The Granularity Gap Problem: A Hurdle for Applying Approximate Memory to Complex Data Layout

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

Ryota Shioya
The University of Tokyo
Tokyo, Japan
shioya@ci.i.u-tokyo.ac.jp

ABSTRACT

The main memory access latency has not much improved for more than two decades. *Approximate memory* is a technique to reduce the DRAM access latency in return of losing data integrity, and it is beneficial for applications that are robust to noisy data. To obtain reasonable outputs from applications on approximate memory, it is crucial to protect critical data while accelerating accesses to non-critical data. A fundamental limitation of approximate memory is that the *approximation granularity*, the minimum size of a continuous memory region that the same error rate is applied, is as large as a few kilo bytes. However, applications may have critical and non-critical data interleaved with smaller granularity (e.g., a pointer and a number in the same C struct). We refer to this as the *granularity gap problem.* We first show that many applications potentially suffer from this problem, and then we propose a framework to quantitatively evaluate the performance overhead of a possible method to avoid it using known techniques. The evaluation results show that the performance overhead is non-negligible compared to expected benefit from approximate memory, suggesting that the granularity gap problem is a significant concern.

CCS CONCEPTS

• **Hardware → Memory and dense storage**; **Analysis and design of emerging devices and systems**.

KEYWORDS

approximate memory; memory systems; performance analysis

ACM Reference Format:
Soramichi Akiyama and Ryota Shioya. 2021. The Granularity Gap Problem: A Hurdle for Applying Approximate Memory to Complex Data Layout. In *Proceedings of the 2021 ACM/SPEC InternationalConference on Performance Engineering (ICPE '21), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3427921.3450259

1 INTRODUCTION

The impact of main memory access latency to the overall performance is much larger on a computer today than in the past. This is because the performance gap between the main memory and the CPU has ever been enlarging. Figure 1 shows the single thread performance of server-class CPUs plotted over time[1]. The figure shows an exponential growth of the single thread performance until recent years. In contrast, the access latency of DRAM that the main memory consists of has been almost the same for more than two decades. As shown in [5], the speedup of the major latency sources of DRAM over time is very marginal, especially when compared to the exponential growth of the CPU performance. Because DRAM access latency occupies substantial amount in a random memory access latency, there is a strong need to reduce the DRAM access latency to catch up with the CPU performance.

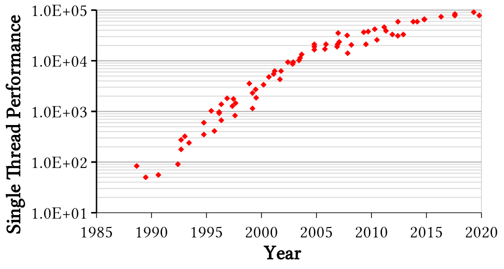


Figure 1: Exponential growth of single thread performance over time (normalized to SPEC CPU 2006 score × 1000).

Approximate memory is a technique to reduce the main memory latency by sacrificing its data integrity [14, 22, 25, 30]. Prior works have proven that the DRAM access latency can be reduced by violating the timing constraints of DRAM internal operations at the cost of increased bit-error rate [5, 7, 10, 17, 31, 34]. Approximate memory exploits this characteristic to reduce the memory access latency by leveraging the error robustness of applications themselves. To obtain reasonable outputs from applications on approximate memory, it is crucial to protect *critical data* while accelerating accesses to *non-critical data.* For example, suppose we want to accelerate a deep learning application using approximate memory. The matrices that express the weights of each layer are non-critical data because it is known that the accuracy of the trained model does not degrade much even when some bit-flips are injected into them [4, 14]. On the other hand, pointers from one layer of a network to another or the counter of the number of epochs are critical data. We must control the error rate of memory regions depending on the criticality of the data stored in them.

A limitation of approximate memory is that the error rate can be controlled only with the granularity of a few kilo bytes due to the internal structure of DRAM. We refer the minimum size of continuous data to which the same error rate must be applied to

[1]Data provided in [26] under CC BY 4.0 (https://creativecommons.org/licenses/by/4.0/).

as *approximation granularity*. The approximation granularity for a given DRAM module is decided by the *row size* of the module. A row is a sequence of data bits inside a DRAM module that are driven simultaneously to catch up with requests coming from a fast CPU. Because approximate memory we focus on is based on tweaking the timing of DRAM internal operations, the approximation granularity is equal to the row size. The row size of a DRAM module is in the range of 512 bytes to a few kilo bytes. For example, the row size of a module from Micron [20] is 2 KB, meaning that the approximation granularity of this module is also 2 KB.

The large approximation granularity makes it difficult to gain benefit from approximate memory for applications that have critical and non-critical data interleaved with a smaller granularity (e.g., 8 bytes). We refer to this problem as the *granularity gap problem*. This can happen when an application manages its data as an array of data structure that has critical members (e.g., pointers) and non-critical members (e.g., numbers whose small divergence do not affect the application's result). For a concrete example, suppose an application that traverses an array of graph nodes, and each graph node has pointers to its neighboring nodes and a score of it that is robust to bit-flips. The non-critical data of this application cannot be stored in approximate memory due to the difference between the approximation granularity and the granularity of interleaving of critical and non-critical data.

In this paper, we show the granularity gap problem is a significant concern in using approximate memory. In concrete, the contributions of this paper are summarized as follows:

(1) A source code analysis of widely used benchmarks to prove that many applications potentially suffer from the granularity gap problem, extended from our previous work [2].
(2) A discussion on pros and cons of a memory layout conversion technique in the context of the granularity gap problem.
(3) A framework to quantitatively evaluate the negative performance impact of the memory layout conversion technique.
(4) Evaluation results of the negative performance impact on widely used benchmarks, which proves the significance of the granularity gap problem in using approximate memory.

2 APPROXIMATE MEMORY ARCHITECTURE AND ITS LIMITATION

This section describes the background and the goal of this work. Our technical report [3] provides a more detailed explanation in its Section 2 and 3.

2.1 Overview of Approximate Memory

Approximate memory is a new technology to mitigate the performance gap between main memory and CPUs. The main idea is to reduce the latency of main memory accesses at a cost of the data integrity by exploiting *design margins* that exist in many DRAM chips today. The CPU may read a slightly different data from what has been written before to the main memory. A design margin refers to the difference between a design parameter defined in the specification of a device and the actual value which the device can be operated with. In particular, we focus on the design margin in the timing of internal (electrical) operations of DRAM. Even when wait-time parameters associated with some internal operations are shortened than the specification, many DRAM chips can read stored data "almost" correctly with a few bit-flips (errors) injected to the data [5, 13]. By controlling the timing of internal operations of DRAM, we can trade reduced main memory access latency with increased bit-error rate.

Approximate memory is especially beneficial for machine learning, multimedia, and graph processing applications, all of which incur many memory accesses and are tolerant to noisy data. For example, Stazi *et al.* [28] show that allocating data in approximate memory for the x264 video encoder can yield acceptable results, and our previous work [1] show that a graph-based search algorithm (mcf in SPEC 2006) can yield the same result as error-free execution even when some bit-flips are injected. Regarding the performance improvement, Koppula *et al.* [14] show 8% speedup in average for training various DNN models on approximate memory, and Lee *et al.* [15] show that using Adaptive-Latency DRAM [16] for approximate memory gives 7% to 12% speedup in average for *"32 benchmarks from Stream, SPEC CPU 2006, TPC and GUPS"* (they do not show numbers for each benchmark though). The performance improvement of a few to 10+ percent is important to these applications because they are typically executed in large scale data centers, where only a few % of relative efficiency improvement results in a huge reduction of energy and/or runtime in total.

Even for applications that can tolerate noisy input and intermediate data, they have critical data that must be protected from bit-flips. For example, deep learning is known to be robust to bit-flips [4, 9, 14] but not all parts of the data are robust to them. Pointers from one layer of a network to another or the loop counter that counts the number of epochs must be protected from bit-flips. Protecting critical data requires two steps:

(1) Detecting which parts of data are critical and which parts are non-critical
(2) Storing non-critical parts of data into approximate memory while storing the critical parts to normal memory

For step (1), there have been much effort [1, 19, 23] and it is out of the scope of this work, so we assume that discrimination of critical and non-critical data is given. For step (2), we must map the critical and non-critical data into different memory regions operated with different timing parameters. We show in the next section that this is challenging due to how DRAM is implemented.

2.2 The Granularity Gap Problem

A limitation of approximate memory exploiting design margins in timing parameters is that the *approximation granularity* cannot be smaller than a few kilo bytes. The approximation granularity refers the minimum size of a continuous memory region to which the same error rate must be applied. This is because the same timing parameter is applied to an entire "row" of data bits [12] and the size of a row is as large as a few kilo bytes (e.g., 2 KB in a 16Gb SAMSUNG chip [27]). This stems from a fundamental constraint that many bits must be driven in parallel so that slow DRAM can catch up with the high rate of requests coming from the CPU.

A challenge in using approximate memory is the gap between the approximation granularity and the granularity at which critical and non-critical data are interleaved. We call this problem the

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

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

Figure 2: Critical and non-critical data interleaved in a single C struct: it is not possible to protect the critical data while storing the non-critical data on approximate memory due to a large approximation granularity (e.g., 2 KB).

granularity gap problem. We say critical and non-critical data are *interleaved* when they co-locate inside one instance of a C `struct` or a C++ `class`. Figure 2 shows an example of interleaved critical and non-critical data. The data structure `struct node_t` contains both critical and non-critical data, and a pointer named `nodes` points to an array of `struct node_t`. To gain benefit from approximate memory for this code, we must protect the critical data (`id`, `r`, and `l`) while storing the non-critical data (`score`) into approximate memory. This is not possible because the approximation granularity is as larger as a few kilo bytes (say 2 KB), while we need to enable or disable approximation with a granularity of 4 bytes to achieve it.

The granularity gap problem has been overlooked by the research community because it is not relevant to applications that have large chunks of non-critical data. For example for deep learning applications, the non-critical data are matrices storing the weights of a network whose sizes range from a few kilo bytes to hundreds of mega bytes. In this case, we can store entire matrices into approximate memory and the approximate granularity is not an issue.

The goal of this paper is to prove the significance of the granularity gap problem with quantitative evidence. First, we show that there are many applications that potentially suffer from this problem. Second, more importantly, we show that avoiding this problem with a known technique has negative performance impact that is as large as almost canceling the benefit of approximate memory.

3 SOURCE CODE ANALYSIS

To show that many real applications can potentially suffer from the granularity gap problem, we analyze source code of widely used benchmarks in this section.

3.1 Analysis Methodology

For a given application, we find if the data structure that can obtain benefit from approximate memory has critical and non-critical data interleaved. Because approximate memory is the most effective when an application's data that incur many cache misses are stored on it, we focus our analysis on a data structure that incurs the largest number of cache misses within an application. We refer to such a data structure as *the most cache-unfriendly data structure.* After finding such a data structure, we analyze it to estimate if the application potentially suffers from the granularity gap problem.

To find the most cache-unfriendly data structure of an application, we first 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., a cache miss) occurs and the time it is recorded [32]. The small skid enables pinpointing which instruction in an application binary causes many hardware events. We execute a benchmark with its sample dataset using linux `perf`, and the actual command line is '`perf record -e r20D1:pp -- b`'. The parameter `r20D1:pp` specifies a performance event that counts the number of L3 misses [11]. The parameter `b` is replaced by an actual command line to execute each benchmark.

After measuring the number of per-instruction cache misses, we find the data structure accessed by this instruction, which is the most cache-unfriendly data structure of this application. Due to the lack of off-the-shelf tools to disassemble an arbitrary binary into C/C++ source code, we rely on human knowledge and labor to do this. One can refer to our previous work [2] for detailed examples of the manual analysis.

3.2 Experimental Setup

Table 1: Analyzed Benchmarks (SPEC CPU 2017)

Name	Domain	Cache Miss Rate
deepsjeng_r	game AI (chess)	77.5 %
nab_r	molecular modeling	64.9 %
omnetpp_r	discrete event simulation	56.1 %
namd_r	molecular dynamics	50.4 %
lbm_r	fluid dynamic	48.8 %
x264_r	video encoding	47.3 %
mcf_r	optimization	43.5 %
gcc_r	c compiler	36.6 %
blender_r	image processing	35.0 %
xz_r	data compression	31.6 %
perlbench_r	perl interpreter	21.4 %

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)

Table 1 describes the benchmarks we analyze. Each line shows a benchmark's name, its domain, and the cache miss rate measured by the linux `perf` tool. From SPEC CPU 2017, we analyze benchmarks whose cache miss rates are more than 20 %. 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 memory layout conversion technique we discuss in Section 4 is mainly researched for programs written in C. We include ones written in C++ because the difference between C++ and C (classes, templates, and some new syntax) do not affect the applicability of the memory layout conversion technique.

Table 3: Results of Source Code Analysis (S: is a C struct or a C++ class, P: has a pointer, F: has a fp, I: has an integer)

Benchmark	Data Type	S	P	F	I
deepsjeng_r	ttentrty_t[]	✓			✓
nab_r	INT_T[]				
omnetpp_r	sVector	✓	✓	✓	✓
namd_r	CompAtom[]	✓		✓	✓
lbm_r	double[]				
x264_r	uint8_t[]				
mcf_r	arc[]	✓	✓		✓
gcc_r	-				
blender_r	VlakRen[]	✓	✓	✓	✓
xz_r	uint8_t[], uint32_t[]				
perlbench_r	char[]				

Table 2 shows the machine we use to execute the benchmarks. We use the largest data set provided by the benchmarks (`refrate`). The LLC miss rate is measured using the linux `perf` tool.

3.3 Results

Table 3 shows the analysis results. Each row shows a benchmark, the most cache-unfriendly data structure, flags that represent the kinds of members that the data structure contains:

- **S**: the data is either a C `struct` or a C++ `class`.
- **P**: the data structure contains a pointer.
- **F**: the data structure contains a floating pointer number.
- **I**: the data structure contains an integer.

The data type column is denoted by `[]` if the data is managed as an array of that data type. We regard any type compatible with an integer (e.g., `char`, `long`) as an integer. If a class inherits other classes, we include the members of the parent classes as well because an instance of a child class in the memory contains all members of the parent classes. We exclude static members and member functions because they are not stored in the memory region allocated for each instance. We do not show the result for `gcc_r` because cache misses are scattered across many instructions. Two data types are shown for `xz_r` because two instructions incur almost the same number of cache misses. For all the benchmarks, the instruction that incurs the largest number of cache misses existed in their own code and not in any standard C/C++ libraries.

The results show that many applications potentially suffer from the granularity gap problem. The most cache-unfriendly data structure is either a C `struct` or a C++ `class` in 5 out of 11 benchmarks in SPEC CPU 2017. Combined with the same analysis applied to SPEC CPU 2006 in our technical report [3], we conclude that many applications "potentially" suffer from the granularity gap problem.

3.4 Drawbacks of the Methodology

Manual effort to find the data type accessed by a given instruction incurs a scalability issue and increases the chances of analysis errors. There are two error patterns stemming from the manual effort:

(1) Mis-identifying the variable in the source code that corresponds to a given memory access instruction

```
struct {
  double x;
  double y;
} points[N];

// calculate the center
double center_x = 0, center_y = 0;
for(i = 0; i<N; i++) {
  center_x += points[i].x / N;
  center_y += points[i].y / N;
}
```

Figure 3: Example of an array of structures.

```
struct {
  double x[N];
  double y[N];
} points;

// calculate the center
double center_x = 0, center_y = 0;
for(i = 0; i<N; i++) {
  center_x += points.x[i] / N;
  center_y += points.y[i] / N;
}
```

Figure 4: Example of a structure of arrays.

(2) Mis-identifying the type of data that is stored in the identified variable in source code

Pattern (1) can happen when the application binary has complex data/control flows for example with multiple levels of indirection (e.g., a->b->c) or when the binary does not look similar to the source code due to compiler optimizations. Pattern (2) can happen when the declared type of a source variable and the type of actual data stored in it are different (i.e., polymorphism). Developing compiler support to reduce the possibilities of these errors is future work.

Another concern for our analysis arises when a member variable of a C `struct` or a C++ `class` is passed to a function by reference. For example, when the same function is called in two contexts by passing the pointer to a member of different C `struct` (e.g., `&s1.v` and `&s2.v` where `s1` and `s2` are of different types), the same instruction can access memory regions inside different data types. In this case, our analysis may additionally require an investigation of stack traces and points-to analysis [29]. However, we did not hit this case in any of the benchmarks in our experiments.

4 MEMORY LAYOUT CONVERSION

This section discusses an applicability of a memory layout conversion technique to avoid the granularity gap problem, and points out that it can degrade the performance for some applications. We show in Section 5 that this performance overhead is as large as almost canceling the benefit of approximate memory in some cases.

4.1 AoS to SoA Conversion

An array of structures (AoS) can be converted into a structure of arrays (SoA) without changing the results of an application. Given an array of C `struct` instances, this technique converts the memory layout of an application so that each member of the C `struct` is stored as a distinct array. Figure 3 and Figure 4 show

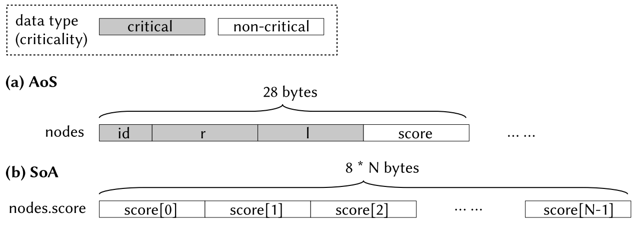


Figure 5: The change of memory layout when the AoS to SoA conversion is applied to the code in Figure 2.

an example of this conversion done explicitly by hand. The code in Figure 3 manages data as an AoS, while Figure 4 shows the converted version that manages each member of the data structure, `x` and `y`, as a distinct array. Code to access the data are converted for example from `points[i].x` to `points.x[i]`.

Besides the technical difficulties that have been tackled by many researchers (e.g., how to find pointer aliases, how to apply it dynamically to programs without the source code, how to ensure safety in weakly typed languages), a fundamental limitation of the AoS to SoA conversion is that there is no method to mathematically calculate its effect on performance. Petrank *et al.* [24] show that predicting the number of cache misses that a given data layout generates for an arbitrary memory access pattern is NP regarding the number of data objects. This means that one must either do exhaustive experiments for memory access patterns under interest or use heuristics to informally estimate the performance implication. This limitation leads us to do the former to evaluate its performance overhead in a later section.

4.2 Pros: Mitigate the Granularity Gap Problem

The AoS to SoA conversion enables using approximate memory even when critical and non-critical data are interleaved by avoiding the granularity gap problem. Because each member of the converted data structure is stored in a distinct array, it can be mapped to a designated DRAM row that has the appropriate timing parameter for the criticality of that member.

Figure 5 depicts how we can selectively store non-critical data of the code in Figure 2 to approximate memory. Gray boxes in the figure show critical data and white boxes show non-critical data. In the original code that manages the data as an AoS, it is not possible to selectively protect the critical data while accelerating accesses to the non-critical data because of the granularity gap problem (Figure 5 (a)). In the converted code that manages the data as a SoA, the non-critical data (`score`) consists a distinct array and it can be mapped directly to approximate memory, while the critical data (`id`, `r`, `l`) can be mapped to normal memory (Figure 5 (b)).

4.3 Cons: Negative Impact on Performance

The disadvantage of the AoS to SoA conversion is that it can degrade the performance due to increased number of cache misses. In Figure 2, it is highly possible that all the members of the same `struct` instance (that is, for any `i`, `nodes[i].id`, `nodes[i].r`, `nodes[i].l`, and `nodes[i].score`) share the same cache line. Thus, accessing more than two members of the same `struct` instance closely in time incurs at most 1 cache miss. However, if we apply the AoS to SoA conversion to the same code, members that are in the same `struct` instance in the original code do not share the same cache line. This might increase the number of cache misses and degrade the performance depending on the memory access pattern to the data to be converted.

For example, imagine a program that traverses a graph managed by the data structure in Figure 2 and it decides which child of the current node (either `l` or `r`) to visit depending on the `score` of the current node. Applying the AoS to SoA conversion to this program increases the number of cache misses because members of one graph node are stored in different cache lines in the AoS version, while they share the same cache line in the SoA version.

5 EVALUATION OF PERFORMANCE IMPACT

The negative performance impact of the AoS to SoA conversion explained in Section 4.3 is a serious concern if it cancels or outperforms the benefit of approximate memory. This section introduces a new methodology to quantitatively analyze the slowdown given by the AoS to SoA conversion, and shows that it is as large as almost canceling the benefit of approximate memory in the worst case.

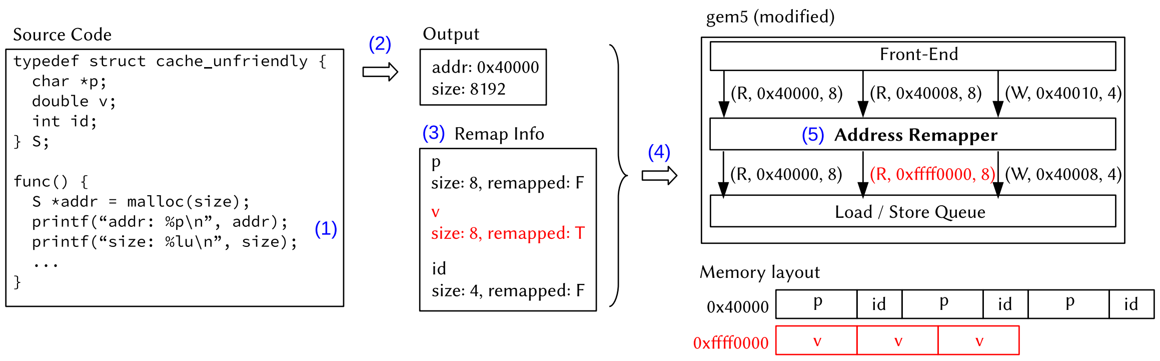
5.1 Pseudo Conversion by CPU Simulator

To quantitatively analyze the slowdown and show its significance, we estimate the effect of memory layout changes incurred by the AoS to SoA conversion by reproducing the memory layout that it would generate from applications' viewpoints using a cycle accurate simulator. Figure 6 shows how our framework works:

(1) The source code of the target application is annotated to print the starting addresses and the sizes of memory regions that contain the most cache-unfriendly data structure.
(2) The target application is executed on a vanilla simulator to gain the starting addresses and the sizes printed by the annotations added in step (1).
(3) The *remap info* that decides which members of the `struct` are stored in distinct arrays is defined. The remap info contains the size of each `struct` member and a boolean value that represents if it is stored into a distinct array (we say that a member is *remapped* if this value is `true`).
(4) A simulation is started on our modified simulator with information obtained in step (2) and step (3).
(5) While in a simulation, the target addresses of memory access instructions are investigated. If the target address points to a remapped member, it is converted to reproduce the memory layout that the AoS to SoA conversion would generate.

The address conversion is done in the *address remapper* in Figure 6 when the front-end of the CPU inserts requests into the load store queue. This is because the border between the front-end and the load store queue is a place right after an accessed address is determined and right before it is used. Therefore, converting addresses at this point prevents cache consistency problems.

Three requests are passed from the front-end to the address remapper in Figure 6: (1) an 8-byte read request to `0x40000`, (2) an 8-byte write request to `0x40008`, and (3) a 4-byte read request to `0x40010`. From the starting address of the memory region that contains the most cache-unfriendly data structure and the remap



```
typedef struct cache_unfriendly {
  char *p;
  double v;
  int id;
} S;

func() {
  S *addr = malloc(size);
  printf("addr: %p\n", addr);
  printf("size: %lu\n", size);
  ...
}
```

Figure 6: Simulation framework to estimate the negative performance impact of the AoS to SoA conversion.

info, the address remapper can find that the first request reads the member p. Because the remap info specifies that p is not remapped, the request is passed as-is to the load store queue. The second request accesses the member v that is remapped. Its target address is converted into an unused address (`0xffff0000` in the figure). The third request accesses the member `id`. Although it is **not** remapped, its address is shifted by 8 bytes to `0x40008` because the previous member v is remapped "away". As a result, the memory layout from the application's point of view is converted as shown in the figure. The member v consists a distinct array and the other members are packed as if there is no v in-between.

5.2 Discussion: Use of Simulator

There are efforts to estimate how memory layout conversion speeds up applications [21, 33] by investigating their memory access traces without applying the memory layout changes themselves (unlike our method). They measure the access frequencies to `struct` members and the access affinities between them from a memory trace of unmodified source code, and suggest which members should be placed closer in memory or separated to different memory regions. However, we cannot directly leverage this method for our purpose because they do not quantitatively estimate the performance impact of the suggested layouts. A difficulty is that memory layout conversion has two effects of opposite directions: (1) **slowdown** caused by increased number of cache misses due to separation of members with strong affinities, and (2) **speedup** caused by decreased size of members that are not separated as distinct arrays.

A challenge of actually applying the SoA to AoS conversion in the compiler-level stems from the fact that pointers can contain any addresses in C/C++ and the values held by pointers cannot be decided by a static analysis. This makes it difficult to robustly implement the conversion in the compiler level although theoretically possible by points-to analysis [29]. In contrast, because our method converts memory addresses inside a simulator at runtime, it is straight-forward to find the address that a pointer contains.

A disadvantage of our method is that a cycle accurate simulation is needed for every single conversion pattern. This is not always possible because the number of memory layout conversion patterns increases exponentially to the number of members in the most cache-unfriendly data structure. On the other hand, if can we somehow estimate the slowdown only from access frequencies and affinities, we can estimate the slowdown of all conversion patterns at once because the access frequencies and affinities can be obtained by one execution of a non-modified application.

Table 4: Simulated Environment

ISA	x86_64
Frequency	3 GHz
Issue Width	8
Reorder Buffer	192 entries
L1 cache	32 KB, 2 way, 32 MSHRs, 2 cycles/miss
L2 cache	2 MB, 8 way, 32 MSHRs 20 cycles/miss
Mem Ctrl Latency	75 ns

5.3 Experimental Setup

Table 4 shows the simulated environment. The "Mem Ctrl Latency" shows the length of time between a point when the CPU sends a request to the memory controller and a point when it receives the response. The memory access latency from software point of view additionally contains the time it takes to miss the caches, which is 7.3 ns (= (2 + 20) cycles × $\frac{1}{3}$ ns per cycle) and makes up a total of 82.3 ns. We use version 20.0.0.0 of gem5 and its SE mode, which emulates syscalls and requires no OS simulation. The benchmarks are compiled by gcc 8.3.0 (Debian 8.3.0-6). We fast-forward the initialization phase of each benchmark with the `AtomicSimpleCPU` model to only emulates the ISA. After that, we simulate a fixed length of time (0.2 seconds in the simulated world) using the `DerivO3CPU` model to faithfully simulates an OoO CPU.

We evaluate three benchmarks from SPEC CPU 2017, namely `mcf_r`, `deepsjeng_r`, and `namd_r`. For each benchmark, we test every possible memory layout conversion pattern and compare the performance. Let N be the number of members in the most cache-unfriendly data structure in each benchmark, we test all 2^{N-1} cases of remapping. We exclude `blender_r` and `omnetpp_r` although their most cache-unfriendly data structures are C++ `class`. The source code of `blender_r` does not have clear separation between the initialization phase and the main computation, and `omnetpp_r` has 21 members in its most cache-unfriendly data structure (`sVector`) and it is not possible to test 2^{20} possibilities.

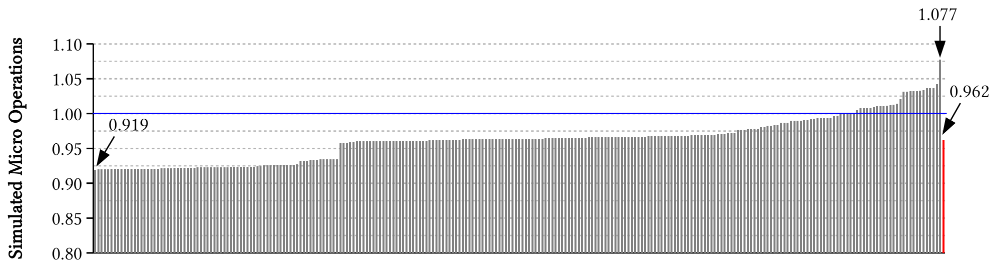


Figure 7: Evaluation result for mcf_r

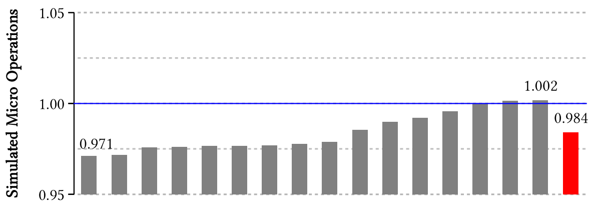


Figure 8: Evaluation result for deepsjeng_r

5.4 Results

Figure 7 and Figure 8 show the evaluation results. Each bar corresponds to a memory layout conversion pattern and each graph has $2^{N-1} + 1$ bars, where N is the number of members of the most cache-unfriendly data structure. The right-most bar shows the average of all patterns. The y values show the number of executed micro operations during simulation normalized to the value when memory layout conversion is not applied. The bars are ordered by their y values. Lower bars show larger negative performance impact (we simulate a fixed length of time in the simulated world).

mcf_r: Among the 256 memory layout conversion patterns, 229 patterns yield worse performance than the no conversion case. The lowest performance is observed when the first three members are remapped to consist distinct arrays, and its performance is 8.13 % slower than the no conversion case. The average negative performance impact is 3.81 %.

deepsjeng_r The most cache-unfriendly data structure "wraps" an array of length 4 and each member of the array is another `struct` (e.g., `struct outer { struct inner array[4]; }`). The inner data structure (`struct inner`) has 5 members and we apply the same remapping policy to all 4 instances of the inner data structure (if array[0].d1 is remapped, array[i].d1 are remapped for any i $\in \{1, 2, 3\}$). The lowest performance is observed when the first and the fourth member of the inner data structure are remapped, with 2.90 % slowdown.

namd_r Among the 64 memory layout conversion patterns, the largest negative performance impact was 0.1 % in the worst case and the average performance impact was an improvement of 0.4 %. We provide a detailed graph in our technical report [3] in its Section 6.

The negative performance impact of the memory layout conversion to avoid the granularity gap problem is sometimes not negligible compared to the benefit of approximate memory. For example, Kim *et al.* [13] report that the average speedup of SPEC CPU 2006 benchmarks when the timing parameters are violated is around 4 - 5 % (Figure 8 of [13]). Note that their system does not reduce the latency to the extent that bit-flips are visible to the applications. Even if we assume that the performance gain by allowing bit-flips to be visible to the application is twice as large, it is almost canceled in the worst case by the performance overhead due to the granularity gap problem (8 - 10 % speedup vs. 8.13 % slowdown). Another research [30] report that their system can save up to around 12.5 % of overall (CPU + memory) energy consumption for `mcf` in SPEC CPU 2006 (Figure 7 (c) of [30]) by approximate memory. Saving energy is another benefit of approximate memory besides performance. If we assume that the negative performance of memory layout conversion to `mcf` is similar to the one to `mcf_r`[2], the 12.5 % gain is deducted by a non-negligible amount.

6 RELATED WORK

To the best of our knowledge, we are the first to study the granularity gap problem. One of the reasons is that it is not relevant when we consider storing only large arrays of numbers such as weight matrices of a neural network to approximate memory. However, as we point out in this paper, it is a significant problem for many realistic applications. Esmaeilzadeh *et al.* mention [8] about this problem a bit, but they provide no further investigation.

Nguyen *et al.* [22] propose a method that partially mitigates the granularity gap 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.* [18] 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 as we discuss in this paper. Chen *et al.* [6] propose a memory controller that maps data into different

[2]The code are similar and the most cache-unfriendly data structures are the same.

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.* [25] advance a previous work [18] 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 and use appropriate pages.

7 CONCLUSION

We investigated the granularity gap problem of approximate memory, which arises due to the difference between approximation granularity and the granularity of actual data criticality. Because the former is as large as a few kilo bytes in today's DRAM and the latter is often a few bytes, we cannot apply different protection levels to critical and non-critical data. We analyzed source code of SPEC CPU 2017 benchmarks and found that 5 out of 11 benchmarks potentially suffer from this problem. We also proposed a simulation framework to quantitatively analyze the negative performance impact to avoid this issue with a known technique, and conclude that the granularity gap problem is a significant concern and it requires more attention from the research community.

ACKNOWLEDGMENTS

This work was supported by JST, ACT-I Grant Number JPMJPR18U1, Japan.

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] Soramichi Akiyama. 2020. Assessing Impact of Data Partitioning for Approximate Memory in C/C++ Code. In *The 10th Workshop on Systems for Post-Moore Architectures (SPMA)*. 1 – 7.

[3] Soramichi Akiyama and Ryota Shioya. 2021. *The Granularity Gap Problem: A Hurdle for Applying Approximate Memory to Complex Data Layout.* Technical Report. 13 pages. arXiv:2101.10605.

[4] N. Chandramoorthy, K. Swaminathan, M. Cochet, A. Paidimarri, S. Eldridge, R. V. Joshi, M. M Ziegler, A. Buyuktosunoglu, and P. 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] J. Choi, W. Shin, J. Jang, J. Suh, Y. Kwon, Y. Moon, and L. Kim. 2015. Multiple Clone Row DRAM: A low latency and area optimized DRAM. In *International Symposium on Computer Architecture (ISCA)*. 223–234.

[8] 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.

[9] Kamyar Givaki, Behzad Salami, Reza Hojabr, S. M. Reza Tayaranian, Ahmad Khonsari, Dara Rahmati, Saeid Gorgin, Adrian Cristal, and Osman S. Unsal. 2020. On the Resilience of Deep Learning for Reduced-voltage FPGAs. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 110–117.

[10] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu. 2016. ChargeCache: Reducing DRAM latency by exploiting row access locality. In *International Symposium on High Performance Computer Architecture (HPCA)*. 581–593.

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

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

[13] Jeremire Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. 2018. Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines. In *IEEE International Conference on Computer Design (ICCD)*. 282–291.

[14] Skanda Koppula, Lois Orosa, A. Giray Yağlıkçı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. 2019. EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM. In *International Symposium on Microarchitecture (Micro)*. 166–181.

[15] Donghyuk Lee, Samira Khan, Lavanya Subramanian, Saugata Ghose, Rachata Ausavarungnirun, Gennady Pekhimenko, Vivek Seshadri, and Onur Mutlu. 2017. Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, Article 26 (June 2017), 36 pages.

[16] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. 2015. Adaptive-latency DRAM: Optimizing DRAM timing for the common-case. In *Int'l Symposium on High Performance Computer Architecture (HPCA)*. 489–501.

[17] Y. Lee, H. Kim, S. Hong, and S. Kim. 2017. Partial Row Activation for Low-Power DRAM System. In *International Symposium on High Performance Computer Architecture (HPCA)*. 217–228.

[18] 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.

[19] Abdulrahman Mahmoud, Radha Venkatagiri, Khalique Ahmed, Sasa Misailovic, Darko Marinov, Christopher W. Fletcher, and Sarita V. Adve. 2019. Minotaur: Adapting Software Testing Techniques for Hardware Errors. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1087–1103.

[20] Micron. 2018. 16Gb, 32Gb: x4, x8 3DS DDR4 SDRAM Description. https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/16gb_32gb_x4_x8_3ds_ddr4_sdram.pdf?rev=77c8db7a371.

[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 Trans. on Circuits and Systems I: Regular Papers* (2020), 1–14.

[23] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. 2018. Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications. In *International Symposium on Microarchitecture (Micro)*. 750 – 762.

[24] Erez Petrank and Dror Rawitz. 2002. The Hardness of Cache Conscious Data Placement. In *Symposium on Principles of Programming Languages (POPL)*. 101–112.

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

[26] Karl Rupp. 2020. Microprocessor Trend Data. https://github.com/karlrupp/microprocessor-trend-data/.

[27] Samsung Electronics Co., Ltd. 2017. 8Gb C-die DDR4 SDRAM x16. https://www.samsung.com/semiconductor/global.semi/file/resource/2017/12/x16%20only_8G_C_DDR4_Samsung_Spec_Rev1.5_Apr.17.pdf.

[28] Giulia Stazi, Lorenzo Adani, Antonio Mastrandrea, Mauro Olivieri, and Francesco Menichelli. 2018. Impact of Approximate Memory Data Allocation on a H.264 Software Video Encoder. In *High Performance Computing. ISC High Performance 2018. Lecture Notes in Computer Science.* 545–553.

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

[30] K. Tovletoglou, L. Mukhanov, D. S. Nikolopoulos, and G. 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.

[31] Yaohua Wang, Arash Tavakkol, Lois Orosa, Saugata Ghose, Nika Mansouri Ghiasi, Minesh Patel, Jeremie S. Kim, Hasan Hassan, Mohammad Sadrosadati, and Onur Mutlu. 2018. Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration. In *Int'ml Symposium on Microarchitecture (Micro)*. 298 – 311.

[32] Vincent M. Weaver. 2016. *Advanced Hardware Profiling and Sampling(PEBS, IBS, etc.): Creating a New PAPI Sampling Interface.* Technical Report. Univ. of Maine.

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

[34] 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.