High Performance, Energy Efficient Chipkill Correct Memory with Multidimensional Parity

Xun Jian, John Sartori, Henry Duwe, Rakesh Kumar, *University of Illiinois at Urbana-Champaign*

Abstract—It is well-known that a significant fraction of server power is consumed in memory; this is especially the case for servers with chipkill correct memories. We propose a new chipkill correct memory organization that decouples correction of errors due to local faults that affect a single symbol in a word from correction of errors due to device-level faults that affect an entire column, sub-bank, or device. By using a combination of two codes that separately target these two fault modes, the proposed chipkill correct organization reduces code overhead by half as compared to conventional chipkill correct memories for the same rank size. Alternatively, this allows the rank size to be reduced by half while maintaining roughly the same total code overhead. Simulations using PARSEC and SPEC benchmarks show that, compared to a conventional double chipkill correct baseline, the proposed memory organization, by providing double chipkill correct at half the rank size, reduces power by up to 41%, 32% on average over a conventional baseline with the same chipkill correct strength and access granularity that relies on linear block codes alone, at only 1% additional code overhead.

1 Introduction

The Memory systems in data centers consume a large amount of power. As online services continue to proliferate, the total power consumption of the memory systems in data centers will continue to rise. This presents a strong need for innovations to lower memory power consumption.

Server memory systems require error protection as a standard feature to prevent the loss of sensitive customer data and increase server availability. Chipkill correct is an advanced type of error correction in memory that corrects errors due to a complete device failure. Large scale studies show that chipkill correct reduces the uncorrectable error rate by 10X [4] to 36X [10] compared to Single Error Correction, Double Error Detection (SECDED).

Despite their higher reliability, conventional chipkill correct memories are power hungry, since they require a large number of DRAM devices per memory access to keep code overhead low. In this paper, we reduce the power consumption of memory systems that implement chipkill correct by separately targeting local faults that affect a single symbol in a word and device-level faults that affect an entire column, sub-bank, or device. We observe that device-level faults cause a large number of errors and therefore can be easily localized with little overhead. Once localized, errors due to device-level faults can be corrected using erasure codes, which only incur half as much overhead as the error correction codes used in conventional chipkill correct memories. We also observe that a local fault only affects a single word at a time; therefore a single unit of error correction resources per 100s of words (to correct only a single corrupted word out of this many words) can correct errors due to such faults with high coverage. By decoupling the correction of errors due to local faults from correction of errors due to device-level faults, the proposed chipkill correct memory with multidimensional parity checksums, or *MPCMem*, can reduce the number of devices required per memory access by half while maintaining the same overall code overhead as conventional chipkill correct memories, which dramatically reduces the dynamic power consumption of memory.

. Manuscript submitted: 18-Mar-2012. Manuscript accepted:1-Jun-2012. Final manuscript received: 12-Jun-2012

2 Background

The access granularity of the memory system has to match the cacheline size of the last level cache. A group of DRAM devices, called a rank, is required to operate in parallel to satisfy this access granularity. A rank is divided into multiple independent *banks*, where each bank consists of a physical *sub-bank* in each device. A limited number of ranks can share the same memory bus *channel*, with it's own dedicated memory controller (*MC*).

Conventional chipkill correct memories are protected by linear block codes, where data are divided into independent blocks, or codewords. Each codeword consists of data and check symbols, where a symbol is a group of adjacent bits [5]. By storing each symbol of a word in a separate device in the rank, the lost symbol can be reconstructed when a device fails. Using one of the most efficient linear block codes, such as the Reed-Solomon code, a linear block code with R check symbols can detect up to R corrupted symbols or correct up to only R/2 corrupted symbols. However, if the position of the corrupted symbols are known through some other means, the linear block code that detects R corrupted symbols can also correct up to R corrupted symbols; a linear block code used in this fashion is called an erasure code. Unlike conventional chipkill correct memories that use linear block codes to both detect and correct all errors, MPCMem uses erasure codes to correct against device-level faults (see Section 3.3).

The check symbols in a chipkill correct memory are stored in redundant devices in the rank. Since the number of redundant devices per rank for a given strength of chipkill correct is constant regardless of the number of data devices in the rank, reducing the number of data devices per rank increases the overall code overhead. On the other hand, reducing the number of devices per rank decreases dynamic access power. Our goal is to reduce the rank size of chipkill correct memory by half without affecting its strength and access granularity, while maintaining roughly the same code overhead.

Posted to the IEEE & CSDL on 6/28/2012
DOI 10.1109/L-CA.2012.21 1556-6056/13/$31.00 © 2013 Published by the IEEE Computer Society

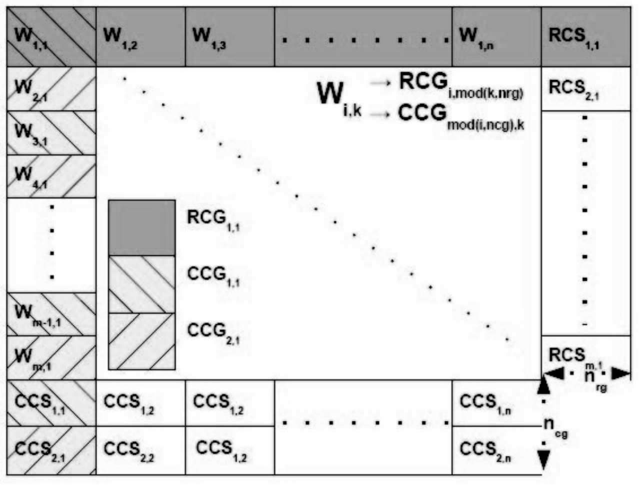

Fig. 1. Example configuration of MPCMem with 1 checksum group per row and 2 per column.

3 MPCMem

3.1 Encoding

Figure 1 shows the logical view of a memory bank in MPCMem with a total of m rows and n columns. $W_{i,k}$ represents a codeword located at the i^{th} row and k^{th} column in memory. A codeword in MPCMem is created using an erasure code, which allows the codeword to detect errors but not correct them by itself. Just as the symbols of a codeword in a conventional chipkill correct organization, each symbol of a codeword in MPCMem is stored in a separate device in the rank to guarantee error detection in the event of device failure(s). To keep the overhead of the codewords the same as that of SECDED (11.1%), in our double chipkill correct configuration, *MPCMemD*, a codeword consists of 18 symbols, where 2 of these are check symbols, which translates to 16 data devices and 2 redundant devices per rank. To incur the same code overhead in the single chipkill correct implementation of MPCMem, *MPCMemS*, a codeword in MPCMemS consists of 9 bits, where 8 bits contain data and the 9th bit is an odd parity bit for the 8 data bits, which leads to a memory rank consisting of 8 data devices and 1 redundant device.

Every codeword in MPCMem belongs to a row checksum group (*RCG*) and a column checksum group (*CCG*). A RCG/CCG is comprised of a set of codewords in the same row/column as well as a row checksum (*RCS*)/column checksum (*CCS*). The RCS and CCS are collectively known as multidimensional parity checksums, or *MPC*s. To keep the overhead of the MPCs low, we assign 256 codewords to each checksum group, which results in an additional overhead of $(1/256) \cdot 2 = 0.78\%$. There are n_{rg} RCGs per row and n_{cg} CCGs per column. A codeword in the i^{th} row and k^{th} column belongs to the x^{th} RCG in the i^{th} row and belongs to the y^{th} CCG in the k^{th} column, where $x = mod(k, n_{rg})$ and $y = mod(i, n_{cg})$. This mapping rule maximizes the physical distance between the members of the same checksum group in order to mitigate the impact of multi-bit cluster faults by spreading each faulty cell across different checksum groups [7]. $RCG_{1,1}$, $CCG_{1,1}$, and $CCG_{2,1}$ in figure 1 are examples of checksum groups when n_{rg}=1 RCG n_{cg}=2. A checksum is the bitwise odd parity of the codewords in the checksum group. Let's take for example a hypothetical checksum group (*CG*) with 2 codewords, where each codeword consists of 4 2-bit symbols. If the codewords are 11′01′10′00 and 01′11′11′10′ (where ′ separates the symbols), then the checksum is 01′01′10′01.

3.2 Correction of Errors due to Local Faults

Error detection in MPCMem is performed on-the-fly with each memory access through the linear block code. When a corrupted codeword is detected, the remaining codewords in the corrupted codeword's RCG are read out and checked for error. These codewords are bitwise xored together as they arrive. If no errors are detected in these codewords or the RCS of the RCG, then the correct codeword can be reconstructed from the xor result and the RCS using bitwise odd parity. During this process, if another codeword or the RCS is discovered to be corrupt, the RCG cannot correct the codeword and the procedure must be repeated using the CCG instead. If a checksum is corrupt, it can be corrected through its CG after any corrupted codewords in its CG have been corrected.

3.3 Correction of Errors due to Device-Level Faults

If an error cannot be corrected by either the RCG or the CCG, the underlying fault may be a device-level fault. In order to use the erasure codewords to correct errors due to device-level faults, the faults must be localized first. MPCMem takes the divide-and-conquer approach of first localizing individual column faults. Once multiple column faults are localized to the same sub-bank, a sub-bank fault is identified. Experimental results from [10], [11] suggest that the ratio of column faults to complete device faults is well below 10 to 1, which can therefore serve as the threshold for merging multiple faulty columns into a single faulty sub-bank. Similarly, a device fault can be identified when multiple sub-bank faults are localized to the same device. MPCMem does not localize row faults for two reasons. First, due to the fact that each codeword belongs to both a RCG and CCG, every codeword in a faulty row can be corrected through its CCG. Second, modern operating systems log all occurrences of errors. Since memory pages are mapped to rows, not columns, the OS can migrate a page that suffers frequent errors (which suggest a row fault) to a different row in memory.

The memory controller then attempts column fault localization. Since each device is fixed to a symbol position, a mismatched checksum symbol corresponding to a particular symbol position is used to locate which device contains the incorrect codeword symbol(s). To localize a faulty column, a large number of CCGs along the column containing the problematic CCG are read out and their checksums are computed and then compared against the CCSs stored in memory. When the number of checksum symbol mismatch corresponding to a device exceeds a threshold, the memory controller localizes the faulty column to that particular device. We observe that in the event of a column fault due to a malfunctioning read datapath, a massive number of errors along the problem column appears immediately, which causes half of the CCGs along the column to contain an odd number of corrupted codewords; these CCGs are able to report symbol mismatches. Assuming that a total of 128 CCGs are read out, a threshold value of 20 is both low enough to render a false negative probability of

$4.29 \cdot 10^{-16}$ yet high enough to rule out the possibility of false positives due to chance alignments of many local faults along the same column. In the event of a column fault due to a malfunctioning write datapath that result in a gradual accumulation of errors that may not immediately exceed the threshold, we can increase the number of errors by artificially injecting a write to every CCG in the column.

We observe that if the CCG read out is only performed when the RCG is unable to correct a local fault, a column fault may remain undetected for a long time until one of the affected cells in the column becomes uncorrectable by the RCG due to chance alignment with a new fault in the same row. This can lead to a long period of performance degradation where every time an affected cell in the column is accessed, an expensive RCG read out is performed. As a result, instead of always performing RCG read out first and CCG second only when the former fails to correct a corrupted codeword, we propose performing CCG read out first and RCG second for $1/10^{th}$ of local fault corrections. When the CCG is read out first, the memory controller counts the total number of corrupted codewords in the CCG. Once a threshold is exceeded, a complete column fault localization procedure is carried out. Taking into consideration the possible alignment of 2 local faults in a single 256-word CCG, we decided on the threshold value of 3.

Once a column fault in a sub-bank has been localized, its address range is stored in an erasure history table. After the number of faulty columns in a sub-bank of a device exceeds a threshold, the address range of the sub-bank is stored in the erasure history table and corresponding column fault entries are cleared. Similarly, when all the sub-banks in a device are faulty, the address range of the device is stored in the erasure history table. After a fault has been localized, errors due to the fault can be corrected using the standard syndrome correction for erasure codes. Errors due to the alignment of a device-level fault and local fault affecting the same codeword can be corrected using a combination of erasure correction for device-level faults and CG correction for the local fault. Note that after a device-level fault has been remembered in the erasure history table, correcting the errors due to the fault can be done on-the-fly via the erasure codewords.

3.4 Error Correction Overhead

Compared to conventional chipkill correct memories, where all error detection and correction operations happen on-the-fly, MPCMem operates on-the-fly only for correction of errors due to device-level faults and detection of all errors. A CG read out operation is required for each error correction of a local fault and up to 128 CG read out operations are required to localize a device-level fault, where each CG readout operation require accessing 256 codewords. Our calculations show that the average latency of error correction for a local fault is $600ns$ for a DDR2 memory with $667MHz$ I/O frequency; this is similar to that of a page migration. The latency of fault localization is $2ms$, which is close to that of a disk access. To estimate the overall latency overhead on the memory system, we need to consider the frequency of these operations. We used data reported in [4], which records the frequency of access to faulty words in memory to estimate the frequency of error correction of local faults. Since a device-level fault needs to be identified only once, the frequency of fault localization can be approximated by the device-level fault rate reported in [10]. Due to the fact that the devices examined in the report were only 1 to 2 years old, to be conservative, we increased the reported fault rate by 3 orders of magnitude. The combined latency overhead of error correction and faulty localization is in the order of $10^{-8}\%/1GB$ of memory.

The memory controller needs to access its erasure history table prior to each memory access to differentiate errors due to local faults from those due to device-level faults. We estimate that a total of 256 entries is sufficient for the erasure history table. Memory controllers that reports an excess of 256 device-faults should have the appropriate ranks in its channel replaced. Since the access to the erasure history table takes place in parallel with the access to memory, it does not increase memory access latency. Due to the fact that access to the erasure history table is on-chip, its energy per access is negligible compared to that of accessing DRAM.

3.5 Physical Implementation Details

We observe that the software management technique and the minor modifications to the CPU proposed in [6] for storing the error correction (not error detection) check symbols of linear block codewords in the data region of memory can be easily generalized to managing arbitrary type of error correction resources via software, such as the MPCs. Note that only linear block codes are used in [6]; although the software-managed error correction reduces the number of redundant devices per rank, thereby improving the effective I/O pin usage of the CPU, it does not reduce the code overhead of chipkill correct since it does not modify the error correction code itself.

While software-managed MPCMem can be implemented using off-the-shelf DRAM devices, we also suggest hardware-managed MPCMem as a viable alternative, since the former induces additional memory traffic to update the MPCs. In hardware-managed MPCMem, the MPCs are to be updated within the DRAM devices themselves, without generating additional memory traffic. Since MPCs uses the simple bit-wise odd parity, they can be updated using an array of primitive xor gates. The checksums are stored in dedicated parity sub-banks corresponding to each sub-bank of a DRAM device. Decoders internal to the DRAM device translate the open row in the data bank as well as the column address of an access command into the appropriate location in the parity banks. The access latency of the parity banks is negligible compared to the data banks since the parity banks are only 1% the size of the data banks. Dual modular redundancy can be applied to the parity sub-banks to provide error detection for the checksum symbols. Each device is only responsible for storing the checksum symbols that are computed using the codeword symbols stored in the device. Due to limited space, we cannot provide every detail of the hardware-managed MPCMem; a reader familiar with the internal structure of DRAM devices can easily fill in the details of an efficient and low overhead implementation.

4 Methodology

M5, a full system simulator [8], was integrated with DRAMsim [9] for memory timing and power modeling. Table 1 describes the CPU microarchitecture, while Table 2 describes

TABLE 1
Processor Microarchitecture

SS Width	IQ Size	Phys Regs	LSQ Size	L1 D$, I$
2	16	72FP/72INT	32LQ	32 kB
L1 lat.	L2$	Assoc	lat.	line size
1 cycle	1MB	16	10 cycles	64b

TABLE 2
Memory Configurations. R. stands for rank.

Name	Tech	I/O	Chan	R./Chan	R. Size
SCC	DDR3	X4	2	1	18
MPCMemS	DDR3	X8	2	2	9
DCC	DDR2	X4	1	1	36
MPCMemD	DDR2	X8	1	2	18

memory configurations for MPCMem and the corresponding baselines. We used the open page and high performance mapping policies in DRAMsim. To provide reduced rank size, both MPCMemD and MPCMemS use devices with twice the I/O width as their respective baselines. To ensure that the power benefits of the MPCMems evaluated are strictly due to their reduced rank size, we kept everything else constant, including the total usable memory capacity, bus width, as well the technology type and density of the DRAM devices. The parameters for the DRAM devices are taken from Micron [1], [2].

We evaluated the hardware-managed implementation of MPCMem. The impact of the write overhead on latency and power are modeled by increasing the write current (IDD4W) by read current (IDD4 - 3P) [9]. The time delay is modeled by extending the Write Latency (WL) by twice the normal number of burst cycles per access (t_burst). Since the checksums take up around 1% of total memory capacity, we increased the total power of memory by 1% to take into account the power consumption of the parity banks. Our test workloads include 21 single-core and 27 quad-core multi-programmed workloads from a selection of 24 SPEC2000 and SPEC2006 benchmarks, as well as 8 quad-core PARSEC benchmarks.

5 Experimental Results

Figure 2 presents the power reduction and IPC improvement of hardware-managed MPCMemD and MPCMemS vs DCC and SCC baselines. On average, MPCMemD reduces power by 31.8%, while MPCMemS reduces power by 17.6%. The longer write access latency is hidden by having twice as many ranks per channel (see Table 2), which allows multiple memory accesses to be served concurrently. This explains the small performance improvements in the figure. Software-management of error correction resources has been thoroughly evaluated in [6]. Despite requiring additional write accesses to memory, a rank of software-managed chipkill correct memory with 18 devices has an average of 27.2% power reduction and only 1% IPC degradation [6] compared to a rank of conventional memory with 36 devices.

We also considered the reliability impact of MPCMem. Since MPCMem enables chipkill correct with half as many devices per rank, it improves reliability compared to the baseline. Just as the baselines, MPCMem uses linear block codes to detect errors and to correct errors due to device-level faults. However, MPCMem uses MPCs to correct errors due to local faults. To evaluate the effectiveness of MPCMem at correcting local faults, we developed analytical and Monte Carlo models to determine the MTTF to first uncorrectable local fault and found it to be with 5% of that of conventional chipkill correct. MPCs offers strong reliability

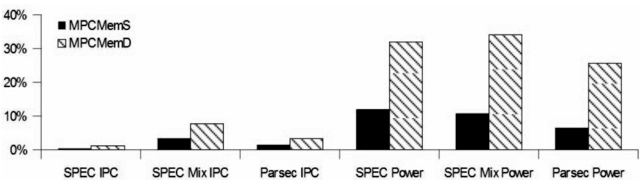

Fig. 2. IPC and power improvement over baselines.

against local faults because (A) the increased utilization of the available error correction resources through sharing, and (B) each codeword belongs to two checksum groups.

6 Related Work

The most recent work that reduces the rank size of chipkill correct memory to reduce power is LOT-ECC [3], which aims to provide single chipkill correct double chipkill detect for the particular rank size of 9 at 21% code overhead. For the same code overhead ($2/(8+2)+1\% = 21\%$), MPCMemD configured with 8 data devices and 2 redundant devices can correct faults in an additional device. Under LOT-ECC, after a device fails, **every** read access to memory requires a second access to obtain error correction resources. In addition, LOT-ECC requires that device-level faults manifest as stuck at 0 or 1 faults [3]. However, many device-level faults do not manifest in this manner. Examples include device-level faults due to a bad write datapath that lead to a gradation accumulation of errors and row faults due to a bad row address decoder causing the wrong row of a device always getting accessed.

7 Conclusions

Conventional chipkill memories require a large rank size to keep code overhead low. MPCMem decouples correction of local faults from that of device-level faults by correcting them with codes that are individually optimized for each type of faults. As a result, MPCMem can reduce rank size by half while maintaining the same access granularity and chipkill correct strength as any given conventional chipkill correct memory, at the cost of 1% additional code overhead. MPCMemD achieves up to 41% power reduction, 32% on average, compared to conventional double chipkill correct memories that relies on linear block codes alone.

References

[1] ”1Gb: x4, x8, x16 DDR2 SDRAM,” MICRON,2007.
[2] ”1Gb: x4, x8, x16 DDR3 SDRAM,” MICRON,2007.
[3] A.N. Udipi et al., ”LOT-ECC: LOcalized and Tiered Reliability Mechanisms for Commodity Memory Systems”, ISCA, 2012.
[4] B. Schroeder and E. Pinheiro and W.D. Weber, ”DRAM errors in the wild: a large-scale field study,” SIGMETRICS, 2009.
[5] C.L. Chen and M.Y. Hsiao, ”Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review,” IBM Journal of Research and Development, vol.28, no.2, 1984.
[6] D.H. Yoon and M. Erez, ”Virtualized and flexible ECC for main memory,” ASPLOS, 2010.
[7] J. Kim et al., ”Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding,” Micro 2007.
[8] N.L. Binkert et al., ”The M5 Simulator: Modeling Networked Systems,” Micro, 2006.
[9] ”University of Maryland Memory System Simulator Manual,” University of Maryland.
[10] V. Sridharan and D. Liberty, ”Field Study of DRAM Errors,” SELSE, 2012.
[11] X. Li and M. Huang and K. Shen and L. Chu, ”A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility,” A technical report sponsored by the NSF, 2007.