equipped with a number of processing cores and large main memory capacity to execute a wide range of applications. Over the past few decades applications have become diverse in CPU, memory and IO usage. Therefore, system designers often over-provision system resources to handle workload variations. This over-provisioning provides an opportunity to use under-utilized system resources to perform various system diagnostics that protect the system from potential failures before they take place. One over-provisioned resource is idle processor cycles. These idle cycles can be utilized to test the system memory for errors and thereby minimizing the vulnerability of the system to an error. The primary goal of a memory diagnostic is to increase resiliency for many application types and, hence, it needs to handle and scale for multi-programmed and multi-threaded workloads with similar efficiency. *In this research, my hypothesis is a diagnostic can be integrated into the live OS so that over-provisioned system resources are utilized to make the system more reliable to the user.*

1.1 REQUIREMENTS OF A GOOD SOLUTION

To help address memory resiliency, in this research I designed and developed an online memory diagnostic inspired by conventional standalone memory testers (e.g., Memtest86+ [4] and PC BIOS Power-on Self Test). These conventional approaches usually operate in a non-transparent and offline setting. To make an online diagnostic a good solution, four requirements should be maintained in it.

The first requirement of a good solution to designing and developing this diagnostic is to check memory health online in a system. Because memory can fail at any time, a solution should aim to check memory health in a deployed system, as it is actively serving a workload. Indeed, the actual conditions, e.g., temperature, age and utilization under which the system is operated can influence the appearance of errors and failure. An online test will check the health in an actual operating context, similar to a wearable heart rate or blood sugar monitor.

The second requirement is the diagnostic must minimize its impact on performance of the system. Due to memory access latency and using processor cycles to access memory, a software-based diagnostic can pose considerable impact on system performance. Given these constraints, the diagnostic aims to maximize test rate and thoroughness.

The third requirement is that the diagnostic must be transparent to applications and the OS. Changes in the OS need to be minimized so that the diagnostic does not hamper original OS functionalities. At the same time, applications must be kept unchanged. This requirement also ensures that the diagnostic will become a good candidate for a quick adoption by the user community.

The last requirement is to test memory constantly and thoroughly in a way that stresses the memory to expose marginal behavior, which may or may not actually manifest itself during application execution. A good solution ensures that the test includes the entire memory subsystem and guarantees that memory health is regularly checked. Failures can happen in many places of the memory system, ranging from the memory controller, to the interconnect, to the DIMMs, and individual chips. Even with error correction and scrubbing techniques, marginally faulty locations can behave in a way that is not covered by the error correction. For example, SECDED is com-

monly used to protect the memory chips. However, this code can only correct a single bit error, but multi-bit errors are possible. The aim is to test the memory for marginality; it is not to provide fault tolerance. By proactively testing the memory, resiliency can be improved.

1.2 CHALLENGES

To meet the requirements for a good solution to a memory diagnostic, five challenges must be addressed. These challenges influence various aspects of my diagnostic approach such as its design, operation, and implementation. The challenges are described below.

The first challenge lies in developing a testing strategy. Memory pages can be tested ahead of time of allocation concurrently with running applications or they can be tested on-demand at the time of a page allocation request. Another strategy can be designed that combines these two possibilities. These strategies will have different impacts on performance degradation of the system.

The second challenge is deriving test parameters for my memory diagnostic. There are a number of parameters (e.g., test rate, page expiration time, etc.) that influence testing. During runtime, the diagnostic needs to set and dynamically adjust values for these parameters to ensure performance efficiency and provide a guarantee that memory health is regularly checked.

The third challenge is scaling the diagnostic to large systems. The diagnostic needs to handle multi-threaded and multi-programmed workloads and increasing number of CPUs and memory capacity. It must test large memory capacity in large systems with similar efficiency as small systems.

The fourth challenge is meeting performance constraints. Given a performance budget, the diagnostic should maximize test rate and guarantee a regular health check of the entire memory.

The final challenge is the implementation of the diagnostic. It will be highly dependent on the OS memory management which is a complex piece of software. So, the implementation needs to be carefully crafted and integrated so that the functionalities and subtleties of the OS design and

implementation are not adversely disturbed. Further, the implementation should aim to localize any OS changes, given the complex, monolithic structure of most modern OSes.

1.3 RESEARCH OVERVIEW

In my dissertation, I developed Continuous Online Memory Testing (COMeT), a software framework that can be used to create online memory diagnostic to perform error diagnosis on physical memory. This diagnostic can sweep through memory at regular intervals with a MARCH test [76, 80]. A MARCH test writes specific bit patterns on physical locations of memory, reads them back, and verifies the correctness of the values to determine marginal erroneous locations. Typically, a more thorough test with many bit patterns can discover more lurking error conditions including unusual but possible ones. Memory pages with detected errors can be retired, scrubbed or possibly salvaged for small kernel buffers [77]. By avoiding pages with errors, especially ones with errors that are uncorrectable by hardware mechanisms (e.g., multi-bit errors for SECDED), reliability is improved. COMeT exercises the entire memory system, including the memory controller(s), memory interconnect, DIMMs, and associated glue logic.

The implementation of COMeT retrofits the OS memory manager with a memory diagnostic capability. COMeT consists of a test controller and tester threads. The test controller works as an intermediary between the memory manager and the memory tester threads. It is designed to be adaptive to memory pressure in the system. A tester thread is responsible for executing MARCH tests on a given memory page. A set of testers can be executed per node which consists of a number of cores and memory. When testing is finished, the tester thread returns fault-free pages to the memory manager. A bad memory page is marked unusable and isolated from the OS by the test controller. COMeT is highly configurable by system administrator who can set resource limitation for COMeT. The design of COMeT aims to minimize the performance impact by fairly distributing testing tasks among tester threads depending on their processing capacity and current load while

staying within the resource limit. In addition, COMeT provides functionalities to thoroughly evaluate diagnostic efficiency and performance impact on the system where it is deployed.

1.4 CONTRIBUTIONS

My dissertation makes a number of contributions to the challenges of online memory diagnostic.

First, I presented the design of a software-only process to continuously test main memory's health while the system is up and running. I developed techniques to test memory ahead-of-allocation in a CMP and adaptively adjust test rate to minimize overhead, while achieving a guaranteed bound on the maximum time between successive tests of a page.

Second, I demonstrated a number of parameters which can be used to control an online testing behavior effectively. I provided in-depth guidance to system administrators on proper setting of these parameters based on their target system of deployment.

Third, I developed new algorithms which work with OS memory management to permit the diagnostic and memory management to work in harmony. These algorithms show how to integrate different test parameters to achieve a maximum memory test rate.

Fourth, I addressed COMeT's scalability issues through a set of a novel techniques in Asteroid. Using numerous experiments, I showed how Asteroid can scale a memory diagnostic on multi-programmed workload with a wide range of memory demands. My techniques also addressed non-uniform memory access while testing in large multi-core systems.

Fifth, I showed a number of interesting optimization techniques which can be crucial to meet the given performance budget of any software-based online diagnostic. Some of these optimization techniques also provide helpful insight into OS memory allocation and access pattern in general.

Sixth, I evaluated performance, energy, and resiliency of COMeT to memory errors, including an analysis of important design and configuration choices. Additionally, I showed COMeT's effectiveness to reduce system downtime by proactively isolating bad memory pages.

Finally, I provided a description of how COMeT can be structured and integrated with an OS kernel. This description can be used as a reference for the OS developer community with ideas and subtle design issues on memory diagnostic in the OS.

1.5 THESIS ORGANIZATION

Rest of my dissertation is organized into four chapters. In Chapter 2, motivation and background of my work are described. Relevant works by other researchers on memory diagnostics are also mentioned in this chapter. Chapter 3 introduces my diagnostic framework and provides a high-level overview of components of the framework. Chapter 4 presents COMeT which is a memory diagnostic based on my framework for small-scale systems. COMeT's effectiveness, evaluation and limitations are discussed in detail in this chapter. The corresponding work is published in [56–58]. Asteroid is presented in Chapter 5 which provides an in depth discussion on my novel techniques to eliminate scalability limitations of COMeT in large-scale systems. This chapter also presents a number of interesting optimization techniques in Asteroid to achieve higher test speed than COMeT. The corresponding work is published in [54, 55]. In conclusion, Chapter 6 summarizes the work and contributions of my dissertation and highlights potential future work.

2.0 BACKGROUND AND RELATED WORK

For several decades, DRAM has been the best choice for main memory due to its relatively low cost, aggressive scalability, low power and good performance. Continued decreases in bit cell size have led to large main memories–a laptop can be purchased with several gigabytes of memory, while a server can easily have tens of gigabytes. The capacity enabled by DRAM has had a direct consequence on the applications, execution models and processors employed today. While the relentless scaling of DRAM has been a key enabler, it also has a down side. As cell size and operating voltage are decreased, the memory becomes more susceptible to errors [10, 40, 78]. Background radiation may lead to more "single event upsets" (i.e., independent random bit flips) in a small cell size. Other error types, such as interdependent multi-bit transient and hard errors, are also possible, and indeed probable, at the extreme scales used in high-density DRAM [8, 71]. With new materials, smaller device geometry and increased process variations, these error types may become as important as single event upsets, particularly at operating margins [15, 34, 36, 72]. A program will be corrupted only if a memory location with an error is accessed and the "bad value" is propagated to sensitive state [36, 37, 47, 48].

A two and half year study of reliability in Google's data centers revealed that nearly one third of their machines and 8% of memory modules—using today's memory technology—experienced at least one memory error [66]. The study suggested that the errors were inter-related, and thus, unlikely caused by independent bit flips. This result is even more surprising given that memory with uncorrectable errors was quickly replaced. Further, in another study [50] by researchers from Microsoft analyzed Windows crash reports collected from thousands of consumer PCs to discover the dominant causes of failure. Consumer PCs usually lack the support for error handling (e.g., ECC).

Their study revealed that DRAM error is one of the three major causes of crash and these errors are recurrent. Their error statistics is collected over only about 1.5% of the average memory capacity across several thousands of consumer PCs. Hence, they suspected that the number of DRAM errors in the entire memory is much worse. They also observed a positive correlation between CPU speed and DRAM errors due to the package temperature. While the CPU speed continues to increase, the package temperature will promote more DRAM errors in future. An overarching conclusion of both of these studies was resiliency techniques are necessary to successfully manage memory reliability.

Two very recent large scale studies [8, 71] have revealed a further interesting trend in DRAM faults. Authors in [71] made two important conclusions. First, SECDED ECC is poorly suited to modern DRAM subsystems and the rate of undetected errors is too high to justify the use of SECDEC ECC in very large scale systems. Second, the choice of DRAM vendor is an important consideration for system reliability. In [8], the authors found that correcting codes optimized for adjacent bit errors are less effective. They also noticed that the number of multi-bit corruptions between 7am and 6pm is double the number of multi-bit corruptions during the night. Their conclusion is that the evidence of SDC occurring in an isolated and independent fashion, making it extremely hard to detect and/or predict. Without proper diagnostic, these errors will have significant adverse impact on system by causing crash and silent data corruption. Memory diagnostic techniques will become vital as DRAM node size is further decreased.

While these studies strongly motivate my work, they do not propose a specific technique to combat errors that occur in DRAM chips. Numerous studies have investigated mechanisms for error diagnosis and recovery [36, 37, 64], checkpointing and rollback [52, 70], detection and repair of bugs [18, 25, 51, 53, 59, 63], software fault tolerance [46, 60, 62, 82] and online self test [26, 39]. Many of these methods, including SWAT [36, 64], SWIFT [61], core self testing [26, 39] and duplicated threads [60, 82], rely on free resources (cores, functional units, etc.). However, these approaches target the processor rather than main memory with added hardware support, application modification, compiler support, and/or replicated state.

2.1 TRADITIONAL MEMORY TESTERS

Traditional testers (e.g., Memtest86+ [4], POST by BIOS, etc.) repeatedly execute write, read and verify operations on physical memory locations. Different bit patterns are written and read back to check for the potential and presence of various error types, including multi-bit errors. The advantage to these techniques is they can stress memory with many patterns to find marginal locations that have errors only in particular situations. Memory testers are used during manufacture, boot up, or machine malfunction (for diagnosis). They are not designed to determine memory health in a live system. Because the testers sweep through the whole memory, they can have large test latency. During manufacture, test time is expensive and may be done for only a short burn-in period [14]. Similarly, it can take a long time to thoroughly test memory during the boot process.

2.2 MEMORY ERROR DETECTION AND CORRECTION

Computer memories are often protected by some form of parity-check code. In a parity-check code, information symbols within a word are processed to generate check symbols. These two symbols together form the coded word which is called ECC (error correcting code). Given the fact that soft errors and hard errors are prevalent in DRAMs, ECC has been used to detect and correct such errors in server computers. Most memory controllers use Single Error Correction Double Error Detection (SECDED) ECC because of its low overhead and easy implementation. ECC uses Hamming codes to encode memory bits and Hamming distance to distinguish between valid and invalid Hamming codes. SECDED uses a combination of SEC Hamming code and parity checking where parity checking augments Hamming code to provide double-bit error detection.

For main memory, the most viable hardware approach relies on embedding some form of information redundancy in DRAM. Such redundancy provides a check on data integrity and/or a capability to restore the original data on detecting an error(s). For example, the most popular

DRAM protection practice employs error correction codes (ECC) at the DRAM module level. A typical ECC is capable of correcting a single-bit error and detecting two errors in a single error correction unit (typically 64 bits), commonly referred to as SECDED. Other schemes [21, 83] distribute memory content (and redundancy) across different chips (or different DIMMs) to provide stronger protection against multiple bit errors. For example, IBM's chipkill [21] can scatter memory bits in a single ECC unit to multiple chips. A smart "virtualized" embodiment of the ECC scheme [81] was recently proposed to decouple the actual ECC from the data in memory and separate the process of detecting errors from the rare need to also correct errors (to save energy). Similar "smart" encoding schemes have been applied to caches [7, 28, 29]. Other encoding schemes have also been proposed to reduce bit-flips in DRAM [67] and in emerging non-volatile memory technologies, e.g., phase-change-memory [42–44, 68].

2.3 OS MEMORY MANAGEMENT

The memory diagnostic in a live system must work closely with the OS memory management. The diagnostic will depend on the OS memory manager to collect memory pages for testing. The fundamental concept on how applications and the OS kernel manages their memory is briefly described below.

2.3.1 Application *malloc()*

malloc() is a function for efficient management of dynamic objects in memory. During execution of a process, different objects sizes in memory need to be allocated and released. Without proper management, memory allocation becomes slow, dynamic memory becomes fragmented and memory utilization drops. Further, dynamically allocated memory in Symmetric Multiprocessors (SMPs) can lead to performance degradation due to false sharing of cache lines among threads of a process running on separate cores or processors [9, 35]. Depending on these performance factors (e.g.,

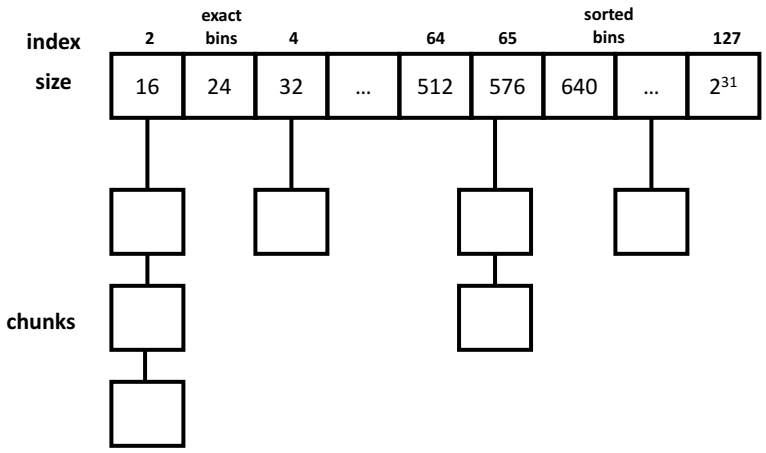

Figure 1: Objects of the same size grouped into bins

speed, memory fragmentation and locality, cache utilization etc.) on various hardware platforms, different variants of *malloc()* have been developed. Fundamentally these implementations create a cache of pre-allocated objects in a memory area called 'heap' in the process address space. These objects are grouped together based on their sizes in the heap. Figure 1 shows a cache with size groups called 'bins'. Each bin contains a list of free objects of corresponding size. Inside a bin, two neighboring free objects might be coalesced to form a free object of larger size and moved to corresponding bin at a higher index. This design was proposed by Doug Lea [33] and used in the GNU C Library. During allocation, objects are taken out of 'smallest-first' object-size bin that 'best-fits' the size of allocation and handed to the requesting process. On release, free objects are put back to the specific-sized bin they belong to. The heap can be extended using the *sbrk()* system call in case a process requires more objects than can fit in the heap capacity. For requests larger than 32 pages, *malloc()* employs *mmap()* to allocate a corresponding number of virtual pages from the OS [35].

2.3.2 Kernel Memory Management

The Linux kernel handles its free memory in two techniques. In the first technique, the kernel uses "Buddy System" to keep track of its free memory pages [20, 41, 45]. Memory allocation is partitioned among global and per-CPU page frame cache (PCP) allocators. Each allocator has free lists that hold blocks of contiguous unused pages. The global allocator uses buddy lists, where a free list is an *order* of rank i that holds free blocks of size 2^i, where $0 \leq i \leq 10$. Modification to buddy blocks during memory allocation and release is done using the *Buddy Algorithm* [32]. For PCP allocation, there are separate free lists (hot and cold) for each core, which locally cache free blocks of size one page. The allocators have separate free lists for different memory zones (DMA, normal and high). Allocation is hierarchical: the PCP allocator is tried first. If this fails, then the global allocator is tried.

The second technique is called "Slab Allocator" [12] which is essentially another implementation of *malloc()*. Each slab in a slab allocator acts as a bin. The slab allocator has all the advantages of application-level *malloc()*. Additionally, to improve cache utilization, the slab allocator uses object coloring to pack objects in a slab at different starting addresses. It reduces the probability that objects from the same slab are stored at the same location in the cache and conflict.

A memory diagnostic must work co-operatively with memory management functionalities provided by the OS. There are several ways a memory diagnostic can be implemented. For example, a diagnostic can be a stand-alone user-mode application which allocates memory using standard library functions, e.g., *malloc()* and test that memory. However, such diagnostic lacks control on the location of the memory pages allocated by the library and the OS. Another technique is to modify the memory management libraries to provide memory testing functionalities. The drawback of this technique is that applications need to be modified and re-built with new libraries. A more effective way is that the diagnostic can be integrated within the kernel as a kernel module or a kernel thread. Such diagnostic has privileged access to system memory and can test memory in background while applications are running. Some code change in the kernel may be required for this diagnostic to gain a better access over system memory resources.

2.4 OS SCALABILITY ISSUES

Contemporary OSes (e.g., Linux, Windows, Solaris) use numerous locking mechanisms (e.g., spin-locks, reader-writer locks, etc.) to synchronize access control to shared resources. For simplicity, these locks were developed to be coarse-grained. Eventually, they were not scalable on multicore machines with more than eight CPU cores [17, 79]. The OS community developed more scalable locks, e.g., RCU locks, MCS locks, etc. and now the community is shifting their focus to the direction where access synchronization scalability is achieved using light-weight message passing [11, 16, 17, 30]. Regardless, these new OSes still need time to mature in design and go through a time-consuming development and rigorous testing before they are used in consumer systems. Further, NUMA is becoming more prevalent in newer systems to manage and scale with increased memory capacity and multiple CPU cores efficiently. Newer Linux kernels support NUMA and provide user-mode libraries (*libnuma* [31]) so that application programmers can use NUMA features. In addition to improved locking and NUMA support, Linux introduced local run queues [6] in process scheduling and improved load-balancing [2] in multicore systems. These improvements and features in the OS can be utilized to make a memory diagnostic scalable with large amount of memory.

2.5 RELATED WORK

Researchers have predicted [13, 15] and observed [8, 19, 50, 66, 66, 71] high incidences of errors in processors and memory chips built at nanometer-scale. Both smaller transistor geometry and higher chip density aggravate the problem. A field study revealed that DRAM error rates are surprisingly high, with 25,000 to 70,000 errors per billion device hours per Mbit and dominated by hard errors [66]. This study found more than 8% of memory modules affected by errors per year. Another study found evidence of SDC occurring in an isolated and independent fashion, making it

extremely hard to detect and/or predict [8]. Other studies [27, 38, 73] made similar observations. This past work motivates COMeT, but does not propose how to combat errors.

Two previous schemes are related to my work, Elm et al. [23] and Singh et al. [69]; like my work, they propose a software memory test strategy. While both proposals implement OS memory testing, the goals and strategies are different than COMeT. First, the past proposals assume ECC and test memory occasionally to catch faults with no consideration for page migration. Instead, COMeT is agnostic to the presence of ECC—it can work both with or without ECC support. If ECC is present, it can be used to triage memory regions that are weak to identify them for more aggressive testing by COMeT. Alternatively, if ECC is not present, COMeT can test memory continuously as described in the paper. Second, neither proposal is adaptive; they allocate a fixed chunk of memory for testing at a fixed rate assigned by the system administrator. They do not relate test strategy parameters and error coverage. Third, they do not study memory test performance on individual programs. For example, Elm et al. [23] measured and reported "system performance degradation" (essentially, lost cycles) due to their memory tester during a one-week experiment with 25 complete tests of a 32MB main memory. Singh et al. [69] employed two programs but detailed memory access behaviors were not discussed or related with the coverage results. Lastly, these previous studies did not consider CMPs.

More recent research (e.g., RAMpage [65] and FlipSphere [24]) has shown progress on software online memory testing. In RAMpage [65], an online scheme is described to test memory during low utilization; it does not guarantee the vulnerability window nor adapt test strategy. RAMpage relies on OS memory allocator interface to get memory pages to test while COMeT is integrated much deeper with the OS memory allocator to sample system memory usage behavior at runtime and get untested memory pages for testing. FlipSphere [24], on the other hand, is a library which provides error detection and correction to malloc system call. It offloads ECC calculation to off-chip hardware accelerator (Intel Xion Phi co-processors [5]). Their technique is on-demand memory testing and suffers from very high performance overhead (85%). COMeT's techniques are orthogonal to FlipShpere and as a future improvement, COMeT can use FlipSphere's hardware-based ECC calculation approach to make MARCH tests more efficient by using off-chip hardware.

3.0 ONLINE MEMORY DIAGNOSTICS

An online memory diagnostic is a software process that checks for errors in physical memory. Errors in DRAM can manifest themselves at any point [8, 71], which necessitates that physical pages are constantly checked. Memory pages with detected errors can be retired and isolated by the OS. By avoiding pages with errors, especially ones with errors that are uncorrectable by hardware mechanisms (e.g., multi-bit errors for SECDED), reliability is improved. In essence, an online memory diagnostic sweeps through memory at regular intervals with a MARCH test [76, 80]. A MARCH test writes specific bit patterns on physical locations of memory, reads them back, and verifies the correctness of the values to determine marginal erroneous locations. Typically, a more thorough test with many bit patterns can discover more lurking error conditions including unusual but possible ones. Multiple MARCH tests can be used with varying periodicity to check for different error types.

3.1 OBSERVATIONS INFLUENCING ONLINE MEMORY DIAGNOSTICS

Although conceptually simple, there are numerous ways that memory testing can be structured and integrated as an online and software-only method. Four observations influenced my design and implementation:

1. The frequency at which memory pages are tested impacts the likelihood that an application will

encounter a page with an error—the more recent an actively used page is tested, the less likely an application will hit an error on the page. Thus, the amount of time between successive tests on the same physical page determines a "vulnerability window" during which a memory access could suffer an error. To test the entire memory capacity can be time consuming, which can lead to a long vulnerability window. However, the vulnerability window should be minimized for a better error resiliency.

2. Memory errors can equally affect single-threaded and multi-threaded applications. An application is vulnerable only to errors on pages that are used. Pages that are allocated, or will be allocated in the near future, must be tested, but unused ones do not have to be checked. Thus, depending on memory utilization, only a portion of physical memory actually needs to be tested. By checking a smaller number of used pages, test frequency can be increased to reduce the vulnerability window. Alternatively, the test frequency can be set for total memory capacity, and less memory could be tested to reduce overhead.

3. Applications often have high page turnover. A page may be requested, allocated and then released quickly. After a page is released, the vulnerability window for that page may not have been reached yet. Thus, a page that has been recently tested and returned does not have to be tested again until the guaranteed duration of the vulnerability window is reached. In essence, this reduces the amount of testing that has to be done. Some pages may be held for a long period and must be tested while they are being used.

4. The test rate only partially determines the actual error exposure of an application. An error may appear shortly after a page is tested, but before the page is tested again. A program will be corrupted only if the error location is accessed and the "bad value" is propagated to sensitive state [36, 37, 47, 48]. Thus, even a modest limit on the vulnerability window duration can be effective.

3.2 MY APPROACH FOR ONLINE MEMORY DIAGNOSTIC

Based on these observations, one way to do memory testing is "on demand". An on-demand strategy can reduce the amount of testing by focusing on pages that are actually allocated. A physical page is tested on-demand when it is allocated and mapped to a virtual page. A page that is held for longer than the vulnerability window is periodically tested by migrating the page to one that is tested.

However, an on-demand strategy is naïve: It introduces large performance overhead since it is inherently sequential. In particular, on a page allocation, the amount of time spent to test a page is fully observed because an application is paused while waiting for its allocation request to be satisfied. Similarly, when page migration is done sequentially with program execution, the program has to wait. Although some migration latency can be masked by testing and copying pages during blocked periods for an application, this process is too unpredictable to guarantee the vulnerability window.

A less time consuming and more predictable approach can check page health *concurrently* to program execution in *anticipation* of allocation. With available idleness in a typical commodity CMP, a free core (or cores) can be used to constantly test memory from which allocation requests are satisfied. Migration can be used to copy long-held pages to ones that are already tested before migration begins. This "ahead-of-time" strategy plans for allocation by executing MARCH tests on pages before requests are placed.

3.2.1 Operation

The key to ahead-of-time diagnostic is the maintenance of a pool of *unused tested pages* from which memory allocation requests are satisfied. The allocation requests may come from an application's memory usage patterns or the test processes employed by the strategy. The tested pages in the pool must be regularly replenished (i.e., newly tested pages are added as pages are removed or grow old). This strategy cooperatively works with the OS memory allocator to provide tested

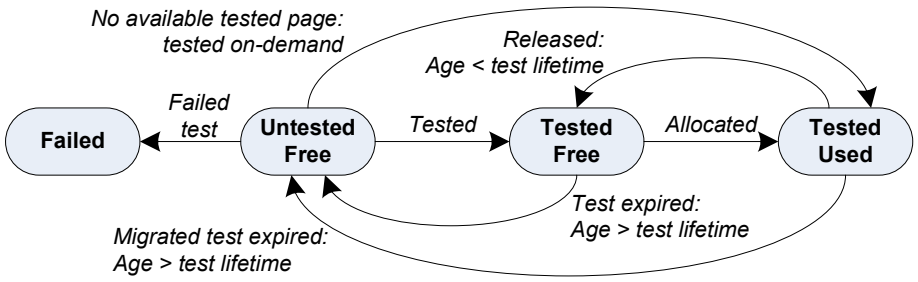

Figure 2: Page State Diagram

pages and gather feedback about memory demand to adjust the replenishment rate.

The ahead-of-time diagnostic guarantees that every page used by an application has had its health checked within a fixed time interval. To express this guarantee, I defined $age(p)$ as the amount of time elapsed since physical page p was last tested. My approach guarantees $age(p)+\delta < test\_lifetime$ for all pages p used by an application. δ is a small constant to allow the diagnostic to check for page expiration prior to it happening. In this way, a bound, determined by $test\_lifetime$, is placed on the vulnerability window which assures a sound memory health.

To understand ahead-of-time diagnostic, first consider a simplified memory allocator. Assume this allocator has a single free list and maintains a page in one of two states – *free* or *used*. A *free* page is on the free list and available for allocation. Physical memory pages are transitioned between these states based on allocation and de-allocation requests. The diagnostic adds states and transitions to the simplified allocator's state diagram to track whether a page is tested or untested. Figure 2 shows state diagram for the diagnostic. $age(p)$ is the amount of time that page p spends in a *tested* state (free or used). Initially, p is *untested free*. At some point, p is selected to be tested and transitioned to *tested free* or *failed*. On a successful test, p is moved to *tested free* and remains in this state until it is allocated or $age(p) + \delta \geq test\_lifetime$. A page that does not pass the test is retired and put in the *failed* state.

When a *tested free* page p is requested, it is transitioned to *tested used*. However, it is possible for p to remain in *tested free* long enough that $age(p) + \delta \geq test\_lifetime$, which is called "test

expiration". To maintain its guarantee, the diagnostic must inspect p's age to ensure the page has not expired. If p expired, then it is transitioned to *untested free* to refresh its test. A different unexpired *tested free* page, q, is selected instead to satisfy the memory request.

Similarly, a *tested used* page p can also expire. In this case, the diagnostic should replace p with a *tested free* page q before p expires. p is migrated to q, and the states of p and q are updated to *untested free* and *tested used*, respectively. This diagnostic must periodically check p's age while the page is in use. Thus, the transition from *tested used* to *untested free* happens on a regular time interval during application execution.

Finally, a transition happens from *untested free* to *tested used* when there are not enough pages in *tested free* to satisfy a memory request. It causes a page to be tested on-demand at the moment of a request, incurring overhead. This diagnostic tries to avoid this transition maintaining enough *tested free* pages to meet instantaneous demand. Although, in reality, errors can manifest while a page is being used by an application, the diagnostic does not handle the situation.

3.2.2 Test Guarantee and Replenishment

My "ahead-of-time" diagnostic relies on two rates. The first rate, termed the "guarantee rate", controls how often to check whether *tested used* pages expire. The second rate—the "replenishment rate"—determines how often to replenish *tested free* pages.

The **guarantee rate** is determined by $test\_lifetime$. To maintain $age(p) + \delta < test\_lifetime$, the rate must be at least $\frac{1}{test\_lifetime - \delta}$. This rate represents only how often my approach needs to check whether *tested used* pages should be migrated and released. Pages that are *tested free* can have their age checked at time of allocation, as previously described. To set the guarantee rate, $test\_lifetime$ has to be determined. A simple strategy based on physical memory capacity and utilization can be used:

$$test\_lifetime = \alpha \times \frac{\left(\frac{mem\_cap}{page\_size}\right) \times mem\_util \times test\_latency}{test\_res}$$

In this equation, $\alpha \geq 1$ is an adjustment factor set by the system administrator to scale $test\_lifetime$ according to system needs. A larger power or performance budget means a higher test rate can be

used. A low error rate would normally permit a slower test rate than the one solely established by memory capacity.

mem_cap is the memory capacity and $page\_size$ is the memory page size. These parameters reflect the physical memory configuration. mem_util is the average percentage of used memory. $mem\_util = 1.0$ sets the guarantee rate conservatively enough that the entire capacity can be tested. However, the whole memory may not be fully utilized (i.e., all pages in *tested used*) at any moment, and the guarantee rate could be set higher by adjusting mem_util. The slack from less than peak utilization can alternatively be exploited to reduce the performance and energy cost of checking memory health since fewer pages have to be checked in unit time.

$test\_latency$ is the amount of time needed to test one page. This latency depends on what tests are done. For example, a comprehensive MARCH test that makes multiple passes over physical memory to read and write different bit patterns could take upwards of 1*ms* per page (there is no caching!). $test\_res$ influences the test rate—the more computational time that the diagnostic is allowed for testing, the faster the guarantee rate. In a "core rich environment", we may be able to dedicate a core(s) to test memory and set $test\_res = 1.0$. In contrast, α can enforce a high test rate even under low memory pressure.

Unlike the guarantee rate, the **replenishment rate** does not effect the vulnerability of an application to errors. Instead, it determines how often the pool of tested unused pages is filled. Its purpose is to avoid the "demand transition" in the page state diagram. To replenish the pool, a set of *untested free* pages need to be acquired, a MARCH test done, and the pages returned. When the pages are released, they are put in the *tested free* state. The page tests are done concurrently to application execution. The higher the replenishment rate, the more likely that an application request can be satisfied with *tested free* pages and the on-demand transition in Figure 2 can be avoided. However, a high replenishment rate puts more load on the memory subsystem. It also removes more pages on average from the memory allocator, impacting the allocator's ability to satisfy requests in overload situations. It also causes the allocator's performance to suffer due to locking and internal bookkeeping (e.g., breaking buddy blocks into smaller ones). If the replenishment rate is set too low, there may not be enough *tested free* pages available, which can cause

overhead due to the on-demand transition.

The "ahead-of-time" diagnostic walks the tight rope between a high and a low replenishment rate by monitoring memory requests to adjust the rate. As described later, my implementation uses an adaptive strategy based on recent history of memory requests to determine a current replenishment rate. The rate by itself is insufficient since it can be satisfied in different ways. Two parameters determine the replenishment rate: $block\_demand$ and $replenish\text{-}\delta$. $block\_demand$ is number of pages to test every interval of $replenish\text{-}\delta$ time. My approach dynamically determines $block\_demand$, but $replenish\text{-}\delta$ is statically fixed.

3.3 ASSUMPTIONS

Several assumptions are made to isolate the fundamental concepts and strategies of my approach from its implementation details in real systems. My assumptions are stated below.

First, my techniques will not test memory that belongs to the OS kernel. The kernel memory is assumed to be fault-free and can be tested with appropriates changes to the OS memory manager. I developed a general methodology for online memory testing and apply that methodology to application memory.

Second, my approach targets detecting hard errors in DRAM including the ones which only manifest under certain operational conditions (e.g., low power budget, temperature, stress on DRAM cells due to reading and writing certain MARCH test patterns, etc.). Detection of SEUs and similar short term transient errors is not a goal of my approach.

Third, on a fault discovery, a memory page is isolated and flagged unusable by the OS instead of replacing the memory module.

Fourth, my approach assumes that the system memory controller exposes information on DRAM row buffer open and close operations. This information is used by tester to ensure test patterns land in DRAM cells from DRAM row buffer during MARCH test.

Fifth, if the target system uses Non-Uniform Memory Access (NUMA) architecture, then the underlying topology is assumed to be known. It can be either automatically collected from the BIOS or manually derived by calculating speed of memory accesses on different nodes from different cores. Discovery of the NUMA topology is not a focus.

Lastly, the target systems are assumed to be over-provisioned and there will be time periods when system recourses are available to perform the diagnostic. However, I evaluated my approaches in overloaded systems to understand how the diagnostic behaves and affects applications in a stressful high-load situation.

3.4 FRAMEWORK FOR ONLINE MEMORY DIAGNOSTIC

In this research, I developed a diagnostic framework, COMeT, which performs error diagnosis on physical memory and scales to multi-core system. At the core, this diagnostic sweeps through memory at regular intervals with a MARCH test. Memory pages with detected errors can be retired, scrubbed or possibly salvaged for small kernel buffers [77]. COMeT exercises the entire memory system, including the memory controller(s), memory interconnect, DIMMs, and associated glue logic.

Figure 3 shows my proposed high-level design of the framework. In this figure, the *Diagnostic Manager* module retrofits the OS memory manager with a memory diagnostic capability. The Diagnostic Manager works as an intermediary between the memory manager and the memory tester threads. A tester thread is responsible for executing MARCH tests on a given memory page. A set of testers can be executed per node which consists of a number of cores and memory. The memory manager is responsible for managing the memory associated with a node. To determine the number of memory pages to test in a given amount of time, the Diagnostic Manager depends on the following five parameters.

1. The **Memory Capacity** of the system.

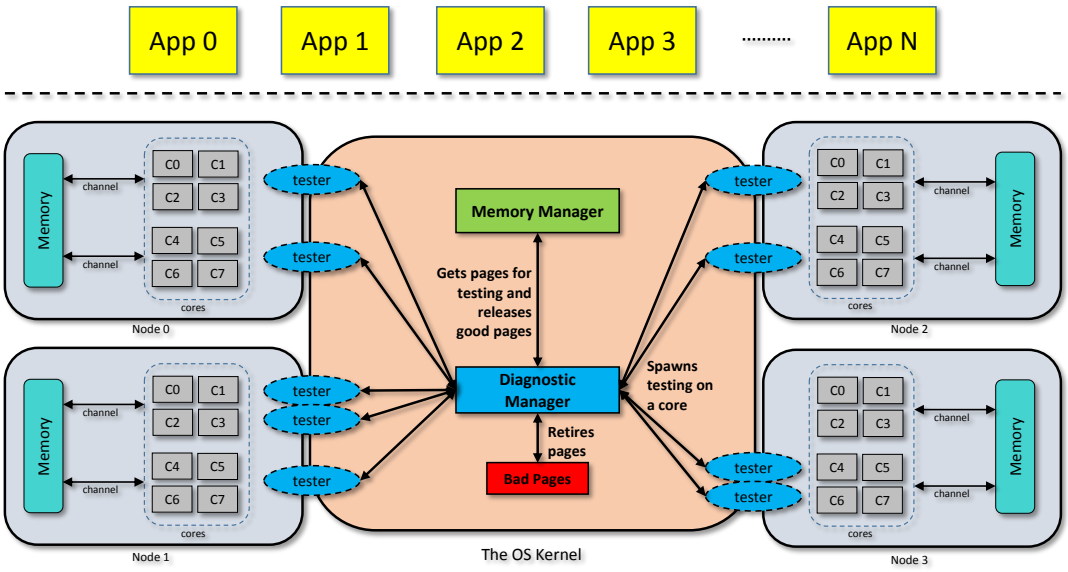

Figure 3: High-Level Design of COMeT

2. The **Memory Utilization** which indicates the level memory pressure the system is currently undergoing.
3. The **Test Latency** to test a memory page depending on the MARCH test.
4. The **CPU Share** the tester threads will get depending on system workload.
5. The **Scalability Requirement** which impacts performance due to testing on large-scale multi-core and multi-socket systems.

After determining the number of pages to test, the Diagnostic Manager pulls pages out of the memory manager and hands those pages to a tester thread. When testing is finished, the tester thread notifies the Diagnostic Manager which returns fault-free pages to the memory manager. A bad memory page is marked unusable and isolated from the OS by the Diagnostic Manager. My design of the framework aims to minimize the performance impact by fairly distributing testing tasks among tester threads depending on their processing capacity and current load.

Figure 4 gives an overview of framework components. It is integrated in the OS kernel and works collaboratively with memory allocation. There are three components: 1) Test Controller

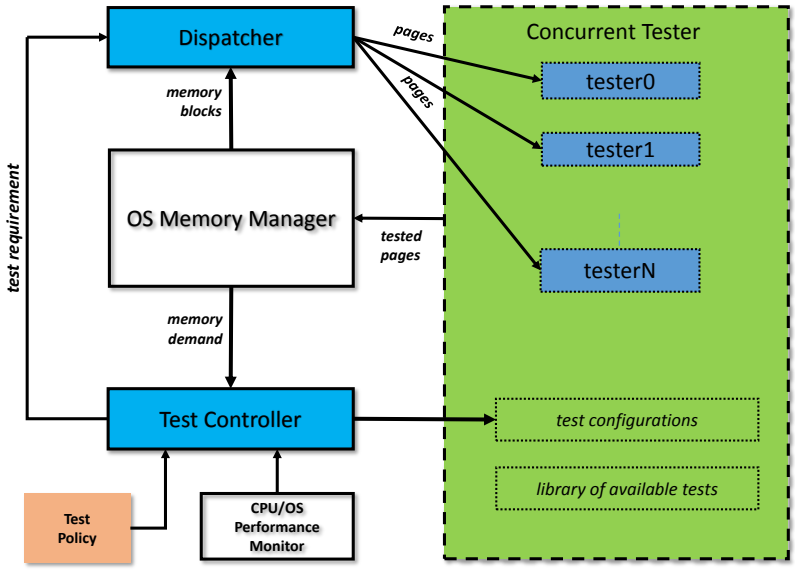

Figure 4: Framework Components

(TC), 2) Test Dispatcher (TD), and 3) Concurrent Tester Threads (CTT).

TC determines *how* memory should be tested and sets diagnostic parameters, or *configuration*, accordingly. A *Test Policy* is incorporated in the Test Controller to set the configuration; the policy is developed by the administrator for a target system. The Test Policy is responsible for only configuring the diagnostic; it decides *how* to test, rather than *what* and *how much* to test. Depending on Test Policy and memory demand information from the OS, TC determines how much and what to test and passes this information to TD. TD extracts physical pages frames from the memory allocator and distributes them to software threads that do memory tests, CTT. CTT is a collection of kernel threads that test memory. TD is node-aware for multi-processor systems and balances work distribution. This design not only ensures scalability but also enables an administrator to implement system-specific diagnostic control policies and memory test algorithms.

The design of the framework is flexible enough to optimize a diagnostic for small-scale systems with limited resource availability or to maximize a diagnostic to support various diagnostic policies on large-scale systems. COMeT is not a substitute for hardware ECC, which should be

used whenever possible. COMeT can work cooperatively with hardware ECC to proactively and thoroughly test memory. In this case, it would operate post ECC to triage memory modules to determine degree of marginality and potential for uncorrected multi-bit errors. Similarly, COMeT can be used to focus checkpointing on marginal memory modules. In the following chapter, I describe how COMeT framework can be used to design and implement diagnostic policy suitable for single-threaded applications on small-scale systems.

4.0 ONLINE MEMORY DIAGNOSTIC IN SMALL-SCALE SYSTEMS

The requirements for an online diagnostic (i.e., transparent operation in a live system, low performance overhead, support for both single-threaded and multi-threaded programming models and bounded memory error vulnerability) pose important questions about its design, operation, and implementation. In this chapter, I answer these questions, including: 1) what is an appropriate design that is both performance efficient and can guarantee that memory health is regularly checked; 2) how should an online diagnostic be implemented and integrated in an actual system; 3) is a software-only memory diagnostic feasible from an implementation and performance perspective and can it improve memory resiliency; and, 4) is this diagnostic capable enough to handle multi-threaded workloads.

For small-scale systems, COMeT is designed to work with both single-threaded and multi-threaded applications. COMeT operates along-side an OS memory manager. So, many of its design choices depend on OS memory manager functionalities. Next sections describe these design choices including the architecture and algorithms behind COMeT. The choices are made in the context of a prototype in Linux that examines how the approach can be integrated with a modern operating system. The design is discussed below.

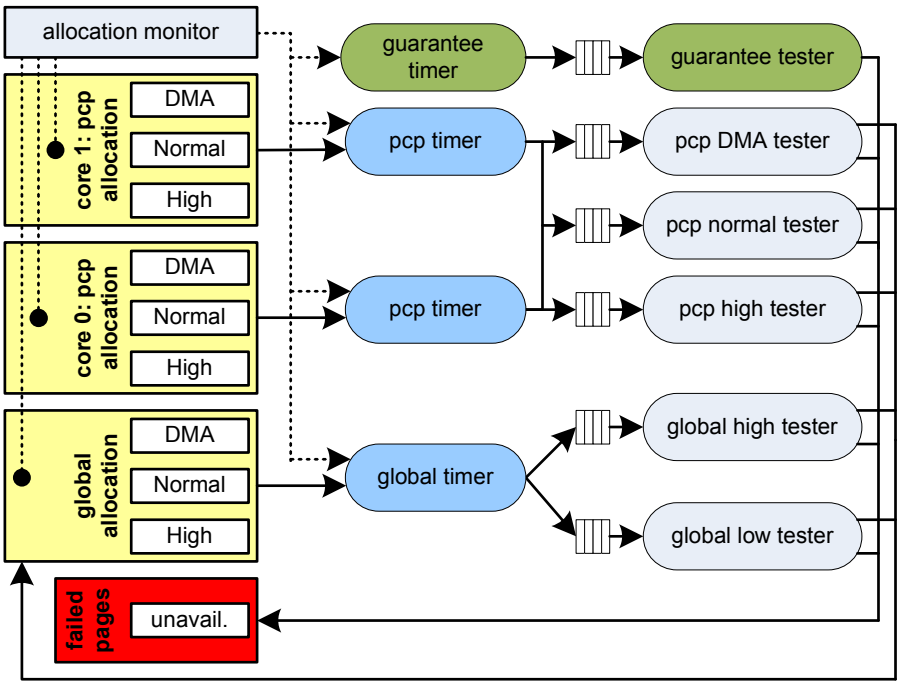

Figure 5: Design of COMeT

4.1 ARCHITECTURE

Figure 5 shows COMeT's implementation, which adds several timers and threads to the OS kernel. As the figure shows, memory allocation is partitioned in a modern OS (e.g., Linux) among global and per-CPU page frame cache (PCP) allocators. Each allocator has free lists that hold blocks of contiguous unused pages. The global allocator uses buddy lists, where a free list is an *order* of rank i that holds free blocks of size 2^i. There are orders $0 \leq i \leq 10$ in Linux. For PCP allocation, there is a separate free list for each core, which locally caches free blocks of size 1. The allocators have separate free lists for different memory zones (DMA, normal and high). Allocation is hierarchical: the PCP allocator is tried first, and if this fails, then the global allocator is tried.

Most changes made to the OS kernel for COMeT are independent of the actual memory allo-

cators, with three exceptions. First, I added interfaces to the memory allocators to request untested blocks and insert tested blocks into free lists. These interfaces can be invoked only by kernel threads. Second, I changed the memory allocators to gather and expose information about memory request demand. Finally, the allocators are modified to return a tested block for an allocation request. If a tested block is unavailable, then the allocator invokes a MARCH test on a block to satisfy the request. This change implements the demand transition from Figure 2.

COMeT introduces new kernel timers to guarantee the vulnerability window and to replenish the free lists with tested pages. Figure 5 shows four timers: one for the vulnerability window guarantee ("guarantee timer"), one for global allocation ("global timer"), and two for per-core allocation on each core ("pcp timer"). When the guarantee timer expires, physical pages are checked for test expiration. When an allocation timer expires, its handler pulls untested blocks, based on monitored demand, from the appropriate free lists to be tested.

New kernel threads are also added to do the actual testing (e.g., "pcp normal tester" and "global low order tester"). Each thread has an input work queue used to accept blocks for testing. The threads are arranged by zone and buddy order rank. For PCP allocation, there are three threads – one tests DMA pages, one tests normal pages, and the last one tests high pages. Similarly, for global allocation, one thread tests the low buddy orders and the other tests the high orders. This organization localizes testing and permits different thread priorities based on latency and importance to satisfying memory requests.

The timer handlers pass memory blocks to these threads for testing. The testers check the blocks, and return blocks to the allocators. When a test finds a page with an error, the page is put into a fault map of failed pages. A page is never allocated again once in the fault map.

4.1.1 Allocation Monitor

The allocation monitor holds a buffer of counters to track allocation events. The counters record memory demand over the time interval $replenish\text{-}\delta$. Each free list has its own counters because demand varies per list. There are two counters for each list: $allocated$ counts the number of blocks allocated and $tested$ counts number of free tested blocks. The counters are set by the allocation

timer handlers. Each counter is 16 bits which rarely saturates for small $replenish$-δ values.

4.1.2 Guarantee Timer and Handler

Figure 6 shows pseudo-code for the guarantee timer's handler. The handler checks whether a page is about to expire (i.e., $age(p) + \delta \geq test\_lifetime$). To check for test expiration, the handler compares the current time to a $timestamp$ for each allocated physical page. When a page, p, is allocated, its $timestamp$ is set to $t_{alloc}(p) + test\_lifetime - \delta$. On line 1, the current time, $time$, is queried. FIND-EXPIRED-PAGES returns a list of pages whose $timestamp \geq time$ on line 2.

```
GUARANTEE-HANDLER()
1  TimeStamp time = GET-TIME()
2  PageList expired = FIND-EXPIRED-PAGES(time)
3  foreach Page page in expired
4    Page newpage = ALLOCATE(0)
5    MARK-PTE-MIGRATING(page)
6    COPY(newpage, page)
7    UPDATE-PTE(newpage, page)
8    RELEASE(page,0)
9  SET-TIMER(δ − (GET-TIME() − time))
```

Figure 6: Check expiration of *tested used* pages

Lines 3 to 8 migrate an expired page by allocating a new tested page and copying the old one to it. During migration, page table entries (PTEs) for the old page are flagged. This action causes a process to fault when it touches a migrating page and to be paused until the migration completes. Once the old page is copied to the new one, the associated PTEs are updated and the old page is released. Line 9 re-arms the timer which is set to expire in δ time, taking page migration latency into account. As long as the handler finishes in δ, the promise on the vulnerability window is satisfied. However, my implementation does not guarantee this condition. δ is large enough (e.g., 10 seconds) that the migration will likely be completed before the next timer event. If such a "deadline miss" actually arises, then migration can be parallelized for better performance.

4.1.3 Global Allocation Timer and Handler

To replenish the pool of tested pages, COMeT regularly extracts and tests blocks of pages. Figure 7(a) shows pseudo-code for this process. Each global allocation zone has a timer. When a timer expires, GLOBAL-HANDLER is invoked with an identifier for the zone ($zone$).[1] GLOBAL-HANDLER forms two lists: $lowlist$ and $highlist$ (lines 1 and 2) that collect untested blocks based on order rank. Blocks put into the lists are tested by the global tester (Section 4.1.4). Lines 3 to 18 build the two lists. The **for** loop iterates over the buddy orders. A descriptor for an order is accessed on line 4.

```
GLOBAL-HANDLER(MemoryZone zone)
 1  PageBlockList lowlist = NIL
 2  PageBlockList highlist = NIL
 3  for Integer id = 0 to MAX-ORDER(zone)
 4    BuddyOrder order = zone.freelist[id]
 5    ADJUST-REPLENISH-RATE(order)
 6    Integer blockdemand = BLOCKS-TO-TEST(order)
 7    if blockdemand ≠ 0
 8      PageBlock block = TAIL(order)
 9      while block ≠ HEAD(order) and blockdemand > 0
10        PageBlock prevblock = PREV(block, order)
11        if IS-TESTED(block) == FALSE
12          REMOVE(block, order)
13          if order ≤ LOW-ORDER
14            ENQUEUE(block, lowlist)
15          else
16            ENQUEUE(block, highlist)
17          N = N − 1
18        block = prevblock
19  SIGNAL-WORK(LOW-TEST-THREAD, lowlist)
20  SIGNAL-WORK(HIGH-TEST-THREAD, highlist)
21  SET-TIMER(this, replenish-δ)
```

Figure 7: Handler for global allocation timer

Next, in line 5, ADJUST-REPLENISH-RATE adjusts the replenishment rate (discussed in Section 4.1.5). Once the new rate is set, the number of blocks, $blockdemand$, to test to meet the rate is determined (line 6). BLOCKS-TO-TEST retrieves the replenishment rate which is updated by ADJUST-REPLENISH-RATE at each $replenish\text{-}\delta$ interval. If blocks need to be tested (line 7), then

[1] Although not shown, the actual implementation is re-entrant to handle two or more timer expirations.

up to $blockdemand$ untested blocks are extracted from the order (lines 8 to 18). The **while** loop on line 9 iterates through the order's free list from tail to head. The list is processed in reverse because it is more likely that untested blocks will be at the list's end (these are the "older" blocks). An untested block is removed from the free list on line 12. If this block came from a low order (determined by the constant LOW-ORDER), then it is put into $lowlist$ for testing. Otherwise, it is added to $highlist$. Lines 19 and 20 signal the low and high global tester threads that new blocks are available. Finally, line 21 resets the global timer to $replenish\text{-}\delta$. This parameter controls how frequently pages are pulled for testing; the page quantity (in blocks) is determined from the demand during the last $replenish\text{-}\delta$ interval. Unlike the guarantee rate, the replenishment rate does not have to be exact. Thus, the timer is set to expire in $replenish\text{-}\delta$ time.

4.1.4 Global Tester

GLOBAL-TESTER, shown in Figure 8(a), is the routine that tests page blocks. It consumes the list of blocks produced by the global timer's handler. GLOBAL-TESTER waits until the handler has produced work, and then it invokes TEST-BLOCK on each block in the work list.[2]

A MARCH test is done by TEST-BLOCK. Lines 2-5 test each page in a block with MARCH-TEST. The page is marked uncacheable prior to the MARCH test to avoid the caches filtering accesses to physical memory. The MARCH test operates on virtual addresses that are known to be mapped to the physical page. If the MARCH test fails, then the **foreach** loop terminates and lines 6 to 9 are done. Line 7 splits the block with the failed page into two or more smaller blocks. The block is split at the failed page. SPLIT-LIST-AT ensures that the new blocks have an appropriate number of pages (i.e., an integral power of 2). The new blocks are put in the work list on line 8 and the failed page is retired on line 9. If the test succeeded, then the block is released back to the allocator on line 11. A small block from a split is released to the appropriate buddy order (this step is not shown, but implied by RELEASE on line 11). Line 12 increments the $tested$ counter for the block's order.

[2] GLOBAL-TESTER handles delivery of new blocks during test, although this is not shown.

```
GLOBAL-TESTER(Signal sig, PageBlockList work)
1  while (TRUE)
2    WAIT-ON-SIGNAL(sig)
3    while (work ≠ NIL)
4      PageBlock block = DEQUEUE(work)
5      TEST-BLOCK(block, work)
```

(a) Test each block in work list

```
TEST-BLOCK(PageBlock block, PageBlockList work)
 1  Boolean isfailed = FALSE
 2  foreach Page page in block and isfailed == FALSE
 3    MARK-UNCACHEABLE(page)
 4    isfailed = MARCH-TEST(page)
 5    MARK-CACHEABLE(page)
 6  if isfailed
 7    PageList frags = SPLIT-LIST-AT(page, block)
 8    ENQUEUE(frags, work)
 9    PERMANENTLY-RETIRE(page)
10  else
11    RELEASE(block)
12    monitor.INCREMENT-TESTED(GET-ORDER(block))
```

(b) Test blocks and release good ones

Figure 8: Tester for global allocation

4.1.5 Adaptive Test Rate

To change the test rate according to application memory requirements at runtime, COMeT uses an adaptive algorithm. This algorithm introduces five states for each of the free lists for each order. Each state tracks how the current memory demand affects the test rate on that free list. These five states are:

1. **State A:** The test rate is set to a minimum value that is configured during COMeT's initialization. The default minimum rate depends on the usage of the free list.
2. **State B:** The current test rate is doubled. This state usually signifies that COMeT is responding to increased memory demand.
3. **State C:** The current test rate is kept constant. This state signifies that memory demand has

been reduced but still high enough not to drop the test rate.

4. **State D:** The current test rate is decreased by half. This state signifies that memory demand is low enough to decrease the rate. If the rate falls below the minimum rate, the minimum rate is set for that free list.
5. **State E:** The current test rate is set to 0. This state signifies that there are enough tested pages in the free list and memory demand is also low.

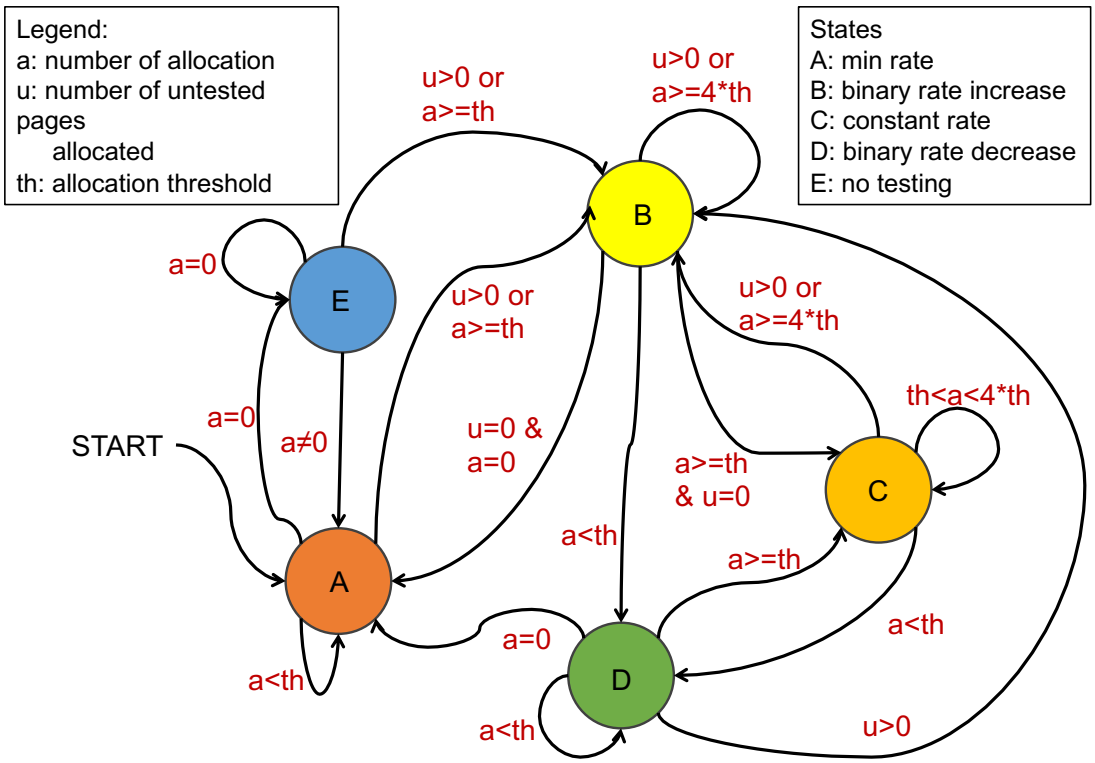

Figure 9: State Transition during Test Rate Change

For each free list, a change from one state to another depends on three factors: 1) **a**: the number of pages allocated since the last rate adjustment; 2) **th**: a threshold value that signifies that the free list is high memory pressure and will soon run out of tested pages; 3) **u**: the number of untested pages allocated since the last rate adjustment.

When COMeT is initialized, a free list is in state **A** to ensure that there will be tested pages on that free list. The simplified algorithm for ADJUST-REPLENISH-RATE is shown in Figure 10. Lines

```
ADJUST-REPLENISH-RATE(BuddyOrder order)
 1  Integer state = CURRENT-STATE(order)
 2  Integer rate = CURRENT-TEST-RATE(order)
 3  Integer a = monitor.GET-ALLOCATED(order)
 4  Integer u = monitor.GET-UNTESTED(order)
 5  Integer th = monitor.GET-THRESHOLD(order)
 6  new-state = GET-NEW-STATE(state,a,u,th)
 7  new-rate = GET-NEW-RATE(new-state,rate)
 8  SET-REPLENISH-RATE(order, new-rate)
 9  SET-NEW-STATE(order, new-state)
10  monitor.RESET-ALLOCATED(order)
11  monitor.RESET-UNTESTED(order)
```

Figure 10: Adjust buddy order's replenishment rate

1-5 of the pseudo-code read the current state $state$, the current rate $rate$, the number of allocated pages a, the number of untested pages u and the threshold th from COMeT's monitoring feature added in the OS. Based on $state$, a, u and th, line 6 invokes GET-NEW-STATE to calculate the next state for the free list. The state changes are summarized in Figure 9. As long as $0 < a < th$, the state remains the same enforcing a minimum rate of testing. If a falls to 0, there is not enough memory demand and the free list goes to state **E**. In state **E**, COMeT stops testing for that free list. From state **A** and **E**, if any untested page was allocated ($u > 0$) or the memory pressure increases ($a \leq th$), the free list switches to state **B**. In state **B**, the test rate is doubled at each invocation of ADJUST-REPLENISH-RATE. As long as memory pressure is very high ($a \geq 4 \times th$) or there are more untested pages allocated ($u > 0$), the free list stays in state **B**.

If the memory pressure decreases but is still over the threshold ($a \geq th$) and there are no untested pages allocated ($u = 0$), the free list moves to state **C**. In state **C**, the test rate is kept constant. As long as memory pressure is relatively high ($th < a < 4 \times th$), state **C** remains in effect. If the memory pressure rises again ($a \geq 4 \times th$) or there are untested allocated pages ($u > 0$), the free list moves back to state **B**. If the memory pressure decreases further ($a < th$), the free list goes to state **D** where the test rate is reduced by half each time ADJUST-REPLENISH-RATE is invoked. The free list remains in state **D** as long as the memory pressure is sufficiently

low ($a < th$). Any untested page allocation forces the list to go to state **B**. If the pressure again rises beyond the threshold, the free list can move to state **C**. If the memory pressure becomes 0 ($a = 0$), the free list moves back to state **A**. This approach scales the rate quickly upward, but avoids decreasing too rapidly, particularly given the relatively small time intervals over which demand is measured. Once GET-NEW-STATE calculates the new next state, GET-NEW-RATE fixes the test rate based on the new state and the current test rate. Lines 8 and 9 set the new test rate and new state into effect for the free list. Lines 10 and 11 resets a and u to track the number of allocated pages and untested allocated pages between two calls to ADJUST-REPLENISH-RATE.

4.1.6 Page Migration

Page migration is required for expired or near-expired pages of an application. COMeT periodically walks through the application page table to examine the time stamps of the pages (see Section A.2.5 for Linux-specific details). If an expired or near-expired page is found, COMeT migrates that page to a free page that was recently tested. Usually the long-held application pages are the primary targets of migration. This migration process is expensive due to the latency of copying data from the old page to the new one and fixing the page table entry. During data copying, the application is stalled only if it tries to access the page. Hence, the frequency of page migration can cause adverse performance impact on an application. Less frequent page migration may result in more expired and near-expired pages that need to be migrated. More frequent page migration will require more page table walks which can be expensive. In COMeT, the page migration interval can be easily configured or turned off using a page migration control knob.

COMeT's page migration supports multi-threaded applications. In multi-threaded applications, usually the code and the data are shared among threads via a single address space which persists throughout the runtime of the application. Thus, in multi-threaded applications, the number of shared pages is higher than those in single-threaded applications. Fixing a shared page table entry during migration takes longer due to the reverse-page-table lookup in the OS kernel and multiple page table entries that must be fixed. This page table modification entails eventual TLB shootdowns on CMP machines. TLB-shootdowns adversely harm performance as page table

Baseline Configuration	
$test\_lifetime$	3*min* vulnerability window
δ	10*sec* (5*sec*, 20*sec*, 30*sec*)
$replenish\text{-}\delta$	25*ms* (15*ms*, 35*ms*, 45*ms*, 0 for On-demand)
α, mem_util, $test\_res$	1.0 (test all memory at maximum rate)
$test\_latency$	0.4*ms* (MATS MARCH test was used [80])
Minimum test rate	Same as guarantee rate

Table 1: Configuration

entries must be reconstructed through expensive page table walks in subsequent accesses. If the vulnerability window is very small, more pages may be migrated causing more TLB shootdowns and further performance degradation.

Most processors support OS-guided TLB-shootdown. Linux uses inter-processor interrupts (IPI) to invalidate shared TLB entries during page migration (Section A.2.6). In the current implementation, COMeT only gathers statistics from the native IPI handling code. The effect of TLB-shootdowns due to COMeT's operation is described in Section 4.2.7.

4.2 EVALUATION

To determine COMeT's effectiveness, I investigated how many pages are tested, the performance overhead and energy of my implementation. I studied several design parameters and compared my approach to a naïve on-demand testing strategy (see Section 3.2), COMeT and a baseline without testing. I also analyzed at the effect of TLB-shootdowns due to COMeT's operation in multi-threaded applications.

	Experimental Setup (Single-threaded Benchmarks)	Experimental Setup (Multi-threaded Benchmarks)
Processor	Pentium 4 D (Presler 930)	Intel i7
Speed/cores	3.0 GHz, dual-core, no hyperthreading	2.1 GHz, four-core with hyperthreading
mem_cap, $page\_size$	1 GB, 4,096 bytes	6 GB, 4,096 bytes
Linux kernel version	2.6.24.3	3.1.1
Benchmarks	SPEC CPU2006, ref. input data sets	PARSEC, native input data sets

Table 2: Experimental setting

Program	Time	Util.	Program	Time	Util.
povray	751.5s	14.9%	omnetpp	994.9s	35.2%
caclulix	2557.2s	19.9%	gromacs	1632.7s	35.7%
namd	1173.0s	20.3%	gamess	1632.0s	36.8%
gcc	111.6s	21.2%	libquantum	1789.6s	42.2%
gobmk	193.7s	21.3%	leslie3d	2179.2s	45.3%
h264ref	189.5s	22.5%	xalancbmk	797.8s	52.6%
sphnix3	1579.6s	24.1%	zeusmp	1555.8s	62.8%
hmmer	494.0s	24.7%	lbm	1340.6s	67.3%
soplex	570.9s	27.5%	milc	1242.0s	82.2%
perlbench	542.6s	29.0%	mcf	948.3s	93.9%
sjeng	1634.1s	31.2%	gemsFDTD	1966.3s	95.4%
astar	518.3s	31.3%	bwaves	1405.2s	97.6%
dealIII	942.7s	32.5%	bzip2	261.8s	98.4%
tonto	1782.8s	34.5%	cactusADM	2395.7s	98.7%

Table 3: SPEC CPU2006 Benchmark Statistics (table is sorted by memory utilization)

4.2.1 Methodology

Table 1 lists on-demand testing and COMeT's base configuration. In the discussion of results, I refer to on-demand testing as "On-demand". It takes about 3 minutes to perform a basic MARCH test on the entire memory in my setup (Table 2). Thus, I used a baseline 3 minute vulnerability window to evaluate On-demand and COMeT in a harsh situation where the test process is kept busy and system resources are taxed. In practice, the expected error rate is likely to be low enough that the vulnerability window can be set to a longer duration. I used the classic MATS MARCH test [80]. The DRAM row-buffer page is opened and closed on every memory access during the MATS test to directly operate on the DRAM array.

For COMeT, a minimum replenishment rate is used to ensure some tested pages are available. For all lists, this rate is set to the guarantee rate. An initial replenishment rate for each free list is set on each program invocation. To get this initial rate, I profiled the benchmarks to find the minimum demand on each list. The minimum among all programs is used for a list's initial rate. To model On-demand, I fix the replenishment rate to 0 and enable page migration.

Table 2 gives two experimental setups. The first setup is used to evaluate single-threaded benchmarks (SPEC CPU2006) and the second is used to evaluate multi-threaded benchmarks (PARSEC). Experiments are done in Linux single-user mode to minimize system activity. Because COMeT imposes overhead on system time, performance is measured as the sum of user and system time (i.e., wall-clock time). I also measure energy using a power meter. Energy is measured for a full SPEC run rather than individual benchmarks due to the meter's limited precision. All cores of the experimental machine are used. The kernel can schedule the benchmark and test threads on the available cores to approximate a "core rich" environment. I used SPEC CPU2006 with reference inputs and PARSEC with native inputs. Table 3 shows the SPEC CPU2006 benchmarks (sorted by memory utilization), their run-time ("Time"), and average memory residency ("Util.") in the first experimental setup with 1GB of main memory. Past work lists the working set sizes for PARSEC [11].

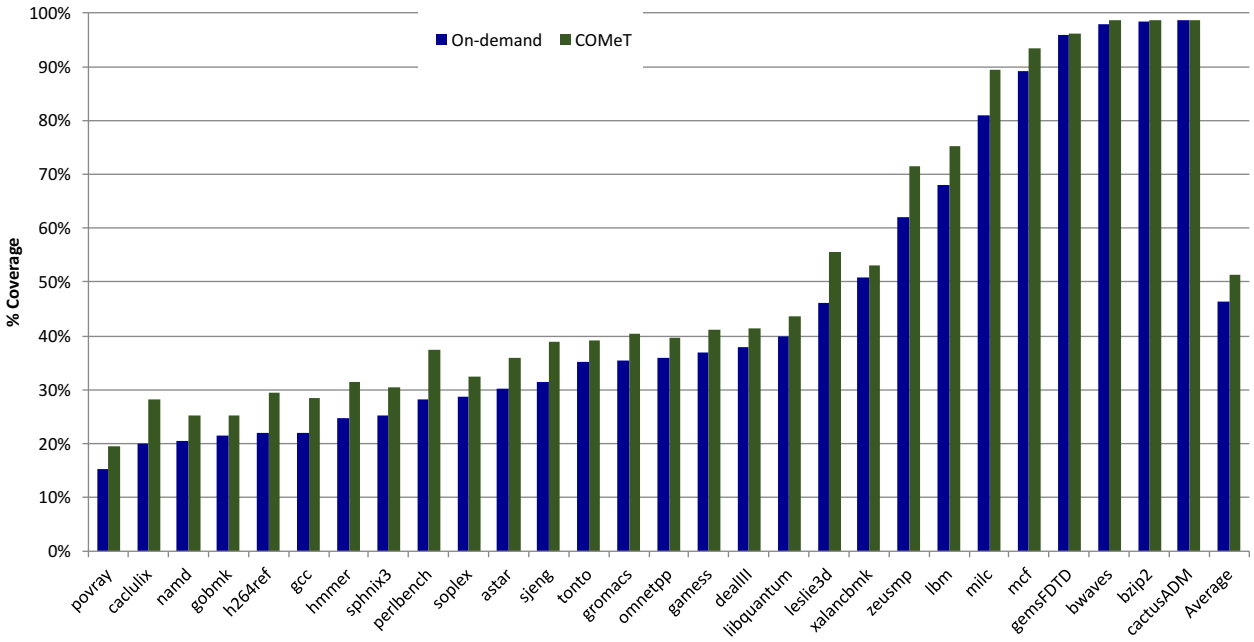

Figure 11: Page coverage (percentage of physical pages tested)

4.2.2 Overall Results

Pages tested: Figure 11 shows the page coverage of On-demand and COMeT. Page coverage is the ratio of the number of pages tested (free or used) to the total number of physical pages. On-demand's coverage is 15.3% to 99%, with an average of 46.4%. The coverage follows the trend of memory utilization. For example, *povray* has the smallest memory utilization (14.9%) and coverage (15.3%). Similarly, *cactusADM* has the highest utilization (98.7%) and coverage (99%). On-demand's coverage is slightly more than memory utilization since pages can be returned before expiration.

COMeT's coverage is 19.5% to 98.6% (51.4% average). The coverage follows the same trend as On-demand, tracking memory utilization. Again, *povray* has the lowest coverage and *cactusADM* has the highest. COMeT always keeps tested pages in the free lists, and naturally, its coverage is higher than On-demand. Coverage also indicates COMeT's prediction accuracy for future

memory demand. The higher coverage relative to utilization reveals there is some overestimation from two aspects. First, COMeT scales up the replenishment rate quickly but degrades it slowly. The delay in degrading the rate causes more pages to be tested than necessary. Second, COMeT conservatively keeps pages in rarely used free lists. It is desirable to overpredict by a small amount to have a reserve of tested pages for unexpected peak allocation.

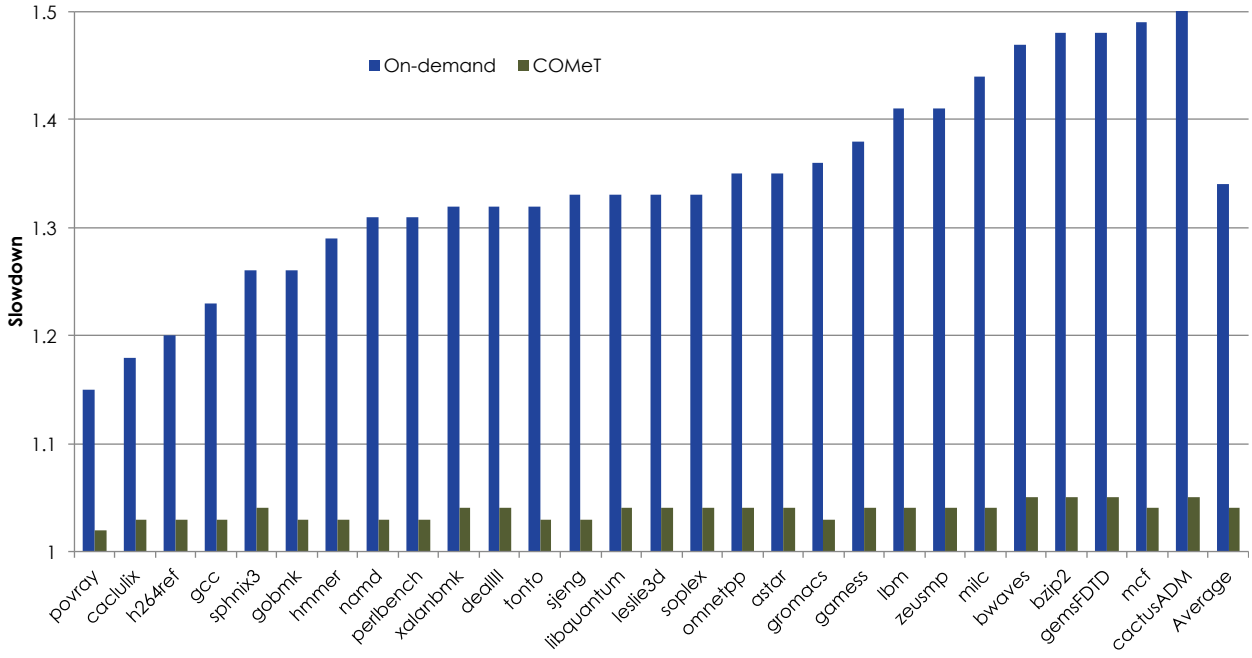

Figure 12: Slowdown relative to baseline without testing.

Performance: Performance overhead for On-demand and COMeT is presented in Figure 12. This figure shows the slowdown experienced by a benchmark versus the baseline without testing. Slowdown is the ratio of a program's wall-clock time with testing to wall-clock time without testing.

On-demand's slowdown is 1.15 (*povray*) to 1.5 (*cactusADM*) with an average 1.34. The overhead generally increases with memory utilization. Because pages are sequentially tested with program execution, the full latency of allocating, testing, and mapping a page is observed, which harms execution time. *bzip2* has this behavior. It has a 1.48 slowdown due to its short execution time (261.8s) but high memory utilization (98.4%). There is less impact in *povray* because its resident size is only 14.9% of memory.

To enforce page lifetime, a large penalty is paid for migration in On-demand. When pages are migrated, target pages are tested on-demand, which dramatically increases migration latency. As a result, in programs with large working sets and high utilization, it is probable that a page under migration will be touched, forcing the program to be paused. In effect, the program cannot make quick progress since it executes sequentially with testing and migration. Several programs (e.g, *mcf*, *gemsFDTD* and *catcusADM*) exhibit this behavior.

Figure 12 also gives slowdown for COMeT, which is a modest 1.02 to 1.05, with an average of 1.04. Programs with high utilization again have the most overhead; however, the actual overhead is small. For example, *cactusADM* and *bzip2* use 98% of memory but suffer only a 1.05 slowdown. For *bzip2*, ahead-of-time testing has helped dramatically. When pages are requested, tested ones can be delivered without the latency of on-demand testing. Similarly, ahead-of-time COMeT has reduced migration latency because target pages are tested ahead of time. Migration is now primarily a page copy operation, which can happen at cache speed. This low latency leads to a smaller chance that a program has to be paused. Even if an application must be paused, it will be stopped for a much shorter duration than with On-demand. COMeT's slowdown comes mostly from increased pressure on memory resources, including the hardware memory subsystem, despite the availability of a free core for testing.

To understand COMeT's behavior on multi-threaded workload, I ran a selection of benchmarks from PARSEC including *canneal*, *streamcluster*, *blackscholes*, *ferret* and *fluidanimate*. These benchmarks have different memory footprints and levels of data sharing [11]. Figure 13 shows the slowdown of these benchmarks. Each PARSEC program was configured with four threads on four cores with my second experimental setup. Similar to the single-threaded benchmarks, the slowdown comes from memory pressure. *Blackscholes*, whose native working set size is about 2MB, suffers less than 2% slowdown. *Canneal*, whose native working set size is 2GB, suffers the most with nearly 8% slowdown. On average, the slowdown is approximately 3.8% across these multi-threaded benchmarks. As the system coverages for single-threaded benchmarks closely followed their memory requirements, I do not show the comparison between on-demand and COMeT's system coverages. In Section 4.2.7, I discuss COMeT's impact due to TLB-shootdowns during page

migration on the multi-threaded benchmarks.

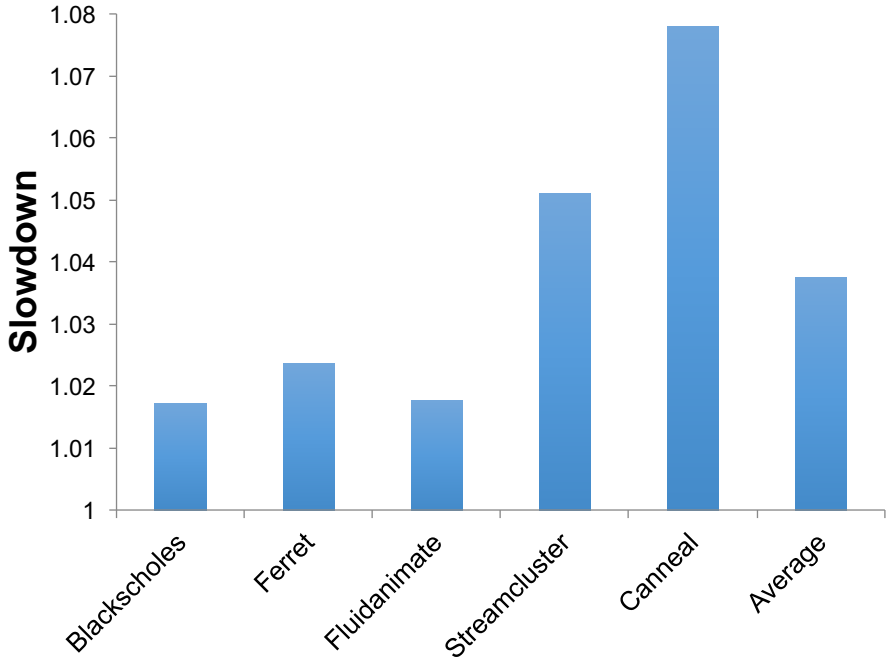

Figure 13: Slowdown relative to baseline without testing on selected PARSEC benchmarks

Energy: Table 4 shows energy for the baseline, On-demand and COMeT. I found that total energy is 47% higher for On-demand than the baseline due to the extra time to run the programs. There is also some cost from more memory and core activity as shown by the 10% penalty in power with On-demand (98 vs. 108 watts). COMeT also increases total energy, but by a smaller 18%. COMeT's energy per hour (113 watts) compared to the baseline (98 watts) shows the impact of using a second core. In comparison to each other, COMeT pays more power than On-demand (113 watts vs. 108 watts) due to increased parallelism, but it has less slowdown, leading to better

	No testing	On-demand	COMeT
Total energy (kWh)	0.91	1.3	1.1
Power (watts)	98	108	113

Table 4: Energy consumption

energy consumption. The experimental machine is an early generation (Pentium 4) CMP, which is a challenging energy target. Current CMPs have less power dissipation per core, and thus, using an idle core for testing should have less energy overhead than these results suggest.

I conclude that ahead-of-time COMeT has feasibly low overhead. The remaining experiments focus on COMeT, and for brevity, a subset of SPEC with mixed behavior (*gromacs*, *zeusmp*, *mcf* and *calculix*) and a few other interesting cases. For multi-threaded benchmarks, I showed experimental results for *canneal*, *streamcluster* and *ferret*.

4.2.3 Configuration

COMeT has three important parameters: vulnerability window duration, guarantee timer (δ), and replenishment timer ($replenish$-δ). I varied these parameters to observe their influence. The other parameters were fixed according to Table 1. I do not report coverage because it does not change from Figure 11.

Vulnerability window: The average "effective page hold time" is influential. This time reflects how long a program needs physical pages, including situations where a page is remapped frequently due to a high page fault rate and memory utilization. When a program has many active pages for a certain vulnerability window size, then it will have the same or more active pages at a smaller size, leading to more migrations. Thus, there is a greater likelihood that a program will need a page that is under migration, causing it to be briefly paused. For example, *calculix* is run with an input that causes many pages to be regularly allocated and released from the heap. It holds these pages for around 2 minutes, which causes its slowdown to increase from 1.03 (3 mins.) to 1.1 (2 mins.). *zeusmp* holds 100% of its pages for at least 3 minutes and varies from 1.04 (3 mins.) to 1.08 slowdown (2 mins.). *cactusADM* and *gemsFDTD* have higher page fault rates and utilization, which cause a slowdown of 1.11 at 2 minutes. On average for all SPEC CPU2006 benchmarks, 2 minutes has the worst performance and 4 minutes has little gain over 3 minutes. Thus, I select a 3 minute window.

Guarantee timer (δ): For a 3 minute vulnerability window, I tried 5, 10, 20 and 30 seconds for

δ. There was only a small overhead change as δ was varied (not graphed). For example, *zeusmp*'s slowdown was 1.04 at 5, 10 and 20 seconds. The overhead increased to 1.06 at 30 seconds. While the relative performance difference is small, varying δ does represent a trade-off. A small δ means that a page p will be migrated closer to its actual $t_{expire}(p)$ time. This can avoid migrations due to an unnecessarily early test expiration (for too large δ). However, a small δ causes more timer events, introducing extra overhead from the expiration check in GUARANTEE-HANDLER. δ also has to be large enough that expired pages can be migrated within the allotted δ time to invoke the guarantee handler. In SPEC, a 10 second δ best balanced the test expiration check's cost and early migration.

Replenishment timer ($replenish$-δ): The replenishment handlers (e.g., GLOBAL-HANDLER) are invoked periodically based on $replenish$-δ. Like δ, this parameter has a trade-off. A small value allows COMeT to react swiftly and it causes a *single* handler invocation to more quickly acquire a smaller number of pages than a large value. However, a small value may momentarily overreact to a spike in demand. It also leads to more handler invocations. Thus, there can be more *total* competition for memory resources.

To investigate this trade-off, I tried 15*ms*, 25*ms*, 35*ms*, and 45*ms* for $replenish$-δ. All programs had little difference for 25*ms*, 35*ms* and 45*ms*, but a larger change was observed for 15*ms*. Figure 14 displays slowdown at 15*ms* and 25*ms*. At 25*ms*, slowdown is 1.03 to 1.05, with a 1.04 average. Slowdown is higher for 15*ms*: 1.05 to 1.11 with a 1.06 average. Programs with high memory utilization or page turn-over (e.g., *calculix*, *gemsFDTD* and *bzip2*) are the most harmed at 15*ms*. It is better to pull more pages less frequently (at 25*ms*), but this effect diminishes as utilization decreases. *gromacs* and *zeusmp* have moderate utilization, with many long-held pages, and suffer less penalty from 15*ms*. In these programs, less replenishment is traded for more migration. Even in the lowest utilization cases (e.g., *povray*), there is still some penalty for a 15*ms* interval from extra handler invocations. From these results, I find that a 25*ms* interval is the best choice since it lets COMeT respond more quickly to memory demand than 35*ms* and 45*ms* with less slowdown than 15*ms*.

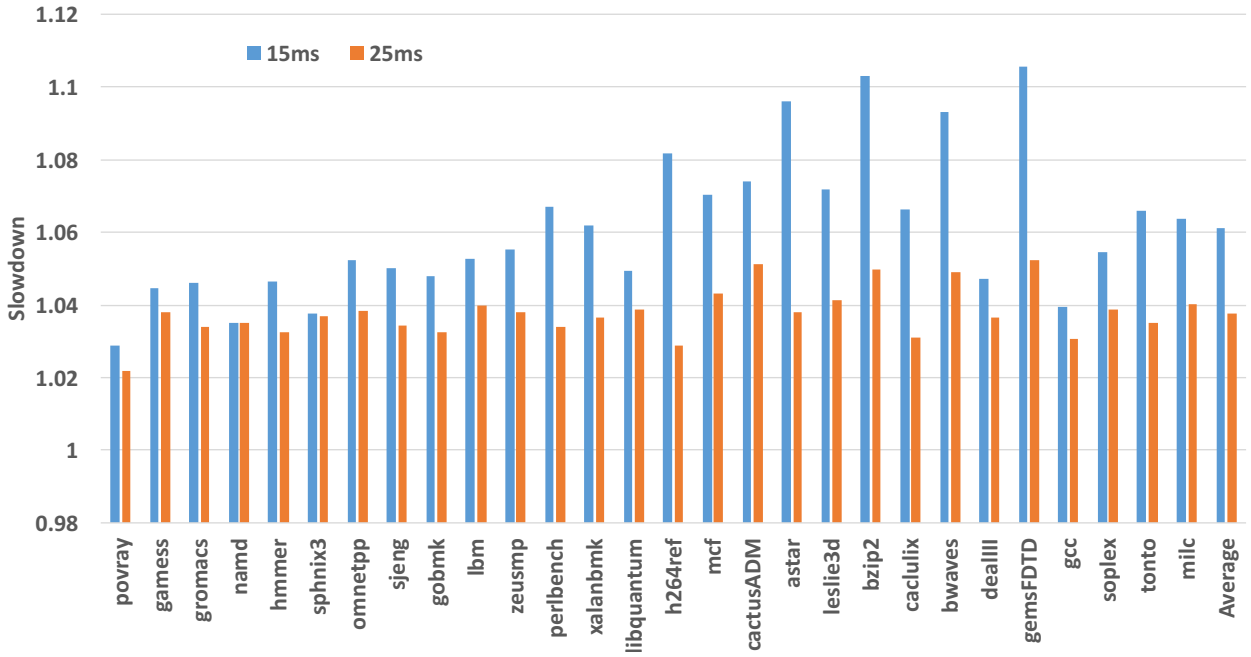

Figure 14: Effect of *replenish-δ*

4.2.4 Test Guarantee and Replenishment

Page migration and replenishment are key aspects to COMeT as its ability to limit exposure to errors partly depends on migration to test long-held pages and its overhead depends on adapting the replenishment rate.

Page migration: Figure 15 shows page coverage without and with page migration. Some programs have a large reduction when page migration is disabled (e.g., *zeusmp*'s coverage is 71.6% to 34.7%). These programs hold many pages beyond the vulnerability window, and thus, migration is especially valuable. *calculix*, *mcf* and *milc* have different behavior – migration has a smaller impact due to page turn-over (which leads to a small number of long-held pages). *gemsFDTD* has high memory utilization with a small number (7.3%) of long-held pages, and thus, its coverage is moderately reduced (96.1% to 80.1%). The average coverage reduces from 51.4% to 41.5%. From these results, I conclude that page migration is necessary to limit error exposure.

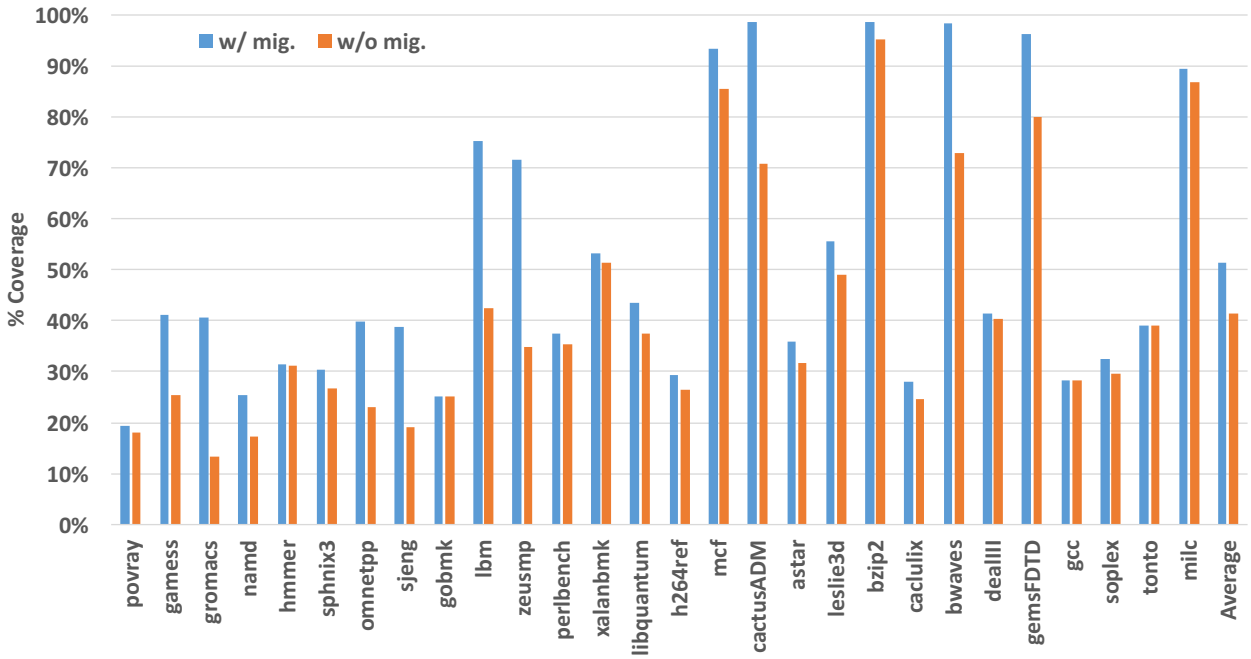

Figure 15: Page coverage due to migration

Replenishment rate: Figure 16 illustrates the benefit from an adaptive replenishment rate. The figure compares slowdown for four rate policies: "Min" statically fixes each free list's rate to the minimum average request rate among all applications, "Max" fixes the rates to the maximum averages among all applications, "Application" fixes the rates to the average rates for a given program, and "Adaptive" is COMeT's scheme. "Min" does well for low demand programs; e.g., *povray* (1.04 slowdown). However, because there is a drought of tested free pages, most programs (e.g., *cactusADM*) suffer badly. Pages are frequently tested on demand, causing "Min" to behave similar to On-demand. "Max" corrects this problem; the rates are set high enough that the free lists are replenished. Nevertheless, "Max" does worse than "Application" due to pressure from unnecessary testing. "Application" is the best because the replenishment rates are set specifically for each program. "Adaptive" comes close to "Application", but it does not need offline profiling. The SPEC average slowdown is 1.03 for "Application" and 1.04 for "Adaptive". The adaptive

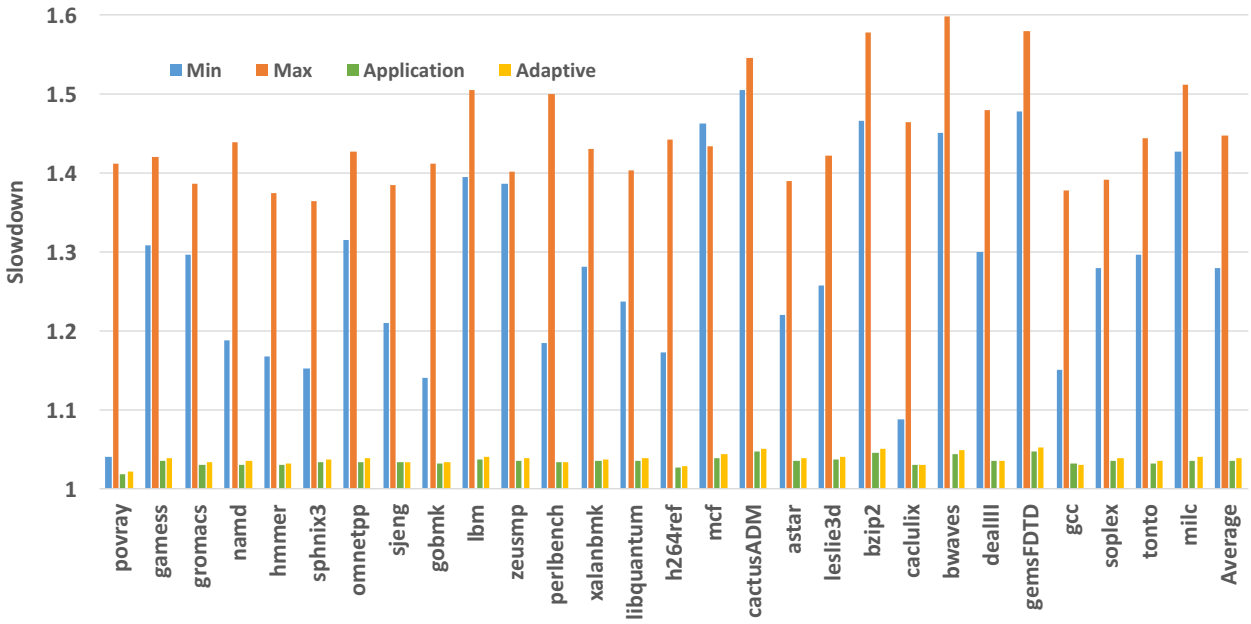

Figure 16: Effect of replenishment rates

scheme can dynamically change to global demand, walking the line between too little and too much testing. For this reason, I find that an adaptive replenishment rate should be used.

4.2.5 Overload Behavior

To examine COMeT in an overload situation for CPU cycles and memory bandwidth, I measured performance as system load is increased by executing multiple program instances. I chose *mcf*, *libquantum*, *calculix*, *lbm* and *zeusmp* due to their adequate memory pressure to create stress on the system without exhausting available memory in one instance.

Table 5 shows slowdown for multiple instances; slowdown is relative to the number of instances without testing. The instances can execute on both processor cores. In some cases, my experimental machine runs out of memory and the slowdown cannot be reported. This condition is labeled "OOM" in the table. Among the benchmarks, *calculix* has a relatively low memory uti-

Benchmarks	Number of Instances				
	1	2	3	4	5
calculix	1.03	1.03	1.03	1.05	1.07
libquantum	1.03	1.07	1.08	1.14	OOM
mcf	1.02	1.08	OOM	OOM	OOM
lbm	1.04	1.07	1.09	OOM	OOM
zeusmp	1.03	1.05	1.10	OOM	OOM

Table 5: Slowdown or Out-of-Memory (OOM) under overload

lization: the slowdown is 1.03 (one instance) to 1.07 (five instances). As more instances are added, allocation rate and memory utilization increase, causing competition for cores and memory bandwidth. *libquantum*, *lbm* and *zeusmp* are memory intensive, causing more overhead. For example, *libquantum* had an overhead of 1.14 at four instances. *mcf* had the highest memory utilization, and thus, the largest overhead at the fewest instances (1.08 for 2 instances). In general, the slowdown increases as there is more pressure on the memory allocator and the memory hardware. Even in the most stressful situations in Table 5, however, the overhead is acceptable, demonstrating that COMeT performs well at overload.

4.2.6 Sensitivity to Test Latency

To examine application sensitivity to test latency, *mcf*, *zeusmp*, *gromacs* and *calculix* were executed with an increasing amount of MARCH testing. Test intensity was varied by applying more patterns in additional test iterations during a page test.

Figure 17 shows that COMeT sufficiently masks test latency up to 3 iterations due to ahead of time testing, with an average 10% slowdown. Beyond 3 iterations, application performance can suffer. *mcf* and *zeusmp* had a large 25% degradation at 5 iterations because COMeT could not maintain a fast enough replenishment rate to keep the pool of tested pages full, which caused increased on-demand testing that harmed performance. In comparison, *gromacs* and *calculix* have

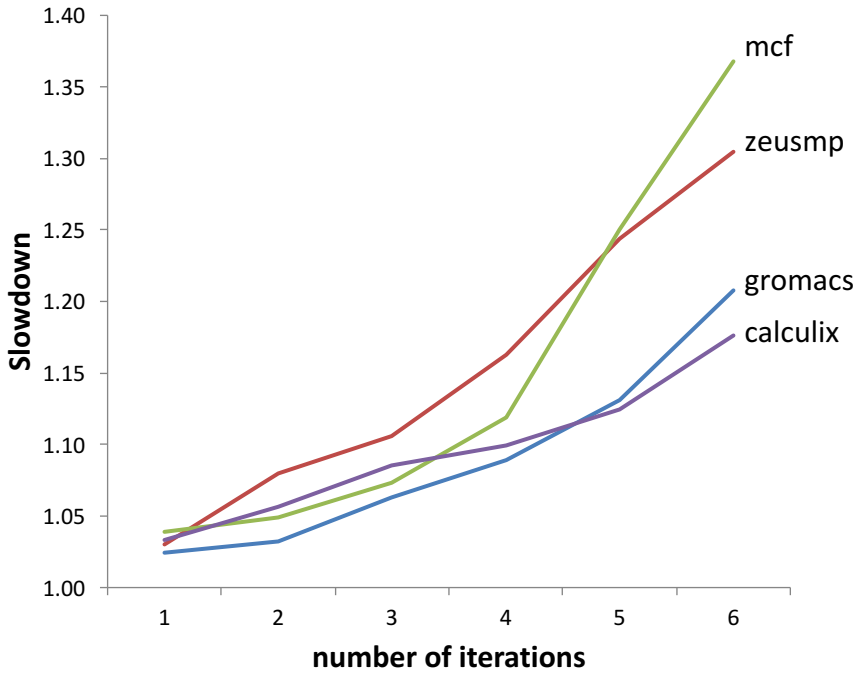

Figure 17: Sensitivity to test latency

low memory utilization, and had only a 10% degradation at 5 iterations. As this experiment shows, there is a trade-off between memory demand and test aggressiveness. Complex test patterns for neighborhood pattern sensitive faults may require additional test time, and consequently, they may not be usable in high demand applications. These applications will likely require fewer and simpler test patterns (i.e., less error coverage) to minimize overhead. Indeed, the patterns and test rate could be dynamically adapted to balance aggressiveness and overhead.

4.2.7 Multi-threaded Workload

For multi-threaded benchmarks, I used PARSEC on the second experimental setup with 2.1GHz Intel i7 processor and 6GB of RAM. The processor has four physical cores (eight hardware contexts). The selected benchmarks were executed with 1, 2, 4 and 8 threads. For each thread count, a benchmark was run 10 times with native inputs and the average runtime was calculated relative

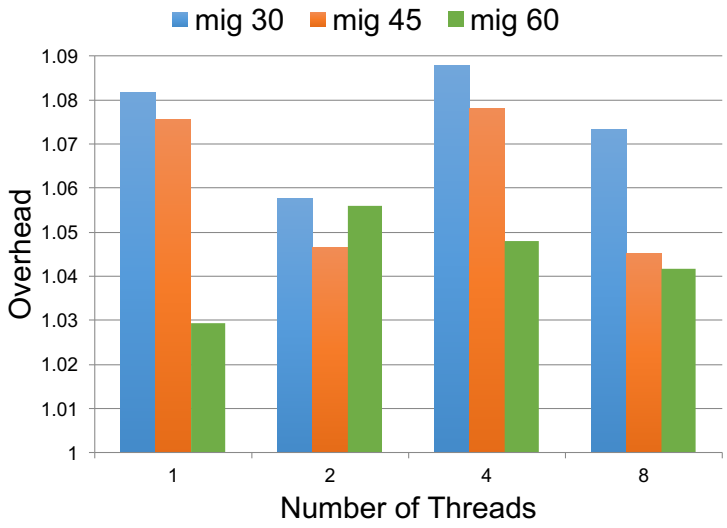

Figure 18: Slowdown on Multi-threaded Benchmarks: *Canneal*

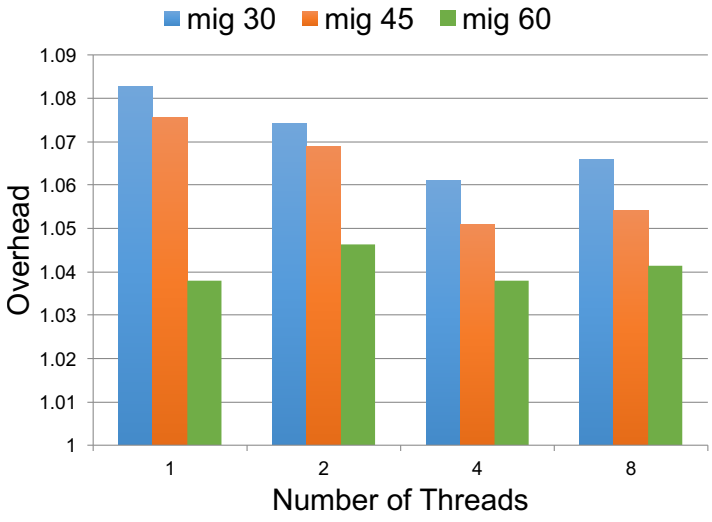

Figure 19: Slowdown on Multi-threaded Benchmarks: *Streamcluster*

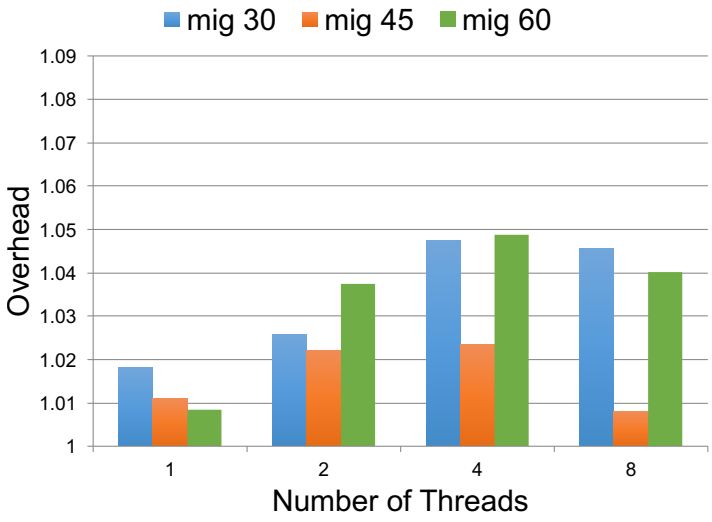

Figure 20: Slowdown on Multi-threaded Benchmarks: *Ferret*

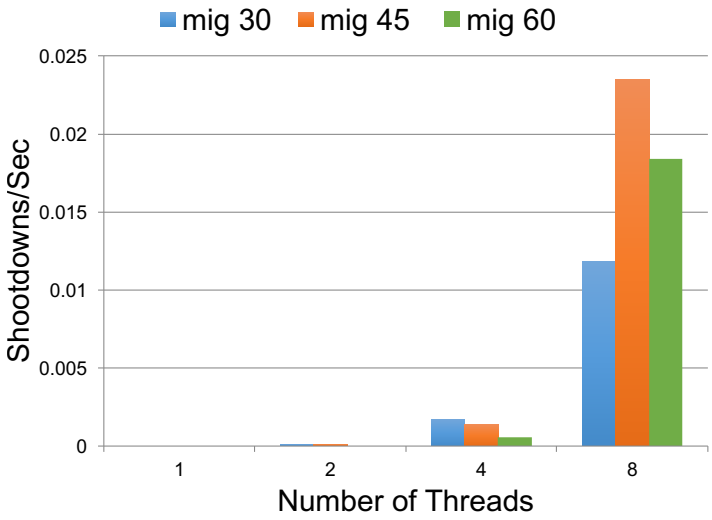

Figure 21: Rate of TLB shootdowns on Multi-threaded Benchmarks: *Canneal*

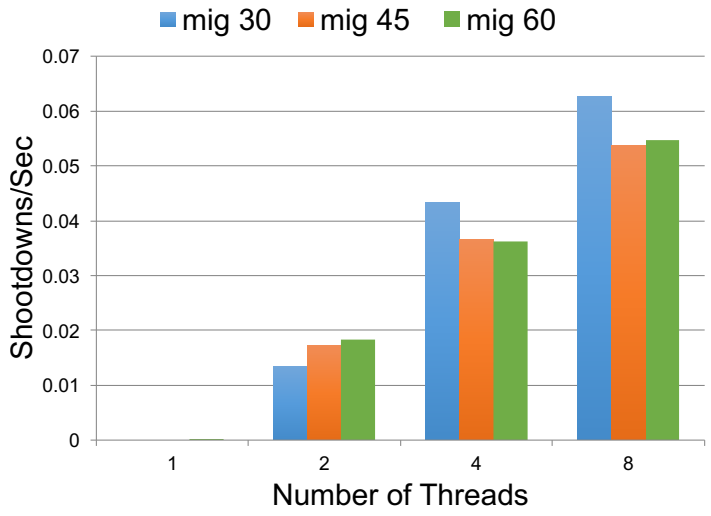

Figure 22: Rate of TLB shootdowns on Multi-threaded Benchmarks: *Streamcluster*

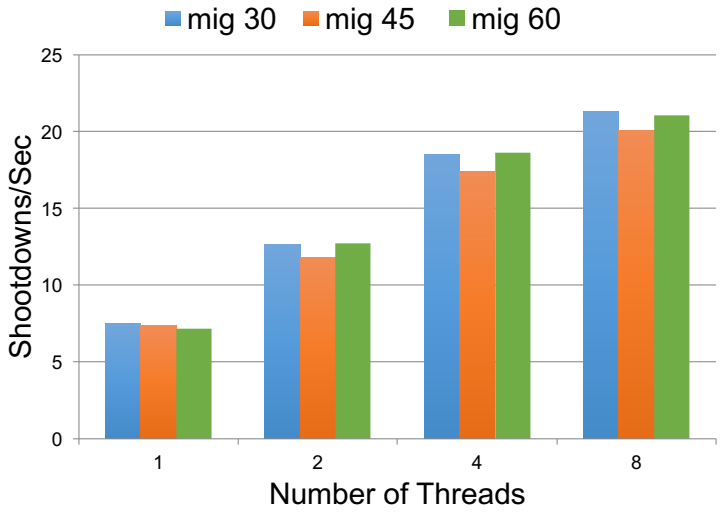

Figure 23: Rate of TLB shootdowns on Multi-threaded Benchmarks: *Ferret*

to the baseline with the corresponding number of threads without COMeT.

Figure 18, Figure 19 and Figure 20 show COMeT's overhead on *Canneal*, *Streamcluster* and *Ferret* respectively. From these figures, *Canneal* has the highest slowdown of 8.7% with four threads and a 30 second migration interval. *Canneal* has the largest native working set size in PARSEC and high data sharing [11]. As the migration interval is gradually increased to 45 seconds and 60 seconds, the overhead drops as *Canneal* releases pages before the expensive migration enforced by COMeT. Figure 21 shows the rate of TLB shootdowns for *Canneal*. I found that the rate of TLB shootdowns increases for 8 threads but its performance impact is lower than the impact on 4 threads. I also observe that the rate TLB shootdowns is quite low (e.g., less than 0.025 per second with 8 threads). Thus, the performance overhead comes primarily from *Canneal*'s high memory demand, putting more pressure on COMeT.

Figure 19 shows that the overhead for *Streamcluster* is slightly reduced due lower memory pressure than *Canneal*. Figure 22 shows that the rate of TLB shootdowns for *Streamcluster* is slightly higher than *Canneal*. However, the increased rate is not high enough to impact slowdown for *Streamcluster*. Figure 20 indicates that overhead for *Ferret* is much lower than *Canneal* and *Streamcluster*. However, from Figure 23, *Ferret*'s TLB shootdown rate is much higher than the other two benchmarks. The high rate of TLB shootdowns adversely impacts *Ferret*'s performance. *Blackscholes* and *Fluidanimate* (not shown) have very low TLB shootdown rates and the performance overhead for these programs is due to memory pressure. In summary, TLB shootdowns will cause the overhead from COMeT to increase. Nevertheless, overhead remains relatively low even for multi-threaded programs.

In summary, COMeT can monitor memory health in a deployed system that is actively executing user applications. COMeT provides an excellent effectiveness in detecting errors "ahead-of-time" with low performance impact on single-threaded and multi-threaded applications in small-scale systems. COMeT is easily deployable and highly configurable. No change in application code or binary is required to run on COMeT-enabled systems. In addition, using StealthWorks, COMeT provides a neat way for system administrators to stress test their configuration for COMeT-enabled systems before deployment.

5.0 ONLINE MEMORY DIAGNOSTIC IN LARGE-SCALE SYSTEMS

COMeT uses a single *test thread* to do diagnostics. The speed at which this thread does writes, reads and comparisons determines the *test latency* of a page. With more bit patterns, the latency increases, reducing the maximum achievable test rate. This rate must be higher than the workload's aggregate memory request rate to avoid on-demand testing. As the number of applications increase with more cores, there is more memory demand, which can overwhelm the test thread.

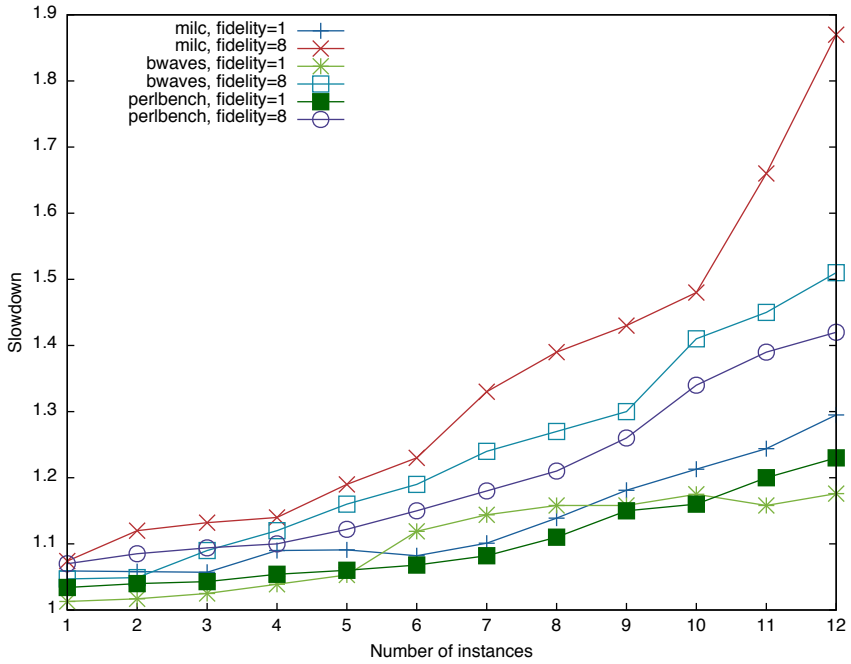

Figure 24: COMeT slowdown on selected multi-instance SPEC CPU2006 workloads

To examine this behavior, I implemented COMeT for an Intel dual-processor server with 16 cores on SPEC CPU2006 (see Section 5.3.1 for the full details of my experimental setup). I evalu-

ated how some SPEC benchmarks behave when run in multiple instance workloads to understand how increasing demand can affect overhead. Figure 24 shows slowdown for three benchmarks in multiple instance workloads. I showed these benchmarks only because they have moderate demand—i.e., they are "typical" of the most SPEC benchmarks. The figure graphs slowdown of testing with COMeT as the number of co-running instances of each benchmark is increased. Slowdown is workload runtime with testing divided by workload runtime without testing. The figure also shows how the quantity of test bit patterns, which is call "fidelity", affects slowdown. Two fidelities are shown: 1 and 8.

On average, for all of SPEC CPU2006, COMeT has only 1.04 slowdown (range 1.01 to 1.07, not shown). In multiple instance workloads, the slowdown remains reasonable until around 5 instances with fidelity=1. The figure illustrates this observation for *milc*, *bwaves*, and *perlbench*. In these cases, COMeT maintains a sufficient test rate to avoid on-demand testing of memory until 5 instances.

Beyond 5 instances, the slowdown is high: Aggregate memory pressure is too much for testing to keep pace, causing on-demand testing. When fidelity=8, the point where slowdown worsens shifts to fewer instances because more testing is required per physical page, incurring higher test latency and lower maximum achievable test rate. For example, *perlbench* has 1.1 slowdown at 8 instances with fidelity=1. The same slowdown happens at 4 instances with fidelity=8. *milc* with 12 instances and fidelity=8 has the worst slowdown of the examples at 1.84. A change in fidelity does not necessarily cause slowdown to linearly change: It depends on the rate of allocation relative to number of instructions executed. Resource competition also contributes to the slowdown, but it is minimal from my measurements.

In addition to increased memory pressure, Non-Uniform Memory Access (NUMA) architecture has been increasingly used by new multicore systems due to scalability and decentralized access control over memory among cores. For example, Figure 25 shows memory access latencies from processing cores to memory in node 0 of an AMD Opteron 48-core machine with 96GB of memory. As this figure shows, memory access latency varies significantly depending on the location of cores and the node hosting the memory due to the interconnect bandwidth. In large

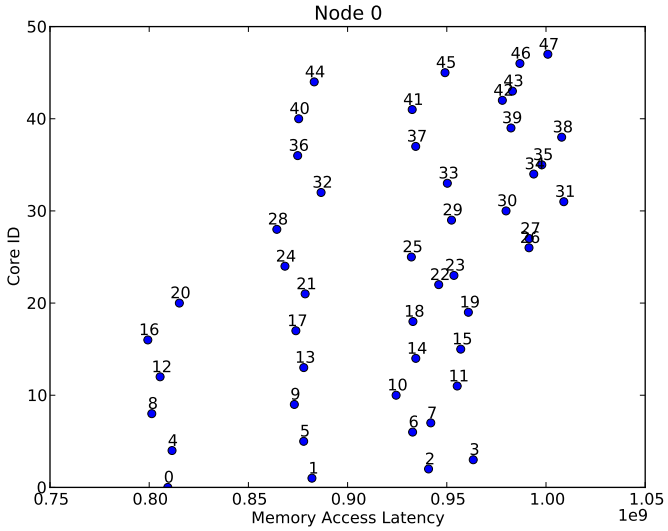

Figure 25: Memory Access Latency in NUMA

systems, NUMA is expected to have an impact on the experimental results of COMeT. A tester thread on a core in a node will observe varying latencies during testing a page depending on the location of that page in NUMA. Remote pages will require more time to test. NUMA latency is very important to address as well while designing a scalable diagnostic.

From these results, I conclude that COMeT is sufficient for a small number of co-running applications at low fidelity. However, it cannot keep pace at larger core counts, executing many applications. Further, COMeT applies only a few patterns, which may not provide enough error coverage. I address these limitations of COMeT by introducing a scalable online memory diagnostic which I describe in the following sections.

5.1 ASTEROID

To address scalability issues explained in the previous section, I designed **A**daptive and **S**calable **Te**chnique for **R**esiliency with **O**nline **D**iagnostics (**Asteroid**). Asteroid allows adapting memory testing based on workload behavior to minimize performance overhead. It is designed to scale to large core counts and aggressive memory tests (i.e., that apply many bit patterns). Asteroid allows the system administrator to specify a diagnostic control policy that decides how to adjust memory tests to trade thoroughness/coverage (*fidelity*), resource usage, and performance overhead. The administrator can also specify the specific tests to use.

Asteroid has three new capabilities. First, it allows multiple memory pages to be tested simultaneously by independent test threads to use multiple idle cores. Second, memory tests done on physical pages can be adapted to apply fewer or more test bit patterns (changing the amount of error coverage), depending on memory pressure. When pressure is high, such as during application initialization, less aggressive tests can be applied to avoid competition with the workload. Likewise, during low demand periods, more aggressive tests can be used to more thoroughly check memory health through increased error coverage with more bit patterns and sweeps. Finally, integration of memory testing and the OS memory allocator are optimized to minimize test latency.

Asteroid supports high memory demands and high fidelity settings for multi-core systems. The key insight is tailoring the diagnostic test to workload demand: the effective test rate can be adjusted by changing the number of test threads and the fidelity. A *diagnostic control policy* determines these parameters, which can be set prior to execution or adjusted online. An online approach permits trading test rate, fidelity and resource competition with runtime changes in memory pressure. Similar to COMeT, Asteroid guarantees that all allocated pages are tested within the vulnerability window. It also migrates pages that are held a long time (beyond the vulnerability window time interval) to test them. Unlike COMeT, Asteroid guarantees minimum fidelity and maximum resource usage. It is also designed to be easily extended by an administrator to implement system-specific diagnostic control policies and memory test algorithms.

5.2 COMPONENTS OF ASTEROID

Asteroid has a significantly more complex architecture than COMeT, which does not support changing the test policy. COMeT uses a simplified version of my diagnostic framework in Figure 3 by using only a single test thread. Consequently, COMeT is more tightly integrated with the kernel and relies heavily on timers/event handling for testing. Instead, Asteroid follows my diagnostic framework in Figure 3 more closely. Asteroid introduces three new components (TC, TD and CTT), a well-defined interface between the components, and extensibility to add new test policies and even new memory test strategies (i.e., MARCH tests).

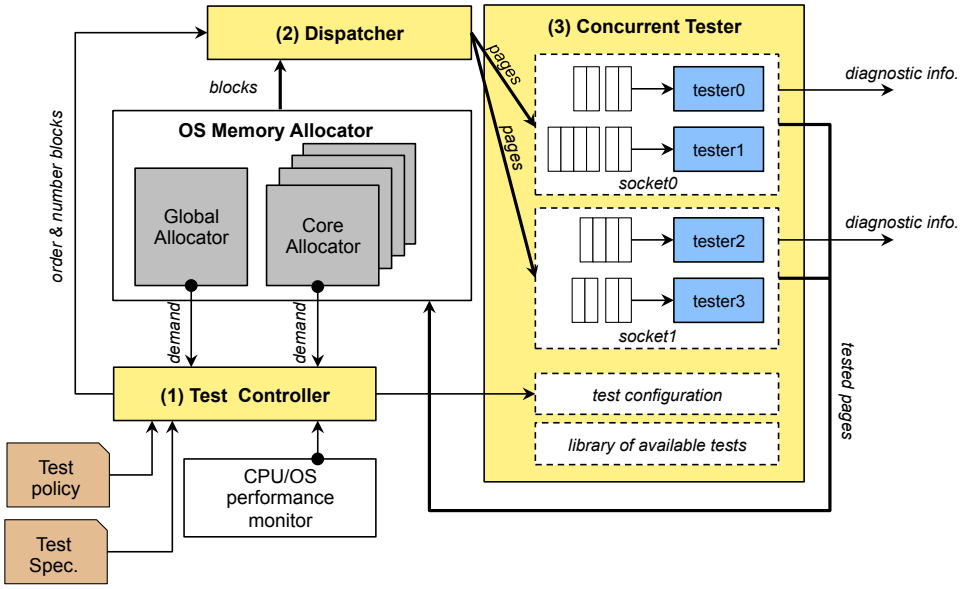

Figure 26: Asteroid memory diagnostic framework

Figure 26 shows Asteroid's design in detail. It is integrated in the OS kernel and works collaboratively with memory allocation. There are three components: 1) Test Controller (TC), 2) Test Dispatcher (TD), and 3) Concurrent Tester Threads (CTT). TC decides what memory to test and how to test it. It applies the diagnostic control policy. TD extracts physical pages frames from the memory allocator and distributes them to software threads that do memory tests, CTT. CTT is a collection of kernel threads that test memory. Each of these components is described in detail in

the next sections.

5.2.1 Test Controller (TC)

TC determines *how* memory should be tested and sets diagnostic parameters, or a *configuration*, accordingly. For example, when the system is relatively idle, TC could configure a DRAM diagnostic to perform aggressive MARCH tests for maximum error coverage. A *diagnostic control policy* is incorporated in the Test Controller to set the configuration, *testconfig*; the policy is developed by the administrator for a target system, or one of my existing policies can be used. The control policy is invoked to determine *testconfig* for an upcoming interval of time, the *test epoch*. The test configuration is a tuple $[fidelity, threads]$, where *fidelity* is a parameter for the test threads and *threads* is how many test threads to use. The meaning of *fidelity* is specific to the actual diagnostic implemented in Asteroid. For example, in a MARCH test, *fidelity* specifies a list of bit patterns to use for the test.

A control policy follows a sense-decide-configure model. TC periodically receives a timer event, which triggers the control policy to evaluate current conditions to derive *testconfig* for the next epoch. Information about memory pressure and CPU events from hardware counters (e.g., cache misses) may be used to derive the configuration. The policy obeys constraints on minimum test fidelity, MIN-FIDELITY, and maximum core count for testing, MAX-CORES, in setting the configuration.

The control policy is responsible for only configuring the diagnostic (*testconfig*); it decides *how* to test, rather than *what* and *how much* to test. This structure factors low-level details about the operating system's memory allocator and physical memory away from the configuration decisions. This factoring is important as details about the memory allocator can be complex and subtle with information scattered throughout the kernel, particularly in an OS that uses hierarchical allocation. Instead, TC provides the functionality to interact with the memory allocator separately from the control policy. The operational details of the memory allocator and page allocation do not need to be considered in the policy itself, simplifying its structure.

To determine the specific pages to test, i.e., the free memory blocks, TC interacts with the

memory allocator. This functionality does not depend on the control policy; an administrator implementing a control policy does not have to consider how to extract blocks for testing. TC automatically extracts pages separately based on monitored demand.

The Test Controller determines a memory block list, *memblocks*, which indicates (1) *what* memory to test (from what buddy allocator ranks) and (2) *how many* blocks to test (from each buddy rank). *Memblocks* is a list of tuples $[rank_i, number_i]$ that indicates the quantity of blocks to test for each rank i of the kernel's buddy memory allocator. The tuple list is a specification of how much and what to test, rather than the specific physical addresses (pages) to be tested.

TC is the key component in Asteroid since it configures the diagnostic and determines what memory to test. Section 5.2.4 describes TC in detail, including software architecture, operation and diagnostic control policy.

5.2.2 Test Dispatcher (TD)

TD receives *memblocks* and *testconfig* from TC; *memblocks* specifies the quantity and orders for extracting actual blocks (i.e., contiguous physical page frames). After extracting blocks from the allocator, TD distributes them to test threads (CTT), which are configured using *testconfig*.

TD is node-aware for multi-processor systems: It dispatches a page to a test thread on that page's home node, avoiding remote memory access latency. Memory blocks associated with a particular memory controller are dispatched to test threads running on that node. TD also balances work distribution by monitoring occupancy of the input work queue of each test thread. Occupancy is measured by number of *pages* in the input queue of a test thread. Work is dispatched in blocks; blocks are not split since they are contiguous memory pages that will be returned to the buddy allocator, assuming they pass the diagnostic.

5.2.3 Concurrent Tester Threads (CTT)

CTT is a pool of test threads, which can be configured to change thread count and fidelity (i.e., *testconfig*). This capability allows CTT to adjust system resources (threads) and memory intensity

(bandwidth) with workload behavior. Figure 27 illustrates how CTT receives blocks of pages to test. The dashed line in the figure shows the flow of extracting untested blocks, testing them, and returning them to the allocator. This "test path" uses TD to extract blocks (according to requirements from TC), which are passed to CTT for testing.

To improve test efficiency, CTT does four "optimizations": (1) core-local on-demand test, (2) page cache recycling, (3) skip middle-ranked blocks, and (4) cache indirect testing. The optimizations are described next.

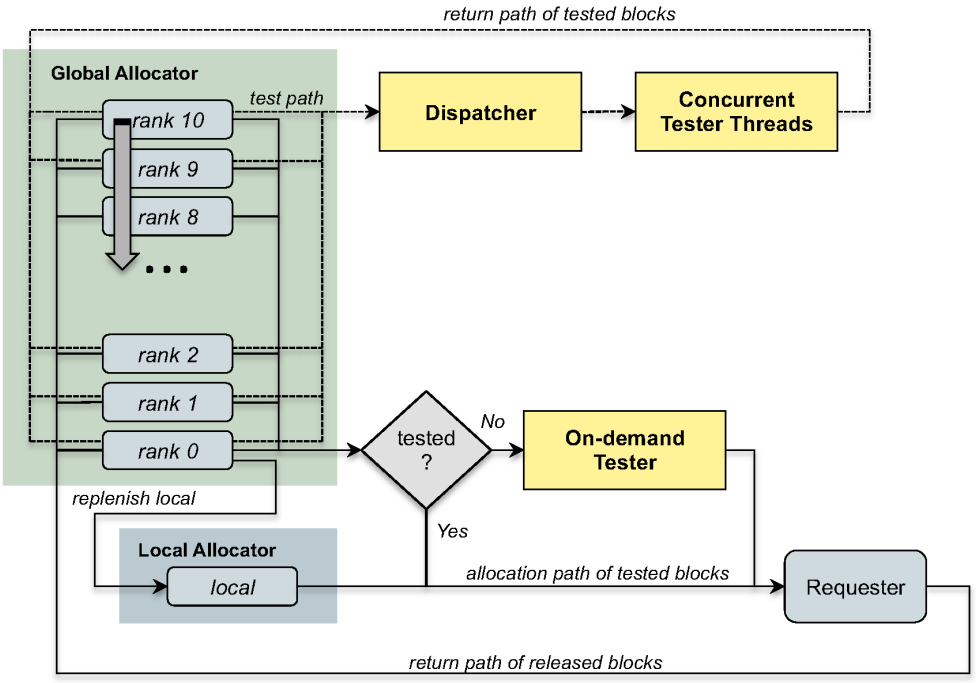

Figure 27: Paths for tested pages (dashed line) and requested pages (solid line)

5.2.3.1 Core-local on-demand test Because modern OSes use hierarchical local and global allocation, CTT is also hierarchical. Local allocation acts as a fast path to satisfy allocation requests; a fixed-length list of singleton blocks are kept per core (the per-core page cache) from which requests can be handled. Local allocation can also have cache benefits from returning pages back through the local allocator.

In Asteroid, pages that need to be tested on-demand are tested locally on the core where the pages were allocated. In Figure 27, the "On-demand Tester" is a kernel thread that runs locally on the core suffering an on-demand request. Multiple on-demand testers can be active simultaneously, if there are several local on-demand requests. This structure lets on-demand testing happen concurrently to execution of other programs and test threads. Thus, only the application local to a core suffering on-demand testing is impacted.

5.2.3.2 Page cache recycling A tested page is marked expired when it is returned to the global allocator from a local allocator. Figure 27 shows this return path. The Requester (e.g., application) can request a singleton block (one page), which is satisfied locally by the Local Allocator. When a singleton block is released by the Requester, it is returned to rank-0 of the global buddy allocator, rather than the per-core page cache. The returned page is expired during return and put to the end of the list of blocks inside the global allocator. Future allocations from the global allocator to the local allocator consist of freshly tested pages. Hence, applications get the most recently tested pages, which are less likely to require migration to ensure the vulnerability window.

The figure shows how the global allocator provides singleton blocks to the local allocator. These pages are delivered from the rank-0 freelist, which is heavily tested (receiving many requests). Although the figure does not show on-demand testing on this path (to simplify the drawing), my implementation ensures that the local allocator receives only tested pages, including the possibility of on-demand testing. Fortunately, on-demand testing from rank-0 to the local allocator is usually off the critical path: The local allocator can satisfy requests with existing free tested pages that it already holds while new pages are concurrently being tested on-demand. In effect, on-demand testing from rank-0 to the local allocator happens in the background to application execution. There can be moments, however, where the local freelist is empty *and* the rank-0 freelist has no tested pages. In this case, a request to the local allocator will be delayed by on-demand testing from rank-0. This situation is rare in practice, particularly as the test rate is adjusted with demand.

5.2.3.3 Skip middle-ranked blocks I observed that low buddy ranks (order 2 or less) sustain most memory allocation during normal system operation. Because the buddy algorithm actively tries to reduce memory fragmentation, the highest-ranked blocks (order 9 and up) also sustain allocation activity to provide memory to lower ranks. Thus, only blocks from lists of order 2 (most heavily accessed) or less and lists of order 9 and up (source of blocks for middle ranks) are tested. All other ranks are skipped. The test time of skipped ranks thereby saved is used to test the more heavily used ranks. Figure 27 shows the test path extracts blocks from rank 10, 9, 2, 1 and 0. The CTT can return tested blocks back to these ranks.

Because blocks may be split from a higher rank, the middle ranks tend to receive recently tested blocks. The figure depicts splitting blocks from the ranks 10 and 9 to fill lower ranks. In this way, tested blocks tend to propagate downward through middle ranks. Consequently, the delivery of an untested block from a middle rank is somewhat unusual as long as the test rate keeps up with demand. There are two reasons for this. First, the vast majority of blocks requested come from the lowest rank, which are actively tested. Second, when a request is made to a middle rank, it will likely be satisfied by a block tested ahead of time due to splitting from a higher rank. If a tested block is not available, then one will be tested on-demand.

5.2.3.4 Cache indirect test Test threads issue memory reads and writes that can either be done directly on memory or indirectly through the caches. CTT uses the latter method for efficiency: Test latency is reduced by utilizing burst-mode cache writes, which is called a "cache indirect test". During testing of a memory page, each cache line is written and flushed to DRAM one at a time. After the whole page is written, the page is read again to check that the patterns were written correctly to DRAM. Burst writes dramatically reduce test latency compared to writing one word at a time bypassing the caches [57]. Test threads also implement strided writes and reads to flush DRAM row buffers before reading them back.

Figure 28 shows a skeletal algorithm for a simple MARCH test that uses cache indirect testing. This test writes, reads and verifies one bit pattern for every cache line on a physical page. For each line on a page (lines 3-5), the line is first flushed from the cache and DRAM (flush from the cache

```
MARCH-PAGE-DIAGNOSTIC(page)
 1  line_addr = page
 2  for (line_num = 0; line_num < PAGE-SIZE; line_num++)
 3    // Initialize cache line to clean (flushed) state
 4    CACHE-FLUSH(line_addr)
 5    DRAM-CLOSE-PAGE(line_addr)
 6    // Write pattern to cache line, flush, close DRAM row
 7    WRITE-PATTERN(line_addr, PATTERN)
 8    CACHE-FLUSH(line_addr)
 9    DRAM-CLOSE-PAGE(line_addr)
10    // Read pattern and check match on pattern
11    line_data = READ-PATTERN(line_addr)
12    if (CHECK-PATTERN(line_data, PATTERN))
13      // Did not match: Report as failed diagnostic.
14      return FALSE
15    line_addr = line_addr + LINE-SIZE
16  return TRUE
```

Figure 28: Example MARCH test using cache indirect testing with one pattern

and close the DRAM row), if present in the cache or DRAM row buffer. Next, the test bit pattern is written to all words in the cache line and flushed to memory (lines 6-9). Finally, the line is read from memory and the words in the line are checked against the pattern (lines 10-14). If the read data does not match the written pattern, then the page is reported as failing the diagnostic. In memory direct testing, the cache and DRAM row buffer would need to be flushed for every word in the page, which is more expensive than performing the same operations at cache line granularity.

5.2.4 Test Controller and Diagnostic Control Policy

In this section, I describe the general operation, constraints and options for the test controller, with an emphasis on how TC can accommodate different diagnostic control policies. I describe two diagnostic control policies but others could be implemented as well.

5.2.4.1 Operation Figure 29 shows the architecture of the test controller. TC has two primary functions: (1) determine which freelists (i.e., the global buddy allocator ranks and the local allocator page cache list) require tested pages and how many pages to be tested and (2) determine the configuration of the test threads for the next test epoch. These functions are described next.

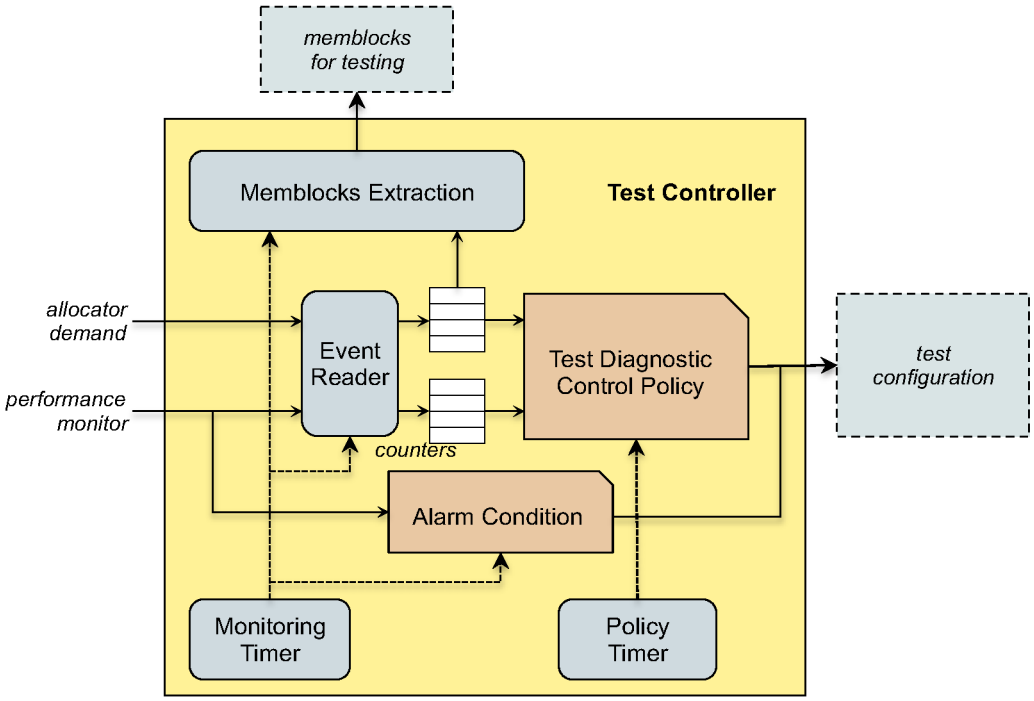

Figure 29: Software architecture and information flow for TC

5.2.4.2 Determining Memory Blocks to Test As the figure shows, TC has a component Memblocks Extraction that determines which blocks and how many to test for the upcoming test epoch. This component generates the memory block list, *memblocks*, which is passed to Test Dispatch to extract the actual physical page frames from the freelists for testing. *Memblocks* is a list of tuples, $[rank_i, number_i]$, that indicate the ranks (freelists) which need replenishment with tested pages and how many blocks to test for each rank.

Periodically, the Monitoring Timer expires, causing the Event Reader to query the memory allocator about which freelists have provided pages (due to allocator requests) since the last query.

The Event Reader also samples any available performance monitoring information, such as hardware counters (e.g., cache misses). All of the sampled information is recorded for use by Memblocks Extraction, the Diagnostic Control Policy and the Alarm Condition.

Using the recorded counter values for memory demand, Memblocks Extraction determines which ranks to test and how many blocks to test from each rank. The number of blocks to test for rank i is the difference between the current counter value and the previous one scaled by a small factor. The current counter values reflect the most recent demand, allowing Memblocks Extraction to react to demand bursts, which will possibly deplete the freelists of tested pages for the next test epoch. In contrast, to avoid reacting to deep valleys in demand (e.g., 0 requests), a lower limit is placed on how many blocks to test. The lower limit is determined by the moving average of demand scaled by one half. The moving average is decayed gradually to smooth out sudden drops in demand. Specifically, the number of blocks tested for rank i is: $number_i = MAX(\alpha \times (counter_i - prev\_counter_i), moving\_average_i/2)$. This strategy ensures a ready supply of tested pages to avoid on-demand testing. It may over test for a period of time, but eventually the moving average decays and stabilizes to reflect the actual demand when it drops sufficiently long.

According to the skip middle rank optimization, Memblocks Extraction considers only the low and high order ranks ($i = 0, 1, 2, 9, 10$) of the global buddy allocator. The previous counter values and the moving average of demand for each rank is updated after the *memblocks* list is determined.

5.2.4.3 Determining Test Configuration A diagnostic control policy aims to configure the diagnostic to achieve objectives for the memory test and the impact on applications, according to memory demand. That is, it tries to configure the diagnostic to keep up with the blocks being scheduled for testing by Memblocks Extraction. TC hosts the control policy, as the figure shows. The control policy and an associated alarm condition (described below), are user defined. The system administrator can change these components to achieve particular system goals (e.g., minimize performance overhead or maximize test coverage).

Similar to Memblocks Extraction, the diagnostic control policy is periodically invoked (determined by a timer, Policy Timer) to configure the test configuration. Whenever invoked, the control

policy decides on the configuration for the upcoming period until invoked again.

Using the monitored information from the Event Reader, the diagnostic control policy attempts to configure the diagnostic to maintain the $test\_rate$ to avoid on-demand testing. In practice, I found that it is best to scale up the target test rate by a small factor, i.e., $test\_rate = alloc\_rate \times \alpha$, where $\alpha \geqslant 1.0$. The amount of scaling, α can be set by the administrator to control how much to overprovision testing of pages; this value also affects how many additional blocks (pages) above monitored demand are scheduled by Memblocks Extraction for testing. I found that doubling the demand works well for my experimental system [58]. Conceptually there is a single test rate, but in actuality, the memory allocator maintains several freelists, each of which has its own test rate, along with the monitored information. For simplicity, in the following discussion, I treat the test rate as a single value that affects how many pages to test every test epoch to keep the memory allocator's freelists filled with a sufficient supply of pages that have been recently tested in the background.

The control policy determines a configuration $testconfig = [fidelity, threads]$, such that $test\_rate$ satisfies:

$$test\_rate \leq threads \times \textsc{Calc-Bandwidth}(fidelity)$$

$$1 \leq threads \leq \textsc{Max-Cores}$$

$$\textsc{Min-Fidelity} \leq fidelity \leq \textsc{Max-Fidelity}$$

where CALC-BANDWIDTH is used to calculate the maximum effective bandwidth to test physical page frames at the specified fidelity. CALC-BANDWIDTH depends on the specific diagnostic tests implemented; I will give an example of this function for MARCH testing in Section 5.2.4.5. In practice, there can be memory contention which may affect the actual bandwidth to test a page. MAX-CORES is the maximum number of cores (system resources) to use for testing pages, MIN-FIDELITY is the lowest acceptable fidelity, and MAX-FIDELITY is the maximum fidelity. These parameters are set by the administrator.

A policy should select the configuration to satisfy the required test rate to keep up with the scheduled *memblocks* but also to meet system goals, such as reducing performance overhead, minimizing core usage or maximizing test thoroughness. A fixed policy can set the parameters to

constants for the *expected* maximum bandwidth demand, while an adaptive policy can change the parameters, according to demand and operating conditions.

The diagnostic policy is periodically invoked by the TC whenever the Policy Timer expires. The periodicity, the *test epoch*, is configurable. Ideally, the periodicity should be set relatively large to avoid changing test configurations too frequently. It helps "smooth out" making configuration adjustments in responding to demand bursts. However, too long of an interval may cause too much smoothing, leading to on-demand testing as the rate cannot be adjusted upward quickly during burst demand.

To address burst demand, TC has an "emergency override" capability through the Alarm Condition. Whenever the Monitoring Timer expires, the Alarm Condition is checked to determine whether monitored allocation demand is high enough to trigger increasing test bandwidth to its maximum for the burst. Note that Memblocks Extraction does not place an upper limit on the number of blocks to schedule for testing (i.e., the current test rate does not limit how many pages to actually schedule for testing). Hence, the emergency override is used to reconfigure the diagnostic to react to peak demand, causing more work (i.e., pages to test) to be suddenly scheduled by Memblocks Extraction. The Alarm Condition is checked with the same periodicity of Memblocks Extraction. Similarly to other aspects of TC, the Alarm Condition is configurable. For the emergency override, three parameters affect its operation: Monitoring Timer periodicity, the alarm triggering event (condition), and the emergency test configuration.

For my experimental system (see Section 5.3.1), I tried different settings of all parameters[1]. I found that an emergency override where demand has increased by $4\times$ since its last reading works well as a threshold (indicating a rapid rise in burst demand) [58]. I also set the emergency configuration to be the one that has the maximum test rate (allowed by the constraints). The emergency configuration remains in effect until the end of the current test epoch. I used a $25ms$ Monitoring Timer value, as suggested by previous work [58]. With these settings, a one second Policy Timer value works well for my system—the system is responsive to demand and emergency override only kicks in during unexpected demand. The test configuration also remains stable during

[1] I do not show a sensitivity study of the parameters because the parameters should be tuned to a given target system with offline profiling. This tuning is orthogonal to my contribution and relatively uninteresting.

epochs without short-lived demand spikes.

Parameter	Description (Values used in experiments)
α ($\alpha \geq 1$)	Scale test rate to over-provision for demand (2.0)
MAX-CORES	Maximum number of cores to use for testing (4)
MIN-FIDELITY	Minimum fidelity to use (1)
MAX-FIDELITY	Maximum fidelity to use (32)
Alarm Condition	Trigger emergency test configuration (4× increase demand)
$e\_testconfig$	Emergency test configuration ([MIN-FIDELITY, MAX-CORES])
Policy Timer	How often to invoke control policy (1s)
Monitoring Timer	How often to sample demand and performance counters (25ms)

Table 6: Test Controller and Diagnostic Control Policy Parameters

In summary, the administrator needs to set several parameters, as shown in Table 6, and select a diagnostic control policy (or implement his/her own policy). Having described the operation and parameters of TC, I describe two exemplar diagnostic control policies: *Fixed* and *Adaptive*. Fixed is static, while Adaptive changes the configuration at runtime. In the results, I experimentally evaluated these policies.

5.2.4.4 Fixed Policy In this simple policy, the system administrator sets a fixed configuration for the diagnostic. The configuration is not changed at runtime; it is initialized at system bootup. In particular, the administrator picks the actual configuration to use, obeying the constraints noted above. That is, $threads$ and $fidelity$ are selected by the administrator and remain fixed throughout system uptime.

In this policy, the test threads (in the CTT) will operate only when they have work available. The amount of work, i.e., pages to test, depends on the monitored allocation demand. If

the selected fixed configuration cannot meet the demand, then the diagnostic will fall behind the demand, and more pages will be on-demand tested at allocation, leading to high performance overhead. If the configuration can meet the demand, then a fixed configuration may unnecessarily use too many resources. It is important to accurately profile the expected system workloads a priori to select the "best" configuration. If the workloads have significant variance in demand, then the fixed policy may prove ineffective, either being under- or over-provisioned. Hence, unless the workload demand is relatively stable, this policy can cause unnecessary resources to be occupied (over-provisioned with threads) or high overhead (under-provisioned, causing on-demand testing of pages).

5.2.4.5 Adaptive Policy Because it can be difficult to anticipate workload demand (i.e., by Fixed), Adaptive adjusts *threads* and *fidelity* at runtime to match memory allocator pressure. Priority is given to the parameters: *threads* is first minimized, and then *fidelity* is maximized, while obeying the constraints to set $test\_rate$ on MAX-CORES, MIN-FIDELITY and MAX-FIDELITY.

```
ADAPTIVE-POLICY(request_bw)
1  threads ← 1, fidelity ← MAX-FIDELITY
2  while request_bw > (threads × CALC-BANDWIDTH(fidelity))
3    fidelity ← fidelity − 1
4    if fidelity < MIN-FIDELITY
5      threads ← threads + 1
6      if threads > MAX-CORES
7        return testconfig(1, MAX-CORES)
8      fidelity ← MAX-FIDELITY
9  return testconfig(fidelity, threads)

CALC-BANDWIDTH(fidelity)
1  return (MAX-TEST-BW × 1/fidelity)
```

Figure 30: Adaptive policy to find test configuration

Figure 30 shows the algorithm for Adaptive. It is "core preserving" because core usage is minimized. It iterates over the test configuration space to find *fidelity* that uses the smallest *threads*

that meets demand. The algorithm starts with one thread and MAX-FIDELITY fidelity on line 1-2 and checks if that setting provides sufficient test bandwidth via CALC-BANDWIDTH. If not, the algorithm searches for a configuration (lines 2-8). To start the search, a lower fidelity is selected on line 3. If valid, the loop continues. Otherwise, *threads* is incremented (line 5). If the number of threads becomes higher than MAX-CORES, the configuration space has been exhausted and the maximum *threads* and minimum *fidelity* are returned (line 8). This configuration has the fastest test rate. If *threads* is less than MAX-CORES, the highest fidelity is selected (line 8) and the loop continues. If the demand can now be met, this setting is returned.

Figure 30 shows how the test bandwidth is computed for a given fidelity in my experimental system, employing a MARCH test. In my implementation, fidelity controls how many test patterns are written, read and verified. The number of test patterns used by the MARCH test is $2 \times fidelity$. For example, at $fidelity = 16$, thirty-two test patterns are used to check memory health. MAX-TEST-RATE is determined through offline profiling, which exercises the diagnostic at the lowest fidelity to perform reads, writes and checks with one test thread. The profiling accounts for all overheads of the diagnostic, including cache invalidations, reads, writes, branches and other control operations. The lowest fidelity is the one that achieves the maximum test rate, since the least amount of work is done. Because my diagnostic has a simple relationship between fidelity and amount of work (number of patterns), the maximum test rate can be linearly scaled down with increasing fidelity to determine bandwidth requirements. In more complex diagnostics, it may be necessary to profile the system under all fidelity settings. This profile information could be represented in a table that is loaded by the system at bootup. A table lookup could then be done in CALC-BANDWIDTH to get the bandwidth requirements.

5.3 EXPERIMENTAL EVALUATION

5.3.1 Methodology

I performed two types of experiments, which I term "primary" and "secondary". The primary experiments evaluate my techniques across a range of workload demands. The focus for the primary experiments is how Fixed and Adaptive compare with one another. I also conducted a set of secondary experiments to establish a baseline for the primary experiments. In the secondary experiments, I examined how the optimizations for the Concurrent Test Threads affect performance. In these experiments, I compared against the original COMeT technique.

I implemented Asteroid in Linux 3.1.7 on a Dell PowerEdge T620 server with two Intel Xeon E5-2650 2.0 GHz processors and 64 GB 1333 MHz DDR3 memory. Each processor has 8 cores (16 cores total); hyperthreading is disabled. In my experimental system, I implemented the classic MATS test algorithm [80]. Each CTT thread does this algorithm on physical pages. Four fidelity levels are used: 8, 16, 24 or 32 patterns. On-demand testing, when activated, uses minimal patterns (8).

To form workloads for my primary experiments (Sections 5.3.3, 5.3.4, and 5.3.5), I mixed benchmarks from SPEC CPU2006, according to their memory demand. The workloads are listed in Table 7; each workload has 12 programs (75% system utilization). The workloads cover three categories of demand—Low (L*), Medium (M*), and High (H*) to examine how demand pressure affects Asteroid. In each category, I increased the demand. L1 has the lowest demand and H2 has the highest.

For the secondary experiments (Section 5.3.2), I considered mixed pairs of benchmarks from SPEC. In these experiments, I ran 5 instances of each benchmark in a pair. I evaluated eight pairs spanning a range of low to moderate memory demand.

I implemented Fixed and Adaptive policies. Fix*x.yy* is Fixed with *x* test threads and *yy* test patterns (fidelity). In this notation, the first number after Fix is always a single digit for my experiments (from 1 to 8) and the second number is up to two digits. Adaptive is abbreviated, Adapt. It

Name	**Benchmarks** (Number instances denoted by "/x".)
L1	namd/2+bzip2/2+povray/2+tonto+hmmer+sphinx3+lbm+libquant+gamess
L2	perlbench+bwaves+gems++gromacs+sjeng+ h264ref+tonto+namd+bzip2+povray+hmmer+lbm
L3	omnetpp/2+mcf+cactus+soplex+gobmk+tonto+namd+ bzip2+povray+hmmer+lbm
M1	omnetpp/2+mcf+cactus+soplex+gobmk+perlbench+bwaves+ gems+zeusmp+leslie3d+gromacs
M2	astar/2+calculix/2+omnetpp+mcf+cactusADM+soplex+ perlbench+bwaves+GemsFDTD+zeusmp
M3	astar/3+calculix/3+omnetpp/2+mcf+cactusADM+soplex+gobmk
H1	milc/2+gcc/2+astar/2+calculix/2+omnetpp+mcf+ cactusADM+soplex
H2	milc/3+gcc/3+astar/3+caclulix/3

Table 7: Workload Mixes.

adjusts configuration in 1 second epochs. Monitored demand is decayed to avoid selecting a low test rate configuration too quickly. Both Fix and Adapt use up to 4 test threads, which are assigned round robin to the two processors, P1 and P2. Pages are assigned to test threads in 1 second test epochs. In the primary experiments, Fix*x.yy* and Adapt are run with all CTT optimizations enabled.

Finally, I considered a number of metrics. For the primary experiments, I reported *slowdown*, *fidelity* and *threads*. Slowdown is runtime with testing divided by runtime without testing. Because some programs finish more quickly than others in a workload, all fast programs are relaunched until the longest running program completes. The programs are pinned to the same cores in all runs to minimize contention differences between runs. For the secondary experiments, I report only slowdown. *Fidelity* is average number of test patterns applied, and *threads* is the average number of cores used. For Fix, fidelity and threads are constant (i.e., the configuration). The test

configuration is sampled every second to compute the metrics.

5.3.2 Effect of Optimizations

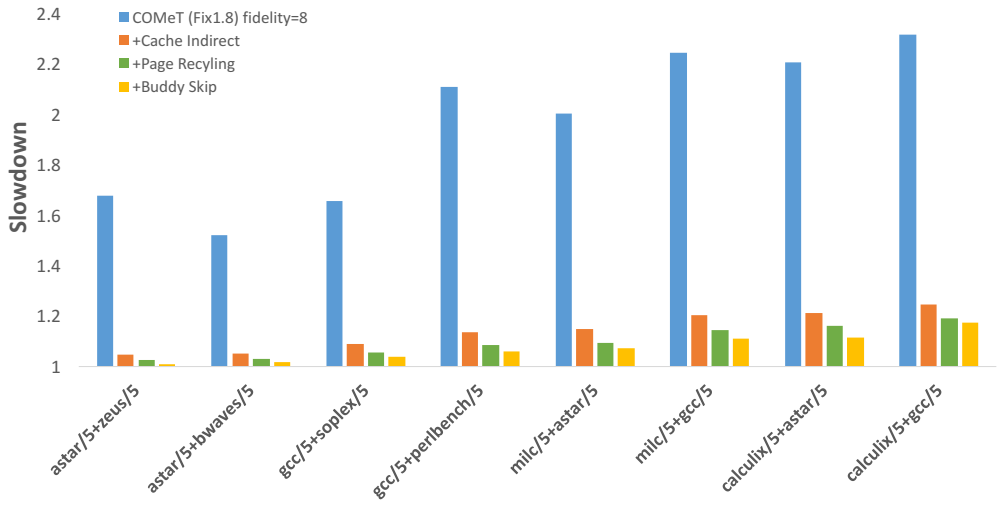

Figure 31: Impact of cache indirect testing, page recycling and skipping middle ranks for CTT

To establish a baseline for my primary experiments, which compare the fixed and adaptive diagnostic control policies, I evaluated how the CTT optimizations affect performance. Figure 31 shows a comparison of Fix1.8 with progressively more optimizations enabled. In the figure, the benchmarks are ordered left to right according to memory demand; the highest demand workload pair is on the right side. The COMeT bar represents performance of COMeT implemented in my experimental system with $fidelity = 8$ and no CTT optimizations. In essence, this bar is Fix1.8 without the optimizations. The figure shows how progressively enabling the optimizations for cache indirect testing, page recycling and skipping middle ranks of the buddy allocator influences slowdown. Going from left to right, each bar in a group adds an optimization.

As the figure shows, slowdown for COMeT is quite high (1.5 to 2.3) for the benchmark pairs; as noted in Figure 24, the basic scheme, with one test thread and no optimizations, simply cannot keep up with demand of running more than a few benchmark instances, which causes on-demand testing. Enabling cache indirect testing dramatically reduces slowdown. In this case, testing at

"memory burst" speed, permitted by grouping words into cache blocks, allows a much higher effective test rate for the MARCH diagnostic. With this higher rate, the diagnostic can better maintain the pace of demand. The figure also shows that page recycling and skipping middle ranks also lead to performance improvement. Relative to cache indirect testing, these optimizations offer a good advantage, especially as demand increases (rightmost pairs in the figure).

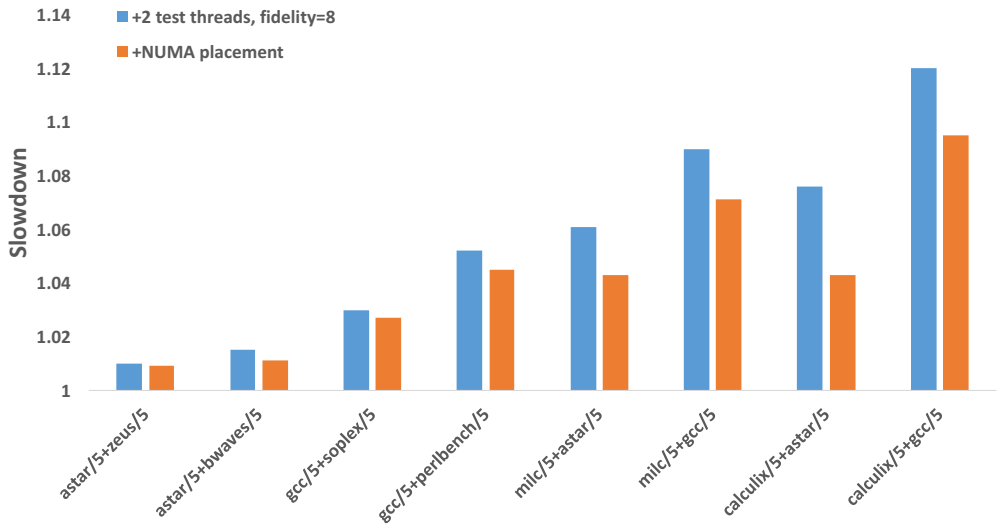

Figure 32: Impact of node local dispatch by TD

Figure 32 shows the impact of dispatching blocks to to a test thread running on the node containing the physical memory for the block (i.e., the node local to the test thread). In this figure, the configuration is Fix2.8 with cache indirect, page recycling and skipping buddy ranks optimizations enabled. The benchmarks are uniformly assigned to the cores of the two sockets (nodes) of my system to avoid imbalance in demand influencing the results.

The figure shows two cases: (1) the left bar is slowdown when blocks are distributed to two test threads based on occupancy (i.e., distribute new blocks to be tested to the least occupied test thread) and (2) the right bar is slowdown when blocks are distributed to the test thread on the local node of the block. As expected, distributing blocks without consideration for local or remote memory access incurs more overhead due to the effect of non-uniform memory access. In the M2 workload, for example, there are numerous remote memory operations, which suffer significantly

more memory latency, slowing down testing.

From these results, I conclude that all four optimizations are useful to improving the performance of testing. In the remaining sections with the primary results, all four optimizations are enabled. Fix1.8 is used in the primary results to approximate COMeT with the four optimizations enabled (since it has one test thread). Consequently, a comparison against Fix1.8 shows how adaptivity for test thread count and fidelity affect performance.

5.3.3 Fixed Configuration

NAME	Adapt	Fix4.8	Fix3.8	Fix2.8	Fix1.8
L1	1.01	1.01	1.01	1.01	1.01
L2	1.01	1.00	1.01	1.00	1.01
L3	1.02	1.01	1.01	1.01	1.01
M1	1.01	1.00	1.00	1.00	1.00
M2	1.02	1.02	1.02	1.02	1.04
M3	1.02	1.02	1.02	1.03	1.07
H1	1.02	1.02	1.03	1.04	1.08
H2	1.04	1.07	1.07	1.08	1.15
NAME	Adapt	Fix4.16	Fix3.16	Fix2.16	Fix1.16
L1	1.01	1.01	1.01	1.02	1.02
L2	1.01	1.01	1.01	1.01	1.02
L3	1.02	1.01	1.02	1.02	1.02
M1	1.01	1.01	1.01	1.01	1.02
M2	1.02	1.02	1.02	1.03	1.09
M3	1.02	1.05	1.05	1.05	1.12
H1	1.02	1.05	1.05	1.07	1.13
H2	1.04	1.12	1.12	1.14	1.20

Table 8: Slowdown for Adaptive and Fix: Fidelity 8 and 16

Table 8 and Table 9 show slowdown of the fixed configurations. The Fix columns are ordered left to right, top to bottom based on priority of slowdown, fidelity and resource usage. The leftmost and top configuration of Table 8, Fix4.8, uses the most resources (4 cores) and has the lowest fidelity (8). It is the "fastest" configuration. The rightmost and bottom configuration of Table 9, Fix1.32, uses the fewest resources and has the highest fidelity. It is the slowest. The relative maximum test rate of a configuration is $threads/fidelity$. Note the table columns are *not* sorted by the test rate.

In general, Fix's slowdown is affected by workload memory demand. Slowdown is minimized by fast configurations due to less on-demand testing. As an example, M1 has 1.1 slowdown with

NAME	Adapt	Fix4.24	Fix3.24	Fix2.24	Fix1.24
L1	1.01	1.01	1.02	1.02	1.03
L2	1.01	1.01	1.02	1.02	1.03
L3	1.02	1.02	1.02	1.04	1.07
M1	1.01	1.01	1.02	1.06	1.09
M2	1.02	1.04	1.09	1.07	1.08
M3	1.02	1.07	1.08	1.08	1.15
H1	1.02	1.08	1.09	1.13	1.15
H2	1.04	1.16	1.17	1.23	1.22
NAME	Adapt	Fix4.32	Fix3.32	Fix2.32	Fix1.32
L1	1.01	1.02	1.02	1.02	1.03
L2	1.01	1.03	1.05	1.05	1.07
L3	1.02	1.05	1.06	1.06	1.08
M1	1.01	1.07	1.08	1.09	1.10
M2	1.02	1.08	1.13	1.12	1.15
M3	1.02	1.10	1.11	1.15	1.17
H1	1.02	1.11	1.15	1.14	1.19
H2	1.04	1.18	1.22	1.25	1.25

Table 9: Slowdown for Adaptive and Fix: Fidelity 24 and 32

Fix1.32 and 1.0 slowdown with Fix4.8. Slowdown increases as memory demand increases. For example, L1 (lowest demand) with Fix1.32 has 1.03 slowdown and H2 (highest demand) with Fix1.32 has 1.25 slowdown. Some configurations to the left and top of others have higher slowdown due to lower test rate. For example, in M2, Fix3.16 has a 1.03 slowdown and Fix1.8 has 1.04. The relative test rate for these configurations is $(3/16) > (1/8)$, and therefore, Fix3.16 has slightly less slowdown.

Some results are influenced by resource competition in the memory subsystem. For example, consider M2 with Fix1.8, Fix2.16, Fix3.24 and Fix4.32. These configurations are test rate equivalent, yet they have different slowdown. Fix1.8 and Fix2.16 have similar slowdowns (1.02 and 1.03), while Fix3.24 and Fix4.32 have noticeably higher slowdown (1.09 and 1.08). The higher slowdown is due to competition: programs on P1 are sensitive to increased testing. Fix3.24 and Fix4.32 use two test threads on P1, whereas Fix1.8 and Fix2.16 use only one thread. The additional test thread consumes more memory bandwidth, causing local competition for memory bandwidth. Asteroid also tries to test memory with a local thread, which can cause imbalance in test work distribution between P1 and P2. For M2, test threads on P1 are busier, leading to additional difference in memory bandwidth consumption on P1 and P2.

The tables show the "best" configuration in shading for each workload. This choice minimizes slowdown, maximizes fidelity and minimizes resource usage (in that order). It is the rightmost and bottom one that has slowdown equal to the minimum of all configurations. For example, L1 has minimum slowdown of 1.01. Looking at L1's row, the rightmost bottom point with 1.01 slowdown is Fix4.24, which maximizes fidelity at a cost of 4 cores. Slowdown does not change much to the left of this point since Fix4.24 matches L1's demand. Lastly, the "best" choice moves leftward and upward as demand increases. The left configurations have higher test rates at the cost of lower fidelity and more cores. For example, Fix3.16 is best for M2 and Fix3.8 is best for H2. Fix3.8's test rate is twice as fast as Fix3.16.

From these results, I observed that no *single* configuration always works best due to varying demand and local effects, such as resource competition. Thus, it is difficult to select one configuration that always works. Adapt tries to resolve this issue. Next I compare Adapt to Fix.

5.3.4 Adaptive Configuration

Table 8 and Table 9 show Adapt's slowdown (left column). Slowdown is generally low: 1.01 to 1.04 (increasing with demand). Because Adapt essentially selects between Fix configurations at runtime, it can identify a configuration to satisfy demand during workload phases. Generally, Adapt should have no worse slowdown than Fix4.8, which is the fastest fixed configuration. However, in a few cases, Adapt is slightly worse. For example, in L3, Adapt has 1.02 slowdown, while Fix4.8 has 1.01 slowdown. Adapt's higher slowdown is due to management overhead and delay in reacting to memory demand.

H2 is is especially interesting: Adapt's slowdown (1.04) is better than Fix4.8's slowdown (1.07). From examining execution traces, I found that Fix4.8 tends to test pages in a burst since the work is dispatched to all threads at once. The burst causes a temporary spike in memory bandwidth competition at the beginning of each test epoch. In contrast, Adapt often selects a configuration with fewer threads (e.g., Fix2.8). This configuration gradually tests the same number of pages as Fix4.8 in each epoch, smoothing the competition.

Slowdown is only part of the story: Adapt should also maximize test fidelity and minimize

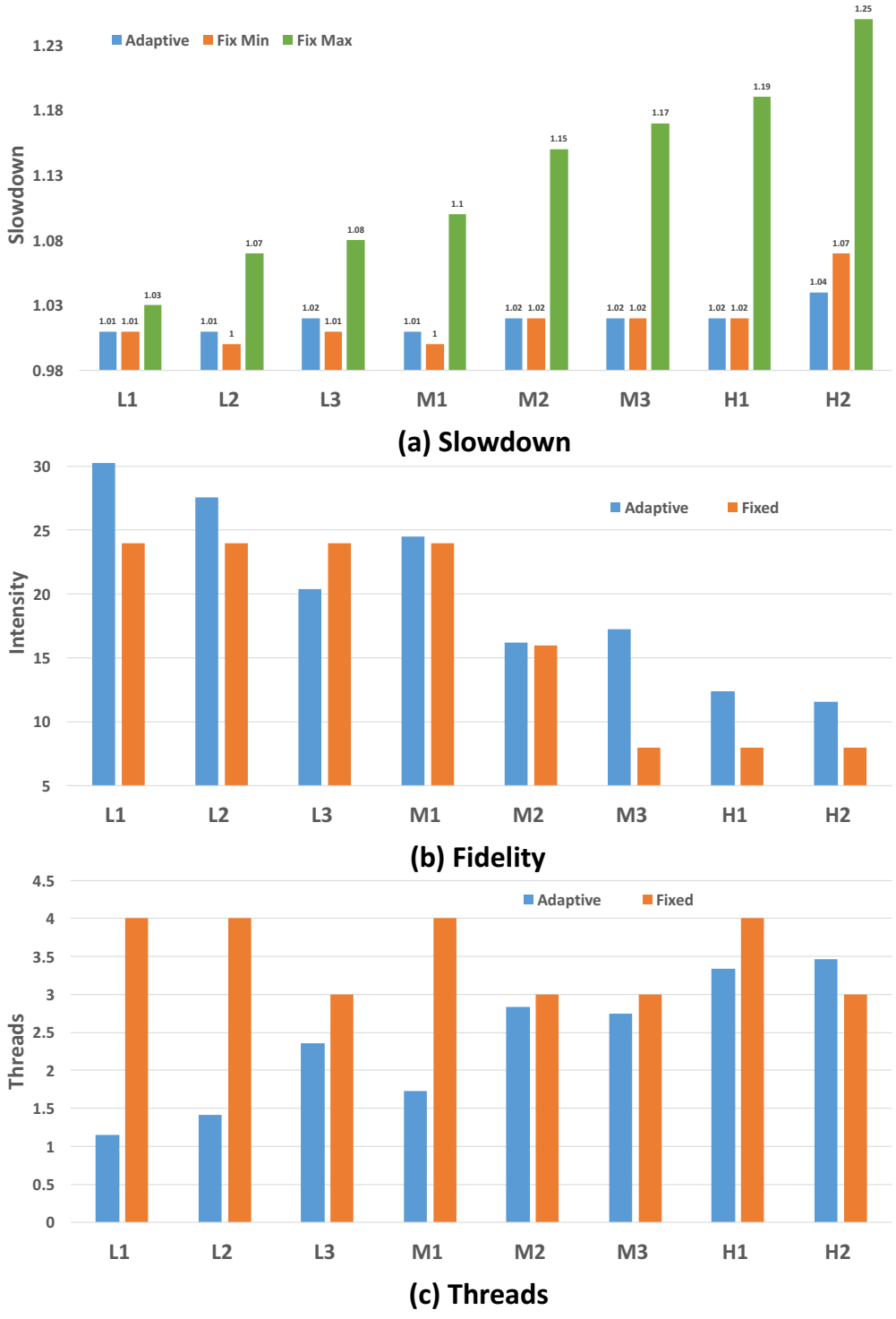

Figure 33: Comparison of performance, fidelity and threads for Adaptive and Fix.

resource usage. Figure 33 compares performance, fidelity and threads. Figure 33(a) compares Adapt's slowdown to the best configuration for Fix (shaded entries in Table 8 and Table 9) and the worst configuration (largest slowdown). The figure illustrates the range of slowdown for Fix; it also shows that Adapt does a good job to select configurations that minimize slowdown.

Figure 33(b) shows fidelity for Adapt and the best Fix choice. Dynamically adjusting configuration, particularly during low demand, is usually beneficial. Fidelity is typically higher with Adapt than Fix for L* and H* workloads. These workloads have several demand peaks and valleys, exposing opportunity to adjust configuration. For example, in L1, average fidelity is 30.3 with Adapt and 24 with Fix4.24. At the other end, H* exhibit phases with low demand that Adapt exploits to select high fidelity configurations. For example, in H2, Adapt has an average fidelity of 11.6 and Fix3.8 has a fidelity of 8. The medium (M*) workloads tend to be stable; they do not exhibit demand variance in phases. Thus, Adapt tends to select one configuration for the full run that is similar to the best Fix configuration, which explains why fidelity does not differ in Adapt and Fix on M*.

Figure 33(c) shows "cost" paid by Adapt compared to the best Fix configuration. For similar reasons as fidelity, Adapt generally has lower cost. It scales down test thread count dynamically to avoid holding onto cores. Fix inherently does something similar. For example, although Fix4.8 uses 4 test threads, the threads are not necessarily always busy. Nevertheless, Fix pays the cost to reserve resources to run testing for the full test epoch. As demand increases, Adapt employs more threads, which begins to behave similarly as Fix. In H2, Adapt used more threads than Fix because it selected high fidelity configurations during some workload phases.

5.3.5 Underlying Behavior

Next I examined Adapt's behavior on a workload's individual programs to determine whether the approach unfairly penalizes some applications, while favoring other ones. I show one workload, H2, as a representative example. It has similar behavior as the other workloads. Figure 34(a) gives the slowdown of individual programs. Some programs have more slowdown than others due to sensitivity to memory demand and local resource competition. The value in parenthesis is the core

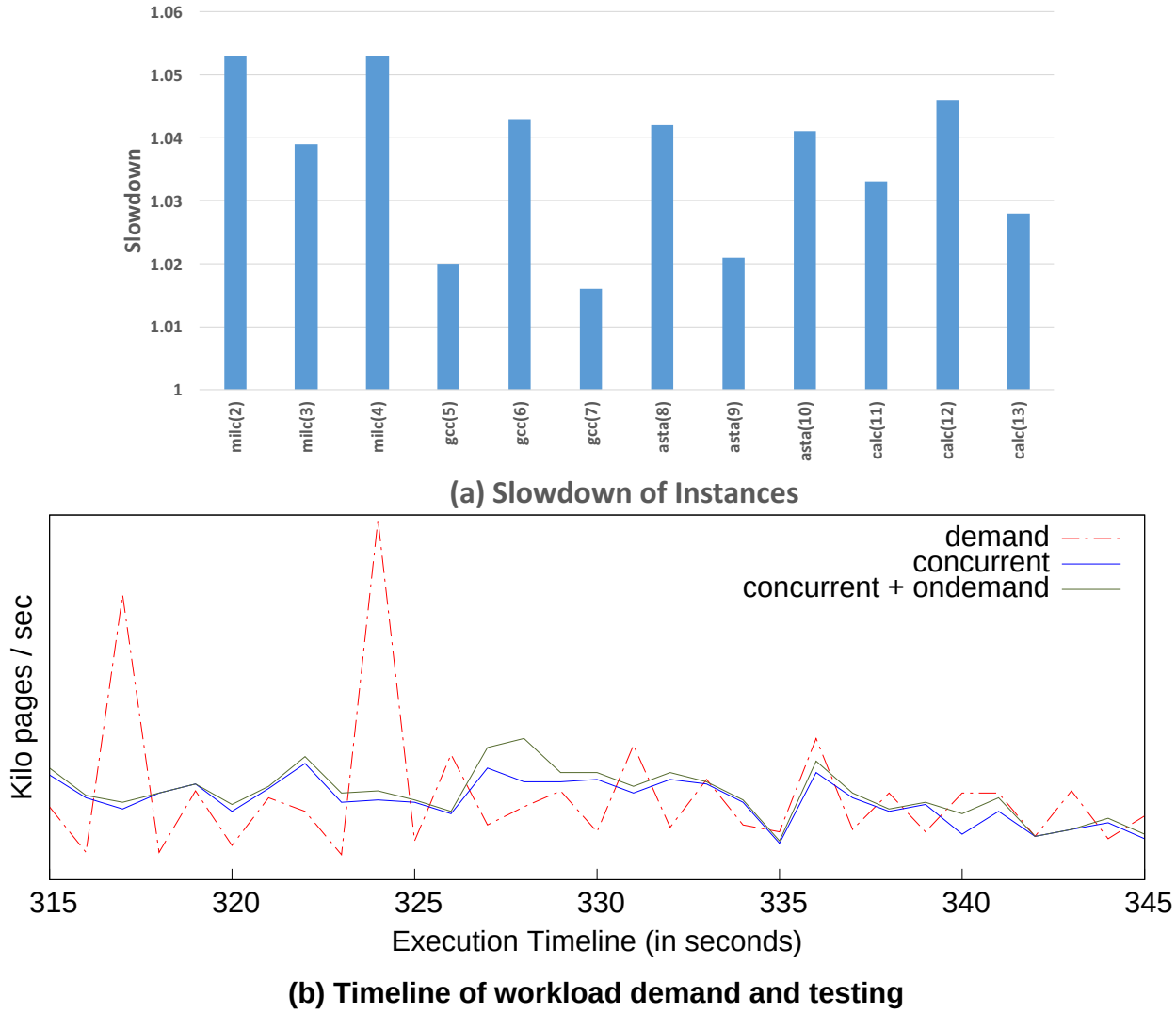

Figure 34: Slowdown of individual benchmark instances in H2 workload.

on which the program was executed. Even cores are processor P1 and odd cores are P2. The figure illustrates how local competition from testing affects slowdown: The same program on P1 has more slowdown than on P2. For instance, *gcc* has 1.04 slowdown on P1 and 1.02 slowdown on P2. There is more testing on P1, and consequently, competition and on-demand testing increases. I observed similar behavior for the other workloads. The effect is less pronounced with less demand due to dilution.

Figure 34(b) gives a snapshot of how Adapt reacts to demand. It shows actual memory demand ("demand") during 30 seconds (*s*) of execution. "Concurrent" is the rate to fill the pool of tested pages, and "concurrent+ondemand" is the rate for concurrent and on-demand testing. The curves track demand, except during bursts. For example, at 324*s*, there is a burst from an application relaunch. Adapt cannot react to the peak: The test threads do not deliver enough pages. Further, the peak is short, which is difficult to quickly accommodate. However, the peak influences later testing because the test rate is decayed. The decay allows refilling the pool during a "quiet period" after a burst empties it. Concurrent+ondemand is higher than Concurrent at 326*s* to 328*s* because the pool was depleted at 324*s*. The burst emptied the pool, leading to on-demand testing later and refilling the pool. The pool is refilled and the test rate is effectively higher than the demand between 330 and 335 seconds.

The graph illustrates that Adapt cannot always meet burst demand. In this situation, more test threads are required, which would require more cores and memory bandwidth. However, Adapt is typically effective by adjusting configuration to demand. From these results, I conclude that Asteroid, using Adapt, minimizes slowdown, while maximizing the number of tests applied to memory. It strikes a good balance to select the "right test configuration at the right time." Asteroid is also flexible enough to support multiple test strategies, as my implementation with Adapt and Fix demonstrates.

In summary, Asteroid shows an impressive scalability in systems with large amount of memory and applications with a wide range of memory requirements. Asteroid's test policies provide system administrators complete control over resources to be used by the memory diagnostic. The diagnostic optimization techniques used by Asteroid provide subtle insight into the interaction be-

tween an application and the OS memory manager that no other study has done before. Scalability, transparency, ease of deployment and configuration make Asteroid a novel memory diagnostic technique.

6.0 CONCLUSION AND FUTURE WORK

Although lack of reliability in DRAM has become a growing concern, there has been less work on online memory diagnostics which improves DRAM reliability. In my dissertation, I presented a transparent software diagnostic framework, Continuous Online Memory Testing (COMeT), to check memory for hard errors. This framework is designed to allow memory diagnostic to run concurrently with other applications in the system. To achieve low run-time overhead, the framework utilizes available cores in a CMP to proactively test memory ahead of allocation. COMeT also uses concurrent page migration to limit the exposure of in-use pages to errors. Based on system configurations and workloads, two designs and implementations of the framework are described and evaluated in my dissertation. In the first implementation, COMeT targets single-threaded and multi-threaded workloads in small-scale systems with a minimal average slowdown of just 1.04 for both SPEC CPU2006 and PARSEC. I also showed that excess TLB-shootdowns due to COMeT's operation do not harm performance of multi-threaded applications. My results demonstrate that a software-only approach to check memory health can be incorporated in a transparent fashion to have feasibly low performance overhead for small-scale CMPs.

In the second implementation, Asteroid, I addressed scalability issue that COMeT suffers from. Asteroid is essentially an improved design of COMeT to make it scalable in large scale system with multi-programmed workload. Asteroid provides for online, scalable software memory diagnostics in multi-core systems. It enables approaches that dynamically adjust test fidelity and resource usage with memory pressure to minimize performance overhead. Asteroid is integrated and optimized with the OS to ensure testing is efficient. I implemented two diagnostic policies, Fixed and Adaptive, in the framework, which illustrates Asteroid's extensibility. Using an implementation

for Linux on a 16-core server, I found that Asteroid with Adaptive has low overhead (1% to 4%), while maximizing test fidelity and minimizing resource usage. I conclude that Asteroid is effective in testing memory and has feasibly low overhead.

6.1 SUMMARY OF CONTRIBUTIONS

Throughout this dissertation, I focused on developing software-based techniques to make DRAM more reliable while system is running. My contributions can be summarized as following.

- The design and implementation of a software-only process is described which continuously tests main memory's health while actively executing user applications. The design is gradually improved to make the diagnostic process scalable.
- Techniques and algorithms are presented to test memory ahead of allocation and adaptively adjust test rate to minimize overhead. These techniques and algorithms help my diagnostic achieve a guaranteed bound on the maximum time between successive tests of a page.
- A number of test parameters and policies are shown with in-depth discussion on their derivation, usage and tuning. These parameters provide system administrator an extensive control over resources consumed by the memory diagnostic.
- The performance impact of an online memory diagnostic on single-threaded, multi-threaded and multi-programmed workloads with a wide range of memory requirements is experimentally shown. The feasibility of COMeT and Asteroid as an effective memory diagnostic is proved from a thorough evaluation of performance, energy, TLB-shootdown effect and resiliency to memory errors, including an analysis of important design and configuration choices.
- Diagnostic scalability is addressed in Asteroid. My approach shows how to trade-off between test thoroughness and CPU usage under high memory pressure on systems with large memory capacity. Asteroid shows fixed and adaptive test policies and their implication on large-scale systems.

- Based on the interactions among applications, the OS memory manager and memory hardware, a number of novel optimization techniques for memory diagnostic are described. These optimizations fit neatly within COMeT framework and work transparently in the system.
- An outline of how COMeT and Asteroid can be structured and integrated with an OS kernel is given. My design tries to modularize the diagnostic software for easy implementation and extensibility while minimizing modification to the OS kernel.
- Complete application transparency is maintained throughout my dissertation. No change in application code and/or binary is required to run in COMeT-enabled systems.

6.2 FUTURE WORK

While I made my best effort to make COMeT framework an efficient and scalable solution to online memory testing, there are still places for improvement and further work. First, in Asteroid, I evaluated "core preserving" Adaptive policy as in Figure 30. A similar "fidelity preserving" algorithm could be used to maximize fidelity. Another possible policy could be implemented in Asteroid to trigger testing for memory regions with a relatively high (above a threshold) number of error corrections, when ECC is present. This would focus the testing on that specific region. I leave these two policies to future work; my goal is to demonstrate that adaptivity, even the simple kind used by a core preserving policy, is effective at mitigating overheads.

Second, COMeT can be deployed in very large scale computing, e.g., supercomputers consisting of hundreds or thousands of nodes. Such deployment can enable further tuning of test parameters in COMeT and Asteroid. A thorough evaluation can span several months to several years which significantly increases the probability that the nodes will be hit by a considerable number of hard DRAM failures [8, 50, 66, 71]. Deployment, parameter tuning and evaluation of COMeT on supercomputing nodes can make a good contribution to the research on software-based online memory diagnostic.

Third, COMeT can use off-chip dedicated hardware to do MARCH testing using DMA, calculate and correct errors via ECC. Researchers have already shown techniques in FlipSphere to use off-chip hardware for checksum and ECC calculation [24]. Having a dedicated hardware for testing purpose is expensive. Hence, the balance between cost and benefit of off-chip hardware support in COMeT can be evaluated as a future work.

Fourth, COMeT uses available (possibly idle) resources in the system for diagnostic. Hence, COMeT can potentially increase hardware power usage. I discussed energy consumption in Section 4.2.2. However, evaluation of energy consumption due to testing in COMeT framework on large-scale systems can be helpful to data-center architects to estimate a power budget and tune COMeT according to their needs.

Lastly, over last decade, graphics processors have become cheap and powerful computing commodity and have recently been widely used in data-centers. GPUs come packed with several gigabytes of memory for video rendering as well as general purpose computing. Similar to DRAM for CPU, GPU memory also suffers from faults [74, 75]. COMeT can be deployed on GPUs to ensure improved reliability for them.

APPENDIX A

NOTES ON IMPLEMENTATION

COMeT framework is written entirely in C and integrated in Linux kernel. The framework code is designed and written as a kernel module. Although bulk of the operations of the framework is implemented in the module, some core kernel modifications were needed to make the kernel work properly when COMeT framework is present and active. In the next two sections describes these kernel changes and interesting testing and optimization techniques implemented in the module.

A.1 CORE KERNEL MODIFICATIONS

The standard Linux kernel memory manager uses two types of algorithms to manage its free memory (DRAM). Linux maintains a list of free pages in 'Buddy System' for memory chunks which are bigger than 4KB (Section 2.3.2). For memory blocks smaller than 4KB size, Linux uses "Slab Allocator" to put different sized chunks into different buckets. Memory from "Slab Allocator" is used much more rapidly by the kernel for various purposes such as allocating various software data-structures to maintain file-system, networking subsystem etc. This *Slab Allocator* relies on the *Buddy System* for memory pages. COMeT framework makes some modification to *Buddy System* to support two sets of operations:

1. Collect statistics on memory manager of the OS.
2. Access physical pages of the system.

These operations are discussed next in the context of *Buddy System*.

A.1.1 More out of Buddy System

Linux kernel provides API to ask *Buddy System* for free memory pages. These APIs have a limitation of accessing physical pages from specific list of blocks within *Buddy System*. To get memory pages from specific lists of blocks of memory pages, COMeT framework introduces another API which work on specific lists of page blocks. This API can do the following operations:

1. Extract a specific number of page blocks from front of a buddy list.
2. Extract a specific number of page blocks from end of a buddy list.
3. Return a specific number of page blocks to the front of a buddy list.
4. Return a specific number of page blocks to the end of a buddy list.

The *Buddy System* data structures are protected by a single lock. Two design choices are made in COMeT framework to reduce thrashing on this lock. First, the API functions work on a list of a particular order. As described in Section 4.1, kernel page allocation and release behavior depends on page timestamps. Due to this unique design, buddy list data structures can be quickly accessed and the time buddy lock is held can be reduced. Second, various statistics counters of COMeT A.1.3 are updated without the lock is held. This later choice may cause slight glitch in various estimated values. However, algorithms in the framework are designed in a way to keep some room for such glitch in estimations.

A.1.2 Per-CPU Page-Cache (PCP)

Linux reserves a number of pages for each CPU core to make single page allocation and release more efficient. The PCP cache is not a part of the *Buddy System*. PCP cache does not have any locking since the pages are accessible only by single CPU and only by the kernel. COMeT

framework has another set of API functions similar to *Buddy System* APIs to interact with the PCP cache.

A.1.3 Statistics Counters

To support calculation of test rate and *Test Controller*(Section 5.2), COMeT framework introduces a number of statistics counters in *Buddy System* of Linux VMM. These counters keep track of the following information of each rank of blocks in *Buddy System*.

1. number of pages allocated untested.
2. number of pages allocated tested.
3. number of pages returned and not expired.
4. number of pages returned and expired.

Using these counters, COMeT calculates adaptive test rate, alarm condition etc. In Asteroid, these counters are used by the *Test Controller* to calculate *testconfig* and *memblocks*. All these counters can be accessed via */sys/comet* interface (see Section A.2.1).

A.2 KERNEL MODULE OF THE FRAMEWORK

While COMeT framework requires some modification to Linux VMM, bulk of the code goes into a kernel module. The design of module is shown in Figure 26. Each tester thread (CTT) is created using kernel thread library provided by Linux. $kthread\_create$ function is used to create a kernel thread and provide a handler (callback) for the thread. $kthread\_bind$ function is used to bind the kernel thread to a specific CPU.

Each kernel thread sleeps until some work is arrived for it. To sleep, I used the *waitqueue* mechanism provided by Linux. Whenever the *Test Controller* has some work for a CTT, a *work* object is prepared with page numbers and the object is put into a queue for the CTT. Then *Test*

Controller signals the *waitqueue* which wakes up the CTT. CTT tests pages, returns to *Buddy System* directly and goes back to sleep. *Buddy System* statistics counters are updated when CTT returns tested pages.

A.2.1 Administrator Control Knobs

I used Linux kernel interface for *kobjects* to create entries in $/sys/comet/$. *kobjects* can be embedded into various data structures in the kernel module and this is an efficient way to manage $/sys$ entries for those data structures. System administrator can use the $/sys/comet/$ entries to tune various parameters as mentioned in Table 1 and Table 6.

A.2.2 Testing a Page

During page testing, COMeT framework must ensure a MARCH test pattern is flushed to DRAM when it is written. Intel x86 processor provides *CLFLUSH* instruction to flush a cache line. Linux kernel provides API to get cache-block size. Using cache-block size, my test function writes MARCH test pattern into one cache-block at a time and flush it before going to write to the next block. When the page is marked uncachable (Section A.2.4), COMeT's test function ignores cache-block size and writes one word at a time.

Page test is done by kernel thread. When kernel thread is initialized, it allocates a page-sized virtual address region in kernel space. I used $vmalloc$ function to allocate a virtual address region of a specific size. When a page is given to the tester for testing, the page is mapped using $vmap$ function on the previously allocated virtual address region and then tested. Upon completion of testing, the page is unmapped using $vmunap$ from kernel space and returned to caller.

A.2.3 Page Timestamps

During testing, COMeT framework has to ensure pages are timestamped properly to mark the point of time when a page was last tested for errors. To do that, the framework uses *RDTSC*

instruction which returns the 64-bit value of the Time Stamp Counter (TSC) register present on all x86 processors. It counts the number of cycles since reset.

A.2.4 Making a Page Uncachable

To make sure MARCH test patterns are written and read from DRAM, we need to bypass the CPU caches. There are two ways to do it. First, Intel Pentium processors come with a special set of registers called Memory Type Range Register (MTRR) [3] which can be programmed to a range of physical address uncachable. Programming MTRRs each time a page test request arrives is usually not a very good approach. The second way is to program the PTE when a page is mapped to virtual address. Linux kernel provides the flag *_PAGE_CACHE_UC_MINUS* [1] which can be supplied in $vmap$ function to make sure a page is marked uncachable.

A.2.5 Page Migration

One of the most interesting features of COMeT is page migration. There is a page migration timer which is triggered at specific intervals. Migration handler is already aware of which user-mode tasks need to be examined for potential expired pages. At each migration trigger, migration handler walks through the page table of a task.

```
1  void migrate_task_pages(task_struct * task)
2  {
3      mm_struct * mm = task->mm;
4      vm_area_struct * vma = mm->mmap;
5      while(vma){
6          addr = vma->vm_start;
7          while(addr<vma->vm_end){
8              pgd = pgd_offset(mm, addr);
9              pmd = pmd_offset(mm, pgd, addr);
10             pte = pte_offset(mm, pmd, addr);
11             if (pte!=NULL){
12                 page = pte_page(mm, pte, addr);
13                 if (expired(page)){
14                     newpage = alloc(PAGE_SIZE);
15                     comet_migrate(mm, addr, newpage, page);
16                 }
17             }
18             addr += PAGE_SIZE;
19         }
20         vma = vma->vm_next;
21     }
```

```
22 }
```

In this listing, $migrate\_task\_pages$ is called with a pointer to $task\_struct$. Each process in Linux is represented by a $task\_struct$ which contains information on the task, e.g., virtual memory regions, memory mappings, open file descriptors, sockets etc. All the virtual memory regions (e.g., heap, code, stack etc.) allocated for the task is saved in mm_struct. Line 2 gets the mm from task. Then in Line 3, we get the list of the virtual memory regions. Then in Lines 5-18, we iterate over each of the regions and look for mapped pages in each region via page table walking. Lines 8-10 retrieves the page table entry (PTE). If the PTE contains some value, Line 12 finds the page specified in the PTE. Once we get the page, we check if the page is already expired and migrate the page with a newly allocated page if needed. $comet\_migrate$ function calls Linux's implementation of page migration which updates page tables and updates LRU data structures which keep track of pages for possible eviction during memory pressure.

A.2.6 TLB Shootdowns

Page migration involves eventual TLB shootdown events on multi-core systems. So, collecting statistics on TLB shootdowns is required to see the effect of different intervals of page migration walks. Linux kernel implements a function $flush\_tlb\_others\_ipi$ (see in $arch/x86/mm/tlb.c$) which is the one responsible for invalidating TLB when a page table update takes place. COMeT framework puts a counter in this function to gather the statistics on this event.

APPENDIX B

DEBUGGING AND TOOLS

B.1 KERNEL DEBUGGING

Throughout my dissertation, I used two debugging techniques to debug my code changes in kernel and in the kernel module. First, *kprintf* was used which is the most effective way if serial debugging is unavailable or not applicable. For example, *kprintf* is the only way to debug issues with NUMA and DRAM access. The second way is to use virtual machine and a virtual serial port to perform serial debugging. This is more efficient way of kernel and module debugging. I used VMWare Workstation for Linux as my virtual machine. However, such debugging technique cannot be used if we need to access real hardware (e.g., accessing DRAM and NUMA).

B.2 USING VIM CODE NAVIGATOR AND CALLGRAPH

Navigating large codebase can be very difficult over terminal. I used *ctags* and *cscope* with *vim* to navigate code. I often found it helpful to generate visual representation of function calls (call-graphs) to get a bigger picture of how various segments of code are related to each other. I used

graphviz to generate callgraphs. I also used online Linux cross-reference sites (e.g., http://lxr.free-electrons.com/) to quickly navigate kernel code.

B.3 MISCELLANEOUS TOOLS

Besides debugging and code navigation, I frequently used *Python* to analyze large data-set on various experimental results. To manage code repository, I used *git* locally. Periodically, I synchronized my work private git repository online (https://bitbucket.org).

BIBLIOGRAPHY

[1] Getting a handle on caching. https://lwn.net/Articles/282250/.

[2] Linux kernel mailing list. http://kerneltrap.org/node/8059.

[3] Memory type range registers (mtrrs). In *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide Section 11.11*.

[4] Memtest86+: Advanced memory diagnostic tool. www.memtest.org.

[5] Intel xeon phi co-processor brief, white paper. 2013.

[6] J. Aas. Understanding the linux 2.6.8.1 cpu scheduler. http://josh.trancesoftware.com/linux/.

[7] A. Ansari, S. Gupta, S. Feng, and S. Mahlke. ZerehCache: armoring cache architectures in high defect density technologies. In *Int'l. Symp. on Microarchitecture*, 2009.

[8] L. Bautista-Gomez, F. Zyulkyarov, S. McIntosh-SmithOsman, and S. Unsal. Unprotected computing : A large-scale study of dram raw error rate on a supercomputer. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'16)*, 2016.

[9] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35:117–128, November 2000.

[10] V. Bhalodia. In *SCALE DRAM subsystem power analysis*. Massachusetts Institute of Technology, 2005.

[11] A. Bhattacharjee and M. Martonosi. Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 29 –40, sept. 2009.

[12] J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[13] S. Borkar. Microarchitecture and design challenges for gigascale integration. In *Int'l. Symp. on Microarchitecture*, 2004.

[14] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[15] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Design Automation Conference*, 2003.

[16] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[17] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[18] E. D. Burger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Conf. on Programming Language Design and Implementation*, 2006.

[19] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *Micro, IEEE*, 23(4):14–19, 2003.

[20] M. C. Daniel Bovet. In *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, Inc., 2005.

[21] T. J. Dell. A white paper on the benefits of chipkill. In *IBM Microelectronics Division*, 1997.

[22] Y. Du, M. Zhou, B. Childers, D. Mosse, and R. Melhem. Supporting superpages in non-contiguous physical memory. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 223–234, Feb 2015.

[23] C. Elm, M. Klein, and D. Tavangarian. Automatic on-line memory tests in workstations. In *Workshop Memory Tech., Design and Testing*, 1994.

[24] D. Fiala, K. B. Ferreira, F. Mueller, C. Engelmann, and R. Brightwell. Flipsphere: A software-based dram error detection and correction library for hpc. *Sandia National Laboratories, Technical Report SAND2014–0438C*, 2014.

[25] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. Soft-error detection using control flow assertions. In *Int'l. Symp. on Defect and Fault Tolerance in VLSI Syst.*, 2003.

[26] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. Adaptive online testing for efficient hard fault detection. In *Int'l. Conf. on Computer Design*, 2009.

[27] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *Conf. on Arch. Support for Programming Lang. and Operating Syst.*, 2012.

[28] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe. Multi-bit error tolerant caches using two-dimensional error coding. In *Int'l. Symp. on Microarchitecture*, 2007.

[29] S. Kim and A. K. Somani. Area efficient architectures for information integrity in cache memories. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 246–255, 1999.

[30] A. Kleen. Linux multi-core scalability. http://www.halobates.de/lk09-scalability.pdf.

[31] A. Kleen. An numa api for linux. http://www.firstfloor.org/~andi/numa.html.

[32] D. Knuth. *Fundamental Algorithms. The Art of Computer Programming. Vol. 1 (Second ed.)*, pages 435–455. Addison-Wesley, Reading, Massachusetts, 1997.

[33] D. Lea. A memory allocator.

[34] S.-H. Lee, C.-H. Choi, J.-T. Kong, W.-S. Lee, and J.-H. Yoo. An efficient statistical analysis methodology and its application to high-density drams. In *International Conference on Computer-Aided Design*, 1997.

[35] C. Lever and D. Boreham. malloc() performance in a multithreaded linux environment. In *USENIX Annual Technical Conference*, USENIX ATC, pages 56–56, Berkeley, CA, USA, 2000. USENIX Association.

[36] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Syst.*, 2008.

[37] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Online estimation of architectural vulnerability factor for soft errors. In *Int'l. Symp. on Computer Architecture*, 2008.

[38] X. Li, M. C. Huang, K. Shen, and L. Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *USENIX Annual Technical Conf.*, 2010.

[39] Y. Li, O. Mutlu, and S. Mitra. Operating system scheduling for efficient online self-test in robust systems. In *Int'l. Conf. on Computer-Aided Design*, 2009.

[40] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 213–224. ACM, 2011.

[41] R. Love. In *Linux Kernel Development, 2nd Edition*. Novell Press, 2005.

[42] R. Maddah, S. Cho, and R. Melhem. Data dependent sparing to manage better-than-bad blocks. *IEEE Computer Architecture Letters*, 12(2):43–46, 2013.

[43] R. Maddah, R. Melhem, and S. Cho. Rdis: Tolerating many stuck-at faults in resistive memory. *IEEE Transactions on Computers*, 64(3):847–861, 2015.

[44] R. Maddah, S. M. Seyedzadeh, and R. Melhem. Cafo: Cost aware flip optimization for asymmetric memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 320–330. IEEE, 2015.

[45] W. Mauerer. In *Professional Linux Kernel Architecture*. Wrox, 2008.

[46] A. Meixner and D. J. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *Int'l. Conf. on Dependable Syst. and Networks*, 2008.

[47] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D. D. Mannaru, A. Riska, and D. Milojicic. Susceptibility of commodity systems and software to memory soft errors. *IEEE Trans. on Computing*, 53(12):1557–1568, Dec 2004.

[48] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Int'l. Symp. on Microarchitecture*, 2003.

[49] P. J. Nair, D.-H. Kim, and M. K. Qureshi. Archshield: Architectural framework for assisting DRAM scaling by tolerating high error rates. In *Int'l. Symp. on Computer Architecture*, 2013.

[50] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In *European Conf. on Computer Syst.*, 2011.

[51] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *Conf. on Programming language design and implementation*, 2007.

[52] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Int'l. Symp. on Computer Arch.*, 2002.

[53] F. Qin, S. Lu, and Y. Zhou. SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Int'l. Symp. on High-Performance Computer Architecture*, 2005.

[54] M. Rahman and B. R. Childers. Asteroid: Scalable online memory diagnostics. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 15:1–15:8. ACM, 2015.

[55] M. Rahman and B. R. Childers. Asteroid: Scalable online memory diagnostics for multi-core, multi-socket servers. *International Journal of Parallel Programming*, pages 1–26, 2016.

[56] M. Rahman, B. R. Childers, and S. Cho. StealthWorks: Emulating memory errors. In *Int'l. Conf. on Runtime Verification*, 2010.

[57] M. Rahman, B. R. Childers, and S. Cho. Comet: Continuous online memory test. In *Pacific Rim Dependability Conf.*, 2011.

[58] M. Rahman, B. R. Childers, and S. Cho. CoMET+: Continuous online memory testing with multi-threading extension. *IEEE Transactions on Computers*, 63(7):1668–1681, 2014.

[59] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In *Int'. Symp. on Defect and Fault Tolerance in VLSI Syst.*, 1999.

[60] S. Reinhardt and S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Int'l. Symp. on Computer Architecture*, 2000.

[61] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *Int'l. Symp. on Code generation and optimization*, 2005.

[62] E. Rotenberg. AR/SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Int'l. Symp. on Fault Tolerant Computing*, 1998.

[63] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou. Using likely porgram invariants to detect hardware errors. In *Int'l. Conf. on Dependable Syst. and Networks*, 2008.

[64] S. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve. mSWAT: low-cost hardware fault detection and diagnosis for multicore systems. In *Int'l. Symp. on Microarch.*, 2009.

[65] H. Schirmeier, J. Neuhalfen, I. Korb, O. Spinczyk, and M. Engel. Rampage: Graceful degradation management for memory errors in commodity Linux servers. In *Pacific Rim Dependability Conf.*, 2011.

[66] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A large-scale field study. In *Int'l. Conf. on Measurement and Modeling of Computer Syst.*, 2009.

[67] S. M. Seyedzadeh, A. K. Jones, and R. Melhem. Counter-based tree structure for row hammering mitigation in dram. *IEEE Computer Architecture Letters*.

[68] S. M. Seyedzadeh, R. Maddah, A. Jones, and R. Melhem. Pres: Pseudo-random encoding scheme to increase the bit flip reduction in the memory. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[69] A. Singh, D. Bose, and S. Darisala. Software based in-system memory test for highly available systems. In *Workshop Memory Techn., Design and Testing*, 2005.

[70] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Int'l. Symp. on Computer Architecture*, 2002.

[71] 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 *ACM SIGPLAN Notices*, volume 50, pages 297–310. ACM, 2015.

[72] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The case for lifetime reliability-aware microprocessors. In *Int'l. Symp. on Computer Arch.*, 2004.

[73] D. Tang, P. Carruthers, Z. Totari, and M. W. Shapiro. Assessment of the effect of memory page retirement on system RAS against hardware faults. In *Int'l. Conf. on Dependable Syst. and Networks*, 2006.

[74] D. Tiwari, S. Gupta, G. Gallarno, J. Rogers, and D. Maxwell. Reliability lessons learned from gpu experience with the titan supercomputer at oak ridge leadership computing facility. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 38:1–38:12, New York, NY, USA, 2015. ACM.

[75] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, et al. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–342. IEEE, 2015.

[76] A. van de Goor and I. Tlili. March tests for word-oriented memories. In *Design Automation and Test in Europe Conf.*, 1998.

[77] R. van Rein. BadRAM: Linux kernel support for broken RAM modules, site last visited July 11, 2010. http://rick.vanrein.org/linux/badram/.

[78] R. Venkatesan, S. Herr, and E. Rotenberg. Retention-aware placement in dram (rapid): software methods for quasi-non-volatile dram. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 155 – 165, 2006.

[79] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43:76–85, April 2009.

[80] C.-F. Wu, C.-T. Huang, K.-L. Cheng, and C.-W. Wu. Simulation-based test algorithm generation for random access memories. In *IEEE VLSI Test Symposium*, pages 291–296, 2000.

[81] D. H. Yoon and M. Erez. Virtualized and flexible ECC for main memory. In *Int'l. Conf. on Arch. Support for Programming Lang. and Operating Syst.*, 2010.

[82] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August. DAFT: Decoupled acyclic fault tolerance. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, 2010.

[83] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. In *Int'l. Symp. on Microarchitecture*, 2008.