MemGuard: A Low Cost and Energy Efficient Design to Support and Enhance Memory System Reliability

Long Chen   Zhao Zhang
Department of Electrical and Computer Engineering
Iowa State University, Iowa, US
{longc,zzhang}@iastate.edu

Abstract

Memory system reliability is increasingly a concern as memory cell density and capacity continue to grow. The conventional approach is to use redundant memory bits for error detection and correction, with significant storage, cost and power overheads. In this paper, we propose a novel, system-level scheme called MemGuard *for memory error detection. With OS-based checkpointing, it is also able to recover program execution from memory errors. The memory error detection of MemGuard is motivated by memory integrity verification using log hashes. It is much stronger than SECDED in error detection, incurs negligible hardware cost and energy overhead and no storage overhead, and is compatible with various memory organizations. It may play the role of ECC memory in consumer-level computers and mobile devices, without the shortcomings of ECC memory. In server computers, it may complement SECDED ECC or Chipkill Correct by providing even stronger error detection.*

We have comprehensively investigated and evaluated the feasibility and reliability of MemGuard. We show that using an incremental multiset hash function and a non-cryptographic hash function, the performance and energy overheads of MemGuard are negligible. We use the mathematical deduction and synthetic simulation to prove that MemGuard is robust and reliable.

1. Introduction

Memory error protection is increasingly a concern as memory density and capacity continue to increase. Two recent, large-scale field studies on data center computers [46, 17] report that DRAM has much higher error rate than previously reported. For computer systems without error protection, memory errors can tamper memory data and crash programs and even the operating systems. It has been reported from real machine failure statistics that main memory is responsible for about 40% of system crashes among those caused by hardware failure [31].

Main memory error protection is usually implemented by storing redundant information in memory devices. A conventional ECC memory module uses (72, 64) Hamming-based [13] or Hsiao code [16] to provide the capability of single-bit error correcting and double-bit error detecting (SECDED). Such a module stores ECC bits in extra DRAM device(s) and requires 72-bit data bus instead of 64-bit data bus used in non-ECC memory modules. Chipkill Correct [7, 30] has additional capability of single device data correction (SDDC) to tolerate multi-bit errors from a single chip. Those two schemes incur storage overhead of 12.5% and are incompatible with non-ECC memory modules. They also constrain the memory organization such that recently proposed memory sub-ranking [51, 54, 1], which improves memory energy efficiency, may not be used. The majority of consumer-level computers, including desktop and laptop computers and mobile devices, have not yet adopted any memory error protection.

We propose *MemGuard*, a system-level scheme with lightweight hardware extension, to support or enhance memory reliability for a wide spectrum of computer systems. The core part of MemGuard is a low-cost and highly-effective mechanism of *memory error detection*, which is revised from more complex designs of memory integrity verification [6, 49] proposed for secure processors. By maintaining a *read log hash* (READHASH) and a *write log hash* (WRITEHASH), 128-bit each, MemGuard can detect multi-bit errors of the main memory with very strong confidence. Conceptually, READHASH is a hashed value of a log of all memory read accesses, and WRITEHASH is one for all memory write accesses. For each access, the logged information is a pair (address, data). Periodically or at the end of program execution, MemGuard synchronizes READHASH and WRITEHASH to the same point of execution and then matches them. A mismatch means that the result of at least one read does not match that of the previous write to the same memory address, and therefore a memory error must have happened.

MemGuard does not have *hardware* memory error correction as ECC and Chipkill Correct do, but is much stronger in error detection. It utilizes (and relies on) an OS-based checkpointing system for error recovery. If a memory error is detected during the execution of a given program, the program will be rolled back to the latest checkpointed state. Since memory error is relatively infrequent (hours for memory of gigabytes), checkpointing can be done at a low frequency and the performance degradation from checkpointing is insignificant. MemGuard is very useful for computers that have no other memory error protection. For consumer-level computers in particular, MemGuard does not require any change to computer motherboards or memory modules, the hardware cost of MemGuard itself is negligible, and its protection can

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

be selectively enabled for programs for which computation reliability is desired. Additionally, MemGuard is compatible with sub-ranked memories as well as narrow-rank memories used in mobile devices.

MemGuard is also useful for large-scale, high-performance computing applications. Many of those applications already use checkpointing or redundant computation or both, otherwise they may not run to completion [19, 10]. With MemGuard, they do not have to run on computers with ECC memory, which leads to reduced cost and improved energy efficiency. For computers of ECC or Chipkill Correct memories, MemGuard can further enhance memory reliability by reducing the probability of silent data corruption (false negative), particularly when single-bit error correction occurs. MemGuard alone, however, may not be suitable for commercial workloads that require high availability and instant recovery, as checkpoint recovery may cause a delay of seconds or dozens of seconds, which may be visible to end users.

MemGuard is motivated by memory integrity verification using read and write log hashes [49] in secure processor design, but the design objective and complexity are very different. In secure processor design, the hash function must be very strong to repel carefully crafted attacks from adversaries, and particularly it has to include a timestamp per memory block to detect replay attacks. Significant design complexity, storage and energy overheads, and high cost can be justified. In memory reliability design, the error pattern is random and unintentional, and for consumer-level computers the cost has to be contained. In the design of MemGuard, we carefully choose a simple, non-cryptographic hash function to minimize the impact on energy efficiency and cost, and has dropped the use of timestamp. Had timestamp been used, there would be significant storage, performance and energy overhead for DDR*x* memories. To our best knowledge, we are the first to adopt such a scheme solely for memory reliability in conventional computers and to comprehensively evaluate and prove its feasibility and reliability for error protection.

The rest of the paper is organized as follows. Section 2 introduces the background of memory error protection and the related work. Section 3 presents the details of MemGuard design. Section 4 describes the experimental methodology, and the results are presented and analyzed in Section 5. Finally, Section 6 concludes the paper.

2. Background and Related Work

2.1. Main Memory Error Rate

Memory errors have been studied for decades [25, 35, 3, 5, 52, 50, 36]. Recent studies on large-scale machines have found that DRAM error rate is orders of magnitude higher than previously reported. A study on Google's computer fleet reports about 25,000$\sim$70,000 FIT per Mbit of DRAM systems [46], where FIT is failure in time represented as number of failures in a billion operation hours. Another study based on IBM Blue Gene (BG) supercomputers at multiple national laboratories also reports high error rate; for example, FIT of 97,614 on BG/L computers at the Lawrence Livemore National Laboratory and 167,066 on BG/P computers at the Argonne National Laboratory [17]. In addition, error correlation with DRAM row or column has been observed, which implies high probability of multi-bit errors.

As the DRAM feature size further scales down, memory error rate is increasing for complex scaling effects. The 2012 updated ITRS roadmap has projected that DRAM fabrication technology will scale to 10nm in 2022 and 6nm in 2026 [21]. DRAM scaling has presented increasing challenges to memory reliability because of gradually prominent physical effects. For example, thinner MOS gate dielectric sustains less time to breakdown. In addition, the scaling raises the issues of process variations and random charge fluctuations, which exacerbate memory cell reliability. There are also other effects discussed in detail in ITRS 2011 [20], such as p-channel negative bias temperature instability (NBTI), the random telegraph noise (RTN), and others, which are increasingly severe. The increasing error rate cannot be fully hidden by conventional ECC design; for example, double-bit memory errors occur on daily basis in the Cary XT5 supercomputer at Oak Ridge National Lab [10].

2.2. Main Memory Error Protection

Conventionally, Hamming-based [13] or Hsiao [16] (72,64) code is used to construct SECDED ECC DIMM for error protection. It is capable of correcting single-bit error and detecting double-bit error. In typical DDR*x* memory system, eight (x8) or sixteen (x4) DRAM devices work in tandem to form a 64-bit data path. A SECDED ECC DIMM has one (x8) or two (x4) extra devices to store ECC code word, and the data bus is extended to 72 bits to transmit 8-bit parity with 64-bit data. It restricts the DIMM organization by limiting the ratio of data to ECC to 8:1, and requires an upgrade of motherboard.

Chipkill Correct, a stronger but more costly design, is able to correct multi-bit errors from a single device. In other words, it supports SECDED plus SDDC (Single Device Data Correction). It introduces significant power overhead as 72 devices work in lockstep. A revised design groups multiple bits from one memory device as a symbol and applies symbol-correction code to recover a device failure [52]. It typically organizes 36 x4 DRAM devices in a particular way to form a 144-bit data path, transferring both data and symbol-correction code. Such a scheme involves 36 DRAM devices in one memory access, still with significant memory power consumption. The recent proposed LOT-ECC [50] uses multi-tier error detection and correction codes to support Chipkill-level protection. It uses nine x8 DRAM devices and carefully organizes data, EDC (error detecting code) and ECC bits at each tier, but with an increased ratio of storage overhead.

2.3. Memory Organization Variants

Various memory organizations and structures have been proposed to improve memory performance or energy efficiency. In sub-ranked DRAM memories [54, 1, 51], the number of DRAM devices in a rank is reduced so as to reduce DRAM operation power spent on precharge and activation. However, the reduced number of devices in a rank presents a new challenge to memory error protection as it breaks the 8:1 ratio of conventional SECDED design. Another type of memory structure stacks DRAM dies by Through-Silicon Via (TSV) technology to reduce its power consumption and improve bandwidth and capacity. One promising product is Hybrid Memory Cube (HMC) [18] with high bandwidth and low power consumption, and it is projected to appear in market in 2014. Such a product drastically changes conventional DRAM organization and therefore presents new challenges for memory error protection. MemGuard does not put any constraint on memory organization and thus can work with those memory organizations.

2.4. Other Related Work

There have been many studies on memory system reliability [52, 50, 36]. Most of them focus on error correction at memory module level. One of the most recent studies, closely related to our work, is ArchShield [36]. It proposes an architectural framework to tolerate fabrication faulty cells (hard error) induced by extreme scaling of DRAM. In their design, a fault map is used to record all the faulty cell locations obtained by built-in self test. By consulting the fault map, it maintains replications of those words in memory space for error correction. MemGuard and ArchShield serve for different purposes, as MemGuard targets soft and intermittent errors rather than permanent errors.

3. MemGuard Design

3.1. Incremental Hash Function

Incremental hash function is first proposed by M. Bellare et al. [4] in 1990s. Such a hash function has the property that the cost to update the result hash upon a modification to the original message is proportional to the modification. Consider a message M and its hash value $H(M)$. With modifications δ to the message, the result message is denoted as $M' = M + \delta$, in which $+$ denotes a modification such as to replace, insert or delete a data block of the message. Incremental hash function can update the result hash of modified message using the following equation:

$$H(M') = H(M) + H(\delta)$$

where $=$ and $+$ are equality and modification operations, respectively, for the defined hash function. The equation means that as long as we have $H(M)$ and $H(\delta)$, we could calculate the result hash. There is no need to retrieve the original message information of M.

Multiset hash functions are a particular type of incremental hash function, operating on multiset [6]. A multiset is defined as a finite and unordered collection of elements where the occurrence of each element can be greater than one. If the occurrence of each element is exactly one time, the multiset is reduced to a set. Multiset hash functions are incremental and the result hash of the multiset is independent of the ordering of the elements within it. Specifically, let $\cup$ be the union operation of multiset, the properties of multiset hash functions can be denoted using the following two equations

$$H(M \cup \{b\}) = H(M) +_H H(\{b\}) \tag{1}$$

$$H(M_1) +_H H(M_2) = H(M_2) +_H H(M_1) \tag{2}$$

where $+_H$ denotes a chosen hash addition operation. Equations (1) and (2) show the property of additivity and commutativity, respectively, for multiset hash functions.

Those two properties can be explored in main memory system to create a fingerprint of memory accesses. In this case, each memory access is regarded as an item and a sequence of memory accesses is considered as a multiset as there exist duplicated memory accesses to same memory address. Define $H(\{q\}) = H_s(s_q)$, where H_s is a hash function that takes input of a string s_q formed of (address, data) pair of the memory request q. The details of selecting H_s hash function are presented in Section 3.4. Based on the two properties of multiset hash function, we thus have the hash for a sequence Q of memory requests q_i $(i \in [1,N])$ below

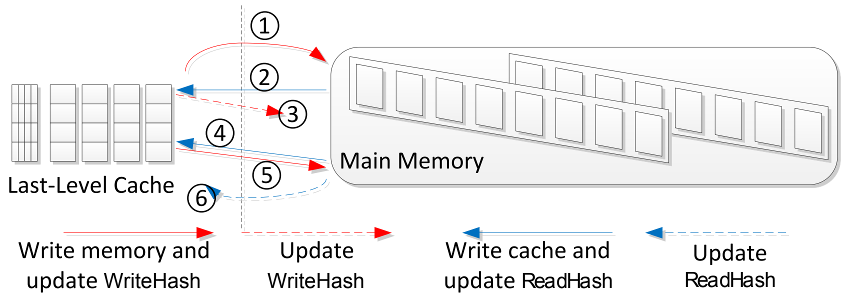
$$H(Q) = H_s(s_{q_1}) +_H H_s(s_{q_2}) +_H \cdots +_H H_s(s_{q_N}) \tag{3}$$

, where Q is a multiset that $Q = \{q_1\} \cup \{q_2\} \cdots \cup \{q_N\}$. Following the two properties of multiset hash functions, the result hash $H(Q)$ is irrelevant to the ordering of the requests and it can be calculated incrementally by adding the hash value of coming memory request. In other words, $H(Q)$ represents a fingerprint of a sequence of memory requests.

The previous study [6] proposes four types of multiset hash functions: *MSet-XOR-Hash*, *MSet-Add-Hash*, *MSet-Mu-Hash* and *MSet-VAdd-Hash*, based on four different operations, binary XOR, conventional addition, multiplication and vector addition, respectively. That means the $+_H$ can be any of these four operations to form a multiset hash function based on a strong hash H_s. *MSet-XOR-Hash*, *MSet-Mu-Hash* and *MSet-VAdd-Hash* are not proper in the MemGuard design as they either merely support set collision resistance or introduce high overhead because of operational complexity. We choose *MSet-Add-Hash* for being multiset collision resistant and simple in operation. The function uses simple addition operation and outputs the lower m bits of the sum, where m is the output length of hash function H_s.

3.2. Log Hash Based Error Detection

In the MemGuard design, error detection is implemented by maintaining and cross-checking a read log hash and a write



①
②
③
④
⑤
⑥
Main Memory
Last-Level Cache
Write memory and update WriteHash
Update WriteHash
Write cache and update ReadHash
Update ReadHash

Figure 1: **Memory operations for memory error detection**

log hash maintained in the memory controller. The two hashes are denoted as READHASH and WRITEHASH, respectively, which *conceptually* log the (address, data) pairs of memory reads and memory writes. *Here memory reads and memory writes are different from actual reads and writes to the main memory*, which will be discussed later. At runtime, the two hashes are updated upon last-level cache and memory events. Periodically and at the end of a program execution, MemGuard synchronizes the READHASH and the WRITEHASH to ensure every memory read is logged in READHASH and every write is logged in WRITEHASH. If no error occurs, the two hash values must match to each other.

Figure 1 and Algorithm 1 show the steps of updating the READHASH and the WRITEHASH such that the two hashes are synchronized for error checking. We assume that the last-level cache is write-back and write-allocate. In the beginning, when the OS loads the program to be executed into memory, the memory controller will *log* each write of a memory block into WRITEHASH, i.e. to hash the pair (address, data) of the memory block into WRITEHASH. A memory block is a block of memory of the cache block size. Note that we do not assume the OS will load all memory pages of the program into memory at this time. If a memory page is loaded during program execution because of page fault, all blocks of the faulted page will be logged into WRITEHASH. We assume that there is a special DMA mode for program loading, which triggers the update of WRITEHASH at the memory controller. Normal memory accesses during OS kernel execution are not logged. This step is shown as ① in Figure 1.

When the program is running, the memory controller enters a logging mode (set by the OS). The read_block operation is executed at the memory controller when a last-level cache miss triggers a memory read. The (address, data) pair of the memory block is logged into READHASH, shown as ② and ④ in Figure 1. If a memory block is loaded into cache for the first time after program loading, the read will match the write to the same memory block address at the time of program loading.

For each cache replacement (eviction), the write_block operation is executed to log a memory write in WRITEHASH. Note that an actual memory write happens only when the replaced block is dirty; however, even if the replaced block is clean, the eviction is treated as a (artificial) write and is logged in WRITEHASH. The purpose is to ensure the same set of (address, data) pairs will be logged in READHASH and WRITEHASH: if the evicted block is loaded to cache again, the read will match the logged write at the time of eviction (assuming no error happens between the eviction and the load), no matter whether an actual memory write happened or not at the time of eviction. Steps ③ and ⑤ in Figure 1 show the write_block operations for clean and dirty block, respectively.

At the end of program execution or during periodical memory error checking, the integrity_check operation is executed as shown in Figure 1 as ⑥. For each physical memory block used by the program, the OS checks if the block is cached or not. If it is not cached, the OS loads the memory block and has the (address, data) pair logged in READHASH. Again, this is to ensure that the same set of (address, data) pairs will be logged in READHASH and WRITEHASH: for every memory block not cached at this time, there is an (address, data) pair that has been logged in WRITEHASH but not in READHASH. This step effectively synchronizes READHASH and WRITEHASH. Then, if READHASH and WRITEHASH match, the OS determines that no memory error has happened. Otherwise, it determines an error has happened and will trigger checkpoint recovery. We assume that READHASH and WRITEHASH are memory-mapped registers (or special I/O registers) for the OS to access. We also assume that the memory controller has a special mode, in which it does not log the normal memory accesses from the OS, and the OS may write a memory block address to a special memory-mapped register for the memory controller to load a memory block and log its (address, pair) into READHASH.

MemGuard does not use per-block timestamp as in the study on secure processor design [49]. The purpose of using timestamp is to prevent replay attack from an adversary, which is very unlikely in a normal system. Assume that an error occurs to a memory block of block address A and changes its value from X to X'. During program loading, (A, X) has been logged in WRITEHASH. When the block is loaded into cache, (A, X') is logged into READHASH, which does not match the logged write. To form an error like one from a replay attack, the program execution may have to generate a write of (A, X') at a later time, and then another memory error shall happen to the same memory block to change the block's value from X'

Algorithm 1 MemGuard error detection algorithm

```
Initialization Operation
function init_block (ADDRESS, INITDATA) {
  update WRITEHASH with the hash of (ADDRESS, INITDATA)
  write INITDATA to ADDRESS in the memory
}

Run-Time Operation
— when there is a cache miss
function read_block (ADDRESS) {
  read DATA from ADDRESS in the memory
  update READHASH with the hash of (ADDRESS, DATA)
}

— when there is cache eviction
function write_block (ADDRESS, DATA) {
  update WRITEHASH with the hash of (ADDRESS, DATA)
  if the block at ADDRESS is dirty then
    write DATA to ADDRESS in the memory
  end if
}

Integrity Check Operation
function integrity_check () {
  foreach ADDRESS in the memory space
    if the block at ADDRESS is not cached then
      call read_block (ADDRESS)
    end if
  end foreach
  if READHASH = WRITEHASH then
    return no error
  else
    return error exception
  end if
}
```

back to X. The chance of this happening is very low.

3.3. Reliability Analysis

MemGuard compares READHASH and WRITEHASH to decide if a memory error occurs in a sequence of N accesses. A hash collision happens if a memory error has happened but READHASH matches WRITEHASH. The probability of hash collision is the probability of false negative of memory error detection. The study of *MSet-Add-Hash* [6] defines that two result hashes are equivalent if the modulus sums of the hashed values by hash function H_s are the same. Given that the collision rate is $\frac{1}{2^m}$ for a strong m-bit hash function and assume the condition that exact i out of N read requests return erroneous data, the probability of collision is given by Formula (4):

$$P(i) = (\frac{1}{2^m})^i + (\frac{1}{2^m})^{i-1}(1-\frac{1}{2^m}) + \cdots + \frac{1}{2^m}(1-\frac{1}{2^m})^{i-1} \tag{4}$$

Let p_0, p_d and p_w denote the probability of no error, detectable error and undetectable error in one data block, respectively. The probability that i out of N accesses have error is $C_i^N(p_d+p_w)^i(1-p_d-p_w)^{N-i}$. We thus have the collision rate for MemGuard design, represented by

$$F_{MemGuard} = \sum_{i=1}^{N} P(i) \cdot C_i^N (p_d+p_w)^i(1-p_d-p_w)^{N-i} \tag{5}$$

For conventional error protection scheme, it fails as long as undetectable error occurred to one of the N request. Therefore, the failure rate for conventional error protection is given by

$$F_{conventional} = 1-(1-p_w)^N \tag{6}$$

Based on Formulas (5) and (6), we build a simple model to compare the error detection capability of MemGuard design with the conventional SECDED protection. Assume the probability that any one bit error occurs to a data block with n_b bits is p in a time period t. Then, $p_0 = (1-p)^{n_b}$ denotes the probability of no error in a data block. The probability of having an exact one-bit error is denoted as $p_1 = n_b p(1-p)^{n_b-1}$. The probability for two-bit error is denoted as $p_2 = C_2^{n_b} p^2 (1-p)^{n_b-2}$. As SECDED detects up to two-bit error, the detectable error rate for SECDED is thus approximately[1] $p_d = p_1 + p_2$; and the undetectable error rate is $p_w = 1 - p_0 - p_1 - p_2$.

With error rate of 25,000$\sim$70,000 FIT/Mbit from study [46], we compare failure rate of $F_{MemGuard}$ with $F_{conventional}$. The results are presented in Section 5.1.1. In general, MemGuard has much higher error detection rate than SECDED in all the cases we studied. Additionally, as the time period t grows, the undetectable error rate for SECDED grows but that for MemGuard does not increase. Furthermore, because the collision rate for a strong hash function is irrelevant with number of bits flipped in data, MemGuard design has better tolerance for multi-bit errors.

3.4. Selection of Hash Function

In MemGuard, a hash function H_s is used to convert the (address, data) pair of a memory request to a hash value before the multiset hash function is used. Hash functions are generally classified as cryptographic ones and non-cryptographic ones. Cryptographic hash functions are applied in secure applications and systems, i.e. MD4 [42], MD5 [43], SHA1 [26], SHA2 [15], etc., to protect the system from resourceful and malicious adversaries. They are typically complex in computation and have a low throughput. For example, MD5 generates a 128-bit hash code by four rounds of computation with 16 operations in each round. To the best of our knowledge, the best FPGA and ASIC implementation reports 0.73GB/s and 0.26GB/s throughput with cost of 11,498 logic slices and 17,764 logic gates [22, 45], respectively. SHA1 is more complicated, requiring four rounds of 20 operations each to generate a 160-bit hash code.

[1] SECDED can detect three and more bits of error with certain probabilities. The exact probability is evaluated by Monte Carlo simulation in Section 5.

Memory error is completely disparate from the intentional malicious attacks. It is caused by cosmic rays, alpha particles and other sources of stochastic nature. Therefore, simple and cost effective non-cryptographic hash functions can be adopted in MemGuard. There exist multiple non-cryptographic hash functions, for example Pearson Hashing [39], Fowler-Noll-Vo(FNV) [11], CRC [40], MurmurHash [2], lookup3 [23], SpookyHash [24], CityHash [12] and others. We still have to make a careful selection for performance and power efficiency.

Above all, the hash function should be collision resistant to provide a strong capability of error detection. A strong hash function with m-bit output has a collision rate of $1/2^m$ if we assume the function produces each hash value with exactly the same probability. With birthday attack, the rate increases to approximately $1/2^{\frac{m}{2}}$. Therefore, the selected hash function should be able to create output with a decent length. The original Pearson hashing algorithm generates only 8-bit output, which cannot be adopted in this design. Second, the output hash values need to follow uniform distribution independent of the distribution of inputs. Otherwise, it will introduce clustering problem which can result in high collision rate. Third, a good hash function is required to have a good level of avalanche effect. Avalanche effect presents the ability for a hash function to produce a large change in output bits upon a minor modification to input bits. The avalanche effect thus can dissipate minor modification in input data to a large structure of output bits, which enhances error detection capability. Previous studies [8, 24] present that lookup3 Hash, SpookyHash, MurmurHash and CityHash all have good properties in avalanche effect.

In addition, the required hash function should be non-linear. Being linear in this context means $H_s(A+B) = H_s(A) + H_s(B)$ mathematically, where A and B are two inputs, and $+$ can be general addition operation or binary xor operation. The reason is that it can introduce high collision when being applied to *MSet-Add-Hash*. Following Formula (1), the hash values of two tampered data can cancel the modification out with a high possibility if the hash function is linear. For example, given two data A and B, assume there is a single-bit error to both of these data at the same bit position. Using A' and B' to denote the tampered data, we have $A+B = A'+B'$. Therefore, $H_s(A') + H_s(B') = H_s(A) + H_s(B)$ even if the single bit error in data A and B creates a great change in its hash value, given a linear hash function. Thus, CRC hash function can not be applied as $CRC(A \oplus B) = CRC(A) \oplus CRC(B)$.

Based on the criteria required, we opt for SpookyHash designed by Jenkins in 2011 [24]. SpookyHash produces well distributed 128-bit hash values for variable length of input. It has been tested by the author for collision up to 2^{72} key pairs, which presents a good collision resistance. SpookyHash is said to achieve avalanches for 1-bit and 2-bit inputs, which means that any 1-bit or 2-bit change in inputs results in a flip in each output bit with 1/2 possibility. In addition, SpookyHash is simple and fast and it costs merely 64-bit addition, xor and rotation operations. SpookyHash classifies keys as short if the length of input is less than 192 bytes and thus compute hash code with a simpler function. In the MemGuard design, we combine each 64-byte data block with its address as input keys. The address is assumed to be 8 bytes with paddings to make a sufficiently large space. The input key size is thus 72 bytes, which is regarded as short by SpookyHash.

The implementation cost and energy overhead of SpookyHash are almost negligible to modern processors. We first carefully scrutinize SpookyHash function and observe that it takes 45 64-bit addition operations, 35 xor and 35 rotation operations to do the hashing. A previous study [32] presents that the energy consumption of a 64-bit adder with 65nm process is 8.2 pJ. Using this number, we calculate that SpookyHash consumes less than 1.0 nJ for each hashing operation. On the other hand, DDR3 memory power calculation is well established by Micron [34]. A complete memory access cycle includes precharge, activation, I/O drive and termination, and data transfer operations. In addition, there is consecutive background power consumption and it is increasing as the number of DRAM devices in a system grows. Taking Micron MT41J256M8 [33] device as an example, a DIMM of eight x8 devices can consume 62.0 nJ energy for a complete memory access cycle. The energy consumption of a real system can be higher as it conventionally comprises multiple channels with multiple DIMMs per channel. Therefore, the energy consumption for SpookyHash is almost negligible compared to that of the DRAM access.

3.5. Checkpointing Mechanism for Error Recovery

In the MemGuard design, errors in the program can be efficiently detected. We turn to OS-based checkpointing mechanism for error recovery. Checkpointing methods have been studied for decades [29, 41, 44, 28, 37] and most recent studies [27, 53, 9] discuss checkpointing recovery scheme for failures in high performance computing (HPC). In general, checkpointing takes the snapshot of entire state of a program at the moment it was taken. It thus maintains all the necessary information for a process to restart from the checkpoint.

Upon an error detection, checkpoint recovery is initiated and the program is rolled back to the most recent checkpoint. The program state is overwritten with the stored checkpoint state. In this case, the computation back to the checkpoint is discarded and the system pays the performance overhead for error recovery. The more frequent the checkpoints are taken, the less the rollback overhead. However, checkpointing itself introduces penalty as it requires time and storage cost. Book [28] presents an analytical model of checkpointing and discusses in detail of checkpointing placement issue and its optimization. As a memory error is an uncommon event, failure recovery will be called relatively infrequently.

In MemGuard, the checkpointing frequency is lower than that of error checking frequency as there is at least one error checking before checkpointing. There can be multiple memory error checking between two consecutive checkpoints to detect errors timely. Otherwise, the system rollback overhead is high as error detection is delayed. In case of error, the system is rolled back to the most recent checkpoint. If the error still exists, the OS can improve checkpointing and integrity checking frequency to exactly capture errors in a shortened period.

3.6. Integrity-Check Optimization and Other Discussions

The integrity_check step in Algorithm 1 requires to scan the entire memory space allocated to the program process and load any data block that is not present in the last-level cache. Although integrity_check period can be prolonged and it will not affect reliability significantly, the checking frequency may be limited by practical requirement, i.e. error detection is required before each checkpointing. In such cases, integrity_check can introduce visible overhead if the allocated memory space is significantly large. We thus further propose *lazy-scan* or also called *touched-only* scan scheme to reduce the scan overhead. The *lazy-scan* scheme only fetches pages that have been touched during an integrity checking period instead of all the pages allocated to the process. Typically, the required memory space is allocated at very beginning while merely part of them is touched during a period. Therefore, a *lazy-scan* may effectively reduce the memory traffic. Current processors already provide the information to the OS.

In order to guarantee that all the allocated memory pages will be added into READHASH for hash comparison, the hashes of all the untouched pages are still required. We thus propose to maintain a 128-bit (16-byte) sum hash for each memory page. In integrity_check step, hashes for untouched pages are added into READHASH directly and hashes for touched pages are calculated based on the fetched data blocks from main memory. A 128-bit hash can be applied for each page or a group of pages to reduce the cost with penalty of possibly increased touched page size. The *lazy-scan* scheme has an additional advantage that it only detects errors in program correction related pages and the untouched pages are assumed to be correct as they are not read from main memory in the current period. This can reduce program recovery rate as the OS may allocate a large memory space for a given application but the actually used memory size can be small. For example, program 434.zeusmp from SPEC CPU2006 is allocated with 1,131MB of memory but only touches 502MB during its running time (See Section 5 for the detail).

In practice, MemGuard can be used in combination with SECDED ECC as single-bit errors are mostly common. In this case, all single-bit errors can be corrected by SECDED and it reduces the overhead for checkpointing rollback recovery caused by single-bit errors. High-performance computing servers can both adopt SECDED and MemGuard design to tolerate single-bit errors and detect multiple-bit errors. For consumer-level computers and mobile systems without ECC, MemGuard design can be employed to efficiently provide error protection.

Real Machine Configuration	
Processor	Intel Xeon E5520 Quadcore 2.26GHz
OS kernel	Linux 2.6.27.6-117.fc10.x86_64
Compiler	GCC 4.6.2
L1 caches (per core)	32KB Inst/32KB Data, 8-way, 64B line
L2 caches (per core)	256KB unified, 8-way, 64B line
L3 cache (shared)	8MB, 8-way, 64B line
Memory	DDR3-1066 2DIMMs with 2GB/DIMM

Marss Simulator Configuration	
Processor	1 ooo core, 4GHz,14-stage pipeline
Functional units	2 IntALU, 4 LSU, 2 FPALU
IQ, ROB and LSQ	IQ 32, ROB 128, LQ 48, SQ 44
Physical registers	128 Int, 128 FP, 128 BR, 128 ST
L1 caches (per core)	64KB Inst/64KB Data, 2-way, 16B line
L2 cache (shared)	8MB, 8-way, 64B line
Memory	4GB, 200 cycles latency

Table 1: Major configuration parameters.

4. Experimental Methodology

We build a memory request generator to generate synthetic traces of memory write and read for reliability evaluation. A configurable error injector is built to inject specific errors into the memory traces to evaluate the error detection capability of the proposed MemGuard scheme and conventional SECDED ECC design. SECDED ECC is implemented following the reference design RD1025 [47] from Lattice Semiconductor and SpookyHash function is implemented using open source code [24]. To evaluate SECDED ECC, we inject specific error types into each memory request and use SECDED error checking to detect any error. The experiment is repeated by 1 billion times and then the error detection ratio is reported. For MemGuard, we generate 1 and 10 billion memory requests and inject specific errors to evaluate its error detection probability. The experiment is repeated for 100 times and the average error detection ratio is reported.

In order to evaluate system performance overhead introduced by MemGuard design, particularly from the integrity_check step, we run all 26 compilable benchmarks from SPEC CPU2006 suite [48] with different input sets to the completion on a real machine. The machine uses an Intel Xeon E5520 2.26GHz processor of 8MB last level cache and 4GB main memory. The detailed configuration is described in Table 1. We follow the study [14] to collect virtual and physical memory size for the total 51 benchmark-input sets to estimate memory scan overhead.

In addition, we use a full system simulator Marss-x86 [38] to further study the introduced memory traffic by MemGuard. Marss is an x86-64 architecture based cycle-accurate simulator and its detailed configuration is also listed in Table 1. We run the 26 benchmark programs on Marss for 10 billion instructions to collect memory traces for the baseline machine and then estimate the extra memory traffic incurred by MemGuard.

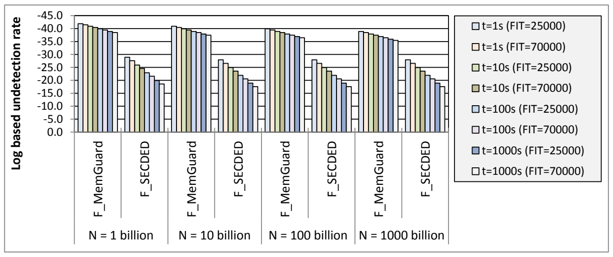


Figure 2: Error detection failure rate comparison of MemGuard and SECDED. The higher the bar the lower the error un-detectable rate.

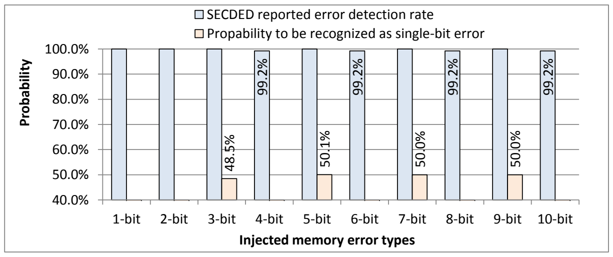


Figure 3: SECDED error protection capability. Note that although SECDED can fully report that error occurs for odd-bit (>1) errors, it might mistakenly recognize them as single-bit errors and correct them incorrectly.

5. Analytical and Experimental Results

5.1. Reliability Study

We study error detection capability of MemGuard design and SECDED ECC from two aspects. The analytical results are based on the error model and mathematical deduction presented in Section 3.3, while the simulated results are based on synthetic memory traces and error injection.

5.1.1. Analytical Results of Error Detection Rate. We calculate the error detection capability of MemGuard and SECDED design following the analytical model discussed in Section 3.3 for 25,000 and 70,000 FIT, respectively. In the calculation, we vary the time period t from 1 to 1,000 seconds and number of accesses N from 1 to 1,000 billion. Figure 2 presents the results. The y-axis is inverse log based to make the figure readable; the higher the bar, the lower the undetected error rate.

In all those cases, MemGuard is orders of magnitude stronger than the conventional SECDED ECC in error detection. The reason is that the strong hash function presents a low collision rate of $\frac{1}{2^{128}}$. With the number of memory accesses in a checking period grows, the error detection capability decreases slightly for increased chance of collision. However, the error detection rate is mainly decided by collision rate of the selected hash and the decrease is gradually attenuated. Therefore, the error checking frequency of MemGuard design can be prolonged almost arbitrarily with very limited loss of reliability. As SECDED ECC protection detects error in each data block, its error detection rate almost remains constant in this case.

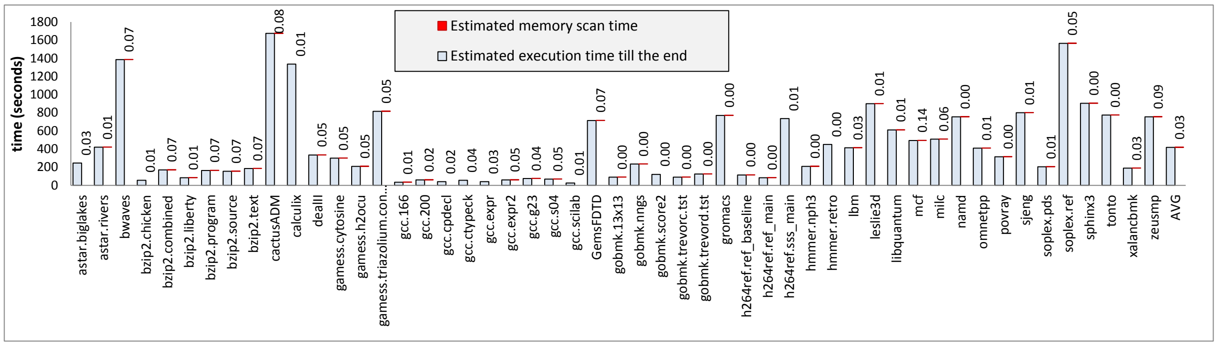
As the time period t grows from 1 to 1,000 seconds, the error rate for each bit in a data block grows, which increases multi-bit (>2) error rate. The error detection capability of SECDED ECC is reduced significantly. However, in MemGuard, it employs strong hash function of which the collision rate is irrelevant with number of flips in a data block. As the error rate for each data block grows, the error detection capability reduces slightly on MemGuard since the number of tampered data blocks increases and so does the possibility of collision. However, the rate of decrease is lower than that of SECDED ECC.

Although we do not have the detailed analysis for Chipkill Correct, we believe that MemGuard is stronger than it. Chipkill Correct can detect (and correct) multiple errors from a same device but not those from difference devices, while MemGuard can work in both cases.

5.1.2. Simulated Error Detection Rate. For SECDED ECC, we inject 1 to 10 bits of errors to a 64-bit data block at random positions and apply SECDED ECC to detect the error. The experiment is repeated for a billion times to report error detection rate. Figure 3 presents the results. SECDED ECC can completely detect all single- and double-bit errors as it has a Hamming distance of 4. In addition, the implemented SECDED ECC can detect all odd-bit errors similar to parity checking, which counts the total number of ‘0’s or ‘1’s in the data. Any odd-bit flip will modify the even/odd parity bit which can be detected. In theory, SECDED code cannot correct any errors with more than one bits flipped. The particular SECDED ECC implementation reports if any error has occurred. If so, it reports whether the error is single-bit or double-bit, no matter how many bits are flipped in the data block. For odd-bit (except 1-bit) errors, they will be mistakenly recognized as double-bit errors with a 50% chance and the other 50% chance to be mistakenly recognized as single-bit errors and corrected by SECDED ECC incorrectly. In addition, SECDED ECC can not fully detect other even number of bits errors. The error detection rate is around 99.2% and all of them are mistakenly recognized as double-bit errors.

For MemGuard, we inject one, two, three, four, five, ten, 100 and 1000 errors into 1 billion and 10 billion memory requests, separately. We group errors into six categories: single-, double-, triple-, quad-, multi- (randomly generated > 4) bit and mixed (of these five) errors. We do not present the results as MemGuard yields 100% error detection rate across the experiments. In reality, the probability of error detection is not 100% but is too high for a false negative to be observed in our experiments. A previous study has shown that SpookyHash is strong in collision resistance, passing through 2^{72} or roughly 10^{21} key pair tests [24].

The high reliability of MemGuard in memory error detection comes from the applied hash function. As illustrated, MemGuard with SpookyHash is stronger than the conventional SECDED error protection. A hash function with higher collision resistance may further improve its reliability. Given that Chipkill Correct may fail when multi-bit error rate is high,


(a) SPEC CPU2006 total execution time with memory scan overhead.

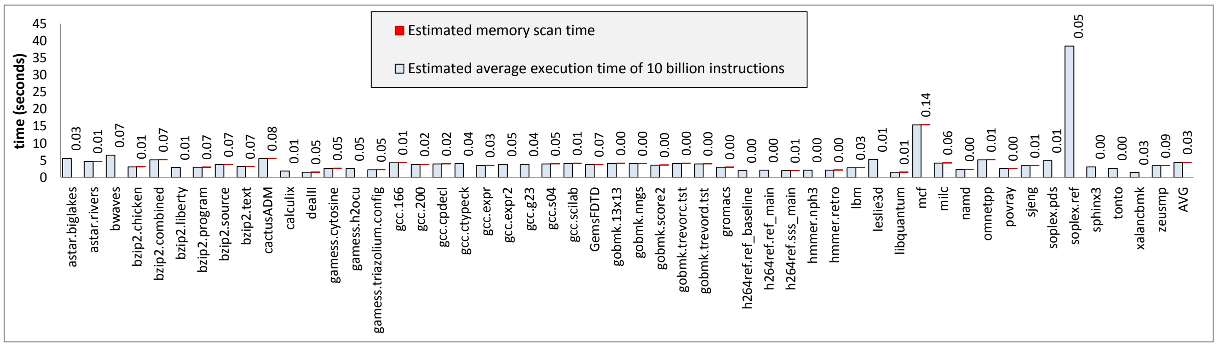

(b) SPEC CPU2006 execution time for 10 billion instructions with memory scan overhead.

Figure 4: Memory integrity-checking overhead of SPEC CPU2006 benchmark-input sets.

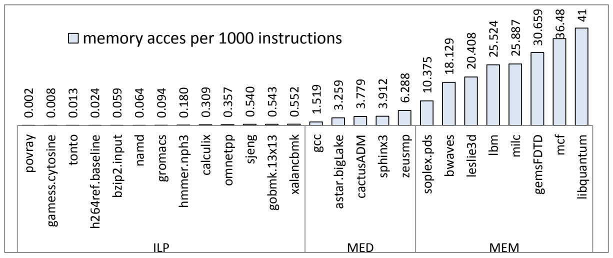


Figure 5: SPEC 2006 memory traffic characterizations.

MemGuard design can be stronger than Chipkill Correct in error detection, and it is designed to be irrelevant to number of flips in a data block.

5.2. System Performance Study

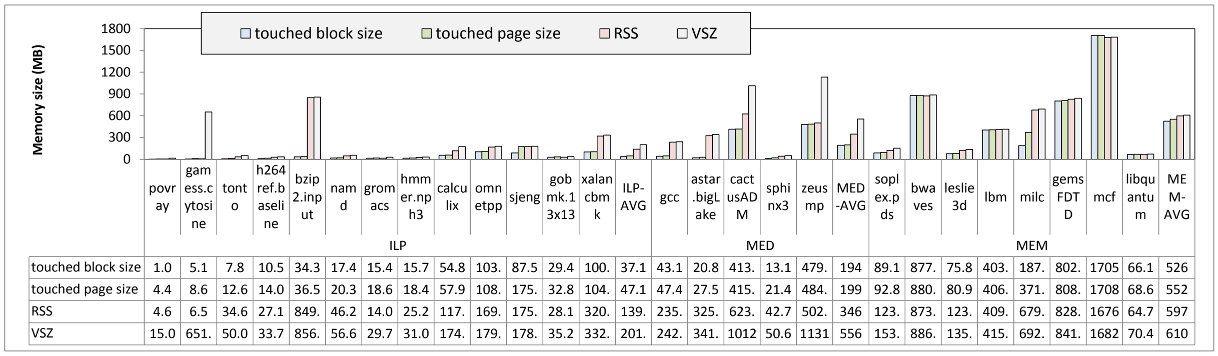
In the MemGuard design, the major performance overhead is from scanning the allocated memory space during integrity check. We thus estimate the memory scan overhead and compare it to the program execution time. We run the executables of the selected benchmark programs on a real machine to obtain the program execution time and collect the allocated virtual memory space, following the method of a previous study [14], to estimate the total memory scan latency. We assume that reading a 64-byte memory block takes 5ns on DDR3-1600 DRAM memory.

Figure 4 presents the results. Assume that memory integrity checking is executed solely at the end of a program, the scan overhead is minimal. As illustrated in Figure 4a, the program execution time is orders of magnitude higher than the memory scan time. On average, the program execution time is 418 seconds, while the memory scan takes merely 0.03 second. The average memory footprint is 372MB. In practice, the low integrity checking frequency can increase checkpointing recovery delay as errors detection is postponed. We conduct further analysis, assuming that the checking is activated every 10 billion instructions. Figure 4b presents the results. For those benchmark programs, the average execution time is 4.35 seconds for 10 billion instructions, and the memory scan overhead is approximately 0.7% of the execution time. Therefore, as long as the integrity checking rate is greater than 10 billion instructions, the system performance overhead is negligible.

In the case that an application program has a large memory footprint and integrity checking frequency is required to be high, *lazy-scan* scheme can help reduce the scan overhead by only scanning the pages touched in the current time period. The result will be presented in the next section.

5.3. Memory Traffic Overhead

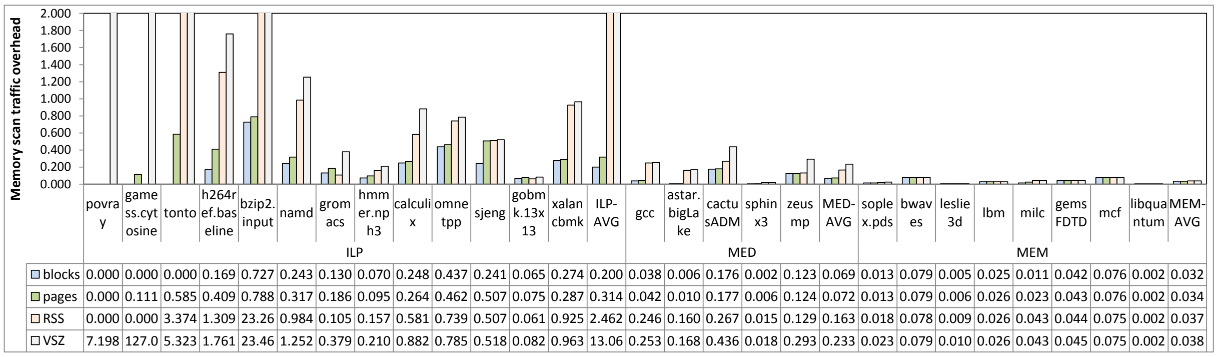
We assume that the memory error checking frequency is every 10 billion instructions to study the memory traffic overhead and the effect of the proposed *lazy-scan* scheme. We first use the Marss-x86 simulator to characterize memory traffic of each SPEC 2006 benchmark without error protection. We group the benchmark programs into three categories, ILP (computation-intensive), MED (medium) and MEM (memory-intensive),



	povray	gamess.cytosine	tonto	h264ref.baseline	bzip2.input	namd	gromacs	hmmer.nph3	calculix	omnetpp	sjeng	gobmk.13x13	xalancbmk	ILP-AVG
	ILP													
touched block size	1.0	5.1	7.8	10.5	34.3	17.4	15.4	15.7	54.8	103.	87.5	29.4	100.	37.1
touched page size	4.4	8.6	12.6	14.0	36.5	20.3	18.6	18.4	57.9	108.	175.	32.8	104.	47.1
RSS	4.6	6.5	34.6	27.1	849.	46.2	14.0	25.2	117.	169.	175.	28.1	320.	139.
VSZ	15.0	651.	50.0	33.7	856.	56.6	29.7	31.0	174.	179.	178.	35.2	332.	201.

	gcc	astar.bigLake	cactusADM	sphinx3	zeusmp	MED-AVG	soplex.pds	bwaves	leslie3d	lbm	milc	gemsFDTD	mcf	libquantum	MEM-AVG
	MED						MEM								
touched block size	43.1	20.8	413.	13.1	479.	194	89.1	877.	75.8	403.	187.	802.	1705	66.1	526
touched page size	47.4	27.5	415.	21.4	484.	199	92.8	880.	80.9	406.	371.	808.	1708	68.6	552
RSS	235.	325.	623.	42.7	502.	346	123.	873.	123.	409.	679.	828.	1676	64.7	597
VSZ	242.	341.	1012	50.6	1131	556	153.	886.	135.	415.	692.	841.	1682	70.4	610

Figure 6: Comparison of touched memory blocks, touched memory pages in 10 billion instructions and used physical memory space and allocated virtual memory space during a program's lifetime.



	povray	gamess.cytosine	tonto	h264ref.baseline	bzip2.input	namd	gromacs	hmmer.nph3	calculix	omnetpp	sjeng	gobmk.13x13	xalancbmk	ILP-AVG
	ILP													
🟦 blocks	0.000	0.000	0.000	0.169	0.727	0.243	0.130	0.070	0.248	0.437	0.241	0.065	0.274	0.200
🟩 pages	0.000	0.111	0.585	0.409	0.788	0.317	0.186	0.095	0.264	0.462	0.507	0.075	0.287	0.314
🟧 RSS	0.000	0.000	3.374	1.309	23.26	0.984	0.105	0.157	0.581	0.739	0.507	0.061	0.925	2.462
⬜ VSZ	7.198	127.0	5.323	1.761	23.46	1.252	0.379	0.210	0.882	0.785	0.518	0.082	0.963	13.06

	gcc	astar.bigLake	cactusADM	sphinx3	zeusmp	MED-AVG	soplex.pds	bwaves	leslie3d	lbm	milc	gemsFDTD	mcf	libquantum	MEM-AVG
	MED						MEM								
🟦 blocks	0.038	0.006	0.176	0.002	0.123	0.069	0.013	0.079	0.005	0.025	0.011	0.042	0.076	0.002	0.032
🟩 pages	0.042	0.010	0.177	0.006	0.124	0.072	0.013	0.079	0.006	0.026	0.023	0.043	0.076	0.002	0.034
🟧 RSS	0.246	0.160	0.267	0.015	0.129	0.163	0.018	0.078	0.009	0.026	0.043	0.044	0.075	0.002	0.037
⬜ VSZ	0.253	0.168	0.436	0.018	0.293	0.233	0.023	0.079	0.010	0.026	0.043	0.045	0.075	0.002	0.038

Figure 7: Memory traffic overhead introduced by MemGuard integrity checking.

based on their MPKI which we define as memory accesses per 1,000 instructions. The ILP benchmarks are those with MPKI less than 1.0, MEM with MPKI greater than 10.0 and MED with MPKI in between.

Figure 5 presents the MKPI numbers of those programs. Figure 6 compares the actually touched memory pages during the execution of 10 billion instructions, and the used physical memory space (RSS: resident set size) and the allocated memory space (VSZ: virtual memory size) during the entire program execution. The physical and virtual memory spaces are mostly consistent except a few exceptions, i.e. gamess, zeusmp, etc. For gamess with cytosine workload, it allocates 651 MB virtual memory and the actual physical memory usage is merely 6.5MB. In addition, across all the benchmarks, the actually touched pages are smaller than total allocated memory space. The reduction is from 201.9MB for VSZ to 47.1MB, from 555.8MB to 199.3MB and from 609.8MB to 552.1MB on average for ILP, MED and MEM programs, respectively. Therefore, *lazy-scan* may effectively reduce memory traffic overhead.

Figure 7 presents the ratio of memory traffic overhead from MemGuard memory scan. We limit the y-axis to less than 2.0 to make the figure readable, and therefore we also present the data in the table. For ILP workloads, MemGuard will introduce 13x memory traffic on average if the entire VSZ is scanned. By scanning only the RSS and the touched pages, the overhead is reduced to 2x and 31%, respectively. As those ILP workloads are computation intensive, the extra memory energy consumption is insignificant. For MEM workloads, the extra memory traffic is 3.8% and 3.7%, respectively, by scanning VSZ and RSS. The cost is further reduced to 3.4% if *lazy-scan* is applied. This cost is less than the conventional SECDED ECC, which has 12.5% memory storage and energy overhead. As memory scanning presents a good amount of page locality, the practical power consumption can be lower than traffic overhead. For MED workloads, the overhead is 23.3% by scanning VSZ and it can be reduced to 7.2% by *lazy-scan*. The overhead is also lower than the conventional SECDED ECC. Note that the reported overhead from MemGuard is based on the assumption that the memory scanning is done every 10 billion instructions. The checking period can be much longer than that, which will further reduce the overhead. The figure also compares the combined size of touched memory blocks with that of touched memory pages. On average, the former is about 65%, 95%, and 94% of the latter for ILP, MED, and MEM workloads, respectively.

6. Conclusion

We have presented MemGuard, a system-level memory error protection scheme without any memory storage overhead or constraint on memory organization. By maintaining and checking two registers WRITEHASH and READHASH inside memory controller, the scheme can effectively detect errors in a sequence of memory requests. A detailed analysis of reliability strength and selection of hash functions are presented.

The evaluation using mathematical deduction and synthetic simulation proves that MemGuard is more reliable than the conventional SECDED ECC design with lower energy overhead.

Acknowledgments

We like to thank the anonymous reviewers for their constructive comments. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1117604. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber, "Future scaling of processor-memory interfaces," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2009, pp. 1–12.

[2] A. Appleby, "Murmurhash," 2011. Available: https://sites.google.com/site/murmurhash/

[3] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[4] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: The case of hashing and signing," in *Proceedings of International Cryptology Conference (CRYPTO)*, vol. 839, 1994, pp. 216–233.

[5] L. Borucki, G. Schindlbeck, and C. Slayman, "Comparison of accelerated DRAM soft error rates measured at component and system level," in *Proceedings of IEEE International Reliability Physics Symposium (IRPS)*, 2008, pp. 482–487.

[6] D. Clarke, S. Devadas, M. V. Dijk, B. Gassend, and G. E. Suh, "Incremental multiset hash functions and their application to memory integrity checking," in *Asiacrypt Proceedings Advances in Cryptology*, vol. 2894, 2003, pp. 188–207.

[7] T. J. Dell, *A white paper on the benefits of chipkill-correct ECC for PC server main memory*, 1997.

[8] C. Estebanez, Y. Saez, G. Recio, and P. Isasi, "Performance of the most common non-cryptographic hash functions," in *Software: Practice and Experience*, 2013.

[9] K. Ferreire, J. Stearley, J. H. L. III, R. Oldfield, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 44:1–44:12.

[10] D. Fiala, F. Mueller, C. Engelmannand, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 78:1–78:12.

[11] G. Fowler, P. Vo, and L. C. Noll, "FNV hash," 1991. Available: http://www.isthe.com/chongo/tech/comp/fnv/

[12] Google, "Cityhash 1.1," 2010. Available: http://code.google.com/p/cityhash/

[13] R. Hamming, "Error correcting and error detection codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[14] J. Henning, "SPEC CPU2006 memory footprint," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 84–89, 2007.

[15] R. Housley, "A 224-bit one-way hash function: SHA-224," *RFC 3874*, Sep. 2004.

[16] M. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[17] A. A. Hwang, I. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design," in *Proceedings of International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 111–122.

[18] Hybrid Memory Cube Consortium, "Hybrid memory cube specification 1.0," 2013.

[19] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "McrEngine: A scalable checkpointing system using data-aware aggregation and compression," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 17:1–17:11.

[20] ITRS, "Process integration, device, and structures," *International technology roadmap for semiconductors*, 2011 Edition, 2011.

[21] ITRS, "International technology roadmap for semiconductors," 2012. Available: http://www.itrs.net/Links/2012ITRS/Home2012.htm

[22] K. Jarvinen, M. Tommiska, and J. Skytta, "Hardware implementation analysis of the MD5 hash algorithm," in *Proceedings of Annual Hawaii International Conference on System Sciences (HICSS)*, 2005, p. 298a.

[23] B. Jenkins, "Hash functions for hash table lookup," 2009. Available: http://www.burtleburtle.net/bob/hash/evahash.html

[24] B. Jenkins, "Spookyhash: a 128-bit noncryptographic hash," 2011. Available: http://www.burtleburtle.net/bob/hash/spooky.html

[25] A. H. Johnston, "Scaling and technology issues for soft error rates," in *Proceedings of Annual Conference on Reliability*, 2000.

[26] P. Jones, "US secure hash algorithm 1 (SHA1)," *RFC 3174*, Sep. 2001.

[27] K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for HPC," in *Proceedings of International Conference for Distributed Computing Systems (ICDCS)*, 2012, pp. 615–626.

[28] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2007.

[29] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 5, pp. 874–879, 1994.

[30] D. Locklear, "Chipkill correct memory architecture," 2000.

[31] "Reliability data sets," Los Alamos National Laboratory. Available: http://institutes.lanl.gov/data/fdata/

[32] S. Mathew, M. Anders, R. Krishnamurthy, and S. Borkar, "A 6.5GHz 54mW 64-bit parity-checking adder for 65nm fault-tolerant microprocessor execution cores," in *Proceedings of IEEE Symposium on VLSI Circuits*, 2007, pp. 46,47.

[33] "DDR3 SDRAM MT41J512M8 -32Meg x8 x8 banks," Micron Technology Inc., 2006.

[34] "Calculating memory system power for DDR3," Micron Technology Inc., 2007.

[35] S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, 2005, pp. 243–247.

[36] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2013, pp. 72–83.

[37] X. Ni, E. Meneses, N. Jain, and L. V. Kale, "ACR: automatic checkpoint/restart for soft and hard error protection," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013, pp. 7:1–7:12.

[38] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86 CPUs," in *Proceedings of Design Automation Conference (DAC)*, 2011, pp. 1050–1055.

[39] P. K. Pearson, "Fast hashing of variable-length text strings," *Communications of ACM*, vol. 33, no. 6, pp. 677–680, 1990.

[40] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," in *In proceedings of IRE*, vol. 49, 1961, pp. 228–235.

[41] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kinsley, "Memory exclusion: optimizing the performance of checkpointing systems," *Software: practice and experience*, vol. 29, pp. 125–142, 1999.

[42] R. Rivest, "The MD4 message-digest algorithm," *RFC 1320, MIT Laboratory for Computer Science and RSA Data Security, Inc.*, April 1992.

[43] R. Rivest, "The MD5 message-digest algorithm," *RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc.*, April 1992.

[44] J. C. Sancho, F. Petrini, G. Johnson, J. Fernandez, and E. Frachtenberg, "On the feasibility of incremental checkpointing for scientific computing," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2004, pp. 58–.

[45] A. Satoh and T. Inoue, "ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS," in *Proceedings of International Conference on Information Technology: Coding and Computing (ITCC)*, vol. 1, 2005, pp. 532–537.

[46] B. Schroeder, E. Pinheiro, and W.-D. Weber, “DRAM errors in the wild: A large-scale field study,” in *Proceedings of International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, vol. 37, 2009, pp. 193–204.

[47] L. Semiconductor, “ECC module,” *Reference Design 1025*, 2012.

[48] *SPEC CPU2006*, Standard Performance Evaluation Corporation, http://www.spec.org.

[49] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, “Efficient memory integrity verification and encryption for secure processors,” in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2003, pp. 339–350.

[50] A. N. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi, “LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems,” in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2012, pp. 285–296.

[51] F. A. Ware and C. Hampel, “Improving power and data efficiency with threaded memory modules,” in *Proceedings of International Conference on Computer Design (ICCD)*, 2006, pp. 417–424.

[52] D. H. Yoon and M. Erez, “Virtualized and flexible ECC for main memory,” in *Proceedings of International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 397–408.

[53] G. Zheng, X. Ni, and L. V. Kale, “A scalable double in-memory checkpoint and restart scheme towards exascale,” in *Proceedings of International Conference for Dependable System and Networks Workshops (DSN-W)*, 2012, pp. 1–6.

[54] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu, “Minirank: Adaptive DRAM architecture for improving memory power efficiency,” in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2008, pp. 210–221.