DrMP: Mixed Precision-aware DRAM for High Performance Approximate and Precise Computing

Xianwei Zhang[+], Youtao Zhang[+], Bruce R. Childers[+] and Jun Yang [*]
University of Pittsburgh
[+]{xianeizhang, zhangyt, childers}@cs.pitt.edu, [*]juy9@pitt.edu

***Abstract*—Recent studies showed that DRAM restore time degrades as technology scales, which imposes large performance and energy overheads. This problem, *prolonged restore time (PRT)*, has been identified by the DRAM industry as one of three major scaling challenges.**

This paper proposes DrMP, a novel fine-grained precision-aware DRAM restore scheduling approach, to mitigate PRT. The approach exploits process variations (PVs) within and across DRAM rows to save data with mixed precision. The paper describes three variants of the approach: DrMP-A, DrMP-P, and DrMP-U. DrMP-A supports approximate computing by mapping important data bits to fast row segments to reduce restore time for improved performance at a low application error rate. DrMP-P pairs memory rows together to reduce the average restore time for precise computing. DrMP-U combines DrMP-A and DrMP-P to better trade performance, energy consumption, and computation precision. Our experimental results show that, on average, DrMP achieves 20% performance improvement and 15% energy reduction over a precision-oblivious baseline. Further, DrMP achieves an error rate less than 1% at the application level for a suite of benchmarks, including applications that exhibit unacceptable error rates under simple approximation that does not differentiate the importance of different bits.

I. Introduction

Today's chip-multiprocessors, with an increasing number of cores, demand large-capacity main memory for scalable performance. For more than four decades, DRAM has been the *de facto* choice for main memory. JEDEC's standard specifies the timing values for DRAM device operations [16]. Servicing either a read or write request requires a restore operation to charge cells in a DRAM row to a target voltage for reliable access.

DRAM faces severe scaling challenges at 20nm and below [15], [38], [37]. Scaling DRAM leads to smaller cells that are slower and leakier [13], [40] with larger process variations [30], [3]. As a result, it takes more time to charge the cells to a target voltage due to (1) increased resistance of the bitlines and the transistor-capacitor contact, and (2) decreased drivability of smaller transistors [37]. This phenomenon, termed *prolonged restore time (PRT)*, has been identified as one of the three most critical impediments to DRAM scaling [19], [8]. This paper addresses PRT and its impact on performance.

Existing timing values are in jeopardy under extreme technology scaling. Since scaling leads to large process variation (PV), future DRAM modules will have so many weak cells that feasible repair schemes cannot mask all [40], [52], [41]. Consequently, staying with the current standard for timing will harm chip yield and increase manufacturing cost dramatically. Relaxing (i.e., increasing) timing helps maintain yield and supports tight profit margins. Unfortunately, timing relaxation introduces significant performance degradation [19]. While the performance loss can be effectively mitigated with recent schemes [58], [60], PRT remains a major performance bottleneck.

Recently, approximation for DRAM has been proposed to improve energy consumption by trading off computation accuracy [27], [28]. By exploiting the intrinsic error resilience of many modern applications, a DRAM sub-system can save approximate data, while still achieving satisfactory computational results. Existing works on DRAM, e.g., *Flikker* [27], focus on refresh energy reduction, which unfortunately has limited impact on improving memory access latency [44], [28]. Consequently, these schemes cannot mitigate the large performance degradation due to PRT. In addition, the benefits of these schemes can only be realized through error-resilient applications, which greatly limits their applicability for general-purpose computation.

This paper proposes a fine-grained precision-aware restore scheduling technique, DrMP, that aggressively reduces restore time to achieve high performance. DrMP is a suite of progressively capable techniques to support approximate, precise and hybrid approximate-precise computing. We summarize our contributions as follow:

- We propose DrMP-A to achieve high-performance approximate computing. By exploiting the variance in restore timing exhibited at different row segments of a DRAM row, DrMP-A reduces the restore time such that only two or four row segments are fully reliable. By mapping the important data bits of different data types to the reliable row segments, errors are avoided in critical bit positions to keep application-level errors low. Using this scheme, application performance is improved for approximate computing.
- We propose DrMP-P to achieve high-performance precise computing. DrMP-P stitches together fast, reliable row segments from a pair of rows. One row of the pair has a fast restore time, while the other row has a slow restore time. Both rows are fully reliable. DrMP-P reduces the average restore time to improve memory performance with minimal architectural overhead.
  We then propose DrMP-U that integrates DrMP-A and DrMP-P to support both approximate and precise computing. DrMP-U constructs two fast physical rows from a row pair — one fast row for storing precise data and one fast row for storing approximate data.
- We evaluate and compare our proposed schemes to the

state-of-the-art. Our experimental results show that, on average, DrMP achieves 20% performance improvement, 15% energy reduction and below 1% application-level error rate over the precision-oblivious baseline.

II. Background

A. DRAM Restore Time Degradation

Although technology scaling has long been the impetus behind building memory with ever-more capacity, there are several threats to scaling at nano-scales [15]. A recent industry study by Samsung identified three major threats to scaling: prolonged restore time (PRT), frequent refresh, and variable retention time (VRT) [19].

DRAM *restore* is the device operation required to charge memory cells after normal accesses. A restore operation is needed for both read and write accesses. For read, the *restore time* is the duration required after the read command (`RD`) and before issuing a precharge command (`PRE`) to service the next request. For write, the restore time is the time required after receiving the data sent with the write command (`WR`) and before `PRE`. The above timing for write is the cell recovery time (*tWR*) specified in the JEDEC standard [16] [1].

Figure 1 shows the tWR [2] distribution of memory cells in one memory module at different DRAM technology nodes. The figure illustrates two scaling effects on restore time. First, the mean value becomes larger. Due to larger contact resistance, bit-line resistance, and the on-current of the transistor, it takes an increasingly longer time to charge the capacitor to the target voltage level [19], [58]. Second, the cells from one module show increasingly larger process variations due to the difficulty of precisely controlling the nano-scale geometry parameters, such as gate length and capacitor size [30], [3].

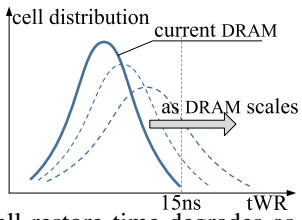


Fig. 1: The cell restore time degrades as DRAM scales.

The current cell recovery time value, tWR, is 15ns, which has been kept across several generations [48]. This value represents a trade-off between system performance and manufacturing yield and cost. However, as DRAM is further scaled, a larger number of cells in future chips will have longer recovery time. It quickly becomes infeasible to rescue these slow cells (using a reasonable ECC scheme or number of spares) to meet timing specifications. Instead, tWR has to be relaxed to maintain high chip yield and low per bit manufacturing cost, causing significant performance degradation and/or serious yield loss [58], [59].

[1]The JEDEC standard specifies many other timing relationships as well. In this paper, we also refer to tRAS and tRCD. *tRAS* is the required time that a row has to be active to ensure reliable access. *tRCD* is the time required between sending a row address and sending a column address.

[2]For ease of illustration, we mainly discuss tWR. In the paper, the effects on reads (tRAS) and writes (tWR) are both fully considered and evaluated.

To mitigate the performance degradation from PRT, Kang *et al.* proposed the co-design of a memory controller and in-DRAM ECC to address the scaling challenge [19]. Zhang *et al.* utilized DRAM organization to construct fast restore regions [58], [60]. Zhang *et al.* proposed to truncate restore operations on basis of the distance to refresh operation [59]. DrMP expands the design space with approximate computing, which are orthogonal to past approaches for mitigating PRT.

In addition to PRT, memory refresh is expected to introduce performance and energy overhead in scaled DRAM [37], [4]. Recently, many schemes have been proposed to address this issue [11], [39], [55], [26]. DRAM scaling also leads to variable retention time (VRT), which demands profiling with strong ECC and a large guardband [20], [43]. Several schemes have been described to reduce timing constraints for improved memory performance [23], [51], [9], [50], [5], [24]. Our design is aligned in this direction with the goal to address the PRT challenge.

Model of scaling in this paper. To study the restoring degradation, we generated DRAM chips using a scaling model in [58], [60]. The model is similar to others in the literature [22], [61], [26], [3]. It considers scaling effects on restore time at both circuit and architecture levels. We verified the model's distribution of weak cells (with slow restore time) and the impact on timing are correctly aligned to studies involving real devices [6], [19] as well as predicted relaxed timing from recent industry patents [1], [2].

B. Approximate Computing

Approximate computing is an emerging paradigm that exploits the inherent error resilience of many modern applications where a small number of hardware and software errors have little impact on the quality of program output [27], [46]. For these applications, user data are often categorized as either *critical* or *non-critical*. While non-critical data has error tolerance, critical data must be protected to ensure correctness.

For main memory, most approximate computing approaches focus on DRAM refresh energy. Flikker partitions data into critical and non-critical groups [27]. It uses different refresh rates for the groups to save energy. Raha *et al.* further divide memory pages into quality bins with different refresh and error rates, and enable quality-aware data allocation to the bins [44]. Sparkk refreshes different chips in a DIMM with different rates to reduce refresh power [28]. Refresh can also be disabled completely for a subset of dedicated applications with negligible impact [17]. Jung *et al.* describe an approximate DRAM simulation framework [18]. Compared to refresh operations, the degradation of restore time with scaling slows down both read and write operations, leading to more significant performance degradation.

In addition, approximate computing caches have been proposed to improve effective cache capacity and achieve energy savings [34], [35], [36]. For non-volatile memory, such as Phase Change Memory (PCM), reduced write precision can improve device lifetime and performance [47]. A progressive transform codec can be used to maximize storage density with minimized image distortion [12].

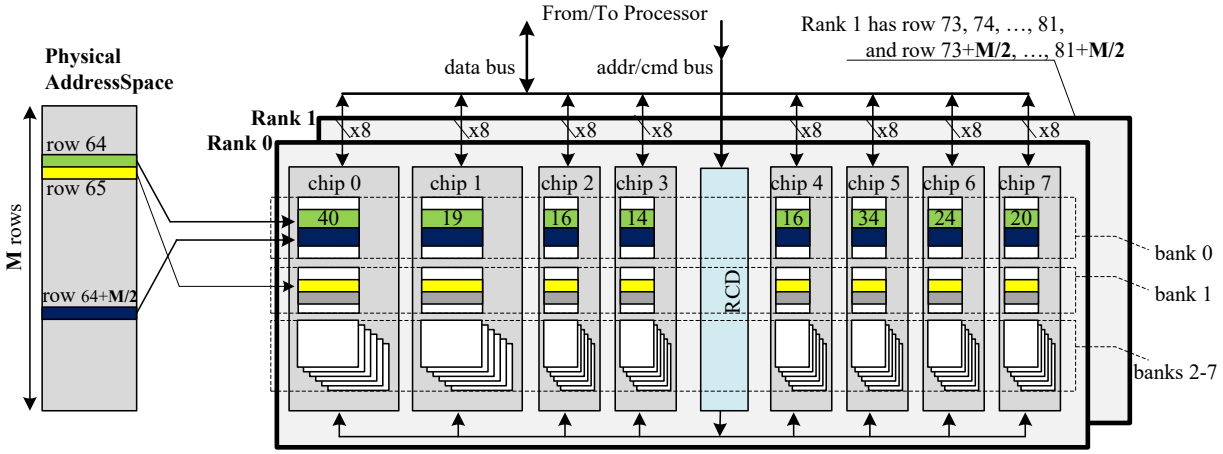


Fig. 2: Baseline DIMM configuration. One row consists of eight row segments, one segment per chip. The number in the row segment indicates the segment's tWR value (in memory cycles).

To support approximate computing analysis, EnerJ uses type qualifiers to map approximate data to low power storage [46]. Schmoll *et al.* presented a flexible software-based error handling [49]. Khudia *et al.* [21] and Mahajan *et al.* [29] proposed to dynamically monitor errors and adjust computation accuracy to meet the quality demand on the final results. These schemes are orthogonal to our work so that they can be combined with our design to further improve system performance.

III. DrMP Design Details

A. Baseline Memory Organization

Figure 2 depicts the baseline memory organization in the paper. A DIMM has two ranks while each rank has eight banks that are spreading across eight DRAM chips (the baseline has no ECC chip). We assume linear row layout across all banks and ranks as follows. One row has 8KB data.

- Consecutive *physical* rows (i.e., the rows in physical address space) are mapped to different banks. For example, rows `0,1,...,15` are mapped to memory banks `0,...,7` in rank `0` and then to banks `0,...,7` in rank `1`.
- The physical address space is divided in half. Rows from the two halves are interleaved. That is, after mapping rows `0,...,15`, rows `0+K,...,15+K` are mapped to banks `0,...,7` in rank `0` and banks `0,...,7` in rank `1`, respectively. We then continue to map rows `16,...,31` and so on. There are `2K` memory rows in the memory space.

In this configuration, each DRAM row has 8 segments spread across 8 chips. Each segment is termed a ***row segment***.

The memory controller sends addresses and commands to the Register Clock Driver (RCD) on a DIMM, which enables the synchronous operation of all eight DRAM chips. The data are sent from each chip to the data bus independently. The baseline follows JEDEC's DDR3 specification [16]; we give timing details in the experiments (see Section IV).

B. DRAM Restore Time Profiling

DrMP requires timing information about the memory. To determine the required information, post-fabrication profiling is done. In essence, this profiling tests memory under different settings of tWR and tRAS to find the best restore timing for DRAM row segments. The memory controller and the OS are enhanced to do the profiling similarly to prior work [24], [56]. The enhancements are: (1) the OS and the memory controller can alter timing values (tWR and tRAS) to check whether specific timing values work correctly; (2) a March test [56] checks the data integrity of each row by writing, reading and verifying test bit patterns in different access orders; and, (3) the test bit patterns are checked multiple times to ensure reliability with given timing values.

Since DRAM restore has worse timing at lower temperature [19], [1], profiling after a cold boot is safe. The timing profile is then used in the online operation of DrMP. For higher assurance, a temperature sensor could monitor memory temperature and disable DrMP if the temperature falls below the one during profiling, likewise to LPDDR [31]. Profiling may need to address VRT through enhanced ECC and large guardband, as described in prior work [20], [43].

Since DRAM cells use a similar charging process for both reads and writes, tWR and tRAS are correlated, i.e., a row segment having faster tWR also has a faster tRAS. Thus, we only need to test a subset of typical combinations. For example, tRAS and tWR have ranges *[19,42]* and *[12,25]*, respectively, in cycles. When setting tRAS=19, we try tWR=12 or 13. In total, we conduct binary search for 30 tRAS and tWR combinations, resulting in around 5 tries to find the best timing. Past work has demonstrated that a March test can be performed at high speed, e.g., 0.4ms per row [56], [45]. Therefore, the complete profiling can be done within 20 minutes.

The profile keeps two timing values (tWR and tRAS, 6 bits each) for each row segment, i.e., 12B per row, or 6MB of profile data for a 4GB DIMM. The OS saves the profile (including processed data, as we show next) in system storage, i.e., hard drive or SSD, and loads it at boot-up.

C. Motivation

To motivate DrMP, we first examine the severity of PRT. We followed the models in [58], [60] as described above. Although PRT appears as a major scaling challenge, the number of weak

cells (i.e., cells that take a longer time to restore) actually remains small in our observations. However, the portion of slow cells is significantly beyond the desired DRAM reliability in modern computer systems. For example, recent in-filed study [54] shows that the reliability is as low as 25 FIT (failure in time per billion device hours) per Mbit, making it impractical to integrate ECC with practical space overhead to rescue the weak cells.

Rather than trying to rescue the weak cells, an alternative solution to boost system performance is to aggressively reduce tWR for approximate computing. This approach exploits the error-resilience of many modern applications. Intuitively, if the errors induced by restoring bit cells faster than their required restore timing cause a tolerable number of application errors (similar to the errors induced by reducing refresh frequency [27]), then we may adopt existing approximate designs to mitigate PRT.

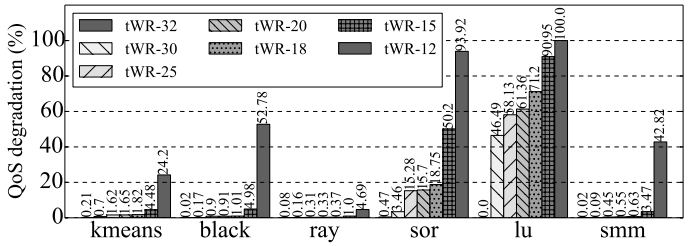


Fig. 3: The QoS degradation with larger tWR values.

We did an experiment to gradually reduce tWR to evaluate the QoS degradation [3] for a suite of benchmark programs that were used for approximate computing from the literature. The results are summarized in Figure 3. The experimental setting is described in Section IV. From the figure, it is clear that application-level errors increase with decreasing tWR. This effect happens because, given the same DRAM row, restoring data with smaller tWR leads to more cells failing to reach the target voltage level for reliable operation. The figure also illustrates that different programs exhibit vastly different error resilience. Some benchmarks, such as `smm`, only begin to suffer from high application-level errors when tWR is reduced to 15ns or smaller. Other applications, such as `lu`, begin to suffer from significant errors at a larger tWR value of 20ns. The high error rate (over 60%) observed for `lu` at 20ns is generally considered unacceptable for approximate computing [46], [34]. The figure shows that programs are sensitive to the selected tWR value, and consequently, it is impractical to uniformly reduce tWR.

Instead, we propose to reduce tWR at row granularity. DrMP exploits process variations within and across DRAM rows such that it can perform precision-aware restore scheduling.

D. DrMP-A: Approximate DRAM Restore

This section presents DrMP-A for high-performance approximate computing. In analyzing the ineffectiveness of the simple approximation scheme in Figure 3, we observe that restore errors may occur at random places in a row. Although these rows store non-critical data, depending on their data types, the importance of erroneous data bits varies. As an example, the sign bit and exponent bits of a floating point representation tend to be more important than the last several bits of the mantissa [28], [12]. For an integer that records RGB colors, the first two or four bits of each byte are often more important. Figure 4(a) shows the importance of bits of a memory row located at different places for different data types. If tWR is reduced below a reliable value, then errors from the "too fast restore" will occur at fixed positions in an application data value. If these positions are important bits, then catastrophic errors might be induced.

Based on this observation, we propose DrMP-A to enable per row approximation that achieves extremely low error rate. Figure 4(b) illustrates how DrMP-A works.

Assume an approximate data declaration "**float** vf[...]", where the first 16 bits of each array element are tagged as important. The OS support and the process to tag important bits are discussed in later sections. DrMP-A attaches three flags *(U_i, M_i, tWR_i)* to each row `i` in the memory space ($0 \leq i < 2K$). The flags are used by the memory controller:

(i) U_i is a 3-bit usage flag to indicate how the important bits of a row are categorized. DrMP-A uses the flags in Table I; more flags can be added if needed.

(ii) M_i is an 8-bit bit vector to record the four fastest row segments. For example, we have M_i="01111000" in Figure 4(b), which indicates that the row segments from chips 1/2/3/4 are faster and precise.

(iii) tWR_i is a 6-bit tWR value that records the tWR of the second or the fourth (depending on U_i) fastest row segment of a row. In the example, tWR_i=19 indicates that reducing the row tWR to 19 memory cycles ensures there is no restore error in row segments from chips 1/2/3/4 while there might be errors in other row segments.

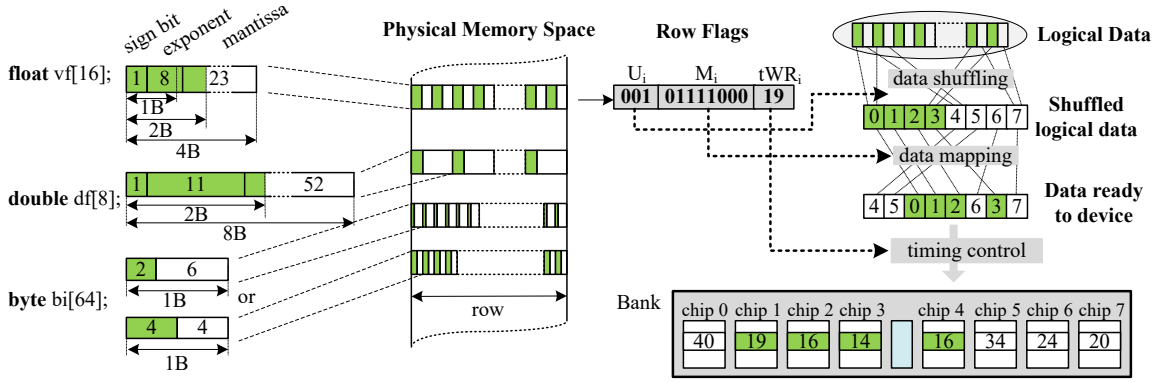
TABLE I: Definition of Usage Flag

"001"	the first 2B of each 4B data are important
"010"	the first 4b of each 1B data are important
"101"	the first 2B of each 8B data are important
"110"	the first 2b of each 1B data are important
"000"	all bits are important; this is the baseline
"111"	all bits are important; used in DrMP-P and DrMP-U

The memory controller schedules requests using the flags as depicted in Figure 4(b). First, the controller uses U_i to shuffle data bits into groups of important bits. Second, the controller uses M_i to map the important bits to four reliable faster row segments, i.e., the segments from chips 1/2/3/4 in the example. At this point, the data are ready to be sent to the memory module. Lastly, the memory controller uses tWR_i to determine when to schedule the next memory command to maximize memory bandwidth usage.

DrMP-A is designed to achieve a good trade-off between memory performance and computation accuracy. Using the fourth fastest row segment's tWR to determine the tWR of the whole row ensures quick access to the row, improving performance over a fully reliable baseline using the worst-case row segment tWR. The majority of cells can be reliably accessed with the fast tWR. In addition, by mapping important bits to row segments with tWR values less than or equal to

[3]QoS degradation is used to evaluate the output quality of approximate computing, which is calculated by comparing the approximated outputs to precise ones. The metrics are application dependent [46], [34]



(a) High order bits (green colored boxes) are more important (b) Map important bits to faster segments (no restore errors)

Fig. 4: The details of DrMP-A.

the row tWR, DrMP-A reduces the impact of restore errors, which minimizes the error rate at the application level.

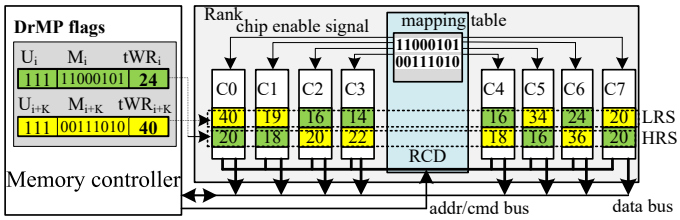


Fig. 5: DrMP-P constructs one fast row for each row pair. Each chip stores two row segments: LRS and HRS.

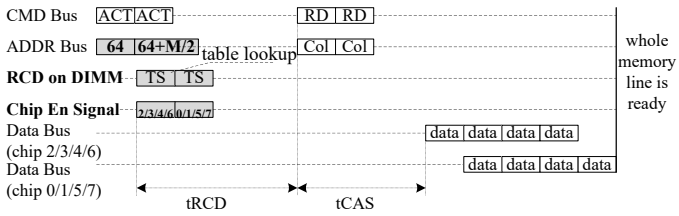


Fig. 6: The scheduling details of DrMP-P.

E. DrMP-P: Pairing Rows for Fast Precise Computing

While DrMP-A speeds up approximate computing by shortening the restore time, its effectiveness is often limited by the amount of non-critical data. We next reuse this hardware enhancement to speedup precise general-purpose computing.

Our study of tWR values at row segment level shows that the slowest row segments of different rows often fall on different chips. Figure 5 illustrates typical tWR values for two consecutive device rows. In the figure, the slowest row segments of row `i` and `i+K` are from chip 0 and chip 6 (with tWRs of 40 and 36), respectively. Thus, we pair two consecutive device rows to construct a fast row and a slow row so that the average restore time can be effectively reduced, which speeds up precise computing.

DrMP-P first creates K row pairs so that pair `i` contains row `i` and row `i+K` (there are 2K rows in total and $0 \leq i < K$). These two rows are K rows apart in physical address space and, as shown in Figure 2, next to each other on the device. Each chip j contributes two row segments to each row pair, referred to as $\mathbf{LRS}_j$ and $\mathbf{HRS}_j$, respectively, as in Figure 5. Here, we have:

$\mathbf{LRS}_j$ Low device address row segment from chip j
$\mathbf{HRS}_j$ High device address row segment from chip j

Without pairing, row `i` always contains LRS segments while row `i+K` always contains HRS segments. With pairing, we redistribute the 16 row segments in a pair such that a new row, while still having 8 row segments from 8 chips, contains a mix of LRS and HRS segments. Forming a pair is simple: for two device row segments of a pair in a chip, DrMP-P assigns the faster one to row `i` and the slower one to row `i+K`, i.e., we get one fast row, `i`, and one slow row, `i+K`, respectively.

For the example shown in Figure 5, row `i` is composed of HRS_0, HRS_1, LRS_2, LRS_3, LRS_4, HRS_5, LRS_6, and HRS_7. Row `i+K` consists of the complementary row segments for the pair. The tWR of each row is determined by the worst tWR of the composed segments, i.e., 24 and 40, respectively for these two constructed rows.

Bit flags. DrMP-P reuses the bit flags from DrMP-A. For row pair `i`, we attach (U_i, M_i, tWR_i) and $(U_{i+K}, M_{i+K}, tWR_{i+K})$ to the two rows, respectively. (i) U_i and U_{i+K} are always set to "111" for precise computing. (ii) M_i records the faster row segment from each chip — '0'/'1' indicate LRS/HRS. For the above example, M_i="11000101" so that $M_{i+K} = \sim M_i$ = "00111010". (iii) DrMP-P sets the tWR of each physical row using the largest tWR value from its component row segments.

Memory scheduling. Figure 6 illustrates the scheduling of memory operations in DrMP-P. The memory controller always fetch a row's DrMP bit flags before accessing the row (similar to DrMP-A). If U_i is "111", then ACT/RD/PRE commands are sent to operate eight DRAM chips. The commands may operate on either LRS or HRS of a row pair. DrMP-P activates and accesses the LRS or HRS segments in a chip in two consecutive bus cycles as follows. In this discussion, we use a closed page policy.

When sending ACT, DrMP-P supplies the row address `i`, which is used to index into the mapping vector table on the DIMM. In the example, this retrieves M_i = "11000101". With this mapping vector, chips 2/3/4/6 are activated first. The complementary others (chips 0/1/5/7) are activated in the next cycle. The activations are shown in bold fonts in Figure 6. Given that the two chip groups receive ACT and RD commands in two consecutive cycles, their output data have one memory cycle difference in the time of arrival at the processor (assuming two rows have the same tRAS).

While DrMP-P occupies two consecutive memory cycles for activation, it does not trigger tRRD constraint (i.e., the row to row delay, which restricts the number of ACT commands with a time window) as the total number of row segments activated is eight, the same as the baseline.

Extra timing. In DrMP-P, the RCD of each DIMM integrates a 256-entry mapping vector table that holds recently used mapping vectors, M_i. In addition, a one-bit enable wire is added from the RCD to each chip, as shown in Figure 5.

Compared to traditional memory scheduling, DrMP-P introduces two extra memory cycles on an access (modeled using CACTI [14] as detailed in Section 3.8): (1) one cycle is used to search the mapping vector table to determine which chips to activate and access; (2) a second cycle is required due to the delayed access to the second chip group.

DrMP-P for precise computing. DrMP-P speeds up precise computing by adopting the new tWR timings when accessing memory rows. As a comparison, in the row pairing-oblivious baseline, row `i` has all LRS segments and row `i+K` has all HRS segments — their tWR values are 40 and 36, respectively. The average tWR is 38 assuming both rows are accessed with same frequency. For the same access pattern, DrMP-P has an average tWR of 32 = (24+40)/2+2 including the 2-cycle extra access overhead.

F. DrMP-U = DrMP-A + DrMP-P

In DrMP-A and DrMP-P, a row pair is used either for approximate or precise computing, but not a mix of both. In this section, we propose DrMP-U, which combines DrMP-A and DrMP-P to fully exploit fine-grained differences in restore time of row segments.

DrMP-U relies on the fact that the slow logical row of a pair for DrMP-P still has several fast row segments. For example, row i+K in Figure 5 has two row segments with tWR values smaller than or equal to 19. If this row is used to save approximate data with usage flag "110", the 1/4 important bits can be saved in fast segments, such that the row tWR can be reduced to 19, achieving a large performance improvement for approximate computing. In addition, the simple segment grouping strategy in DrMP-P is sub-optimal to support both. Given that the tWR of row `i` is 24, it is unnecessary to take the faster row segment (LRS, tWR=16) from chip 2. Using a slower row segment (HRS, tWR=20) has no impact on the final row-level timing. Yet, this choice improves the chance of a smaller tWR when using the slow row for approximate computing — the important bits can be saved in LRS of chip 2.

DrMP-U exploits this observation to construct two fast rows, instead of only one in DrMP-P. A row pair is created from two physical rows that are K rows apart, similar to DrMP-P. For the pair containing rows `i` and `i+K` ($0 \leq i < K$), DrMP-U uses row `i` to save precise data and row `i+K` to save approximate data. The pair bit flags (PU_i, PM_i, $PtWR_i$) combine the flags from the two rows, e.g., PU_i consists of two sub-flags U_i and U_{i+K}.

DrMP-U integrates DrMP-A and DrMP-P in one framework with the flags. When PU_i="000/000", the baseline is adopted for the row pair and DrMP is disabled (no approximation). When PU_i="111/111", the row pair is in the DrMP-P mode (precise-only computing). When PU_i="aaa/bbb" and "aaa" is neither 111 or 000, the mode is DrMP-A and "bbb" cannot be 111 or 000 (approximate-only computing). If PU_i is not one of these cases, then PU_i must be "111/bbb" where *bbb* can be 001, 010, 101, or 110, depending on what approximate data to save in row `i+K`. This mode is DrMP-U (hybrid approximate- and precise-computing).

PM_i has two 8-bit bit vectors. The first vector assists the access of row `i` for precise computing, as discussed for DrMP-P. DrMP-U uses (1) the negation of the first bit vector; and (2) the second bit vector to access row `i+K` . The former determines the row segments to hold approximate data while the latter determines the subset of segments to hold important bits, as shown in Figure 7.

$PtWR_i$ saves two tWR values for accessing row `i` and `i+K`, respectively. Both are fast accesses.

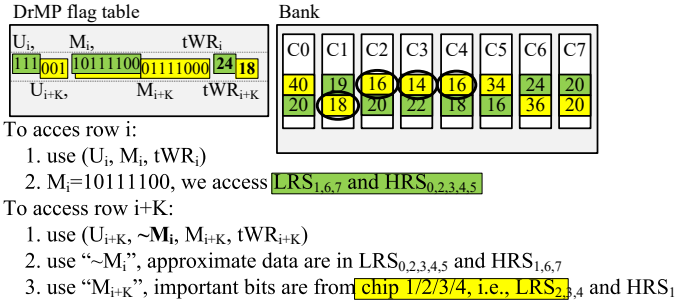


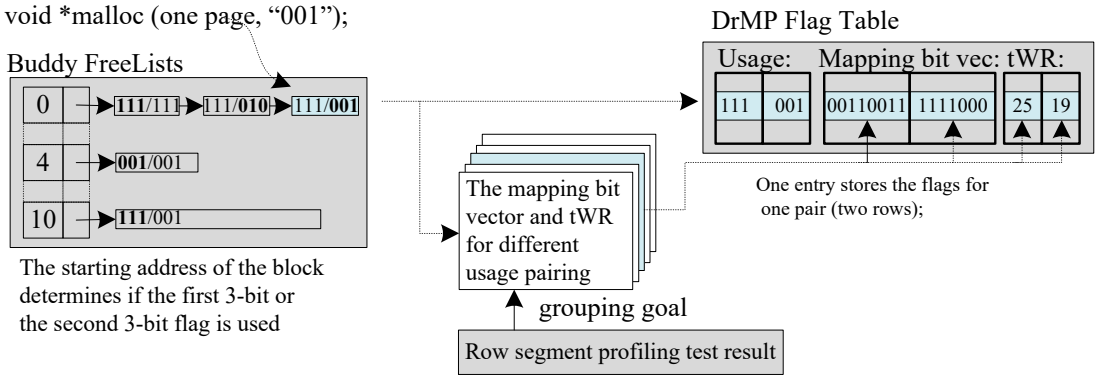
Fig. 7: DrMP-U combines DrMP flags and uses two mapping vectors to enable approximate computing.

For the example in Figure 7, when accessing row `i+K` for approximate computing, DrMP-U uses "$\sim M_i$", i.e., "01000011', to find that the approximate data are saved in LRS_0, HRS_1, LRS_2, LRS_3, LRS_4, LRS_5, HRS_6, and HRS_7. Given that M_{i+K} is "01111000", the middle four segments save important bits in chips 1/2/3/4, i.e., HRS_1, LRS_2, LRS_3, and LRS_4.

Row segment grouping. To maximize the scheduling opportunity for approximate computing, DrMP-U needs to find a better row segment grouping solution. The optimization goal depends how the row pair is used. After profiling to determine the best row segment tWR values, the OS performs an exhaustive search to find the best row segment grouping for different usage patterns. For a 4GB memory, it takes less than 20 seconds for one pattern, and less than 2 minutes for all five usage patterns. If chip manufacturers conduct post-fabrication test and regrouping, we will need a better heuristic. We leave this to future work.

G. Precision-aware Memory Management

DrMP-P and DrMP-U couple the usage of paired rows, which brings new constraints on memory allocation. For example, assume rows `i` and `i+K` ($0 \leq i < K$) are paired by DrMP-U such that rows `i` and `i+K` save precise and approximate data, respectively. Row `i`, after being reclaimed by the memory allocator, may not be allocated to store approximate data. This is because storing approximate data in row `i` needs a new M_i



(a) DrMP enhances MMU with usage flag (b) DrMP flag table is filled in based on usage flag

Fig. 8: The OS assisted memory management for DrMP.

while row `i+K` needs the negation of the old M_i to determine data locations, which prevents loading another M_i.

We use Figure 8 to illustrate precision-aware memory allocation. For simplicity, we apply DrMP only to allocate *normal* user data. We do not allocate memory space for device drivers or DMA operations. Modern OSes, like Linux, adopt buddy allocation to allocate blocks of consecutive memory pages to user applications. Linux's buddy allocator maintains an array of 11 freelists that link free blocks of 2^l pages ($0 \leq i \leq 10$). A request asking for more than 2^{10} pages is served by multiple blocks. Given a 2^l-paged block whose first page address is `x`, its **paired block** is defined as the 2^l-paged block whose first page address is either `x+P/2` (if $0 \leq x < P/2$) or `x-P/2` (if $P/2 \leq x < P$), here P is the total number of memory pages.

In DrMP, a memory request specifies not only the size but also the required precision:

void *malloc(int size, char UsageFlag);

Here, *UsageFlag* is a 3-bit flag as shown in Table I. To service this request, DrMP adds a 6-bit usage flag "aaa/bbb" to each block in the freelist. The flag describes the usage of the block and its paired block. That is, a block whose starting address is in the first half of the memory uses "aaa" while its paired block uses "bbb". The bold font in the figure indicates the flag that is actually used by a free block.

When linking a 2^{10}-paged block to the freelist, the OS links its paired block at the same time. The usage flags are initialized for both blocks. DrMP saves the usage flag in the first byte of each free block and this flag is carried to smaller blocks when a large block is split, as shown in Figure 8. A valid usage flag is one of the following:

1) "000/000" indicates that the two blocks are used as they are in the baseline. DrMP is disabled.
2) "111/111" indicates that the two blocks are paired and are used for precise computing only.
3) "111/`aaa`" indicates that the two blocks are paired and used for mixed precision computing. The block with the starting address in the low half of the memory is for precise computing and the paired block (with higher address) is for approximate computing. The flag `aaa` can be one of "001","010","101", and "110", depending on what approximate data to save in the block.
4) "`aaa`/`aaa`" (when `aaa` is neither "000" nor "111") indicates the two blocks are for approximate computing only. They are not paired. `aaa` is set similarly as above.

With DrMP, a memory allocation request is serviced by the memory allocator to provide a block of a matching size and usage flag. In Figure 8, the request for one page of "001"-type approximate data is satisfied by the third block in the #0-freelist. The buddy allocator without DrMP extension would return the first block instead.

The OS maintains a DrMP flag table to assist precision-aware restore scheduling. Given each row pair, the table keeps one entry that saves the usage flag, mapping bit vector, and tWR values for both rows, as shown in Figure 8.

The OS fills in the DrMP flag table when it updates a corresponding page table entry. The usage flags are extracted from the allocated memory block. Based on the usage flag, the OS loads the mapping bit vectors and tWR values from the grouping results (as described in Section 3.6).

Fragmentation optimization. A concern for memory allocation is that DrMP may increase system fragmentation. In Figure 8, a request for a 2^4-paged block for precise computing may not be satisfied even though there is a block with a matching size. The request triggers a bigger block to be split, creating additional small blocks in the system. We next discuss optimization to minimize fragmentation.

DrMP combines compatible usage flags such that the OS may return a block with a compatible (i.e., not exactly the same) usage flag. For the example in Figure 8(a), to satisfy the request for a 2^4-paged block with "010" flag, it is acceptable to return a block with "111/001" flag and its starting address is in the second half of the memory. We dynamically alter the usage flag of the block to "111/010". It is safe to do so because either "001" or "010" set the tWR to the fourth fastest row segments, and thus, they share the same mapping bit vectors and tWR values for the row pair. In addition, DrMP can satisfy the request with "010" flag with a block of a "001" flag. Here, the OS returns a more reliable but slightly slower block.

H. Architecture Enhancements

Figure 9 shows an overview of the architectural enhancements. The light color shaded boxes indicate the enhancements

to support DrMP-A and the dark color shaded boxes indicate the additional enhancements to support DrMP-P and DrMP-U.

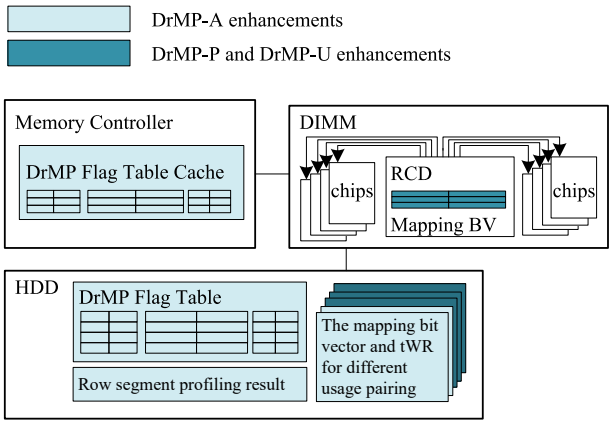


Fig. 9: An overview of architectural enhancements.

Space overhead. In the HDD, we save the row segment profiling information that marks the best tRAS and tWR for each row segment. The profile needs 6MB of storage for a 4GB main memory. The OS then applies the row segment grouping algorithm to get the mapping bit vectors and tWR/tRAS values for each different pairing pattern. Given that we have five patterns and each row is of 8KB, we need (8b+6b)×4GB/8KB×5 = 4.4MB HDD space for the mapping bit vectors. The DrMP flag table occupies 1.1MB (= 4GB/8KB×(3b+8b+6b)) space.

Given that the bit flags are for 8KB rows, they show good access locality, i.e., similar to the locality of a TLB buffer. We use a 512-entry on-chip DrMP flag table to buffer the most frequently accessed entries, which requires about 2KB space. To support DrMP-P and DrMP-U, we add a 256-entry mapping bit vector buffer in each DIMM, which occupies about 512B. The buffer is organized as a direct-mapped cache with tag fields maintained in the memory controller. Overall, the space overhead is modest.

Timing overhead. We used CACTI [14] to model timing overhead. DrMP has minimal timing overhead for DrMP-A. It introduces two CPU cycles of extra latency to extract the DrMP flags and shuffle the data based on approximate usage flag. The overhead is added for both read and write accesses because the data need to be remapped between the device layout and logic layout.

DrMP introduces extra latency to pair rows for DrMP-P and DrMP-U. As described in DrMP-P, two extra memory cycles are needed — one cycle determines which chips to activate and the other cycle is needed to send the device command.

Energy overhead. By introducing chip enable wires, DrMP activates the same number of subarrays as the baseline. We use CACTI to model the flag cache in the memory controller and the map table buffer in the DIMM. The energy overhead is negligible as shown in the experiments.

IV. Experimental Methodology

To evaluate the effectiveness of DrMP on approximate computing, we adopted the two-phase methodology from past work [34], [36]. In the first phase, we used a Pin-based simulator to instrument programs annotated for approximate computing. The simulator tracks all loads and stores of integer and floating-point variables and, based on the memory map of weak cells (where timing exceeds reliable operation), injects faults into memory operands at runtime. The memory map was generated following the model of Zhang et al. [58], [60], and the timing parameters were aligned with the industrial values [19], [1], [2]. We integrated the usage flags in the memory map so that different row segment pairing strategies lead to different error rates at runtime. For this phase, we ran the benchmarks to completion and compared the final output with the one from the baseline run (i.e., with no restore errors).

In the second phase, we used a cycle accurate simulator, USIMM [7], to compare performance and energy consumption. USIMM executed the instructions tracked in the first phase to make sure the two runs had the same instruction flow. We modeled a 4-core chip multiprocessor following past research [50], [39]. For the DRAM main memory, we used the Micron SDRAM DDR3 [33] [4]. Table II lists the details.

TABLE II: System Configuration

Processor	four 3.2Ghz cores; 128 ROB size Fetch width: 4, Retire width: 2, Pipeline depth: 10
Memory Controller	Bus freq.: 800 MHz; Write queue capacity: 64 Write queue high/low watermark: 40/20 Address mapping: rw:cl:rk:bk:ch:offset Page management policy: close-page with FRFCFS
DRAM	1 channel, 2 ranks/channel, 8 banks/rank, 32K rows/bank, 8KB/row, 64B cache line tCK=1.25ns, width: x8 tCAS(CL): 13.75ns, tRCD: 13.75ns, tRC: 48.75ns tCWD: 6.25ns (5 cycles), tBURST: 5.0ns (4 cycles) tRAS: 35ns, tRP: 13.75ns, tFAW: 24 cycles, tRRD: 5 cycles, tRFC: 208nCK, tREFI: 7.8μs

A. Benchmarks

We selected a suite of benchmark programs that were used in the literature to evaluate approximate computing. As shown in Table III, the benchmarks are from different domains, including machine learning, financial analysis and scientific computing. In addition, we included two memory intensive applications `libq` and `leslie` from SPEC CPU2006 [53]. These two applications always demand precise computing. They are used to form workloads with mixed precision demands.

TABLE III: Evaluated Applications

Application	Description (Input)	Quality Metric	% of approx mem accesses
kmeans	Machine Learning (Color image)	Image diff	45.4%
blackscholes	Financial Analysis (Portfolio options)	Avg. price error	6.3%
raytracer	3D Image Renderer (Light, texture, etc)	Image diff	4.0%
sor	Scientific Comp (Grid pattern)	Mean entry diff	79.5%
lu	Scientific Comp (Dense matrix)	Mean entry diff	98.0%
smm	Scientific Comp (Dense matrix)	Mean normalized diff	73.5%

For the evaluation, we manually annotated the benchmarks to identify the data that can be approximated. This approach

[4]Whereas we are studying DDR3 in the paper, the proposed techniques are applicable to other DRAM types, e.g., DDR4, LPDDR and HMC

is the same as past work [35], [46]. Table III summarizes the percentage of memory accesses that access approximate data.

B. Evaluation for Approximate Computing

To evaluate QoS, we compared the results from approximate execution to those from the baseline with precise execution and followed prior studies [46], [34] to compute application-specific metrics. Traditionally, a flat threshold of 10% error rate was set as the upper bound [46], [10], [34], [57]. However, this error rate often leads to a large deviation [42], [25]. For example, `blacksholes` is a financial analysis application from PARSEC 3.0. Its QoS metric is stock/option price difference. A 10% error for a $20 option leads to $2 difference, which is significant and generally unacceptable [25]. Therefore, we did not set a fixed percentage threshold. Our design goal is instead to minimize the error rate with a performance improvement.

V. Results and Analysis

We studied and compared the following schemes.

— `Baseline`. This scheme mitigates PRT with fully relaxed restore timing, i.e., `tRAS`=42, `tWR`=25, and `tRCD`=15 [59]. The baseline adopts built-in spare rows and columns to rescue the worst set of cells. The same timing is applied to all chips.

— `PRT-Free`. This scheme assumes future DRAM chips are free from PRT, and thus use the current JEDEC timing, i.e., `tRAS`=28, `tWR`=12, and `tRCD`=11 [33].

— `Approx-base-#`. This scheme is baseline approximation schemes without dedicated techniques to protect important bits. # is either 2 or 4, indicating whether the row tWR is set to the 2*nd* or 4*th* fastest row segment.

— `DrMP-A-#`. This scheme is DrMP-A, where rows are being utilized for approximate computing with important data bits being protected.

— `DrMP-P-#`. This scheme is DrMP-P. The paired rows are used to save precise data only.

— `DrMP-U-#`. This scheme is DrMP-U. Given one row pair, the row with the low address saves precise data, while the row with high address saves approximate data.

A. QoS of Approximate Computing

We first evaluated the effectiveness of our approximate computing strategy. Figure 11 compares the QoS of different schemes. `Approx-base-n` uses the tWR of the n*th* fastest row segment. It is similar to `DrMP-A-n` except important bits in a row are not mapped. From the figure, we observe that mapping important bits greatly mitigates QoS degradation. For example, mapping reduces 100% QoS degradation of `lu` to 0.31% in `DrMP-4` when ensuring the reliability of four row segments in each row.

We examined `kmeans` to check the visual effect of the output image, as shown in Figure 10. Figure 10(a) shows the precise output image with no restore errors. We observe that Figure 10(b)(c)(d) shows a color change — the gray tail of the eagle turns pinkish; and Figure 10(b)(d) shows visible noise. DrMP-A-4 (Figure 10(e)) has no visible change compared to the baseline. Therefore, it is important to reduce the error rate in approximate computing. The bit remapping in DrMP-A is effective in mitigating the QoS degradation.

B. Performance

Figure 12 reports the execution time of different schemes. The results are normalized to `Baseline`. In the figure, `Gmean` is the geometric mean of all workloads. For `DrMP-A` and `DrMP-U`, we compared the schemes when setting the row tWR to the 2*nd* and 4*th* fastest row segment for approximate computing. For `DrMP-P`, we studied two page allocation schemes — `DrMP-P-rand` is the baseline allocation that returns a random page; `DrMP-P-fast` returns a random fast page first, i.e., it uses all fast pages before allocating slows ones.

On average, `DrMP-A-4`, `DrMP-P-fast`, and `DrMP-U-4` have a 17%, 10.2% and 19.8% improvement over the baseline, respectively. Not unexpected, random page mapping in `DrMP-P-rand` lowers the speedup to 5.2%. The difference between `DrMP-A-2` and `DrMP-A-4` (similarly, `DrMP-U-2` and `DrMP-U-4`) is usually small. For applications that have dominant approximate data accesses, such as `lu` and `smm`, DrMP-U does better than `PRT-Free`. This improvement happens because many rows that save approximate data have a tWR faster than the standard 15ns tWR in `PRT-Free`. DrMP-U achieves lower improvement for applications that are less memory intensive with fewer approximate accesses, such as `kmeans` and `raytracer`. Given the moderate difference of performance and the notable contrast in QoS as reported in Figure 11, we use the 4*th* fastest row segment in subsequent sections.

C. Timing Values

Table IV compares the average restore timing in different schemes. `PRT-Free` has the best timing, while `Baseline` has the worst. `Baseline`, `PRT-Free`, `DrMP-A-n` do not use row pairing, and thus, they have one set of average values. The tRAS of DrMP-A is even better than `PRT-Free` because the timing is aggressively reduced, which introduces restore errors in some row segments. `DrMP-A-2` has better timing than `DrMP-A-4` as it reduces the tWR more aggressively.

TABLE IV: Restore Timing Value of Each Row Pair

Scheme	Low Address Row			High Address Row		
	tRAS	tWR	tRCD	tRAS	tWR	tRCD
	(memory cycles)					
Baseline	42	25	15	same as left		
PRT-Free	28	12	11	same as left		
DrMP-A-2	20	13	15	same as left		
DrMP-A-4	22	15	15	same as left		
DrMP-P	30	20	15	40	24	15
DrMP-U-2	30	20	15	19	12	15
DrMP-U-4	33	21	15	20	13	15

In DrMP-P, each pair has a fast row and a slow row. The average of slow rows in DrMP-P is close to `Baseline` as the slowest row segments do not change. The average of approximate rows in `DrMP-U-n` is close to `DrMP-A-n`, indicating that `DrMP-U` makes better use of device rows. `DrMP-P` and `DrMP-U` report similar timings for low address

(a) Precise baseline (0%) (b) Approx-base-2 (5.97%) (c) DrMP-A-2 (1.98%) (d) Approx-base-4 (2.16%) (e) DrMP-A-4 (0%)

Fig. 10: Visual effects for approximated runs for `kmeans`.

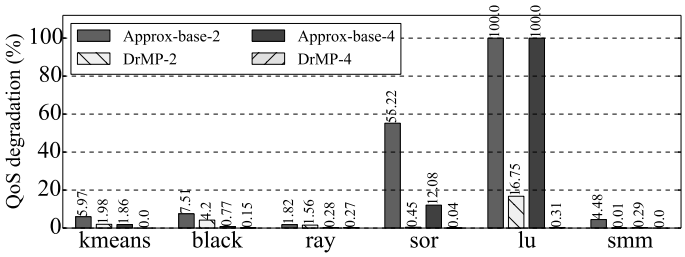


Fig. 11: QoS degradation in different schemes.

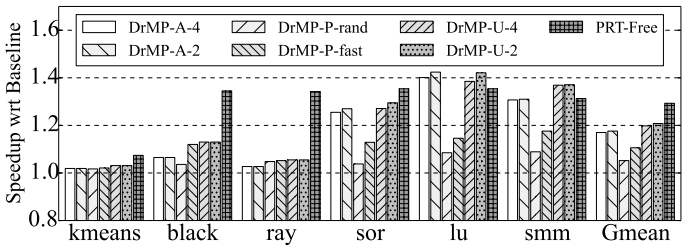


Fig. 12: Performance comparison.

rows because `DrMP-U` exploits mainly the fast row segments in the slow rows.

D. Energy Consumption

DrMP reduces system energy consumption by speeding up the program execution. Next, we study the memory energy reduction in more details. Figure 13 reports memory energy consumption in terms of background (`bg`), active/precharge (`act/pre`), read/write (`rd/wr`) and refresh (`ref`). We followed the Micron power equations and parameters [32], [33]. We used CACTI to model the DrMP flag cache in the memory controller and in the DIMM. We observed that, by improving application performance, the DrMP schemes reduce the background energy the most. Overall, `DrMP-U-4` achieves 15% energy consumption reduction, which is within 7% gap of `PRT-Free`. The primary contributor is the reduced background energy because of the shortened execution time. Read/write power/energy is also optimized with the reduction in the restore time.

E. System Overhead

In DrMP, the 512-entry flag cache in the memory controller is organized as 4-way set associative cache. On average, it has 97.8% hit rate. The CACTI simulation shows that the cache has 0.35ns access latency, 0.02mm^2 area, 1.5mW standby leakage power and 5.45pJ energy per access. The DIMM mapping table is organized as a 256-entry direct-mapped cache

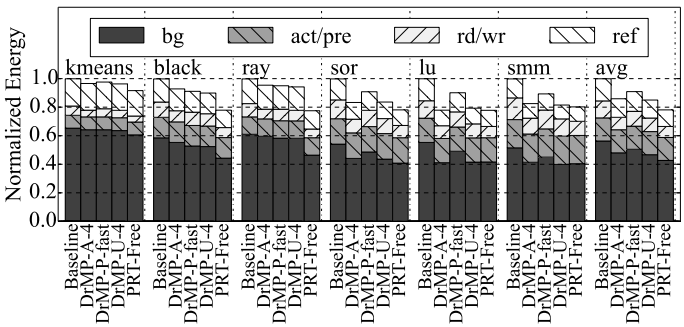


Fig. 13: Energy comparison.

with tags maintained in the memory controller. It has an average 97.9% hit rate. The CACTI simulation shows that this structure has 0.22ns access time, 0.016mm^2 area, 1.14mW standby leakage and 2.96pJ energy per access.

Frequently used flag cache entries are captured in the L2 cache and loaded to the flag cache and mapping table on misses of these structures. The performance overhead is less than 1%.

Due to limitations of our evaluation framework, we did not evaluate the overhead of the buddy memory allocation. We expect it to be low due to its invocation, i.e., to allocate blocks of consecutive pages to processes, is less frequent per process.

F. Integration with Restore Truncation.

Restore truncation (RT) [59] is a recent PRT mitigation approach. It partially restores memory cells to a low voltage level, depending on the distance of an access to the next row refresh. Since DrMP exploits error resilience through approximate computing, these two designs are orthogonal. Figure 14 reports the results when both schemes are adopted (RT+DrMP). From the figure, RT+DrMP achieves a larger performance improvement over RT: on average, RT+DrMP is 13.7% better than RT.

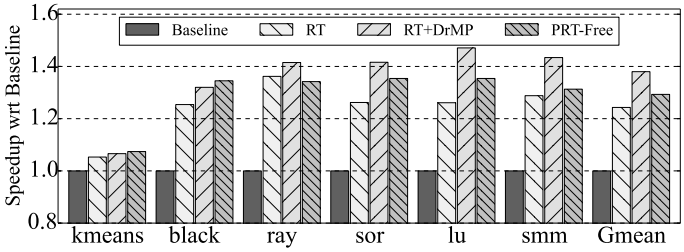


Fig. 14: Performance comparison of DrMP and RT [59].

VI. Conclusions

This paper proposes DrMP, a fine-grained precision-aware DRAM restore scheduling design, to mitigate performance

degradation due to PRT in future DRAMs. We devised three schemes to achieve the best trade-off between performance, energy consumption, hardware overhead, and computation precision. Our experimental results show that, on average, DrMP achieves 20% performance improvement over a conventional relaxed timing design, while minimizing QoS degradation for a suite of benchmark programs.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive feedbacks. This work is supported in part by NSF under grants CCF-1422331, CNS-1012070, CCF-1535755 and CCF-1617071.

REFERENCES

[1] "Memory Device With Relaxed Timing Parameter According To Temperature, Operating Method Thereof, and Memory Controller And Memory System Using The Memory Device," US 20140359242 A1, 2014.
[2] "Backward Compatible Dynamic Random Access Memory Device and Method of Testing Therefor," US 9123441 B1, 2015.
[3] A. Agrawal, *et al.*, "Mosaic: Exploiting the Spatial Locality of Process Variation to Reduce Refresh Energy in On-Chip eDRAM Modules," in *HPCA*, 2014.
[4] I. Bhati, *et al.*, "DRAM Refresh Mechanisms, Penalties, and Trade-Offs," in *TC*, 2015.
[5] K. Chandrasekar, *et al.*, "Exploiting Expendable Process-margins in DRAMs for Run-time Performance Optimization," in *DATE*, 2014.
[6] K. Chang, *et al.*, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
[7] N. Chatterjee, *et al.*, "USIMM: the Utah SImulated Memory Module," *Technical report*, University of Utah, 2012.
[8] B. R. Childers, *et al.*, "Achieving Yield, Density and Performance Effective DRAM at Extreme Technology Sizes," in *MEMSYS*, 2015.
[9] J. Choi, *et al.*, "Multiple Clone Row DRAM: a Low Latency and Area Optimized DRAM," in *ISCA*, 2015.
[10] H. Esmaeilzadeh, *et al.*, "Neural Acceleration for General-Purpose Approximate Programs," in *MICRO*, 2012.
[11] M. Ghosh and H.-H. S. Lee, "Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs," in *MICRO*, 2007.
[12] Q. Guo, *et al.*, "High-Density Image Storage Using Approximate Memory Cells," in *ASPLOS*, 2016.
[13] S. Hong, *et al.*, "Low-voltage DRAM Sensing Scheme with Offset-cancellation Sense Amplifier," in *JSSC*, 2002.
[14] http://www.hpl.hp.com/research/cacti/, "CACTI: An Integrated Cache and Memory Modeling Tool," http://www.hpl.hp.com/research/cacti/, 2009.
[15] ITRS, "The International Technology Roadmap for Semiconductors Report," , 2013, URL http://www.itrs.net/.
[16] JC-42.3, "JESD79-3F, DDR3 Stdandard," JEDEC, 2010.
[17] M. Jung, *et al.*, "Omitting Refresh: A Case Study for Commodity and Wide I/O DRAMs," in *MEMSYS*, 2015.
[18] M. Jung, *et al.*, "Efficient Reliability Management in SoCs - An Approximate DRAM Perspective," in *ASPDAC*, 2016.
[19] U. Kang, *et al.*, "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *MEMORY FORUM*, 2014.
[20] S. Khan, *et al.*, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
[21] D. S. Khudia, *et al.*, "Rumba: An Online Quality Management System for Approximate Computing," in *ISCA*, 2015.
[22] W. Kong, *et al.*, "Analysis of Retention Time Distribution of Embedded DRAM - A New Method to Characterize Across-Chip Threshold Voltage Variation," in *ITC*, 2008.
[23] D. Lee, *et al.*, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.
[24] D. Lee, *et al.*, "Adaptive-latency DRAM: Optimizing DRAM timings for the common-case," in *HPCA*, 2015.
[25] S. Lee, *et al.*, "Synthesis of Quality Configurable Systems," in *WAX*, 2016.
[26] J. Liu, *et al.*, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.
[27] S. Liu, *et al.*, "Flikker: Saving DRAM Refresh-power through Critical Data Partitioning," in *ASPLOS*, 2011.
[28] J. Lucas, *et al.*, "Sparkk: Quality-Scalable Approximate Storage in DRAM," in *Memory Forum*, 2014.
[29] D. Mahajan, *et al.*, "TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning," in *HPCA*, 2016.
[30] J. Mandelman, *et al.*, "Challenges and Future Directions for the Scaling of Dynamic Random-access Memory (DRAM)," in *IJRC*, 2002.
[31] Micron, "TN-46-12: Mobile DRAM Power-Saving Features/Calculations," Technical Note, 2001.
[32] Micron, "TN-41-01: Calculating memory system power for DDR3," *Technical report*, Micron, 2007.
[33] Micron, "TwinDie DDR3 SDRAM, MT41J512M8 Datasheet," http://www.micron.com, 2011.
[34] J. S. Miguel, *et al.*, "Load Value Approximation," in *MICRO*, 2014.
[35] J. S. Miguel, *et al.*, "Doppelganger: A Cache for Approximate Computing," in *MICRO*, 2015.
[36] J. S. Miguel, *et al.*, "The Bunker Cache for Spatio-Value Approximation," in *MICRO*, 2016.
[37] J. Mukundan, *et al.*, "Understanding and Mitigating Refresh OverhOver in High-density DDR4 DRAM Systems," in *ISCA*, 2013.
[38] O. Mutlu, "Memory Scaling: A Systems Architecture Perspective," in *IMW*, 2013.
[39] P. Nair, *et al.*, "A Case for Refresh Pausing in DRAM Memory Systems," in *HPCA*, 2013.
[40] P. J. Nair, *et al.*, "ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates," in *ISCA*, 2013.
[41] P. J. Nair, *et al.*, "XED: Exposing On-Die Error Detection Information for Strong Memory Reliability," in *ISCA*, 2016.
[42] J. Park, *et al.*, "AXGAMES: Towards Crowdsourcing Quality Target Determination in Approximate Computing," in *ASPLOS*, 2016.
[43] M. K. Qureshi, *et al.*, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.
[44] A. Raha, *et al.*, "Quality-Aware Data Allocation in Approximate DRAM," in *CASES*, 2015.
[45] M. Rahman, *et al.*, "COMeT+: Continuous Online Memory Testing with Multi-Threading Extension," in *TC*, 2014.
[46] A. Sampson, *et al.*, "EnerJ: Approximate Data Types for Safe and General Low-power Computation," in *PLDI*, 2011.
[47] A. Sampson, *et al.*, "Approximate Storage in Solid-State Memories," in *MICRO*, 2013.
[48] Samsung, "DRAM tWR," http://www.samsung.com/global/business/semiconductor/file/product/tWR-0.pdf, 2001.
[49] F. Schmoll, *et al.*, "Improving the Fault Resilience of an H.264 Decoder using Static Analysis Methods," in *TECS*, 2013.
[50] W. Shin, *et al.*, "NUAT: A Non-Uniform Access Time Memory Controller," in *HPCA*, 2014.
[51] Y. H. Son, *et al.*, "Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations," in *ISCA*, 2013.
[52] Y. H. Son, *et al.*, "CiDRA: A Cache-inspired DRAM Resilience Architecture," in *HPCA*, 2015.
[53] SPEC, "SPEC CPU2006," http://www.spec.org/.
[54] V. Sridharan, *et al.*, "Feng Shui of Supercomputer Memory Positional Effects in DRAM and SRAM faults," in *SC*, 2013.
[55] J. Stuecheli, *et al.*, "Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory," in *MICRO*, 2010.
[56] A. van de Goor and I. Tlili, "March Tests for Word-oriented Memories," in *DATE*, 1998.
[57] D. Wong, *et al.*, "Approximating Warps with Intra-warp Operand Value Similarity," in *HPCA*, 2016.
[58] X. Zhang, *et al.*, "Exploiting DRAM Restore Time Variations in Deep Sub-micron Scaling," in *DATE*, 2015.
[59] X. Zhang, *et al.*, "Restore Truncation for Performance Improvement in Future DRAM Systems," in *HPCA*, 2016.
[60] X. Zhang, *et al.*, "On the Restore Time Variations of Future DRAM Memory," in *TODAES*, 2017.
[61] B. Zhao, *et al.*, "Variation-Tolerant Non-Uniform 3D Cache Management in Die Stacked Multicore Processor," in *MICRO*, 2009.