Designing a Fast and Adaptive Error Correction Scheme for Increasing the Lifetime of Phase Change Memories

Rudrajit Datta and Nur A. Touba

Computer Engineering Research Center
University of Texas at Austin, Austin, TX 78712
rudrajit.datta@mail.utexas.edu, touba@ece.utexas.edu

Abstract

This paper proposes an adaptive multi-bit error correcting code for phase change memories that provides a manifold increase in the lifetime of phase change memories thereby making them a more viable alternative for DRAM main memory. A novel aspect of the proposed approach is that the error correction code (ECC) is adapted over time as the number of failed cells in the phase change memory accumulates. The operating system (OS) monitors the number of errors corrected on a memory line, and when the number of errors on a line begins to exceed the strength of the ECC present, the ECC strength is adaptively increased. As this happens, the performance of the memory system gracefully degrades because more storage is taken up by check bits rather than data bits thereby reducing the effective size of a cache line since less data can be brought to the cache on each read operation to the PCM main memory. Experimental results show that the lifetime of a phase change memory can be significantly extended while keeping the fraction of data to check bits as high as possible at each stage in the lifetime of the phase change memory.

1. Introduction

Memory technology scaling drives increasing density, increasing capacity, and falling price-capability ratios. Storage mechanisms in prevalent memory technologies require inherently un-scalable charge placement and control. This in turn has put memory scaling, a first-order technology objective, in jeopardy. Dynamic Random Access Memory (DRAM) has been used as the main memory in computer systems for decades due to its high-density, high-performance and low-cost. However, DRAM technologies, facing both scalability and power issues, will be difficult to scale down beyond 50nm [Zhang 09] due to various limitations associated with device leakages and retention time.

Phase change memory (PCM) provides a non-volatile storage mechanism amenable to process scaling. Phase change memories function by alternating between low resistance crystalline and high resistance amorphous states. The thermally induced phase transition is brought about by injecting current into the storage material during writes. The state of the cell is then detected during reads with the high resistance state being interpreted as a zero and the low resistance state as one. PCM, relying on analog current and thermal effects, does not require control over discrete electrons. As technologies scale and heating contact areas shrink, programming current will also scale linearly. PCM scaling mechanism has been demonstrated in a 20nm device prototype and is projected to scale to 9nm [Lee 09]. As a scalable DRAM alternative, PCM could provide a clear roadmap for increasing main memory density and capacity.

However, one major challenge that needs to be addressed for PCM is its limited write endurance. PCM writes induce thermal expansion and contraction within the storage element, degrading injection contacts and limiting endurance to hundreds of millions of writes per cell at current processes. In current devices, a PCM cell typically supports around 10^7 writes [Ferreira 10]. Thus, PCM will wear-out quickly if used as a main memory.

This is a significant limitation and a prime reason why PCM is not yet a ready substitute for DRAM main memory. Current PCM prototypes are not designed to mitigate PCM endurance. One major challenge in designing ECC for PCM based systems is that the number of cell failures is a monotonically increasing function of memory writes. In order to mitigate the time-dependant nature of failures, this paper proposes a novel adaptive error correction technique that increases PCM endurance several times. The core idea here is to start with a nominal ECC, depending on experimentally determined error rates for PCM, and then adaptively boost the ECC strength to keep up with increasing failure rates of PCM. The dynamic control over the ECC is achieved by involving the underlying operating system (OS). The OS monitors the maximum number of errors corrected per PCM line and compares this against the strength of the ECC currently in place. When number of errors corrected on a memory line read approaches the capacity of the existing error code, the ECC strength is increased. This can be done on the next reboot or done by writing main memory to disk and reconfiguring the ECC when it is paged back in. The increase in ECC strength is achieved by breaking up a memory line into segments and then implementing separate ECC for each segment. Taken together, the combined effect of the segmented ECCs can correct up to tens of bits per memory line.

978-1-61284-656-9/11/$26.00 ©2011 IEEE

Note that as the ECC of the memory is increased, the performance of the memory system gracefully degrades because more storage is taken up by check bits rather than data bits. However, this strategy is much better than using a worst-case ECC which would give worst-case performance throughout the lifetime of the system. The degradation comes in the form of reducing the effective size of a cache line since less data can be brought to the cache on each read operation to the PCM main memory. Note that if the cache itself is implemented with more reliable SRAM, then the number of check bits stored in the cache does not need to be increased. So the total cache capacity remains the same. If the cache line size is reduced, then the cache can store more lines. Strategies for designing the cache to accommodate graceful degradation of the line size is discussed as well as the performance impact which is highly dependent on the number and locality of memory references for an application.

2. Related Work

Various approaches have been adopted to counter the limited write endurance of phase change memories. [Zhang 09] presents a hybrid PRAM/DRAM memory architecture that uses an OS level paging scheme to improve PRAM write performance and lifetime. They use a 7-error correcting BCH code for the ECC. BCH codes provide the desired level of reliability but require increasing number of cycles for correcting multi-bit errors [Lin 83]. [Xu 10] proposes a novel sensing mechanism for multi-level PCM structures to address the reliability issue using either BCH or LDPC codes, for both of which the decoding time scale with number of errors being detected.

[Ferreira 10] shows how PCM writes can be minimized thereby increasing their lifetime. Note that this methodology could be used on top of the methodology proposed in this paper. [Lee 09] uses buffer reorganization and partial write techniques to mitigate high energy PCM writes but improves PCM lifetime to only about 5.6 years.

Traditional schemes like using spare rows and columns as well as bit interleaving, as shown in [Stapper 92], are likely to prove insufficient because of the prohibitively high error rate in PCM systems.

While the PCM reliability issue has primarily been addressed from an architecture standpoint, solutions using novel ECC have yet to be fully explored. PCM differs from standard DRAM in a fundamental way in that the number of failures for a PCM cell is a function of time or more accurately a function of the number of writes/cell. The following section presents a detailed overview of the proposed scheme.

3. Overview of Proposed Approach

Given that bit failure rates for phase change memories increase with continuous usage, the proposed approach of adaptively increasing the strength of the ECC to keep up with the increasing failure rate is a good strategy. Note that this strategy requires involving the operating system (OS).

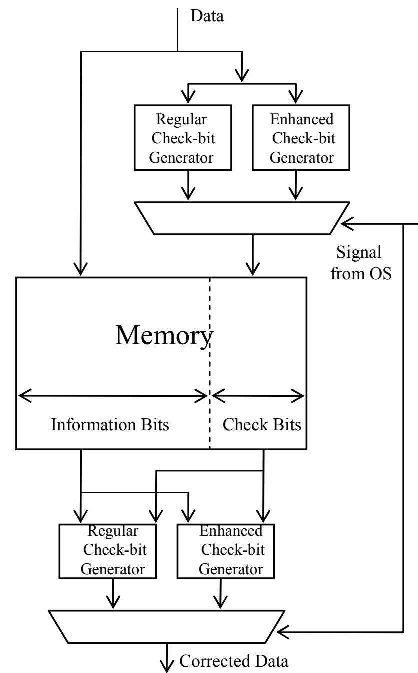

Figure 1. Adaptive ECC implementation

ECC decoding and correction is performed by the memory controller after the data and check bits have been read from the memory chip and are in the processor core. The memory controller can be programmed as to what granularity of ECC is to be implemented. In the proposed method, every time the memory controller reads a memory line, the number of errors corrected is compared against a threshold which depends on the existing strength of the ECC in place. When the number of errors corrected for a particular read begins to approach the given threshold for the implemented ECC, the OS switches to a stronger ECC. This is accomplished by writing all of the main memory to disk and paging the memory back into physical memory with the stronger ECC. Note that this process of reconfiguring the ECC occurs very infrequently (on the order of months or years). Another way to implement it would be to record the need for an ECC reconfiguration and then perform it on the next reboot.

For the memory controller to be able to switch to an ECC of greater strength, the hardware has to be in place from before. Depending on up to how many levels the ECC will be stepped up, the full hardware to perform the necessary encoding-decoding will have to be implemented in the memory controller from the beginning. The memory controller will then choose which of the existing encoding-decoding schemes to employ using information received from the OS.

Figure 1 shows a possible implementation of the scheme that can switch between two modes of ECC. The OS signals when the memory controller needs to switch from the basic ECC scheme to the advanced scheme. In the general case where several different levels of ECC hardware are implemented, the OS will signal the memory controller when to step up the strength of ECC.

Strengthening the ECC requires increasing the redundancy of the code, which means more check bits and fewer data bits can be stored in each line of the memory. When a cache miss occurs, fewer data words can be brought into the cache in each main memory access. For example, if a line in the memory initially contained 1024 data bits (i.e., 16 64-bit words), but then if the redundancy of the ECC is increased by 25%, then each line would only have 768 data bits (i.e., 12 64-bit words).

Note that if the cache is implemented with more reliable SRAM rather than PCM technology, then it is not necessary to store as many check bits in the cache as is needed in a PCM main memory, nor is it necessary to scale up the number of check bits in the cache over time. So as the number of check bits stored in the phase change main memory is increased over time, the total storage capacity of the cache is not affected. What is affected is the bandwidth coming into the cache. However, the reduction in system performance would be very application dependent and would depend on the locality of the data and number of required memory accesses. Experimental results are shown in Sec. 6 exploring the impact on performance.

There are a number of different options for how the cache is implemented to accommodate a gracefully degrading line size coming from the main memory.

One would be to have multiple valid bits for each original full size line in the cache. As the effective bandwidth is reduced when the ECC is strengthened in the phase change main memory, then each read from the main memory will only partially fill a line in the cache which would be indicated with the appropriate subset of valid bits for the line. If for example, the number of data bits was reduced by 50%, then each memory access would fill half of the cache line. Another way to think about it is that a cache with 1024 lines where each line originally stored 16 words would be effectively transformed to a cache with 2048 lines where each line stores only 8 words. The cache capacity doesn't change, only the effective line size changes.

Another way to implement the cache would be to adjust the associativity when the bandwidth from the PCM main memory is reduced. For example, a two-way set associative cache could be converted to a four-way set associative cache when the line size is reduced in half.

In either of these cases, the impact of reducing the line size will depend on the locality and frequency of memory references in the application. The reduction in line size is partially offset by the increase in either the associativity or the number of lines.

This gives an overview of the approach which is general and could be used for any type of ECC. Next, one way for implementing the ECC with this scheme will be proposed which utilizes OLS codes. The advantage of using OLS codes over traditional multi-bit ECC such as BCH, LDPC codes, is that correction can be performed in a single clock cycle and the amount of time required is independent of the number of errors.

4. Orthogonal Latin Square Codes

OLS codes, as the name suggest, are based on Latin squares. A Latin square [Hsiao 70] of order (size) m is an m x m square array of the digits 0, 1, . . . , m - ***1,*** with each row and column a permutation of the digits 0,1, … , m - 1. Two Latin squares are orthogonal if, when one Latin square is superimposed on the other, every ordered pair of elements appears only once.

As explained in [Datta 10], a t-error correcting majority decodable code works on the principle that $2t + 1$ copies of each information bit are generated from $2t + 1$ independent sources. One copy is the bit itself received from memory or any transmitting device. The other $2t$ copies are generated from $2t$ parity relations involving the bit. By choosing a set of h Latin squares that are pair-wise orthogonal, one can construct a parity check matrix such that the number of 1's in each column is $2t = h + 2$. The orthogonality condition ensures that for any bit $\boldsymbol{d}$, there exists a set of $2t$ parity check equations orthogonal on d_i, and thus makes the code self-orthogonal and one-step majority decodable. One-step majority decoding is the fastest parallel decoding method. The t-error correcting codes generated by OLS codes [Hsiao 70] have m^2 data bits and $2tm$ check bits per word.

5. Adaptive Error Correction Code

The principle behind the proposed method is that as the number of permanent errors keeps increasing over time, the ECC needs to be increased in strength. The trade off comes in the form of using up more of the memory for storing the ECC check bits.

If for k data bits in a memory line, a t-error correcting code is used, then this is sufficient for all errors $\leq t$. To mitigate more than t errors, more check bits are required. They cannot be added without reducing the number of data bits per line because for off-chip main memory, the n-bit bus width transferring data and check bits from the memory to the processor is fixed such that

$$n = \textit{data bits} + \textit{check bits}$$

A straightforward approach for strengthening the ECC for an OLS code would be to simply directly increase t for the whole line. However, rather than doing that, it is more efficient to divide the line into fragments and increase the number of errors corrected in each fragment as will be shown in this section.

If an n-bit line consists of k data bits, it can be broken up into fragments each of size k_i, and r_i bits of ECC are separately implemented for each data fragment k_i such that

$$\sum_i (k_i + r_i) = n$$

Consider the case where all k bits have a t-error correction OLS code implemented on it. Then the total number of bits, data plus check bits, would be

$$k + 2t\sqrt{k} \quad \ldots\ldots\ldots\ldots\ldots\ldots(5)$$

Now if the line is broken up into fragments, each of size k_i = k/f and a $t/\sqrt{f}$ -error correcting OLS code is implemented for each fragment. The total number of bits still remains

$$\{k/f + 2(t/\sqrt{f})\sqrt{k/f}\} * f = k + 2t\sqrt{k} \quad \ldots(6)$$

Although the total number of bits is the same in both the cases, (5) and (6), the error correction capacity is different for both. In case (5), the line can withstand all error patterns affecting up to t-bits. In case (6), each fragment can handle up to $t/\sqrt{f}$ errors. But overall the line can handle all error patterns affecting up to $t/\sqrt{f}$ bits in each fragment and *some* error patterns affecting up to $t\sqrt{f}$ bits on the entire line. As is shown later in section 6, for randomly occurring errors, the property to correct some error patterns of size greater than the individual capacity of the ECC in each fragment is significant and as simulations show, a fair number of error patterns can be tolerated using this property.

So the overall idea is the following. An initial code over across all n bits is selected to protect the PCM memory based on characterization tests. Then during the course of operation as the number of failed cells accumulates over time, the strength of the ECC is increased by implementing ECC on increasingly smaller fragments.

Consider a numerical example to illustrate the scheme. Consider a memory line with 256 data bits. Initially a 3-error correcting OLS code is employed. Thus the total number of bits in the line is,

$$256 + 2 * 3 * \sqrt{256} = 352$$

In an enhanced ECC mode, 25% of the memory line is used to store extra check bits. Hence the total number of data bits per line now becomes 192. The rest

$$352 - 192 = 160$$

bits are used for storing ECC. But instead of implementing ECC over the entire 192 data bits at a time, the line is broken up into fragments of size 64, 64, 16, 16, 16 and 16 bits. Next a 3-error correcting OLS code is implemented on each of the 64-bit fragments and a 2-error correcting OLS code on each of the 16-bit fragments, bringing the total number of bits to

$$(64 + 2 * 3 * 8) * 2 + (16 + 2 * 2 * 4) * 4 = 352$$

But now instead of being able to correct only 3-error patterns all 2-error patterns, 99.97% of all 3-error patterns, 99.73% of all 4-error patterns and so on, up to a small fraction of 14-bit errors can be corrected. Thus the approach of breaking up a line into fragments and using separate ECC for each fragment is more efficient in terms of error correcting capacity than implementing a single ECC on the whole line.

The selection of fragment sizes and their respective ECC bits is a combinatorial problem. In the cases where there is more than one possible way to break up a memory line into identical division of data and check bits, the combination which can correct maximum number of errors is chosen.

6. Experimental Results

The bit error rate for a memory is defined as the number of failed bits divided by the total size of the memory (or in other words, the probability that each bit has failed). The bit error rate for PCM memories starts very small and grows over time as more cells fail. Figure 3 compares the proposed adaptive error correction scheme with the approach in [Zhang 09] where a 7-error BCH code is used and can tolerate bit error rates of up to 0.145%. The adaptive scheme discussed in this paper was implemented with a line size of 1024 and starts with an initial correction capability of 3-errors. As the OS detects that the bit error rate is exceeding the strength of the ECC, 25% of the data bits are converted to check bits by dividing the memory into fragments and adding check bits for each fragment. This process is repeated as the error rate continues to increase. As can be seen in Fig. 3, the proposed scheme can tolerate bit-error rates from 0.08% to a significantly higher 1.2%.

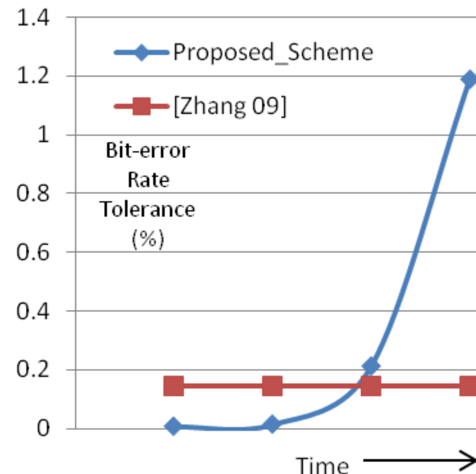

Figure 3. Adaptive fault tolerance

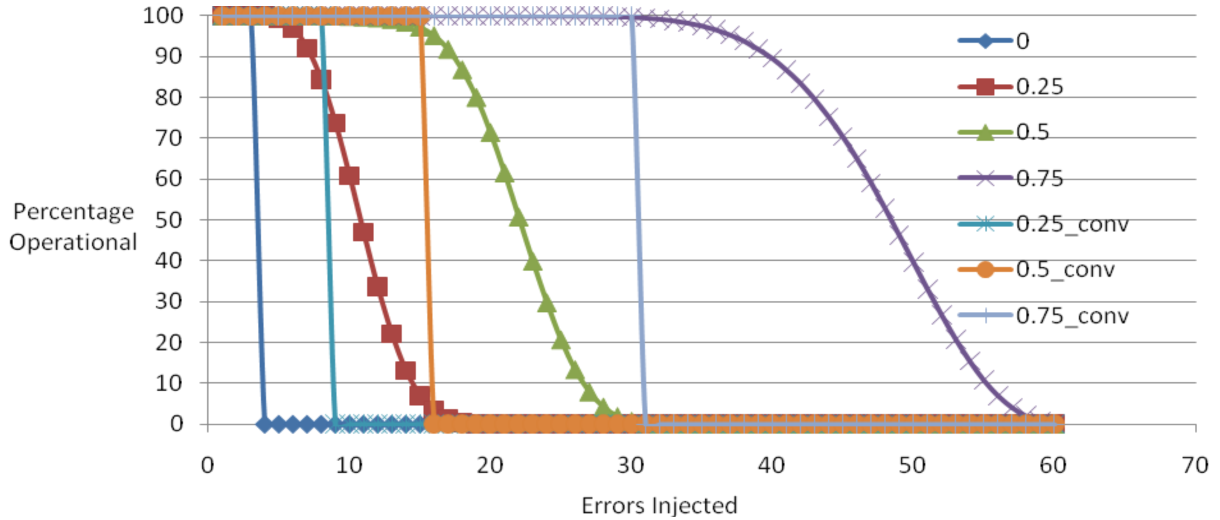

Figure 4. Percentage of operational cache lines versus number of errors injected (out of 100,000 experiments)

Table 1 shows how the error tolerance varies with size of the memory as the number of check bits is increased. As can be seen from the data, very high error rates of around 10^{-2} can be tolerated in an adaptive manner.

Table 1. Error Tolerance (no. of errors / no. of bits * 100) for varying line sizes

Memory Size	Fraction of Memory Used for Storing Extra Check-bits			
	0.0	0.25	0.5	0.75
128MB	0.008	0.015	0.213	1.190
256MB	0.006	0.042	0.205	1.117
1GB	0.005	0.026	0.154	0.989
4GB	0.003	0.020	0.125	0.916

The monotonic decrease in tolerance with increasing size can be explained by the fact that as the number of lines increase, the difference

$$n * E[one\ line] \sim E[n\ lines]$$

is likely to increase, where n is the number of lines, *E[one line]* is expected error tolerance of a single line and *E[n lines]* is the error tolerance of n lines. This mirrors a likely scenario where errors will accumulate faster on some lines than others unless the data read/write pattern is absolutely random, which is not the usual case due to data locality.

Another aspect of evaluating the proposed scheme is to study the distribution of error tolerance in each line for different ECC configurations. Figure 4 shows results for a 1024 bit line in which the initial stating point is a 3-bit error correcting ECC and then 25% of the data bits were converted to check bits, and then 50%, and finally 75%. Errors were injected at random, and Fig. 4 shows the percentage of lines that have not failed across 100,000 experiments. The x-axis corresponds to the number of errors injected in the line, and the y-axis corresponds to the percentage of lines that were able to tolerate that many errors for different configurations of the ECC. Results are shown for both the ECC scheme described in Sec. 5 which breaks up the line into fragments and implements ECC separately for each fragment, and the conventional case where the ECC was implemented across all the data bits at once. As can be seen from the results, the fragmented ECC scheme can easily tolerate more errors than the conventional method. The former is able to tolerate 20% more errors per line at 90% probability, than the conventional method, when half the line is used for storing check bits. When $3/4^{th}$ of the line is used for storing check bits, the fragmented scheme can tolerate 33% more errors at 90% probability.

Figures 5 and 6 show the effect of reducing memory line size to accommodate extra check bits. One way to implement this, as was described in Sec. 3, is to adjust the associativity and line size of the cache. In both Figs. 5 and 6, the y-axis plots cycles per instruction (CPI) for a set of SPEC2006 [Spec 06] benchmarks across different cache configurations defined as follows:

cache_config1 – line size 512B, associativity 4
cache_config2 – line size 256B, associativity 8
cache_config3 – line size 128B, associativity 16

The CPI was calculated assuming single cycle for all non-memory instructions and five cycles [Wulf 95] for instructions that caused a cache miss and needed to access main memory. As can be seen from the figures, there is little degradation on performance for reduced line sizes and increasing associativity. Moreover, the fact that these changes are expected at an interval of every few years lessens the performance impact.

The simulations were done using Pin [Pin 04], a dynamic instrumentation tool. Pin was used to obtain memory traces of the benchmarks for various cache configurations. These memory traces were then used as an input to the DineroIV [Dinero IV] cache simulator to generate cache miss rates.

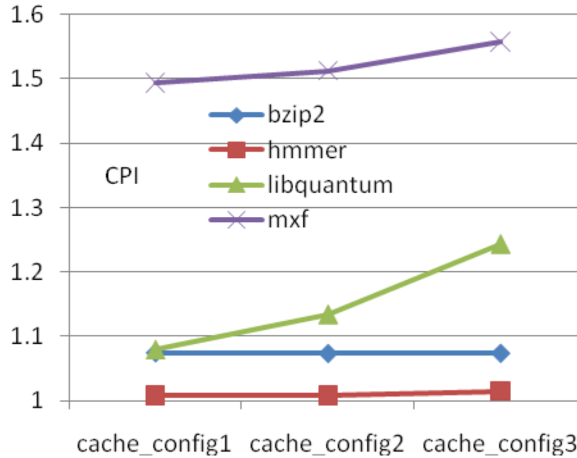

Figure 5. Variation of CPI for different cache configurations for four different SPEC2006 benchmarks for 64KB cache

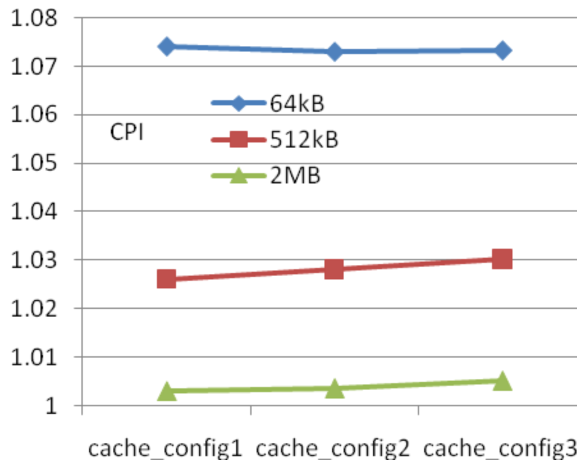

Figure 6. Variation of CPI for different cache configurations across different cache sizes for the SPEC2006 benchmark *bzip2*

7. Conclusions

This paper described an adaptive error tolerance scheme that can extend to 8x more error tolerance than that of [Zhang 09] for similar initial redundancy. As the PCM memory degrades to the point where it exceeds the capability of the method in [Zhang 09] to continue operation, the proposed method can continue operation by adaptively increasing the ECC. The performance of the memory will gracefully degrade due to reducing the effective line size for each memory read to service a cache miss. However, the impact of this can be minimized by careful cache design since the total cache storage capacity as a whole is not impacted.

Acknowledgments

This research was supported in part by the National Science Foundation under Grant No. CCR-0426608.

References

[Datta 10] R. Datta, N. Touba, "Post-Manufacturing ECC Customization Based on Orthogonal Latin Square Codes and Its Application to Ultra-Low Power Caches", *Proc. of International Test Conference*, Paper 7.2, 2010.

[Dinero IV] J. Edler, M. D. Hill, "DineroIV – Trace Driven Uniprocessor Cache simulator for memory references," http://www.cs.wisc.edu/~markhill/DineroIV/

[Ferreira 10] A.P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, D. Mosse, "Increasing PCM Main Memory Lifetime", *Design Automation and Test in Europe*, pp. 914-919, 2010,.

[Hsiao 70] M.Y. Hsiao, D.C. Borren, R.T. Chien, "Orthogonal Latin Square Codes", *IBM Journal of Research and Development,* Vol. 14, No. 4, pp. 390-394, July 1970.

[Lee 09] B. C. Lee, E. Ipek, O. Mutlu, D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative", *Proc. of International Symposium of Computer Architecture*, pp. 2-13, 2009.

[Lin 83] S. Lin, D. Costello, *Error Control Coding: Fundamentals and Applications*. Prentice Hall, 1983.

[Pin 04] V.J.Reddi, A. Settle, D.A.Connors, R.S.Cohen, "PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education", *Proc. of Workshop on Computer Architecture Education*, June 2004.

[Spec 06] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. http://www.spec.org/cpu2006/.

[Stapper 92] C.H. Stapper, Hsing-san Lee, "Synergistic Fault-Tolerance for Memory Chips", *Proc of IEEE Transactions on Computers*, Vol. 41, No. 9, pp 1078-1087, Sep. 1992.

[Wulf 95] W.A. Wulf, S.A. McKee, "Hitting the memory wall: implications of the obvious", *ACM SIGARCH Computer Architecture News*, Vol. 23 Issue 1, Mar.1995.

[Xu 10] W. Xu, T. Zhang, "Using Time-Aware Memory Sensing to Address Resistance Drift Issue in Multi-Level Phase Change Memory", *Proc. of International Symposium of Quality Electronic Design*, pp. 356-361, 2010.

[Zhang 09] W. Zhang, T. Li, "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures", *Int. Conference on Parallel Architectures and Compiler Techniques*, pp. 101-112, 2009.