Assessing Impact of Data Partitioning for Approximate Memory in C/C++ Code

Soramichi Akiyama
Department of Creative Informatics, The University of Tokyo
akiyama@ci.i.u-tokyo.ac.jp

ABSTRACT

Approximate memory is a technique to mitigate the performance gap between memory subsystems and CPUs with its reduced access latency at a cost of data integrity. To gain benefit from approximate memory for realistic applications, it is crucial to partition applications' data to *approximate data* and *critical data* and apply different error rates. However, error rates cannot be controlled in a fine-grained manner (e.g., per byte) due to fundamental limitations of how approximate memory can be realized. Due to this, if approximate data and critical data are interleaved in a data structure (e.g., a C struct that has a pointer and an approximatable number as its members), data partitioning may degrade the application's performance because the data structure must be split to separate memory regions that have different error rates. This paper is the first to conduct an analysis of realistic C/C++ code to assess the impact of this problem. First, we find the type of data (e.g., "int", "struct point") that is assessed by the instruction that incurs the largest number of cache misses in a benchmark, which we refer to as the *target data type*. Second, we qualitatively estimate if the target data type of an application has approximate data and critical data interleaved. To this end, we set up three criteria to analyze it because definitively distinguishing a piece of data as approximate data or critical data is infeasible since it depends on each use-case. We analyze 11 memory intensive benchmarks from SPEC CPU 2006 and 2 graph analytics frameworks, and show that the target data types of 9 benchmarks are either a C struct or a C++ class (criterion 1). Among them, two have a pointer and a non-pointer member together (criterion 2) and three have a floating point number and other members together (criterion 3).

1 APPROXIMATE MEMORY ARCHITECTURE

1.1 Overview of Approximate Memory

Approximate main memory, or approximate memory, is a new technology to mitigate the performance gap between memory subsystems and CPUs of computers. The main idea is to reduce the latency of main memory accesses at a cost of the data integrity (i.e., the CPU may read a slightly different data from what has been written before to the main memory) by exploiting *design margins* that exist in many DRAM chips today. A design margin refers to the difference between a design parameter defined in the specification of a device and the actual value the device can be operated with. For example, many DRAM chips can read stored data "almost" correctly with a few bit-flips (errors) injected to the data when some wait-time parameters are shortened than the specification [5], resulting in access latency reduction.

Approximate memory attracts much research interest due to the ever-increasing performance gap between memory subsystems and CPUs [13]. Chang *et al.* [5] measure the relationship between error rates and latency reduction for a large number of commercial DRAM chips, Das *et al.* [9] and Zhang *et al.* [32] prolong the interval of refreshing[1] for strong memory cells to reduce the average latency, and our previous work [1] estimates effect of approximate memory to realistic applications without simulation by counting the number of DRAM internal operations that induce errors.

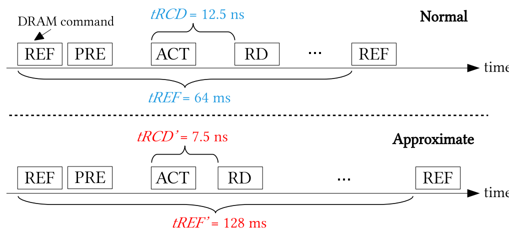

Figure 1: DRAM command sequence of normal memory (top) and approximate memory (bottom): In this example, tRCD is shortened to 7.5 ns and tREF is prolonged to 128 ms, both of which reduce the average latency.

1.2 Relaxing Timing Constraints

Approximate memory on DRAM can be realized by *relaxing timing constraints* of DRAM chips. Although there are other types of approximate memory such as approximate flash memory for storage that leverages multiple levels of flash programming voltages [12] and approximate SRAM based on supply voltage scaling [4, 10, 30], we focus on approximate DRAM using relaxed timing constraints. A timing constraint refers to the specification of the interval between DRAM commands issued from the memory controller, and relaxing a constraint means either shortening or prolonging an interval (i.e., "violating" the specification). Relaxing a timing constraint of DRAM reduces the access latency to main memory but may inject errors (bit-flips) to memory cells with an error rate depending on how aggressively a constraint is relaxed and other aspects such as the chip temperature [16].

Figure 1 shows an example of DRAM command sequence in normal (exact) memory and approximate memory. It shows four representative DRAM commands: refresh (`REF`), precharge (`PRE`), activation (`ACT`), and read (`RD`). In this example, a timing constraint called `tRCD` is shortened from 12.5 ns to 7.5 ns, and one called `tREF` is prolonged from 64 ms to 128 ms. The memory controller must wait for `tRCD` between a `RD` command and the preceding `ACT` command to ensure that the activation has finished. Although `tRCD` is defined as

[1] A DRAM cell is a tiny capacitor and loses its charge as time goes by, thus it needs to be periodically re-charged (also known as "refreshed").

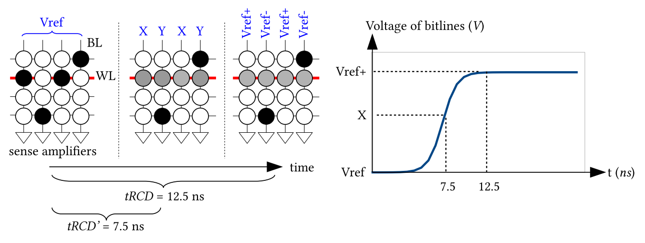

Figure 2: An ACT command drives an entire row at the same time with a given tRCD value, forcing the minimum approximation granularity to be a row size (typically 4 KB or 8 KB).

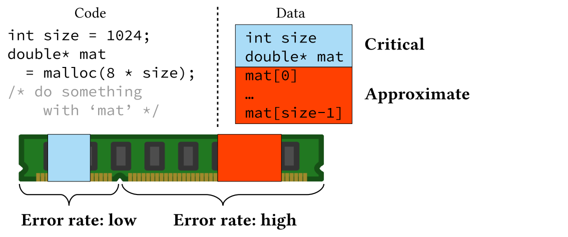
```
int size = 1024;
double* mat
  = malloc(8 * size);
/* do something
    with 'mat' */
```

Figure 3: Data partitioning: (1) identify critical data (size, mat) and approximate data (mat[0], ..., mat[size-1]), and (2) map them to memory regions that have different error rates.

12.5 ns in the DDR3-1600J specification [2], Chang *et al.* [5] found that only a small portion of the cells experience errors even when tRCD is shortened below it to serve the RD command quickly. tREF is another timing constraint that specifies the longest interval between two REF commands, which refresh DRAM cells to prevent them from losing stored data. Das *et al.* [9] and Zhang *et al.* [32] propose to prolong this interval because many DRAM cells can retain data for more than 64 ms in practice. Because prolonging tREF increases the amount of time during which more useful commands are served, it reduces the average DRAM access latency.

1.3 Approximation Granularity

In approximate memory that exploits design margins in the timing constraints, the granularity of approximation cannot be smaller than 4 KB (i.e., the same error rate must be applied to a continuous 4 KB or more). This is because a DRAM command drives many memory cells in parallel to improve performance.

Figure 2 shows how an ACT command drives multiple memory cells in parallel. The circles show an array of memory cells, where each row has a wordline (WL) and each column has a bitline (BL). A black cell has a value of 1 and a white cell has a value of 0, and a gray one is in an intermediate state. An ACT command takes the target row number as its parameter (for example, the 2nd row in the figure). The WL of the target row is enabled to connect the cells in the target row to the BLs. The electric charge of the cells whose values are 1 pull up the voltages of the BLs from Vref (the reference value) to Vref+, which takes up to tRCD. Finally, the sense amplifiers sense the voltages of the BLs to fetch the values in the cells. If tRCD is reduced from 12.5 ns to say 7.5 ns, the sense amplifiers may fetch wrong values because the voltages of the BLs have not yet reached to Vref+ as shown in the right of the figure. The same discussion is applicable to "0" cells and Vref-. Because a shortened tRCD is applied to an entire row, it forces the granularity of approximation to be the size of a row, which is typically 4 KB or 8 KB. Therefore, we cannot control the error rate with a granularity smaller than 4 KB. Similar discussions can be applied to other timing constraints as well such as tREF and a REF command that reads an entire row at a time to refresh the cells in the row [15].

Although this work focuses only on DRAM, this limitation is also applicable to other memory technologies because the requirement of performance is fundamental due to the ever-increasing performance gap between memory subsystems and CPUs. For example, phase change memory (PCM) organizes memory cells as an array [23] and injects electric pulses to an entire row at once. If we consider realizing approximate memory with PCM, for example by reducing the length of pulses for writing, the approximation granularity is still limited by its row size.

2 DATA PARTITIONING AND CHALLENGE

2.1 Data Partitioning

Executing an application on approximate memory requires two additional steps compared to executing it on normal memory. First, we must identify which parts of the application's data are approximate data or critical data. This prevents errors from being injected into critical data (e.g., pointers) and the application can yield meaningful results. Second, we must map approximate data and critical data to different memory regions that have different error rates. We refer to these two steps as *data partitioning*.

Figure 3 depicts an example of data partitioning. Assume that the application can yield acceptable results even with some errors injected to the floating point numbers stored in mat, then we can identify each element of mat (mat[0], ..., mat[size-1]) as approximate data and the other variables as critical data. Note that 'double* mat' itself is critical data because it is a pointer.

After the identification, the approximate data and critical data must be mapped to memory regions with different error rates. The size of each memory region must be at least 4 KB because the approximation granularity (the granularity at which error rates can be controlled) is 4 KB as discussed in Section 1.3.

2.2 Challenge: Interleaved Criticality

The challenge of data partitioning is that it can impose much overhead when approximate data and critical data are *interleaved* in a single data type. The two kinds of data are defined as interleaved when they co-locate inside one element of a C struct or a C++ class.[2] Figure 4 shows an example of interleaved approximate data and critical data. The struct tree_node data type has both approximate data and critical data in it, and nodes points to an array of struct tree_node. Applying data partitioning to this code requires to map the first 20 bytes (an int and two pointers) and the last 8 bytes (a double) of a struct node to separate memory regions due to the large approximation granularity, splitting the

[2]The only difference of them is the access control of members in typical C/C++ implemenetations (for example, even a C struct can have member functions in gcc). Therefore, we only mention C struct hereafter without loss of generality.

```
struct tree_node {
   int id; // id of the node, critical
   struct tree_node *r; // pointer to the right child, critical
   struct tree_node *l; // pointer to the left child, critical
   double score; // score of this node, approximate
};

int size = 1000 * sizeof(struct tree_node);
struct tree_node *nodes = malloc(size);
```

Figure 4: Approximate data and critical data interleaved in a single C struct: applying data partitioning to this code imposes much overhead due to reduced access locality.

```
       31f8:   add    %r10,%rax
       31fb:   cmp    %rax,%r9
       31fe:   jbe    3258 <primal_bea_mpp+0xf8>
       if( arc->ident > BASIC )
48.58  3200:   mov    0x18(%rax),%edi
       3203:   test   %edi,%edi
       3205:   jle    31f8 <primal_bea_mpp+0x98>
```

Figure 5: Sample output of perf report: It shows an instruction, the offset of the instruction from the head of the binary, and the percentage of cache misses that it incurs (if any), from right to left. The C code, if (arc->ident > BASIC), corresponds to the assembly code below it.

region pointed to by `nodes` to many pieces. This can dramatically degrade the performance due to reduced access locality.

Prior works [8, 18, 31, 33] propose to convert "an array of structures" to "a structure of arrays" for improving access locality. For example, the code in Figure 4 can be converted to have a distinct array for each member of `struct tree_node`, and then the array for `score` can be mapped to a consecutive memory region with high error rate. However, this method mitigates the data partitioning problem only when the conversion itself does not degrade access locality. Assume that `r`, `l`, and `score` in Figure 4 are accessed closely in time (for example, the code might change which child to traverse next depending on the score of the current node), then splitting `r`, `l`, and `score` into different arrays can degrade the access locality and performance.

3 SOURCE CODE ANALYSIS

The research question we tackle is stated as: *Is data partitioning a real concern for realistic applications?* To answer it, we analyze source code of widely used benchmarks and show that many applications potentially have approximate and critical data interleaved.

3.1 Analysis Methodology

Approximate memory is the most effective when an application's data that incur many cache misses are stored on it. With this in mind, our analysis consists of two steps:

(1) First, we find a data structure (e.g., a C `struct`) accessed by an instruction that incurs the largest number of cache misses among a target application using hardware performance counters.
(2) Second, we apply our criteria to qualitatively estimate the probability that the data structure found has approximate data and critical data interleaved in it.

In the first step, we measure the number of cache misses **per instruction** using Precise Event Based Sampling (PEBS) on Intel CPUs. PEBS is an enhancement of normal performance counters that uses designated hardware for sampling to reduce the skid between the time an event (e.g., cache miss) occurs and the time it is recorded [3, 29]. The small skid enables pinpointing which instruction in an application binary causes many cache misses. We execute a benchmark with its sample dataset using linux `perf`, and the actual command line is '`perf record -e r20D1:pp -- benchmark`'. The parameter `r20D1:pp` specifies a performance event whose *event number* is 0xD1 and the *umask value* is 0x20, which "*counts retired load instructions with at least one uop that missed in the L3 cache*" (described in Table 19.3 of [14]). The parameter `benchmark` is replaced with an actual command line to execute each benchmark. Figure 5 shows a sample output of `perf report`, executed after a measurement by `perf record`. The measurement is done for a benchmark called `mcf` and the details of the benchmarks we analyze are described in Section 3.2. Each line shows, from right to left, an instruction, the offset of the instruction from the head of the binary, and the percentage of cache misses it incurs (if any). The C code, `if( arc->ident > BASIC )`, corresponds to the lines of assembly code below it.

Figure 5 shows that the instruction that incurs the largest number of cache misses in this application is `mov 0x18(%rax),%edi`. By mapping the assebmly code in Figure 5 with the C source code, we can find that the `mov` instruction accesses the `ident` member of a C `struct` named `arc`[3]. We refer to the type of `arc`, `struct arc`, as the *target data type* because it is the target of the analysis in the next step. Please carefully note that, although this result tells that the `ident` member alone incurs many cache misses, it does not mean that we can split this `struct` to exclude `ident` and put an array of `ident` to approximate memory. Memory accesses to the other members of the same `struct` may hit the cache only because cache misses to `ident` fetch the whole part of the `struct` to main memory, in which case this splitting doubles the number of cache misses. When the assembly code is more complex and the target data type is not obvious, we resort to human labor to find it because systematically reverse-engineering an arbitrary binary to C/C++ code is not the main focus of this work. The same methodology is applicable to a template function as well because there is an independent piece of assembly code for each instantiation of it (i.e., no type-ambiguity exists in assembly code).

In the second step, we analyze the target data type of each benchmark to estimate if it has approximate data and critical data interleaved. The issue is that it is infeasible to definitively distinguish approximate data and critical data without concrete use-cases and expert knowledge of the benchmark. There are some typical cases (e.g., pointers are *typically* critical data, floating point numbers are *typically* approximate data), but even these might be overridden by particular code or use-case. Therefore, instead of

[3]In fact, ident is placed in the 0x18th byte of arc and BASIC is a compile-time constant whose value is 0. This supports our guess that the mov instruction copies ident to %edi and the test instruction following it compares ident and BASIC (constant 0).

definitively distinguishing them, we set up criteria to qualitatively predict the probability that a target data type has approximate data and critical data interleaved:

C1: Is the target data type a `struct` or a `class`?
C2: If **C1** is yes, does it include a pointer and at least one other member with non-pointer type?
C3: If **C1** is yes, does it include a floating point number and at least one other member?

As the number of criteria applicable to a target data type increases, so does the probability that it has approximate data and critical data interleaved. Please note the following two things. (1) If a target data type is a nested `struct` (i.e., a `struct` that has another `struct` in it), we "flatten" it so that every member becomes a non-struct type (e.g., `int`, `char*`) before applying the criteria. This reproduces how a nested `struct` is mapped to a memory region. (2) We exclude member functions because they are stored in function tables located in a separate memory region from data members in typical C++ implementations, meaning that they are already partitioned and irrelevant to our analysis. For example, if a `class` has two integers and a function as its members, we consider the two integers as the only members and the analysis result for this data type is "C1 = Yes, C2 = No, C3 = No".

The intention of each criterion is as follows. **[C1]**: If the target data type is not a `struct`, we can allocate the whole part of it (or an array of it) either in normal memory or approximate memory data and no data partitioning is needed. On the other hand, if it is a `struct`, it might include approximate data and critical data interleaved. **[C2]**: If the target data type includes a pointer as its member, it is highly possible that the pointer is critical data because even a single bit-flip invalidates it. If it includes other members with non-pointer types as well, then these members might be approximate data, in which case approximate data and critical data are interleaved. **[C3]**: If the target data type includes a floating point number, it is possible that the floating point number is approximate data. If it includes other members that are critical data, then the target data type has approximate data and critical data interleaved. Note that **C3** is not symmetric with **C2** because a pointer is most probably critical data, while a floating point number can either be approximate data or critical data depending on each application and use-case. Therefore, **C2** requires a non-pointer member along with a pointer member, while **C3** only requires a member with any type along with a floating point number.

3.2 Analyzed Benchmarks

Table 1 describes the benchmarks we analyze. Each line shows a benchmark's name, its domain, and the cache miss rate with the percentage of cache misses incurred by the instruction that incurs the largest number of cache misses. For example, "74.8 % (88.8 %)" means that the number of cache misses divided by the number of memory accesses is 0.748, and the number of cache misses incurred by a certain instruction divided by the number of total cache misses is 0.888. Note that the latter can be any value between 0 and 1 independently of the former. The values are measured in the environment in Table 2 and the `-O3` option is used to build the binaries.

From SPEC CPU 2006 [7], we analyze 11 benchmarks whose cache miss rates are more than 20%. We use the largest dataset named `ref` to measure the cache miss rates. We exclude others because approximate memory is not beneficial for CPU intensive benchmarks with low cache miss rates. We also exclude ones written in Fortran because the necessity of data partitioning is affected by the programming style, which is largely different in each programming language. This work only focuses on C/C++ that are more often used in modern applications than Fortran.

We also analyze two graph analytics benchmarks, Graph 500 [19] and GraphMat [27]. The former is used to measure the performance of supercomputers, thus the speed is the only concern. The latter, on the other hand, is designed to preserve programmability while maintaining the speed as much as possible. For Graph 500, we generate the dataset with the `scale` parameter set to 19 and execute breadth first search on a single core. We use the reference implementation provided in their website [19]. For GraphMat, we convert the ego-Twitter dataset of Stanford Large Network Dataset Collection [17] into a GraphMat-compatible format and calculate PageRank on it. The source code is taken from the github repository[4] and we run it on a single core.

Table 1: Analyzed Benchmarks: 11 from SPEC CPU 2006 (we select ones written in C/C++ and the cache miss rates are more than 20%) and 2 graph analytics frameworks.

Name	Domain	Cache Misses
milc	quantum simulation	82.6% (9.61 %)
sjeng	game AI (chess)	74.5% (44.9 %)
libquantum	quantum computing	54.6% (47.4 %)
lbm	fluid dynamics	49.2% (44.5 %)
omnetpp	discrete event simulation	47.9% (6.62 %)
soplex	linear programming	41.2% (31.5%)
gobmk	game AI (go)	38.4% (39.3 %)
gcc	c compiler	36.8% (22.8 %)
mcf	optimization	33.7% (48.6 %)
dealII	finite element analysis	33.6% (67.1 %)
namd	molecular dynamics	21.0% (11.3 %)
Graph 500	graph analytics (bfs)	74.8 % (88.8 %)
GraphMat	graph analytics (PageRank)	47.7 % (53.5 %)

Table 2: Experiment Environment

CPU	Intel Xeon Silver 4108 (Skylake, 8 cores)
Memory	DDR4-2666, 96 GB (8GB × 12)
LLC	11 MB (shared across all the cores)
OS	Debian GNU/Linux 10
kernel	4.19.0-6-amd64
gcc/g++	8.3.0 (Debian 8.3.0-6)

4 RESULTS

Table 3 shows the analysis results. Each row shows a benchmark, the target data type of it, and applicability of the three criteria to the benchmark ('Y' means that the criterion is applicable, and 'N' means not applicable). If **C1** is 'N' for a benchmark, we put '-' for **C2** and **C3** because these two criteria are evaluated only if **C1** is

[4]https://github.com/narayanan2004/GraphMat

Table 3: Results of our analysis

Name	Target Data Type	C1	C2	C3
milc	struct complex	Y	N	Y
sjeng	struct QTType	Y	N	N
libquantum	struct quantum_reg_node_struct	Y	N	Y
lbm	double	N	-	-
omnetpp	class cChannel	Y	N	N
soplex	struct Element	Y	N	N
gobmk	Hashnode (struct hashnode_t)	Y	Y	N
gcc	struct rtx_def	Y	N	N
mcf	arc_t (struct arc)	Y	Y	N
dealII	double	N	-	-
namd	struct CompAtom	Y	N	Y
Graph 500	int64_t	N	-	-
GraphMat	float	N	-	-

'Y'. In the target data type column, we put the original type inside parentheses if it is aliased by a `typedef` declaration. For example, "arc_t (struct arc)" means that the target data type is a C `struct` named arc, and it is aliased as arc_t. The benchmarks are executed in the environment described in Table 2. For all the benchmarks, the instruction that incurs the largest number of cache misses exists in their own code and not in any standard C/C++ libraries.

Observation 1: The target data type is a struct or a class in 9 SPEC CPU 2006 benchmarks out of 11 analyzed. Among them, `mcf` and `gobmk` have a pointer and a non-pointer member in their target data types, and `milc`, `libquantum`, and `namd` have a floating point number and another member in their target data types. For the former group, if one of the members other than the pointer is approximate data, they have approximate data and critical data interleaved because the pointer is most probably critical data. For example, our previous work [1] shows that `mcf` can yield the same result as the error-free one even when a member of `arc_t` is approximated. For the latter group, if the floating point number is approximate data (which is the case in many applications) and another member is critical data, they have approximate data and critical data interleaved.

Observation 2: There is no benchmark whose target data type has both a pointer and a floating point number, although this type of benchmarks (if exist) have the highest probability of having approximate data and critical data interleaved. This might be because we exclude Fortran, which tends to be used for numerical applications. Investigating the relationship between suitability for approximate memory and the programming language / style used in an application is a part of our future work.

Observation 3: The target data type is a non-struct type in both of the graph analytics frameworks. Although their source code have C `struct`s that appear to have approximate data and critical data interleaved, these `struct`s do not incur many cache misses. We presume that highly optimized code for performance have less probability of including approximate data and critical data interleaved in a single data type. Investigating the relationship between the suitability for approximate memory and the category of each application (e.g., highly optimized graph analytics) is another important aspect of future work.

5 DISCUSSIONS

5.1 Threats to Validity

Our analysis methodology cannot be applied as-is when a member of a C `struct` is passed to a function by reference. For example in Figure 6, the same function (`f`) is called either by passing `&s1.v` or `&s2.v` as its argument. Finding the data type that the memory region pointed to by `fp` belongs to requires an investigation of stack traces and points-to analysis [26].

```
struct S1 {
    double v; // probably approximate
    double vv; // probably approximate
} s1;

struct S2 {
    double v; // probably approximate
    int *p; // probably critical
} s2;

void f(double *fp) {
    // do something and return the result through *fp
}

f(&s1.v); // (1): invoke f by passing s1.v by reference
f(&s2.v); // (2): invoke f by passing s2.v by reference
```

Figure 6: Calling the same function by passing members of different structs by reference. Identifying the data type that *fp belongs to requires stack traces and points-to analysis.

Although it seems more natural for a function to take a pointer of a whole `struct` such as '`void g(struct S1 *sp)`', this may appear in some cases such as when a library function returns the result through a pointer. In our case study, the original data type was identifiable without hitting this issue in all the benchmarks.

5.2 Analyzing Performance Implication

This paper is focused on estimating if an application has approximate data and critical data interleaved, but we defer quantitative analysis of how the application's performance is degraded if we really partition approximate data and critical data into separate memory regions. One method to conduct this analysis is to actually split the target data type and measure the slowdown of the application on a real machine. However, the issue is that although much work have been done on structure splitting, structure peeling, and structure reordering and data partitioning is essentially the same as structure splitting, there is no off-the-shelf tool to support it as far as we know. One of the reasons is that it is very troublesome to implement these techniques so as to work correctly in general cases. For example, old versions of `gcc` support structure reordering, but the functionality was removed because it "*did not always work correctly*" [11].

An alternative method to qualify how an application slows down after data partitioning is to use heuristic metrics that estimate the access locality between two structure members. Prior works on leverage these metrics to find which members should be in the same structure to maximize the performance improvement. For example, Ye *et al.* [31] introduce a metric called *access affinity* between two members u and v of a structure. Given a memory trace, it is increamented by one if there is a pair of memory accesses to u and v where the number of memory accesses to other members in-between is less than a threashold. Because this metric captures access locality of two members of a structure, it can also be used to estimate how harmful to split two members into separate memory regions. Other studies such as [18, 21] propose similar metrics. Applying these metrics is easier than implementing data partitioning because they are based on memory traces of the original program.

6 RELATED WORK

To the best of our knowledge, this work is the first to investigate the data partitioning problem for approximate memory. We discuss related work in the context of error rate controlling.

Nguyen *et al.* [22] propose a method that partially mitigates the data partitioning problem. It transposes rows and columns of data layout inside DRAM so that a chunk of data is stored across many rows that have different error rates. This enables protection of important bits (e.g., the sign bit of a floating point number) while aggressively approximating less important bits. This mechanism is effective for DNNs because they require the whole part of a large weight matrix at once and the number of memory accesses do not increase regardless of the data layout. However, it is not effective in general cases where memory is accessed with smaller granularity.

Mapping data into memory regions with different error rates depending on its criticality is commonly proposed. Liu *et al.* [20] partition a DRAM bank into bins with proper refresh interval and ones with prolonged refresh interval. Each data is store into either type of bins depending on the criticality specified by the programmer. Although they do not discuss the minimum bin size, it cannot be smaller than a DRAM row (typically 4 KB or 8 KB) as we discuss in Section 1.3. Chen *et al.* [6] propose a memory controller that maps data into different DRAM banks with different error rates depending on the criticality of the data. Because this method is bank-based, the approximation granularity is limited to the bank size. A typical DDR3/DDR4 DIMM module has 2 GB to 16 GB with either 8 or 16 banks, resulting in a typical bank size of 256 MB to 2 GB. Raha *et al.* [24] advance a previous work [20] by measuring each bin's error rate at a given prolonged refresh interval and assigning them to approximate data in the ascending order of the error rate. They realize the bin size (or "page size" in their terminology) of 1 KB by measuring the average error rate per 1 KB. Although this approach could be further pursued to realize smaller page sizes, it still cannot control error rates per byte as it just measures them to use proper pages for given criticality.

Applying lower supply voltages to store approximate data is often done especially with SRAM [4, 10, 30]. Esmaeilzadeh *et al.* [10] propose a dual-voltage SRAM architecture to implement approximate data types proposed by Sampson *et al.* [25]. This architecture also suffers from the same problem we discuss in this paper because it changes the supply voltage at the row granularity of an SRAM subarray, although a row of a subarray is smaller than a row of the entire SRAM.

Tovletoglou [28] *et al.* propose an end-to-end framework that ensures the availability of VMs hosted on approximate memory. The framework provides an OS support and its APIs to allocate approximate data to memory control units operated by reduced refresh rate, while maintaining the ability to exploit memory-level parallelism. Because the framework leverages the existing memory allocators implemented in Linux, the minimum size of a critical / approximate memory region is 4 KB, making it suffer from the same problem we discuss in this paper.

7 CONCLUSIONS AND FUTURE WORK

Approximate memory is a new technique to reduce the main memory access latency, but has a potential problem in data partitioning when approximate data and critical data are interleaved in applications' data structures. We are the first to assess the impact of this problem by means of analyzing realistic C/C++ code. We introduce three criteria to qualitatively estimate if the data type that incurs the largest number of cache misses in an application has approximate data and critical data interleaved, and applied them to 11 SPEC CPU 2006 benchmarks and 2 graph analytics frameworks. As a result, we found that the data type that incurs the largest number of cache misses are either a C `struct` or a C++ `class` in 9 benchmarks. Among them, two have a pointer (possibly critical) and a non-pointer member interleaved and three have a floating point number (possibly approximate) and other members interleaved.

Future work includes two directions: (1) categorizing workloads by their domains, programming language / styles used and other aspects, and (2) quantitatively analyzing the performance implication of data partitioning to each workload. For the second part, no off-the-shelf tool that enables data partitioning is available even though much work have been done on structure splitting, mostly because implementing structure splitting for general cases is very difficult. Instead, we can use existing metrics [18, 21, 31] that predict the benefit of structure splitting for a given structure.

ACKNOWLEDGMENTS

This work was supported by JST, ACT-I Grant Number JPMJPR18U1, Japan. The author thanks the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Soramichi Akiyama. 2019. A Lightweight Method to Evaluate Effect of Approximate Memory with Hardware Performance Monitors. *IEICE Transactions on Information and Systems* E102-D, 12 (Dec. 2019), 2354–2365.

[2] JEDEC SOLID STATE TECHNOLOGY ASSOCIATION. 2010. JEDEC STANDARD: DDR3 SDRAM Standard. JESD79-3F. (July 2010).

[3] Denis Bakhvalov. 2018. Advanced profiling topics. PEBS and LBR. (June 2018). https://easyperf.net/blog/2018/06/08/Advanced-profiling-topics-PEBS-and-LBR.

[4] Nandhini Chandramoorthy, Karthik Swaminathan, Martin Cochet, Arun Paidimarri, Schuyler Eldridge, Raji V. Joshi, Matthew M Ziegler, Alper Buyuktosunoglu, and Pradip Bose. 2019. Resilient Low Voltage Accelerators for High Energy Efficiency. In *International Symposium on High Performance Computer Architecture (HPCA)*. 147–158.

[5] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. 2016. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *International Conference on Measurement and Modeling of Computer Science (SIGMETRICS)*. 323–336.

[6] Yuanchang Chen, Xinghua Yang, Fei Qiao, Jie Han, Qi Wei, and Huazhong Yang. 2016. A Multi-accuracy Level Approximate Memory Architecture Based on Data Significance Analysis. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 385–390.

[7] Standard Performance Evaluation Corporation. 2018. SPEC CPU 2006 Benchmarks. (2018). https://www.spec.org/cpu2006/.

[8] Stephen Curial, Peng Zhao, Jose Nelson Amaral, Yaoqing Gao, Shimin Cui, Raul Silvera, and Roch Archambault. 2008. MPADS: Memory-Pooling-Assisted Data Splitting. In *International Symposium on Memory Management (ISMM)*. 101–110.

[9] Anup Das, Hasan Hassan, and Onur Mutlu. 2018. VRL-DRAM: Improving DRAM Performance via Variable Refresh Latency. In *Design Automation Conference (DAC)*. 1–6.

[10] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture Support for Disciplined Approximate Programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 301–312.

[11] Free Software Foundation, Inc. 2019. GCC 4.8 Release Series Changes, New Features, and Fixes. (2019). https://gcc.gnu.org/gcc-4.8/changes.html.

[12] Qing Guo, Karin Strauss, Luis Ceze, and Henrique S. Malvar. 2016. High-Density Image Storage Using Approximate Memory Cells. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 413–426.

[13] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc.

[14] Intel. 2018. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide. (2018).

[15] Bruce Jacob, Spencer Ng, and David Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[16] Jeremie S. Kim, Minesh Patel, Hasan Hassan, Lois Orosa, and Onur Mutlu. 2019. D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput. In *International Symposium on High Performance Computer Architecture (HPCA)*. 582–595.

[17] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. (2014). https://snap.stanford.edu/data/.

[18] Jin Lin and Pen-Chung Yew. 2010. A Compiler Framework for General Memory Layout Optimizations Targeting Structures. In *Workshop on Interaction between Compilers and Computer Architecture (INTERACT)*. 8:1 – 8:8.

[19] The Graph 500 List. 2017. Graph 500 | Benchmark Specification. (2017). https://graph500.org/?page_id=12.

[20] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 213–224.

[21] Svetozar Miucin and Alexandra Fedorova. 2018. Data-Driven Spatial Locality. In *International Symposium on Memory Systems (MEMSYS)*. 243–253.

[22] Duy Thanh Nguyen, Nguyen Huy Hung, Hyun Kim, and Hyuk-Jae Lee. 2020. An Approximate Memory Architecture for Energy Saving in Deep Learning Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2020), 1–14.

[23] Yoshio Nishi and Blanka Magyari-Kope. 2019. *Advances in Non-volatile Memory and Storage Technology, 2nd Edition*. Woodhead Publishing.

[24] Arnab Raha, Soubhagya Sutar, Hrishikesh Jayakumar, and Vijay Raghunathan. 2017. Quality Configurable Approximate DRAM. *IEEE Trans. Comput.* 66, 7 (July 2017), 1172–1187.

[25] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. *SIGPLAN Not.* 46, 6 (June 2011), 164–174.

[26] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Symposium on Principles of Programming Languages (POPL)*. 32–41.

[27] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proceedings of the VLDB Endowment* 8, 11 (July 2015), 1214–1225.

[28] Konstantinos Tovletoglou, Lev Mukhanov, Dimitrios S. Nikolopoulos, and Georgios Karakonstantis. 2020. HaRMony: Heterogeneous-Reliability Memory and QoS-Aware Energy Management on Virtualized Servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 575–590.

[29] Vincent M. Weaver. 2016. *Advanced Hardware Profiling and Sampling(PEBS, IBS, etc.): Creating a New PAPISampling Interface*. Technical Report UMAINE-VMW-TR-PEBS-IBS-SAMPLING-2016-08. University of Maine.

[30] Lita Yang and Boris Murmann. 2017. Approximate SRAM for Energy-Efficient, Privacy-Preserving Convolutional Neural Networks. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 689–694.

[31] Louis Ye, Mieszko Lis, and Alexandra Fedorova. 2019. A Unifying Abstraction for Data Structure Splicing. In *International Symposium on Memory Systems (MEMSYS)*. 173–183.

[32] X. Zhang, Y. Zhang, B. R. Childers, and J. Yang. 2016. Restore truncation for performance improvement in future DRAM systems. In *International Symposium on High Performance Computer Architecture (HPCA)*. 543–554.

[33] Peng Zhao, Shimin Cui, Yaoqing Gao, Raúl Silvera, and José Nelson Amaral. 2007. Forma: A Framework for Safe Automatic Array Reshaping. *ACM Transactions on Programming Languages and Systems* 30, 1 (Nov. 2007), 1 – 29.