IEICE TRANSACTIONS

on Information and Systems

VOL. E102-D NO. 12
DECEMBER 2019

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER *Special Section on Parallel and Distributed Computing and Networking*

A Lightweight Method to Evaluate Effect of Approximate Memory with Hardware Performance Monitors*

Soramichi AKIYAMA[†a)], ***Member***

SUMMARY The latency and the energy consumption of DRAM are serious concerns because (1) the latency has not improved much for decades and (2) recent machines have huge capacity of main memory. Device-level studies reduce them by shortening the wait time of DRAM internal operations so that they finish fast and consume less energy. Applying these techniques aggressively to achieve *approximate memory* is a promising direction to further reduce the overhead, given that many data-center applications today are to some extent robust to bit-flips. To advance research on approximate memory, it is required to evaluate its *effect* to applications so that both researchers and potential users of approximate memory can investigate how it affects realistic applications. However, hardware simulators are too slow to run workloads repeatedly with different parameters. To this end, we propose a lightweight method to evaluate effect of approximate memory. The idea is to count the number of DRAM internal operations that occur to approximate data of applications and calculate the probability of bit-flips based on it, instead of using heavy-weight simulators. The evaluation shows that our system is 3 orders of magnitude faster than cycle accurate simulators, and we also give case studies of evaluating effect of approximate memory to some realistic applications.

key words: *approximate memory, computer architecture, memory systems*

1. Introduction

Performance of memory subsystems play a significant role to determine the overall performance of computers in two aspects. First, the energy consumption of memory subsystems is getting larger and larger as more and more memory capacity is required by data-center workloads such as artificial intelligence (AI), high performance computing (HPC), and big data analytics. Second, the memory access latency relative to the performance of CPU cores is getting larger and larger. This is because the floating point operations per second of cores have been increasing exponentially for decades, while the memory access latency has been almost the same [1], [2]. As a result, memory subsystems today are the largest concern both in terms of the energy consumption and the performance of large-scale computers.

Prior works have succeeded to lower the latency and energy consumption of main memory by reducing predefined wait parameters of DRAM internal operations so that they consume fewer amount of energy and take less time [2]–[7]. To further lower the latency and energy consumption, using these techniques more aggressively to achieve *approximate memory* is a promising direction. Approximate memory is a type of *approximate computing*, which realizes increased efficiency such as reduced energy consumption [8], larger capacity [9], and faster throughput [10] in the cost of computation accuracy. By aggressively applying the existing techniques [2]–[7], we can achieve main memory that does not guarantee data integrity but has lower latency and consumes less energy. It is expected to be useful for HPC applications that use iterative methods so that small numerical errors can be amortized, and for AI and big-data analytics applications whose data contain noises by nature.

To advance research on approximate memory, evaluating its *effect* that occurs to applications running on it is of great importance. First, it allows researchers of approximate memory to evaluate how their proposals affect applications. Second, it enables potential users of approximate memory to estimate how their applications behave on approximate memory. However, it is challenging due to three reasons:

1. There is yet no off-the-shelf approximate memory device that users can try with their own applications.
2. Device-level studies reveal the relationship between energy/latency reduction and error rate, but users cannot know how their own applications are affected.
3. Applications must be executed repeatedly with different error-rates, but cycle accurate simulators of hardware are too slow to run realistic applications repeatedly.

To this end, we propose a lightweight method to evaluate effect of approximate memory to given applications. The main idea is that we can calculate the probability that each bit is flipped by approximate memory from the number of DRAM internal operations that can be measured by a hardware performance monitoring mechanism rather than cycle accurate simulators. First, users specify data to which errors can be injected (referred to as *approximate data*) with our memory allocator. Then, our system measures the number of DRAM internal operations that occur only to the approximate data while the application is running. Our system calculates the probability that each bit of the approximate data is flipped from the measurement and the error-rate parameter, and periodically injects bit-flips to the approximate data. Finally, the application outputs a result that users can compare with the correct one to evaluate effect of approximate memory to their applications. Our system allows

Manuscript received January 2, 2019.
Manuscript revised May 28, 2019.
Manuscript publicized September 2, 2019.

[†] The author is with Department of Creative Informatics, The University of Tokyo, Tokyo, 113–8656 Japan.

[*] Part of this work was done while the author was with National Institute of Advanced Industrial Science and Technology (AIST).

a) E-mail: akiyama@ci.i.u-tokyo.ac.jp

DOI: 10.1587/transinf.2019PAP0012

users to execute same applications repeatedly with different error-rate parameters because it slows them down only by several times compared to native execution.

This paper is structured as follows. Section 2 introduces the background. Section 3 explains why evaluating effect of approximate memory is important and difficult. Section 4 describes our lightweight method and its implementation. Section 5 shows overhead analysis of our system and case studies. Section 6 gives some discussions. Section 7 reviews related work and Sect. 8 concludes this paper.

2. Background

2.1 Main Memory as Performance Bottleneck

Main memory is a large performance bottleneck of computers today. First, the energy consumption of main memory pressures the energy budget of a computer because AI, HPC, and big-data analytics applications require 100s of GB of memory. For example, AI Bridging Cloud Infrastructure (ABCI) [12] has 384 GiB of memory per node, and NVIDIA DGX-2 [13] has 1.5 TB of memory per node. As a result of installing large memory, even 25% – 40% of the energy of a machine is consumed by the main memory [14], [15].

Second, the memory access latency today affects application performance to the largest extent ever because it has not improve much for the last 20 years while the performance of CPUs have increased exponentially [1], [2]. For example, our previous work observed that the scalability of a memory-access intensive application is worse on a newer machine than on an old machine due to larger performance gap of the CPU and the memory [11]. Figure 1 shows the elapsed time of the Alternate Least Square (ALS) algorithm [16] on an old machine (left, Xeon E5-2603 with DDR3-1600) and a newer machine (right, Xeon E5-2699 **v3** with DDR4-2133) with different number of processes. It has many random accesses to an 8K × 8K matrix, thus memory access latency largely affects the performance. Each row is accessed independently and there is no inter-process communication. The "actual" values show the measured elapsed time, and the "ideal" values show the elapsed time with an assumption that the algorithm scales perfectly. The figure shows that the "actual" and "ideal" values differ much on the newer machine due to a larger memory/CPU performance gap.

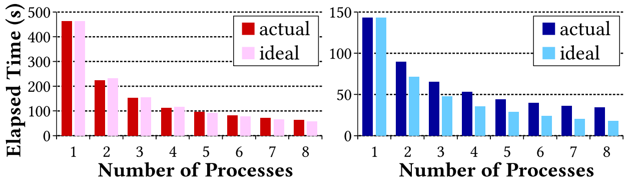


Fig. 1 Elapsed time of ALS algorithm on an old machine (left) and a newer machine (right), reproduced from Fig. 1 of [11]. It does not scale well on the newer machine due to larger memory/CPU performance gap.

2.2 DRAM Internal Operations

Main memory modules used for current computers are made of DRAM. A DRAM device has a hierarchical structure with multiple *channels*, *ranks*, *banks*, and *capacitor arrays*. In this paper, we focus on a single capacitor array for simplicity and better understanding without loss of generality.

Figure 2 shows electric operations that occur on a capacitor array when it is accessed. Capacitors in a capacitor array are organized as rows and columns, along with a sense amplifier (also known as a row buffer). Each row is connected with a wordline (WL), and each column is connected with a bitline (BL). A circle in the figure represents a capacitor, with a color describing the value of it. A black circle has a value of 1, a white circle has a value of 0, and a gray circle shows a capacitor is in the intermediate state. For each access, the capacitor array undergoes three operations [17]:

1. An *activation* enables the WL of a specified row, which makes the capacitors in the row connected to the BLs. The electric charge stored in the capacitors increase the voltage of the BLs from V_{ref} to a slightly higher value of V_{ref+}. At the same time, the electric charge leak from the capacitors and they become intermediate states (Fig. 2 (b)). The sense amplifier senses the slight increase of the voltage of the BLs and amplifies it to the voltage that represents the value of 1 (Fig. 2 (c)).
2. A *restoration* restores electric charge to the capacitors that have leaked in the activation. This operation is mandatory because a BL is long to cover many capacitors and the capacitance of a BL is much larger than that of a capacitor. Thus, slightly increasing the voltage of a BL requires a capacitor to leak much of its charge.
3. A *precharge* resets the voltage of all the BLs, WLs, and the sense amplifier to V_{ref} so that another row can be activated for the next access.

2.3 Existing Device-Level Techniques

DRAM internal operations described in Sect. 2.2 take much time and consume much energy. First, they dominate a large portion of the time that a random memory access takes. Typical memory access latency from software point of view

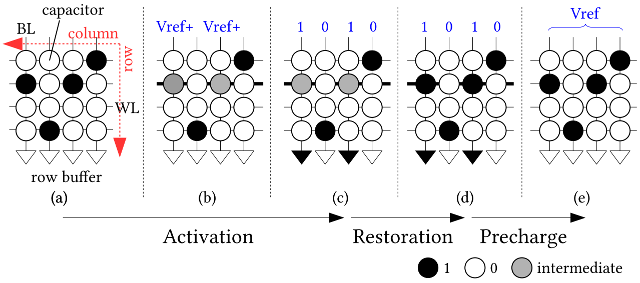


Fig. 2 DRAM Internal Operations: Activation, Restoration, Precharge

is 50 – 100 ns in modern Intel CPUs, and it is around 80 ns specifically in our machine described in Table 5 (measured by Intel Memory Latency Checker [18]). On the other hand, the activation latency, the restoration latency, and the precharge latency are defined as 12.5 ns, 22.5 ns, and 12.5 ns, respectively for DDR3-1600J standard. The sum of these latencies dominates 59.4% ($\approx \frac{12.5+22.5+12.5}{80} \times 100$) of a random memory access latency.

Second, the three operations also dominate a large portion of energy consumed by DRAM because these operations charge and leak electric charge to and from capacitors. Lee *et al.* [3] executed memory intensive workloads in a simulator and broke down the energy consumption of DRAM. Figure 2 of [3] shows that the sum of the energy consumed by the three operations (labeled as "ACT-PRE") consumes more than 20 % of the energy consumed by DRAM in average, and close to 40% for some workloads. Note that an activation and a restoration are combined and referred to as "a row activation" in their terminology, but how to count them (either combined or decoupled) does not matter here.

The fact that the activation, restoration, and precharge operations incur large overhead both in terms of energy and latency has motivated many prior works to reduce the overhead with novel device-level techniques. Chang *et al.* [2] give a thorough analysis on how reduced activation latency incurs bit-flips, and propose a flexible-latency DRAM system based on their insight. Wang *et al.* [4] shorten the restoration latency by predicting rows that will be accessed in the near future and applying shorter restoration time than defined for these rows because these rows will be fully charged when they are activated again for a future access. There are many other works that focus on reducing the latencies of DRAM internal operations [5]–[7] to achieve faster and energy efficient memory. Reducing the latencies also has a positive effect on the energy consumption because of two reasons: (1) Shorter execution time of applications achieved by lower memory access latency consumes smaller amount of energy, and (2) Reducing latencies means charging less electricity to capacitors, resulting in lower energy consumption.

2.4 More Aggressive Reduction: Approximate Memory

Approximate memory is an idea that embraces bit-flips inside main memory to achieve efficiency to the extent that conventional methods cannot achieve. This is particularly suitable for AI, HPC, and big-data analytics applications where the results are calculated by iterative methods and small numerical drifts caused by bit-flips can be amortized. For example, Fang *et al.* [19] showed that substituting 0s instead of true values referenced by broken pointers yields acceptable results in some HPC applications, and Liu *et al.* [20] showed that reducing the refresh rate of DRAM is acceptable for an optimization algorithm, a natural language parser, an mpeg2 decoder, a ray shader, and a turn-based game.

Given (1) that reducing the activation, restoration, and precharge latencies can greatly reduce the memory access latency and the energy consumption, and (2) that some applications are robust to bit-flips injected to their data, using the device-level techniques [2], [4]–[7] more aggressively to achieve approximate memory is a promising direction to further reduce the overhead. In this paper, we presume a use of the device-level techniques aggressively so that the integrity of data is no longer guaranteed in return of larger reduction of the latency and energy consumption. The approximate memory model we assume in this paper is as follows:

1. Main memory is divided into two regions: the *approximate region* and the *critical region* and they are applied different latency parameters. One possible way to implement this is to assign different sets of DRAM banks to each region (e.g. banks 1–4 for the approximate region, banks 5–8 for the critical region) and apply different latency parameters to these sets.
2. The integrity of data on the critical region is guaranteed by hardware, while data on the approximate region may be injected bit-flips when accessed. Applications that run on this memory place *approximate data* on the approximate region and *critical data* on the critical region. Critical data refers data whose integrity must be guaranteed (such as a program binary), and approximate data refers data in which a small number of bit-flips can be injected (such as weights of a neural network).
3. Bit-flips occur with a small probability every time a DRAM row in the approximate region is activated, restored, and precharged. Other operations such as accessing the critical region and touching registers do not incur bit-flips. The probability of bit-flips depends on the extent of latency reduction (a larger reduction comes with a larger error-rate).

We use a simple error model (assumption 3) that does not perfectly reflect every aspect of DRAM internals. For example, prior works show that bit-flips are not equally distributed but localized to weak bitlines for activations [21] and to weak rows for refreshes [22]. Even with the simple model, our method is advanced because it can use a real value for the average number of bit-flips. Existing methods either (1) are light-weight but use random error models [23], [24], or (2) use more realistic error model but rely on heavy-weight simulation [20], [25]. Our method allows more realistic evaluation than random error models with low overhead.

Given a specific bit-error rate, the amount of latency reduction that can be achieved within the bit-error rate hugely depends on which particular DIMM model is used for approximate memory. From real measurements, Chang *et al.* [2] confirmed that reducing the activation latency (tRCD) from 12.5 ns to 7.5 ns yielded a bit-error rate of 10^{-10} for some DIMM models, but the same tRCD resulted in a bit-error rate of 10^{-1} for others. Thus, this paper focuses on revealing the relationship between bit-error rates and application results, but we defer modeling the relationship between bit-error rates and application speedups for future work.

3. Evaluating Effect of Approximate Memory

3.1 Importance and Difficulties

Evaluating *effect* of approximate memory that occurs to applications running on it is important both for researchers of approximate memory and potential users of it. It allows researchers of approximate memory to investigate how their proposals affect applications so that they can be confident of their work. It also enables potential users of approximate memory to estimate how their applications behave on it so that they can rationally consider an use of it.

To this end, we need an easy-to-use method to evaluate effect of approximate memory. First, approximate memory devices are still being actively researched, and there is no off-the-self device that users can try. Thus, we need a method that requires no real hardware. Second, device-level studies show the relationship between (1) their latency/energy reduction and introduced error rates and/or (2) their latency/energy reduction and the effect to some applications used for their evaluation. However, potential users of approximate memory cannot know effect of it to *their own applications*.

Approximate memory imposes multiple types of effect to applications. Table 1 describes the three types of effect (*crash*, *endless execution*, *drifted result*). What makes the situation more complex is that different types of effect may occur alternately even for the same application with the same error rate. For example, a numerical application may experience a crash, a drifted result with an acceptable amount of drift, and a drifted result with an un-acceptable amount of drift for three different runs with the same error rate.

3.2 Requirement

Evaluating effect of approximate memory requires to run same applications repeatedly with different *bit-error rates* and *data separation patterns*. Bit-error rate refers the probability that a bit is flipped for each DRAM operation (row activation, restoration and precharge in our scenario). The bit-error rate is controllable by changing configurations of underlying approximate memory devices. For example, larger bit-error rate reduces more energy and achieves lower latency, but it obviously increases the bit-error rate. Therefore, it is important to find bit-error rate that gives the best efficiency with acceptable effect to a given application.

A data separation pattern refers which part of data to store in the approximate region and which part to store in the critical region. For example, matrix-matrix multiplication has three matrices as its data. Because each matrix has different memory access pattern, the effect to the final results can differ depending on which one among the matrices (or what combination of them) to store in the approximate region. It is needed to find the best data separation pattern that leverages approximate memory the most and yields the smallest extent of effect.

Table 1 Types of Effect caused by Approximate Memory

Crash	Applications crash due to fatal errors such as defererencing invalid pointers or applying arithmetic operations to NaNs.
Endless Execution	Iterative algorithms such as graph traversal and path search do not finish if the on-memory data is changed to unexpected states.
Drifted Result	Applications finish safely, but the results are different from the correct ones.

3.3 Use of Cycle Accurate Simulators

A naïve way to evaluate effect of approximate memory is to use cycle accurate simulators. They simulate internal mechanisms of a target machine such as CPUs and memory devices for each cycle (a.k.a. clock). Because they are software-based, users can modify them to mimic approximate memory devices and probe the internals with lighter burden than preparing a real hardware.

An issue of using cycle accurate simulators is that they are extremely slow due to their accurate simulation. Figure 3 shows slowdown of benchmarks executed on a cycle accurate simulator relative to cases when they are executed on a bare-metal machine. As a representative of cycle accurate simulators, we used `gem5` [26]. We used `DerivO3CPU` and `DRAMCtrl` included in `gem5` to simulate an out-of-order CPU and DDR3 memory, respectively. Other parameters of the simulated machine are in Table 2. The SE simulation mode delegates system calls to the host machine and only simulates an userspace environment. The x axis indicates benchmarks included in SPEC CPU 2006 Benchmarks, and the y axis shows the relative slowdown. We used the

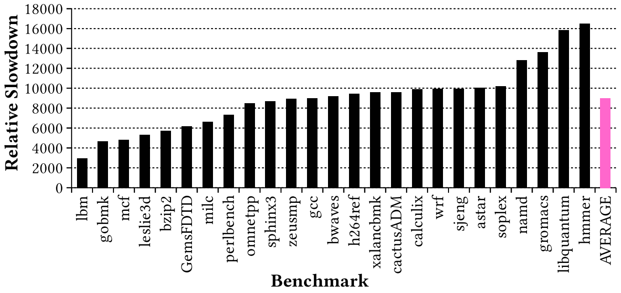


Fig. 3 Slowdown of SPEC CPU 2006 benchmarks executed on gem5

Table 2 Details of the Simulated Machine

Core	x86 ISA, Out-of-order execution, 1 GHz
L1 Cache	16 KB data + 16 KB instruction, 2 cycles for a miss
L2 Cache	256 KB, 20 cylces for a miss
Memory	DDR3-1600, 2 ranks, 8 banks/rank
Simulation	System-call Emulation (SE) mode
OS	Debian 9.5 (shared with the host machine)

smallest dataset called `test`. The result shows that executing benchmarks on `gem5` takes 16500× longer time than native execution in the worst case, and 9000× longer time in average. Cycle accurate simulators are too heavy to evaluate effect of approximate memory and a lightweight alternative is needed.

3.4 Use of Other Software-Based Techniques

PIN [27] and LLVM [28] are two often used software-based techniques to inject errors into applications to investigate their error-robustness. PIN is a dynamic binary instrumentation tool that can intercept, investigate, and modify program instructions at runtime. In the context of this paper, it can be used to inject errors into operands of instructions including both registers and memory locations. However, in order to consider internal states of DRAM such as the number of activations, it has to be modified to simulate them to know if an activation occurs for each memory operand. This is not an easy task because internal states of DRAM depends not only on DRAM commands issued in the past, but also the time they have been issued.[†]

LLVM is a compiler suite that is designed to be highly modular and easy to reuse its components. For example, Wei *et al.* [29] use LLVM to inject errors in the IR (Intermediate Representation) level. Their method also injects errors into operands of instructions, but is superior to PIN-based methods in the sense that mapping an error-injected instruction into its corresponding line in the source code is easier. The issue of LLVM-based methods is the same as PIN-based methods; they cannot consider internal states of DRAM unless detailed simulation is implemented.

We discuss the details of existing PIN- and LLVM-based methods in Sect. 7 to clarify the differences between our work and these methods.

4. Proposal: A Lightweight Method

4.1 Overview of the System

We propose a lightweight method to evaluate effect of approximate memory to given applications with only several times of slowdown compared to native execution. The main idea is to calculate the probability that each bit in approximate data is flipped from the number of DRAM internal operations that can be measured on a bare-metal machine. We enable this by assigning a designated NUMA node for approximate data and measure the number of operations for that NUMA node. The overview of our method is as follows:

1. Users allocate approximate data of their applications using our memory allocator. The approximate data is placed on a different NUMA node from critical data.

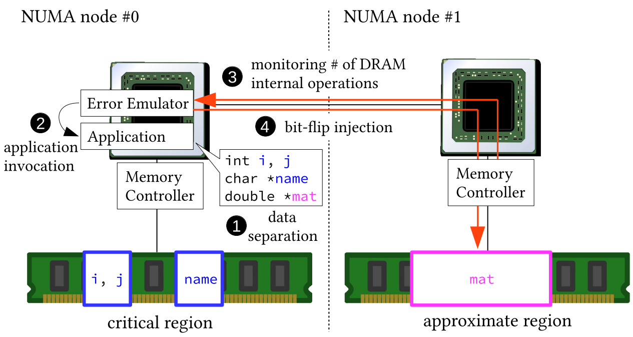


Fig. 4 Evaluating effect of approximate memory using our system.

2. A target application is executed *on a bare-metal machine* with our error emulator attached.
3. While the application is running, the error emulator counts the number of DRAM internal operations on the NUMA node where the approximate data is placed.
4. The error emulator calculates the probability that a bit is flipped from the number of DRAM internal operations and inject bit-flips to the approximate data.

Figure 4 shows how our system works. First, the user specifies which data of the target application is approximate data (❶). We provide a special memory allocator for this. The memory allocator stores the approximate data to a designated NUMA node at run time. The details of the memory allocator are described in Sect. 4.2. Second, the error emulator invokes the target application (❷). The error emulator takes the bit-error rate per DRAM internal operation as a parameter. Third, the error emulator monitors the number of DRAM internal operations on which the approximate data is placed (❸). In Fig. 4, it monitors the DRAM internal operations of NUMA node #1. The implementation details of the monitoring mechanism are shown in Sect. 4.3. Fourth, the error emulator periodically suspends the target application and injects bit-flips into the approximate data (❹). The target application is suspended every time its memory IO after the previous suspension reaches a threshold, which is given to the error emulator as its parameter. We use the amount of memory IO to define the interval because it is easier to set than directly using the number of DRAM internal operations. While the target application is suspended, the error emulator injects bit-flips that would have occurred during the time between the previous and the current suspension (we refer this time period to as an *epoch*). The algorithm to calculate the probability that each bit is flipped is explained in Sect. 4.4. After injecting bit-flips finishes, the target application is resumed with some bit-flips injected to its approximate data. The procedures ❸ and ❹ are repeated until the target application finishes.

4.2 Memory Allocator

Figure 5 shows the programming interfaces of our memory allocator. It is called `bmalloc`, with the prefix “b” standing for <u>b</u>it-flipping. `b_init` is called once at the beginning of

[†] An activated row is closed in less than 10 micro seconds for refresh operations (that use the same row buffer) of other rows.

```
// For initialization
void b_init(double error_rate, uint64_t threshold);

// Data attribute
typedef struct {
  char type; // 'F' or 'I'
  size_t size;
} b_attr;

b_attr default_attr = { .type = 'I', .size = 4 };

// Memory allocation/deallocation functions
void *b_malloc(size_t size, b_attr *attr);
void *b_calloc(size_t nmemb, size_t size, b_attr *attr);
void *b_realloc(void *ptr, size_t size, b_attr *attr);
void b_free(void *ptr);
```

Fig. 5 Programming Interfaces of Our Memory Allocator

```
//-- Without data separation -----------------
double *mat = malloc(sizeof(double) * SIZE);

for(i=0; i < SIZE; i++) {
  mat[i] = ...
}


//-- With coarse-grained data separation -----
// Approximate data is allocated with b_malloc
double *mat = b_malloc(sizeof(double) * SIZE, &attr);

for(i=0; i < SIZE; i++) {
  // the same as the original code
  mat[i] = ...
}
```

Fig. 6 Example of coarse-grained data separation.

a target application to specify the bit-error rate per DRAM internal operation and the threshold to determine the interval of error injection. `b_malloc`, `b_calloc`, and `b_realloc` are used to allocate memory regions for approximate data. They are used in the same ways as the normal `malloc`, `calloc`, and `realloc` functions except that they take the data attribute (`*attr`) as an additional parameter. A data attribute is expressed with a struct `b_attr` that includes the type of data (either ‘F’ or ‘I’) and the size of each element of the data. When the `type` field of an attribute is ‘I’, bit-flips are injected with the equal probability among all bits of the data and the `size` field does not matter in this case. When the `type` field is ‘F’, it implies that the data stored in this region are floating point numbers. In this case, when a bit-flip occurs to an element of this region, a random floating point number with the size specified by the `size` field is assigned. This mechanism prevents a floating point number from being changed to a NaN by flipping arbitrary bits of it. If users want to emulate this phenomenon as well, the `type` field can be set to ‘I’ even when the stored data are floating point numbers. Note that how to *prevent* this type of serious errors is related but a different story.[†]

Figure 6 and Fig. 7 show usages of `bmalloc`. There are two types of usages depending on the way approximate data is used in the original source code.

1. *Coarse-grained separation* is applied to applications whose approximate data is continuous (e.g. a large matrix). Figure 6 shows an example. It requires modification of the source code that allocates approximate data, but accessing the approximate data is done by the same way as in the original source code.
2. *Fine-grained separation* is applied to applications whose approximate data is embedded in a larger data hierarchy. For example, when particular members of a `struct` are stored into the approximate region, fine-grained data separation is used. Figure 7 shows an example. In addition to modifying the allocation code of approximate data, it requires to modify lines of code that access approximate data. Note that newly introduced pointers (e.g. `double *v` in Fig. 7) are located in the critical region, thus memory IO to the pointers themselves do not affect the accuracy of our method.

```
//-- Without data separation ----------------
typedef struct {
  int id;
  double v;
} Node;

Node *nodes = malloc(sizeof(Node) * SIZE);

for(i=0; i < SIZE; i++) {
  nodes[i].v = ...
}


//-- With fine-grained data separation ------
typedef struct {
  int id;
  double *v; // points to approximate region
} Node;

// Critical data is allocated with malloc
Node *nodes = malloc(sizeof(Node) * SIZE);

// Approximate data is allocated with b_malloc
double *vs = b_malloc(sizeof(double) * SIZE, &attr);

for(i=0; i < SIZE; i++) {
  // approximate data access via pointers
  nodes[i].v = &(vs[i]);
  *(nodes[i].v) = ...
}
```

Fig. 7 Example of fine-grained data separation.

The implementation of `bmalloc` leverages the NUMA support APIs of Linux (`libnuma`). The memory allocation functions of `bmalloc` use `numa_alloc_onnode` to allocate approximate data to a specific NUMA node. We rely on `libnuma` because allocating a memory region on a specified NUMA node requires placing the region in a particular *physical* address range, which requires to modify the underlying OS. We discuss the validity of this choice in Sect. 6.1.

4.3 Counting DRAM Internal Operations

Our system counts the number of DRAM internal operations with the help of hardware performance monitoring mechanism of the memory controllers connected to the NUMA node of the approximate region. Modern Intel Xeon processors have an integrated memory controller (iMC) for each memory channel and expose them as PCI devices that can

[†] Our ongoing work provides an OS support to repair NaNs that may appear by bit-flips [30].

Table 3 Vendor ID and Device IDs of PCI devices for iMCs of Recent Gegenerations of Intel Xeon CPUs

Generation	Vendor ID	Device IDs
Skylake	0x8086	0x2042, 0x2046, 0x204a
Broadwell	0x8086	0x6fb0, 0x6fb1, 0x6fb4, 0x6fb5
Haswell	0x8086	0x2fb0, 0x2fb1, 0x2fd0, 0x2fd1
Ivy Bridge	0x8086	0x0eb0, 0x0eb1, 0x0eb4, 0x0eb5
Sandy Bridge	0x8086	0x3cb0, 0x3cb1, 0x3cb4, 0x3cb5

Table 4 A Part of Performance Metrics Supported by iMCs of Modern Xeon Processors, reproduced from Sect. 2.3.5 of [31]

Metric	Event Code	Description
CLOCKTICKS	0x00	DRAM Clockticks
ACT_COUNT	0x01	DRAM Activate Count
PRE_COUNT	0x02	DRAM Precharge commands
CAS_COUNT	0x03	DRAM CAS (Column Address Strobe) Commands
DRAM_REFRESH	0x04	Number of DRAM Refreshes Issued
…	…	…

be read/written from software. The PCI devices can be used to configure the hardware performance monitoring mechanism and retrieve performance metrics.

To access an iMC, a pair of a bus number (`bus_no`), a device number (`dev_no`), and a function number (`fn_no`) associated to the iMC is required. Once the pair is determined, Linux allows reading data from and writing data to the iMC via a file named `/proc/bus/pci/bus_no/dev_no.fn_no`. The pairs can be found by matching the first 4 bytes of each file in `/proc/bus/pci` with the pre-defined vendor ID and device IDs shown in Table 3.

Table 4 shows a part of the performance metrics supported by iMCs of modern Intel Xeon processors. The table is reconstructed from Sect. 2.3.5 of an official manual [31]. The event code of each metric is written to the iMCs in order to count the metric. Each iMC supports counting four metrics simultaneously. We use `ACT_COUNT` to count the number of activations and restorations (note that a restoration happens after every activation), and use `PRE_COUNT` to count the number of precharges. We also use `CAS_COUNT` to measure the amount of memory IO to the approximate region to detect the end of epochs.

4.4 Emulating Bit-Flips

Our system emulates the behavior of approximate memory by injecting bit-flips to approximate data at the end of each epoch. To achieve this, we calculate the probability that each bit is flipped during an epoch (referred to as x) and traverse every bit of approximate data to flip each bit with this probability. Let the number of DRAM internal operations (activations, restorations, precharges) during the epoch be N_{op} and the bit-error rate per DRAM internal operation be R, the expected number of bit-flips C during this epoch is:

$$C = N_{op} \times S_{row} \times R \tag{1}$$

where S_{row} is the number of bits in a DRAM row (typically 4096). Next, we assume that x is constant across all the bits of approximate data, which stems from two assumptions: (1) each bit is flipped independently from the others, and (2) data accesses to approximate data are distributed to all the bits. Then, C is expressed in a different form as:

$$C = x \times S_{data} \tag{2}$$

where S_{data} is the amount of approximate data expressed in bits. One can refer Appendix A for the details. From Eq. (1) and Eq. (2), we get:

$$N_{op} \times S_{row} \times R = x \times S_{data} \tag{3}$$

$$\therefore x = \frac{N_{op} \times S_{row} \times R}{S_{data}} \tag{4}$$

Equation (4) is used to calculate x. Among the variables, N_{op} is counted by our system, S_{data} is retrieved from our memory allocator, and S_{row} is a constant. R is a configuration parameter that is adapted to underlying approximate memory hardware.

To decide the value of R, two methodologies exist depending on use cases.

1. Users can identify several Rs that their applications produce reasonable results by swinging R among various values, then compare the identified values with hardware characteristics (either given by real measurements or modeling) to decide latency parameters they can potentially use.
2. If there are particular latency parameters that come from users' energy and performance requirements, the users can set a specific value of R that a particular memory device yields with the given latency parameters. For example, if an approximate memory device in interest flips m bits for every DRAM internal operation in average, the value of R for that device is $\frac{m}{S_{row}}$.

Because the focus of this paper is to reveal how a given R affects application results, how to acquire the bit-error rate from a real device is out-of-scope of this work. Characterizing and modeling the relationship between bit-error rates and latency parameters is an on-going research field.

5. Evaluation

5.1 Methodology

To show that our system can evaluate effect of approximate memory to given applications, we conduct two types of evaluation. First, we show that the runtime overhead incurred by data separation is small enough to run same applications repeatedly with different parameters. Second, we apply our system to selected applications of SPEC CPU Benchmarks to evaluate effect of approximate memory for these applications. Table 5 shows the evaluation environment.

Table 5 Evaluation Environment

OS	Debian GNU/Linux 9.5
CPU	Intel Xeon Silver 4108 (8 cores) ×2
LLC	11 MB per NUMA node (1.375 MB per core)
Memory	DDR4 2400 MHz, 48 GB per NUMA node

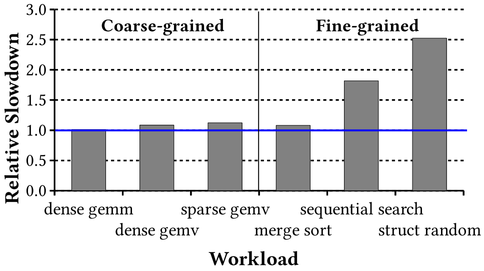


Fig. 8 Slowdown caused to various workloads by data separation.

5.2 Runtime Overhead due to Data Separation

Data separation with `bmalloc` slows down some applications because (1) it changes the data layout inside main memory and (2) it requires extra pointer dereferencing that do not exist in the original source code. In this section, we show that this slowdown is small enough to run the same application repeatedly, which is mandatory to evaluate effect of approximate memory with different parameters. In terms of the overhead caused by injecting bit-flips, we evaluate it in the next section for more realistic applications.

Figure 8 shows slowdown of synthetic workloads. The x axis shows workloads and the y axis shows relative slowdown compared to the elapsed time of the original source code. The labels "Coarse-grained" and "Fine-grained" show the applied data separation patterns. The values are averaged over 10 runs. We used six synthetic workloads with sufficiently large data compared to the LLC size.

- **dense gemm** calculates $C = A \times B$ for dense matrices A, B, and C of size 2048 × 2048. An element of the matrices is a `double`. The matrices are stored in the approximate region by coarse-grained data separation.
- **dense gemv** calculates $y = A \times x$ for vectors x, y and a dense matrix A of size 2048 × 2048. An element of the vectors and the matrix is a `double`. The vectors and the matrix are stored in the approximate region by coarse-grained data separation.
- **sparse gemv** calculates $y = A \times x$ for vectors x, y and a sparse matrix A of size 2048 × 2048. A is expressed in the csr format and 20% of the elements are non-zeros. An element of the vectors and the matrix is a `double`. The vectors and the matrix are stored in the approximate region by coarse-grained data separation.
- **merge sort** sorts a linked-list by the merge sort algorithm. An element of the list has two members: its value and the pointer the next element. The value elements are stored in the approximate region by fine-grained data separation.
- **sequential search** finds elements whose values are less than a threshold from an array of elements. An element has two members: its id and value. The value elements are stored in the approximate region by fine-grained data separation.
- **struct random** traverses an array of elements with a random order. An element has two members: its id and value. The value elements are stored in the approximate region by fine-grained data separation.

The workloads applied coarse-grained data separation experience almost negligible amount of slowdown because data separation does not change the memory access pattern nor require additional pointer dereferencing. As shown in Fig. 6, approximate data is accessed with the same way as it is accessed in native execution. On the other hand, the slowdown is sometimes visible for the workloads applied fine-grained data separation because it changes the memory access pattern and requires extra instructions for pointer dereferencing. The reason why the slowdown of **merge sort** is smaller than that of **sequential search** and **struct random** is that **merge sort** has more work to do between two approximate data accesses, thus the cycles needed for pointer dereferencing do not count that much. The different amount of overhead for **sequential search** and **struct random** comes from their data access patterns. For **sequential search**, approximate data pointed by extra pointers are prefetched by hardware prefetchers, while for **struct random** fetching the extra pointers and approximate data are serialized.

The results show that the runtime overhead incurred by data separation is 1× to 3× compared to native execution. It is 3 orders of magnitude faster than `gem5` (9000× of native execution in average). This allows users to evaluate effect of approximate memory to applications repeatedly with different parameters.

5.3 Case Studies with SPEC CPU Benchmarks

In this section, we use our system to evaluate effect of approximate memory to realistic applications. We chose two applications from SPEC CPU Benchmarks for case studies:

- **mcf** *uses a network simplex algorithm (. . .) to schedule public transport* [32]. The source code has data structures for nodes and arcs of a network. We store the cost values associated with them as approximate data with fine-grained separation.
- **milc** is a *gauge field generating program for lattice gauge theory programs with dynamical quarks* [33]. We select a data type named `su3_vector` as approximate data and store it with coarse-grained separation.

The number of lines modified in the source code is 62 for **mcf** and 43 for **milc**. Note that we cannot publish the modified code because SPEC CPU Benchmarks are not free software. We used the middle-sized data set called `train`.

For each workload, we choose three bir-error rates with which the workload produces various results, so as to show

Table 6 Effect of Approximate Memory to **mcf**. 'o': application finished correctly, 'x': application yielded a wrong result, 'e': endless execution.

$R = 1.0 \times 10^{-12}$

run	1	2	3	4	5	6	7	8	9	10
bit-flips	10	5	7	12	5	10	6	5	5	6
effect	x	o	x	o	o	x	o	o	o	o

$R = 2.5 \times 10^{-12}$

run	1	2	3	4	5	6	7	8	9	10
bit-flips	20	-	24	30	23	20	16	19	15	25
effect	o	e	x	o	x	x	o	o	x	x

$R = 5.0 \times 10^{-12}$

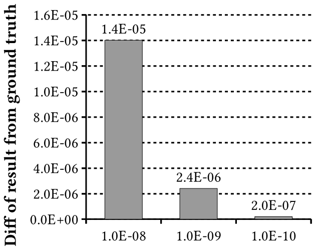
run	1	2	3	4	5	6	7	8	9	10
bit-flips	-	25	36	57	44	49	-	34	-	38
effect	e	x	x	x	x	x	e	x	e	x

that executing the same application with various parameters is important to evaluate effect of approximate memory. This corresponds to the 1st methodology of deciding the value of R explained in Sect. 4.4. We do not use bit-error rates from real measurements because the bit-error rate for a particular latency value differs by orders of magnitude depending each DIMM model (see Sect. 2.4 for the details).

5.3.1 mcf

Table 6 shows the evaluated effect of approximate memory to **mcf**. We set the amount of memory IO for the error injection interval to 10 GB. The bit-error rate per DRAM internal operation (R) is set to 1.0×10^{-12}, 2.5×10^{-12}, and 5.0×10^{-12}. Each column of the table corresponds to one run. The 'bit-flips' rows show the number of bit-flips injected during each run. The 'effect' rows show the effect of approximate memory observed: 'o' means that no effect was observed, 'x' means that the final result differed from the correct one, and 'e' means that the application was caught into endless execution and did not finish in a reasonable amount of time. Because **mcf** is a discrete application, we define any results that are different from the correct one as "wrong" (that is, there is no "close" result). The number of bit-flips for non-finished runs are not shown because bit-flips are kept injected during endless execution. The application finished correctly even when some bit-flips are injected into the data. This is because **mcf** is a search-based application and data that is no longer referred by the search algorithm is tolerant to any number of bit-flips. Note that this does not mean that there is no need to store this data because we cannot know that any portion of data is not necessary before completing a search.

The results show the importance of executing target applications repeatedly to evaluate effect of approximate memory. For example, the 4th run for $R = 2.5 \times 10^{-12}$ case had a correct result with 30 bit-flips while the 3rd run for $R = 1.0 \times 10^{-12}$ case had a wrong result even with 7 bit-flips. This shows that the behavior of a discrete algorithm does not directly depend on the number of injected bit-flips. Therefore, the capability of our system to allow users to run same applications repeatedly is mandatory to evaluate effect of approximate memory for these applications.



R	bit-flips
1.0×10^{-8}	3151 bits
1.0×10^{-9}	308 bits
1.0×10^{-10}	31.5 bits

Fig. 9 Effect of Approximate Memory to **milc**. The bar char (left) shows the drift of the final results from the correct one with various R. The table (right) shows the number of injected bit-flips.

5.3.2 milc

Figure 9 shows the evaluated effect of approximate memory to **milc**. We set the amount of memory IO for the error injection interval to 300 MB. The bit-error rate per DRAM internal operation (R) is set to 1.0×10^{-10}, 1.0×10^{-9}, and 1.0×10^{-8}. We only show the averaged results for 10 application runs because **milc** is a numeric application and every run finished with either a drifted or undrifted result. The application outputs a value called `GACTION` as a part of the final result. The bar chart of Fig. 9 shows the difference of `GACTION` between an error-free run and the cases when bit-flips are injected. The error-free value of `GACTION` is 2.222088. The table in Fig. 9 shows the number of injected bit-flips. Unfortunately, the manuals of **milc** [33], [34] do not provide any information on how much error is acceptable in the result, nor does the author have proper knowledge to interpret the physical meaning of it. However, given that the code supports a compiler option to change the precision of floating point operations (either 32 bit or 64 bit) [34], we believe that some use cases of **milc** accept drifted results.

The results again show the importance of executing target applications repeatedly to evaluate effect of approximate memory. The number of bit-flips increases linearly as R is increased, but the difference of `GACTION` compared to the correct value does not obey the linear correlation. For example, increasing R from 1.0×10^{-9} to 1.0×10^{-8} increases the number of bit-flips by 10×, but the difference of `GACTION` compared to the correct value increases only by 5.8×. This means that the capability of our system to allow users to run same applications repeatedly with different R is an urgent need to evaluate effect of approximate memory.

5.3.3 Execution Time

Figure 10 shows the overhead of our system incurred to the **mcf** and **milc** benchmarks. The labels "vanilla" show the elapsed time of them with no modification to the source code, the labels "separation" show the elapsed time when data separation is applied but no bit-flips are injected, and the labels "bmalloc" show the elapsed time when our

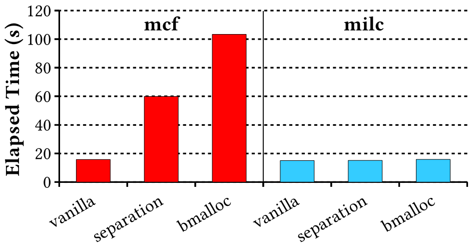


Fig. 10 Elapsed time of mcf and milc on our system. “vanilla” means the original code, “separation” means that the approximate data is allocated with our allocator, and “bmalloc” means that our method is fully applied.

method is fully applied (data separation and bit-flip injection). The values are averaged over 10 runs.

The slowdown due to data separation is negligible for **milc** while **mcf** is slowed down by 3.8×. The reasons is that the approximate data of **milc** is continuous and applied the coarse-grained data separation, but the approximate data of **mcf** is a member of a larger data hierarchy and is applied the fine-grained data separation with extra pointer dereferencing. The slowdown due to the bit-flip injection is also negligible for **milc** while it is not negligible for **mcf**. The reason is that the size of the approximate data of **mcf** is larger than that of **milc** and injecting bit-flips takes longer time. The results show that our system with all the overhead taken into account is still 3 orders of magnitude faster than cycle accurate simulators.

6. Discussion

6.1 Validity of Using NUMA Nodes

An alternative implementation choice to measure the number DRAM internal operations only for approximate data is to store it to designated *memory channels* within a single NUMA node. Although this choice needs only one CPU, the disadvantage is that placing data on an arbitrary memory channel is hard to implement in a portable way.

The mapping between physical memory addresses and memory channels cannot be known in a portable way because they are not defined in an open specification to the best of our knowledge. Even if the memory interleaving functionality is disabled, we can only guess that the physical memory is evenly divided, which may be true for some machines but may not for others. In contrast, the mapping between physical memory addresses and NUMA nodes is notified with System Resource Affinity Table by ACPI (see Sect. 5.2.16 of the ACPI spec [35] for the details). Therefore, using NUMA nodes is the most portable way to meet our need.

6.2 Overhead Due to Bit-Flip Injection

The overhead due to bit-flip injection (the time it takes to inject bit-flips into approximate data) is not negligible for some applications as shown in Sect. 5.3. One possible way to suppress this overhead is to randomly select C bits (which can be calculated by Eq. (1)) from approximate data and flip them at the end of every epoch, instead of traversing every single bit. Although this choice incurs almost no overhead for bit-flip injection, the issue is that it could reduce the accuracy of mimicking effect of approximate memory. As seen in Sect. 5.3 for **mcf**, the number of injected bit-flips differs run by run even for the same bit-error rate because C only refers the expected value. Injecting C bit-flips in every epoch dismisses this phenomenon and might only produce “averaged” effect in return of reduced overhead.

6.3 Limitations

Our current system is applicable only to approximate memory models where bit-flips are incurred by DRAM internal operations. Therefore, it cannot directly be applied to an approximate memory model where lower refresh rate is applied to approximate memory or to other approximate *computing* models that assume ALUs and caches do not guarantee data integrity. However, we hope that our basic idea of leveraging hardware performance monitors to emulate bit-flips will be further studied to be applied for these approximate models. Hardware performance monitors exist not only in iMCs but also in CPU cores and caches and these monitors could be leveraged to extend our system to other approximate models that we do not handle in this work.

7. Related Work

Error-robustness of applications is widely studied in the context of fault tolerance. Ashraf *el al.* [36] study error-robustness of MPI applications by injecting bit-flips into registers at compile-time using an LLVM-based fault injector [29]. Although their method does not use hardware simulators, it cannot be applied to our scenario because it cannot consider the number of DRAM internal operations. They depend on available cache size and input data size, thus it is not possible to deal with bit-flips stemming from approximate memory at compile time. Carbin *et al.* [37] analyze source code to estimate error-robustness by tracking how errors propagate from one place to another. A shortcoming of their method is that it is very conservative. For example, the trustworthiness of a value modified by an approximate operation inside a loop is 0 in most cases because the number of loop iterations cannot be known from the source code.

Error-robustness analysis is applied to GPUs as well because they are used for mission-critical applications. Li *et al.* [38] extend the same LLVM-based fault injector as used by Ashraf *el al.* [36] to inject bit-flips into GPU applications. Their method not only injects bit-flips but also tracks how injected errors are propagated across multiple kernels. Nie *et al.* [24] prune the number of possible places to which bit-flips are injected for highly parallel GPU applications. The main idea is that GPU applications apply the same instructions to much data using many threads, thus it is not needed to consider all possibilities when injecting bit-flips.

In the context of approximate computing, evaluation of proposed approximation models is done with heavyweight methods. Flikker [20] prolongs the refresh interval of DRAM to reduce the energy consumption. It uses PIN [27], a dynamic binary instrumentation tool, to detect if a byte is accessed after the previous refresh. Although not cycle accurate, PIN is already tremendously slow. It is reported that PIN takes 4 – 6 days to record a full trace for SPEC CPU 2006 Benchmarks [39]. Another shortcoming of PIN is that it cannot consider effect of caches, as it traces instructions from a viewpoint of software. This does not matter much in a mobile environment with small caches that Flikker assumes, but the effect of caches cannot be dismissed in server environments with large LLCs. EnerJ [25] is a new instruction set architecture for approximate computing. They use an in-house instrumentation tool to trace method calls, object creation and destruction, arithmetic operators, and memory accesses. Although they do not show how long it takes to retrieve a trace, it must be as slow as PIN given that they do similar things. Device-level techniques to reduce the latency and energy consumption of DRAM use either cycle accurate simulators or real hardware with FPGA-based memory controllers to evaluate their proposals [2]–[7]. Cycle accurate simulators are too slow for our scenario as shown Sect. 3.3 and FPGA-based hardware is not easy-to-use. On the other hand, our system is both lightweight and easy-to-use.

8. Conclusion

This paper proposed a lightweight method to evaluate effect of approximate memory to applications to tackle the latency and energy problem of main memory. Instead of using heavy-weight cycle accurate simulators, we emulate bitflips caused by approximate memory by measuring the number of DRAM internal operations on a bare-metal machine while an application is running. Thanks to the small overhead, our system enables to evaluate effect of approximate memory repeatedly for a given application with different parameters. We showed that it is 3 orders of magnitude faster than cycle accurate simulators and we gave case studies of evaluating approximate memory for realistic applications.

Acknowledgments

This work was supported by JST, ACT-I Grant Number JPMJPR18U1, Japan. Part of this paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO). The author thanks Prof. Ryota Shioya (Univ. of Tokyo) for giving advice on how to set up `gem5`.

References

[1] J.L. Hennessy and D.A. Patterson, Computer Architecture, Fourth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[2] K.K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization," International Conference on Measurement and Modeling of Computer Science (SIGMETRICS), pp.323–336, 2016.

[3] Y. Lee, H. Kim, S. Hong, and S. Kim, "Partial row activation for low-power dram system," International Symposium on High Performance Computer Architecture (HPCA), pp.217–228, 2017.

[4] Y. Wang, A. Tavakkol, L. Orosa, S. Ghose, N.M. Ghiasi, M. Patel, J.S. Kim, H. Hassan, M. Sadrosadati, and O. Mutlu, "Reducing DRAM latency via charge-level-aware look-ahead partial restoration," IEEE/ACM International Symposium on Microarchitecture (Micro), pp.298–311, 2018.

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

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

[7] J. Choi, W. Shin, J. Jang, J. Suh, Y. Kwon, Y. Moon, and L.-S. Kim, "Multiple clone row DRAM: A low latency and area optimized DRAM," International Symposium on Computer Architecture (ISCA), pp.223–234, 2015.

[8] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," Tech. Rep. TR-2009-042, MIT CSAIL, 2009.

[9] Q. Guo, K. Strauss, L. Ceze, and H.S. Malvar, "High-density image storage using approximate memory cells," SIGARCH Comput. Archit. News, vol.44, no.2, pp.413–426, March 2016.

[10] D. Fujiki, K. Ishii, I. Fujiwara, H. Matsutani, H. Amano, H. Casanova, and M. Koibuchi, "High-bandwidth low-latency approximate interconnection networks," International Symposium on High Performance Computer Architecture (HPCA), pp.469–480, 2017.

[11] S. Akiyama, T. Hirofuchi, and H. Ogawa, "Performance prediction of memory access intensive apps with delay insertion: A vision," International Conference on Cloud Computing Technology and Science (CloudCom), pp.492–496, 2016.

[12] "ABCI AI bridging cloud infrastructure." https://abci.ai/.

[13] "NVIDIA DGX-2 the world's most powerful deep learning system for the most complex AI challenges." http://images.nvidia.com/content/pdf/dgx-2-print-datasheet-738070-nvidia-a4-web.pdf.

[14] L.A. Barroso, J. Clidaras, and U. Holzle, The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition, Morgan & Claypool Publishers, 2013.

[15] D. Meisner, B.T. Gold, and T.F. Wenisch, "Powernap: Eliminating server idle power," International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp.205–216, 2009.

[16] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the Netflix prize," 4th International Conference on Algorithmic Aspects in Information and Management (AAIM), pp.337–348, 2008.

[17] B. Jacob, S. Ng, and D. Wang, Memory Systems: Cache, DRAM, Disk, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[18] "Intel memory latency checker v3.5." https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[19] B. Fang, Q. Guan, N. Debardeleben, K. Pattabiraman, and M. Ripeanu, "Letgo: A lightweight continuous framework for hpc applications under failures," International Symposium on High-Performance Parallel and Distributed Computing (HPDC), pp.117–130, 2017.

[20] S. Liu, K. Pattabiraman, T. Moscibroda, and B.G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," In-

ternational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp.213–224, 2011.

[21] J. Kim, M. Patel, H. Hassan, and O. Mutlu, "Solar-DRAM: Reducing dram access latency by exploiting the variation in local bitlines," IEEE International Conference on Computer Design (ICCD), pp.282–291, 2018.

[22] Y.-H. Gong and S.W. Chung, "Exploiting refresh effect of dram read operations: A practical approach to low-power refresh," IEEE Trans. Comput., vol.65, no.5, pp.1507–1517, 2016.

[23] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," International Conference on Dependable Systems and Networks (DSN), pp.467–478, 2014.

[24] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault site pruning for practical reliability analysis of GPGPU applications," IEEE/ACM International Symposium on Microarchitecture (Micro), pp.750–762, 2018.

[25] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," SIGPLAN Not., vol.46, no.6, pp.164–174, June 2011.

[26] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood, "The gem5 simulator," SIGARCH Comput. Archit. News, vol.39, no.2, pp.1–7, Aug. 2011.

[27] "Pin - A Dynamic Binary Instrumentation Tool." https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[28] "The llvm compiler infrastructure." https://llvm.org/.

[29] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," International Conference on Dependable Systems and Networks (DSN), pp.375–382, 2014.

[30] S. Hamada, S. Akiyama, and M. Namiki, "Reactive NaN repair for applying approximate memory to numerical applications," The 8th Workshop on Systems for Multi-core and Heterogeneous Architectures (SFMA), pp.1–6, 2018.

[31] "Intel Xeon processor scalable memory family uncore performance monitoring reference manual." https://software.intel.com/en-us/download/intel-xeon-processor-scalable-memory-family-uncore-performance-monitoring-reference-manual.

[32] "SPEC CINT2006 Benchmarks." https://www.spec.org/cpu2006/CINT2006/.

[33] "SPEC CFP2006 Benchmark Descriptions." https://www.spec.org/cpu2006/CFP2006/.

[34] A. Bazavov, C. Bernard, T. Burch, T. DeGrand, C. DeTar, J. Foley, S. Gottlieb, U. Heller, J. Hetrick, L. Levkova, C. McNeile, K. Orginos, J. Osborn, K. Rummukainen, B. Sugar, and D. Toussaint, "The milc code." http://www.physics.utah.edu/~detar/milc/milcv7.pdf.

[35] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation, "Advanced configuration and power interface specification." http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf.

[36] R.A. Ashraf, R. Gioiosa, G. Kestor, R.F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in hpc applications," International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp.72:1–72:12, 2015.

[37] M. Carbin, S. Misailovic, and M.C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA), pp.33–52, 2013.

[38] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in gpgpu applications," International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp.21:1–21:12, 2016.

[39] D. Chen, C. Ye, and C. Ding, "Write locality and optimization for persistent memory," International Symposium on Memory Systems (MEMSYS), pp.77–87, 2016.

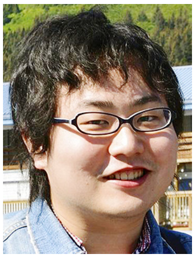
Appendix A: Proof of Eq. (2)

Let the probability that a bit is flipped be x and the size of approximate data be S bits, the expected number of bit-flips in the approximate data is calculated by definition as follows:

$$\sum_{s=1}^{S}\left(s \times \text{Combination}(S, s) \times x^{s}(1-x)^{S-s}\right). \qquad (\text{A}\cdot 1)$$

The below proves that Eq. (A· 1) is equal to Eq. (2).

$$\begin{aligned}
&\sum_{s=1}^{S}\left(s \times \text{Combination}(S, s) \times x^{s}(1-x)^{S-s}\right)\\
&= \sum_{s=1}^{S}\left(s \times \frac{S\,!}{s!(S-s)!}x^{s}(1-x)^{S-s}\right)\\
&= xS \times \sum_{s=1}^{S} \frac{(S-1)!}{(s-1)!(S-s)!}x^{s-1}(1-x)^{S-s}\\
&= xS \times \sum_{k=0}^{K} \frac{K!}{k!(K-k)!}x^{k}(1-x)^{K-k}\\
&\qquad\qquad\qquad\qquad (s-1=k, \quad S-1=K)\\
&= xS \times (x+(1-x))^{K}\\
&= xS
\end{aligned}$$



Soramichi Akiyama is a faculty member of Department of Creative Informatics, The University of Tokyo, Japan. Dr. Akiyama received a B.Eng. from Kyoto University in 2010 and a Ph.D. from The University of Tokyo in 2015. Dr. Akiyama's research interest centers on how to efficiently execute large-scale workloads in the AI and HPC fields with as little programmer effort as possible, by leveraging knowledge of operating systems, virtualization techniques, memory systems, and performance analysis.