Memory Equalizer for Lateral Management of Heterogeneous Memory

Chencheng Ye
Service Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology
Wuhan, Hubei, China
yechencheng@gmail.com

Chen Ding
Computer Science Department
University of Rochester
Rochester, New York, USA
cding@cs.rochester.edu

Hai Jin
Service Computing Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology
Wuhan, Hubei, China
hjin@hust.edu.cn

Abstract

Modern computers increasingly use more types of memory such as phase-change memory and high-bandwidth memory. Consequently, it is more complex and difficult to manage memory effectively.

This paper presents a vision of memory management and its key component called *fraction cache*, which is a type of two-level exclusive cache. The fraction cache is flexible and encompasses a broad solution space. More importantly, its parameters can be automatically optimized. As a demonstration, the fraction cache is used to minimize the DRAM need to sometimes less than 1% of the program data size.

ACM Reference format:
Chencheng Ye, Chen Ding, and Hai Jin. 2017. Memory Equalizer for Lateral Management of Heterogeneous Memory. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 10 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As Peter Denning explained in his 1980 survey paper, the problem of memory management dates back to the design of the Multics computer, where Jerome H. Saltzer saw it as "an adaptive control that would allocate memory ... in order to maximize performance. The resulting system could have a knob by which the operator could occasionally tune it." Denning called it "Saltzer's Problem". His drawing of the "knob" is shown in Figure 1 [8].[1]

The solution to Saltzer's Problem is well known and widely used. It is the virtual memory support in every general-purpose operating system. In the classic design, virtual memory has two memory types: the fast main memory and the disk. It is concerned with mainly two memory properties: speed and capacity.

Today, there are additional memory types, not just DRAM but also high-bandwidth memory (HBM) and phase-change memory (PCM); and there are equally important considerations beyond speed and capacity: energy, power, endurance and persistence.

Conventionally, virtual memory is hierarchical. Each running application is allocated a portion of the main memory. "Turning the knob" means judicious allocation based on the application's dynamic memory demand, i.e., its locality [11]. Memory management has one primary objective — to maximize the throughput with limited fast memory.

Today's memory is heterogeneous, and its structure flat. A processor has direct access to different types of memory, not just DRAM but also HBM and PCM. Memory management has multiple objectives, not just throughput, but also energy

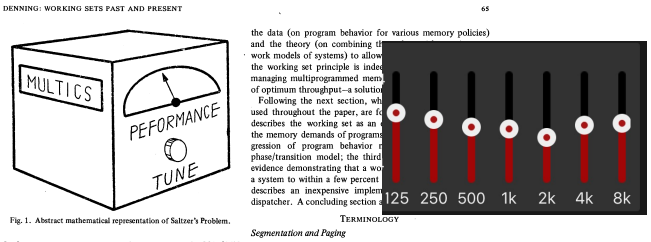

Fig. 1. Abstract mathematical representation of Saltzer's Problem.

Figure 1. (left) A visual representation created by Denning in 1980 for the classic memory management problem [8]. (right) The new problem of managing multiple types of memory is visualized as an audio equalizer.

We envision a modern memory manager analogous to an audio equalizer shown in Figure 1. An equalizer has a row of sliders which are frequency-specific volume knobs. What the figure implies is that the equalizer problem (EQ) replaces the single-knob memory tuning Saltzer's Problem (SP) with multiple sliders.

We call this new type of memory managers *memory equalizer*. It addresses a set of observable memories. As an audio equalizer is used to adjust the sound effect of instruments and correct acoustic defects of a room or an auditorium, a memory equalizer is used to combine multiple types of actual memories to effect the most desirable overall quality. The metaphorical sliders may be viewed as representing allocations, one in each memory. They may also be viewed

[1] Permission for re-publication was granted by the Association for Computing Machinery.

as specifications from a user, expressing local constraint for each memory.

From the outset, virtual memory is the abstraction and optimized use of actual memories. The goal of a memory equalizer is to manage more diverse and complex m

This shift from SP to EQ is to address two challeng a memory equalizer performs lateral memory mana In EQ, an application stores its data laterally in types of memories. In comparison, SP manages just of fast memory. In EQ, memory allocation must b for all memory types. In SP, the trade-off happens between capacity and throughput. In EQ, each mem has different strength and weakness compared to and requires different trade-offs. Recent examples the trade off between energy and speed (see Section between latency and bandwidth [14].[2]

Second, an equalizer serves multiple objectives are measures such as performance, power, persiste lifetime, in which no one strictly dominates the re specific problem may still have a single goal as a w mixture of multiple objectives, but the weights in ture may change from system to system, user to u application to application. A general solution shou optimization for an arbitrarily weighted objective (o user if no such solution exists).

This position paper starts a tentative step in this rection. As the first step, we present an abstract desig *fraction cache*. It divides program data into two pa cached fraction and the uncached fraction. For the f tion, it uses the familiar cache mechanisms to dyna place data on two memories and move data betwe Given the size of the memories, i.e. the cache size, it existing techniques of cache modeling to predict th at each memory and the amount of data migration.

The fraction cache creates a parameterized solutic a memory equalizer can choose how many fraction which fraction uses which memories, and at what sizes. The creation of this solution space is the key EQ. First, the fraction cache is flexible. The large space is likely to contain a good solution that sa complex objective. Second, the fraction cache prec quality of all solutions and uses the prediction to best one. We call the latter *deductive optimization*, s result is deduced rather than obtained through test

The rest of the paper is organized as follows. S presents the fraction cache and shows that it is gene ble, and more importantly, permits multi-objective o tion. Section 3 evaluates how the fraction cache ma the DRAM demand in a set of test programs. Finally two sections discuss related work and summarize.

[2]Ramos and Hoefler showed that although the integrated Mic Channel DRAM (MCDRAM) on Intel Knight Landing has mu bandwidth than DRAM, it also has slightly higher access latency

2 Fraction Cache Theory and Optimization

This section first reviews a previous study, then shows the fraction cache and its parameterized solution space, and finally its deductive optimization.

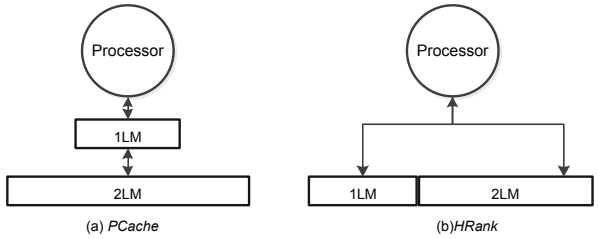

Figure 2. The two architectures of hybrid DRAM and PCM memory, reproduced from Su et al. in MemSys 2015 [18].

Su et al. showed two example applications. *pF3D* has good locality and therefore better performance and lower energy in the vertical PCache than in the horizontal HRank, while *LULESH* has poor locality, so PCache is not effective, and as a result, HRank is more energy efficient. Su et al. developed a new memory controller called *HpMC* to switch between PCache an HRank and make the best choice for each application.

The study by Su et al. shows convincingly that neither PCache or HRank is sufficient to achieve both high performance and energy efficiency across different workloads. Following their approach of combining existing solutions, we next develop a new technique that provides a greater range of solution choices.

2.2 Fraction Cache

A *fraction cache* divides the program data into fractions. In the basic design, we randomly divide program data into two fractions and store them in two types of memories: DRAM and PCM. One fraction is cached in DRAM and evicted to PCM. The other fraction is stored directly in PCM. We may distinguish by calling them the *cached fraction* and the *uncached fraction*.

A fraction cache has the following parameters:

- *Total data size m*, the unit of which is usually a page.
- *The cached fraction* sz_{fr}, which is a size between 0 and m. The cached fraction uses DRAM as the cache and PCM the target of eviction.

- *The DRAM size sz_{dram} and PCM size sz_{pcm}.* To not waste memory, as cache, the DRAM size is no greater than the cached data, i.e., $sz_{dram} \leq sz_{fr}$. When the cache data is more than the DRAM size $sz_{fr} > sz_{dram}$, the overflow, $sz_{fr} - sz_{dram}$, is stored in PCM.

The organization of a fraction cache is shown in a diagram in Figure 3. Program data is of size m and stored in both DRAM and PCM, so we have $m = sz_{dram} + sz_{pcm}$. The cache fraction sz_{fr} is stored partly in DRAM sz_{dram} and partly in PCM.[3] The vertical dotted line in Figure 3 marks the separation in PCM between the cache fraction and the uncached

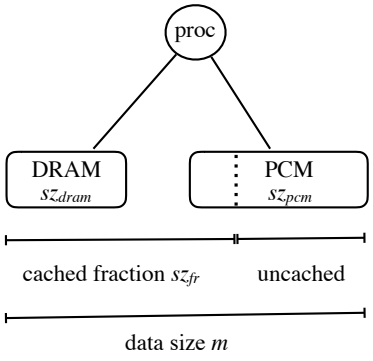

Figure 3. A fraction cache divides program data into the cached fraction and the uncached fraction. It stores the cached fraction in DRAM and PCM and the uncached fraction in PCM.

The memory organization is flat. At any time, the program data is partitioned between DRAM and PCM, and each is accessed directly. An access to the data of the cache fraction will happen in DRAM, if the access is a cache hit, and in PCM, if it is a cache miss. At a cache miss, the missed data is also promoted into DRAM, incurring an eviction in DRAM if it is full. Effectively, the cache replacement policy is used to control data migration between DRAM and PCM. The new lateral memory management by the fraction cache reuses the familiar solution of cache design. A difference is that in a conventional cache, the miss data is fetched through the cache. In the fraction cache, the miss data is fetched from PCM directly, at the same time of the migration of the missed data to DRAM.

The key novelty of the design resides in its two parameters, sz_{fr}, sz_{dram}, which make the fraction cache not a single solution but a collection of solutions, one for each parameter combination.

[3]The two-level, DRAM-PCM cache may be inclusive or exclusive. This paper assumes an exclusive cache, which consumes less PCM but incurs more write backs compared to an inclusive cache. The performance modeling discussed later in the paper is the same for either the exclusive or the inclusive cache.

By choosing the right parameters, the fraction cache can implement the two designs of Su et al., discussed in Section 2.1. If the fraction $sz_{fr} = m$, we have the effect of vertical PCache. In actual design, the fraction cache has two differences compared to PCache. First, the fraction cache is exclusive, while PCache is inclusive. Second, at a DRAM miss, the data is fetched directly from PCM, and at the same time, the data page is migrated from PCM to DRAM. If the cached fraction fits entirely in DRAM $sz_{fr} = sz_{dram}$, then we have the effect of horizontal HRank without page migration.[4]

The performance of fraction cache can be efficiently modeled. Next we present a technique based on a recent locality theory.

2.3 Predicting Cache Performance Using Footprint

Xiang et al. developed the higher-order theory of locality (HOTL), which defines a set of metrics and uses them to compute the miss ratio in shared cache [23]. We review the HOTL theory here and then use it to model and optimize the performance of the fraction cache. In HOTL, the most important metric is the *footprint*.

In an execution trace, each time window is represented by (t, x), where t is the end position and x the window length. The number of distinct elements in the window is the *working-set size* $\omega(t, x)$, as defined first by Denning [6]. The working-set size may vary from window to window, the footprint is its average. For each x, $fp(x)$ is the average working-set size of all windows of length x, i.e., the total working-set size divided by the number of length-x windows as shown by the following equation:

$$fp(x) = \frac{1}{n - x + 1} \sum_{t=x}^{n} \omega(t, x) \tag{1}$$

The footprint is a function $fp(x)$, and its parameter x a timescale such that $0 \leq x \leq n$, where the largest timescale n is the trace length.

For fully-associative LRU cache, the miss ratio $mr(c)$ is the (discrete) derivative of the footprint function. The precise formula has two calculations as follows. Given cache size c, it first finds the timescale x and then takes the derivative at x.

$$mr(c) = fp(x + 1) - fp(x) \text{ where } c = fp(x)$$

In this paper, we refer to it as the HOTL conversion and use a more compact representation based on the Leibniz's notation, where the two calculations are in the same equation separated by the vertical bar.

[4]In HRank of Su et al. [18], periodically the most accessed pages are migrated from PCM to DRAM.

$$mr(c) = \frac{\mathrm{d}}{\mathrm{d}x} fp(x)\Big|_{fp(x)=}$$

As an example illustration, the follo
simple access trace, its footprint, and
puted using the HOTL conversion.

access trace: a b c

(a) An access trace

timescale x	0	1	2	3
$fp(x)$	0	1	2	3

(b) Footprint

cache size c	0	1	2
$mr(c)$	100%	100%	100%

(c) Miss ratios

Figure 4. Example footprint and HOTL conversion

In 1972, Denning and Schwartz were the first to estimate the LRU cache miss ratio by the slope of the mean working-set size [9]. The HOTL conversion has the same form but using the footprint.[5] The mean working-set size was initially defined as a limit value [9]. See Xiang et al. [23] for a comparison between HOTL and the working-set theory.

The HOTL theory includes efficient footprint measurement, concavity of the footprint (hence monotonicity of the miss ratio), and the correctness condition [22, 23]. This paper builds on these results and will use the HOTL conversion to model performance and drive optimization.

2.4 Fraction Cache Performance

The performance is measured by data traffic. We consider three types of traffic, one for each connection between the processor, DRAM, and PCM, shown in Figure 3: the traffic $trfc_{dram}$ between the processor and DRAM, $trfc_{pcm}$ between the processor and PCM, and $trfc_{miss}$ between DRAM and PCM. The last one $trfc_{miss}$ is the misses in the fraction cache. It is also the data migration from PCM to DRAM.

For simplicity, we consider only data reads[6] and only the volume of data reads. The performance is measured by the amount of read traffic from DRAM and PCM and the traffic of data misses in DRAM. These three types of traffic are shown in Figure 5.

[5]HOTL conversion is defined for all execution traces. Take the simple example in Figure 4. Since it has no reuses, it is unclear how the previous working-set theory defines its means working-set size, but its footprint has a clear definition.

[6]Data writebacks in cache can be modeled using write locality developed by Chen et al. [5].

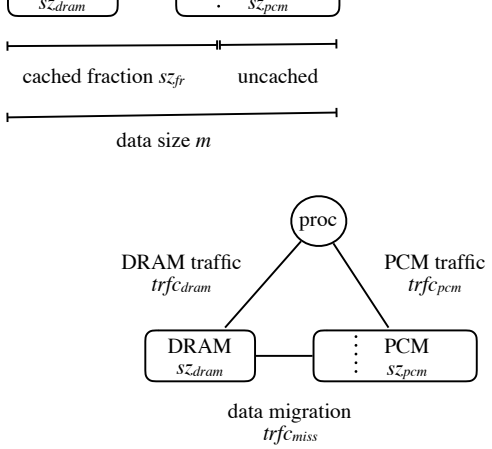

Figure 5. The performance of a fraction cache is measured by the communication between the processor and DRAM, between the processor and PCM, and data migration between DRAM and PCM. This position paper simplifies and considers only reads and misses.

In fraction cache, the cached data is accessed in DRAM if it is a hit. Otherwise, the missed data is accessed directly in PCM, not indirectly through DRAM. However, another copy of the missed data is transferred to DRAM to update the content of the DRAM cache. Hence, the DRAM traffic includes the accesses to the cached fraction but does not include the misses, and the PCM traffic includes all accesses to the uncached fraction and all the misses.

Using the footprint theory in Section 2.3, we compute the performance of the fraction cache as follows. Let $h(x)$ be the footprint of the cached data. We first compute the miss ratio $mr(sz_{dram})$ using HOTL for any $sz_{dram} \geq 0$:

$$mr(sz_{dram}) = \frac{\mathrm{d}}{\mathrm{d}x} h(x)\Big|_{h(x)=sz_{dram}} \tag{3}$$

The miss traffic is naturally the miss ratio times n.[7] Let $r = \frac{sz_{fr}}{m}$ be the ratio of the cached fraction. All three types of traffic are computed as follows:

$$trfc_{miss} = mr(sz_{dram}) * n \tag{4}$$

$$trfc_{dram} = r * n - trfc_{miss} \tag{5}$$

$$trfc_{pcm} = (1 - r) * n + trfc_{miss} = n - trfc_{dram} \tag{6}$$

2.5 Fraction Cache Optimization

The key novelty of the fraction-cache design is its parameterized solution space, which has fully predictable performance. The benefit is two fold: to make it flexible and to enable optimization.

The fraction cache is a versatile tool for lateral memory management. The previous section has shown how to divide data dynamically between DRAM and PCM. The previous solution was a two-fraction cache. On a machine with three

[7]In our implementation described in Section 3, the fraction data footprint $h(x)$ uses the logical clock of all n accesses, and the miss traffic is $mr(sz_{dram}) * n$. If $h(x)$ uses the logical clock of only the accesses to the data fraction, the miss traffic would be computed as $mr(sz_{dram}) * n * r$.

types of memories, HBM, DRAM and PCM, we divide program data into three fractions. There are at least two choices to assign these fractions. The first fraction is cached in HBM and evicted to DRAM, the second fraction cached in DRAM and evicted to PCM, and the last fraction is stored only in PCM. Alternatively, if we avoid storing infrequently accessed data in DRAM, the data of the first fraction can be evicted to PCM instead of DRAM.

The fraction cache enables deductive optimization. It has a clearly defined solution space, parameterized by how many fractions to use, which fraction to store in which memories, and at what memory sizes. With the modeling technique from Section 2.4, we can compute the performance of each solution. Because the solution space is large, it is likely for different objectives of performance, power, persistence and lifetime, a good solution exists in this space, which is then discovered by performance modeling and deductive optimization. At the end, the optimization finds the best parameters.

The fraction cache requires dividing program data into cached and uncached fractions. There are two generic schemes: page interleaving and random assignment. In page interleaving, data pages are assigned round robin based on the fraction. In random assignment, data pages are randomly assigned. For example, if the fraction is 0.7, page interleaving would place 7 out of every 10 pages in the cached fraction, and random assignment would place a page in the cached fraction with 0.7 probability. These schemes are generic because they do not require program knowledge and can be applied on any application process. More specific division schemes may be used if special program information is available. For example, some data may have frequent accesses and should always be placed in the cached fraction. The remaining data can be assigned through the two generic schemes.

The performance model requires to know the footprint of the access to the cached fraction, which can be analyzed in real time using sampling. A recent technique is based on a model called the average-evicting time (AET), and its analysis has a negligible cost in both time and space [10]. In fact, the model can be changed to use reuse distance, which can be as efficiently sampled by a technique called SHARDS [20]. Another possible extension is miniature simulation, which may allow the fraction cache to use non-LRU replacement policies [21].

Fraction cache optimization may not reach optimal performance for two reasons. First, the performance model may have errors, and an error in the model can lead to an error in optimization. Second, its solution space does not cover all possible solutions. In particular, it uses fixed partitioning to assign data between cached and uncached partitions. It is possible that the locality set of an application changes dynamically, and the data in the uncached partition later receives frequently access. The problem of fixed partitioning may be ameliorated by periodically repartition data for the fraction cache.

2.6 DRAM Portion Reduction

In this section, we use the fraction cache to solve a contrived problem called *DRAM Portion Reduction* (DPR). The motivation for this problem is as follows. Consider a machine with hybrid DRAM and PCM. Since DRAM is more expensive and consumes more energy per bit than PCM, we maximize the cost and energy savings by minimizing the portion of program data stored in DRAM. The following is the problem setup.

We assume the machine has DRAM and PCM side by side, with access bandwidths bw_{dram}, bw_{pcm}. To use both DRAM and PCM bandwidth fully, we want the ratio of DRAM and PCM traffic matches the ratio of DRAM and PCM bandwidth, i.e., $\frac{trfc_{dram}}{trfc_{pcm}} = \frac{bw_{dram}}{bw_{pcm}}$. To simplify the discussion, we consider the case $bw_{dram} = bw_{pcm}$.

For this problem, we want to (1) equalize the DRAM and PCM traffic $trfc_{dram} = trfc_{pcm}$, (2) limit the migration cost, and (3) minimize the need for DRAM. We call the first *bandwidth requirement* and the second *migration requirement*. In this paper, we measure traffic by the ratio of accesses. The bandwidth requirement means that both DRAM and PCM traffic be 50% of all memory accesses $trfc_{dram} = trfc_{pcm} = 0.5$.

Let's examine the solution space. The simplest solution is *even division*, which divides the program data m into two halves between DRAM and PCM. Both memory types will be accessed equally frequently, satisfying the traffic requirement. Even division is trivial to implement. We use it as the baseline solution, where the initial DRAM needed is half of the data size $\frac{m}{2}$.

We can reduce the DRAM need by assigning frequently accessed data to DRAM. The fraction cache moves such data to DRAM automatically. It can be proved that, assuming the miss ratio is monotone, the DRAM need is minimized by *complete caching*, that is, caching the entire data $sz_{fr} = m$ and choosing the DRAM size so that $mr(sz_{dram}) = 0.5$. In this solution, the miss ratio is 50% and the $trfc_{dram} = trfc_{pcm} = 0.5$.

Fraction cache has a two-dimensional solution space: the fraction sz_{fr} ranges from 0.5 to 1, and the DRAM need ranges from $\frac{m}{2}$ down. The trivial and best solutions, even division and complete caching, are two extreme points in this space.

A key factor is data migration. Even division needs most DRAM but incurs no migration ($trfc\_miss = 0$), while complete caching needs least DRAM but incurs most migration ($trfc\_miss = 0.5$). While caching depresses the need for DRAM, it incurs the cost of migration. It must not grow without a bound.

Given a migration bound, the fraction cache theory computes the minimal DRAM size. From the formulas in Section 2.4, we can compute the DRAM, PCM and migration traffic for any fraction sz_{fr} and DRAM size sz_{dram}. It is straightforward to find all valid solutions that meet the bandwidth requirement and stay within the migration bound. From the valid solutions, we select one with the minimal DRAM size.

3 Evaluation

We measure the effect of DRAM portion reduction as described in Section 2.6. On a machine with hybrid DRAM and PCM, we want to use DRAM the least and PCM the most to save cost and energy. In this section, we first show an experimental setup and then an evaluation using the results from performance modeling and deductive optimization. The evaluation is partially theoretical and not based on a real system.

3.1 Methodology

Benchmarks we select 7 benchmarks to evaluate the fraction cache, including 3 programs from NPB [1], 3 from PRASEC [3], and 1 real application. All 7 benchmarks have large memory footprint. The NPB benchmarks use *class C* input, the PARSEC benchmarks use *native* input, and Cassandra [**?**] uses the input generated by the Yahoo! Cloud Serving Benchmark (YCSB) [**?**].

PRASEC is a benchmark suites of multi-threaded programs, it includes both server and desktop applications. The 3 programs selected in this paper, *canneal*, *dedup* and *freqmine*, are used for cache-aware simulated annealing, data compression and frequent item-set mining respectively.

Cassandra is distributed storage system widely used in commodity servers. The workload is specified by YCSB, in particular, the portion of reads, updates, scans and/or insert operations. We use the first of the five built-in core workloads. It is update heavy. Specifically, the *workload A* with 8 million records (8GB data) and 10 million operations. Half of the operations are reads, and the rest are updates.

Trace Collection The traces are obtained on an Intel machine with i7-6700K processor(4C/8T) and 16GB memory. All programs are run with 8 threads. We sample accesses missed in last level cache using PEBS. The event

MEM_LOAD_UOPS_RETIRED:L3_MISS

is used. We sample 1 miss in every 10 misses. The information of the sampled traces are showed by Table 1. The page size is 4KB.

Benchmark	Accesses	Pages	Data size (MB)
Cassandra	189,060,547	4,354,988	17,011.67
canneal	70,707,747	235,460	919.77
dedup	1,075,904	229,137	895.07
CG	39,828,501	228,574	892.87
freqmine	6,041,628	192,936	753.66
BT	53,054,258	182,589	713.24
LU	9,433,759	136,669	533.86

Table 1. The length and data size of 7 test programs

The overhead of sampling is less than 10% of the execution. Computing footprint handles about 7 million accesses per second, which means for example 7.7 seconds for *BT*.

Data Partitioning and Cache Modeling We assume that data partitioning is done by one of the two generic schemes, page interleaving and random assignment, described in Section 2.5. For example, if the fraction is 0.7, page interleaving would place 7 out of every 10 pages in the cached fraction, and random assignment would place a page in the cached fraction with 0.7 probability.

Since the partitioning is generic, we assume that the data access pattern in the cached fraction is the same as that of the whole. We compute the footprint of the cached fraction as follows. Let $r = \frac{sz_{fr}}{m}$ be the fraction ratio, $0 \le r \le 1$. Let $fp(x)$ be the footprint of the whole data. The footprint for the cached fraction is computed by $h(x) = fp(x) * r$. Then we compute the cache performance using the formulas from Section 2.4.

As stated in Section 2.6, we use cache modeling to (1) equalize the DRAM and PCM traffic $trfc_{dram} = trfc_{pcm}$, (2) limit the data migration, and (3) minimize the portion of data stored in DRAM.

The generic data partitioning does not require program knowledge or profiling, which is not always practical or accurate. However, when there is program knowledge about the data access pattern, the knowledge can be used to improve data partitioning. For DRAM portion reduction, we may improve the result by allocating frequently accessed data to the cached fraction. The following results are based on the generic data partitioning and show the least benefit that can be achieved by fraction cache optimization.

3.2 DRAM Portion Reduction

Figure 6 shows seven graphs, one for each program showing the minimal DRAM portion on the y-axis and the bound of migration traffic on the x-axis. The DRAM portion is measured as a ratio of the data size, and the migration traffic a ratio of data accesses.

In all graphs, the leftmost is the baseline solution, even division, where the DRAM portion is 50% and migration traffic is 0. The rightmost is complete caching, where the DRAM portion is minimized and migration traffic 50%.

Deductive optimization examines the two-dimensional solution space, where the fraction ratio sz_{fr} ranges from 0.5 to 1 and the DRAM portion ranges from 50% down. For each migration requirement on the x-axis, the fraction cache theory finds the solution with the minimal DRAM size, shown

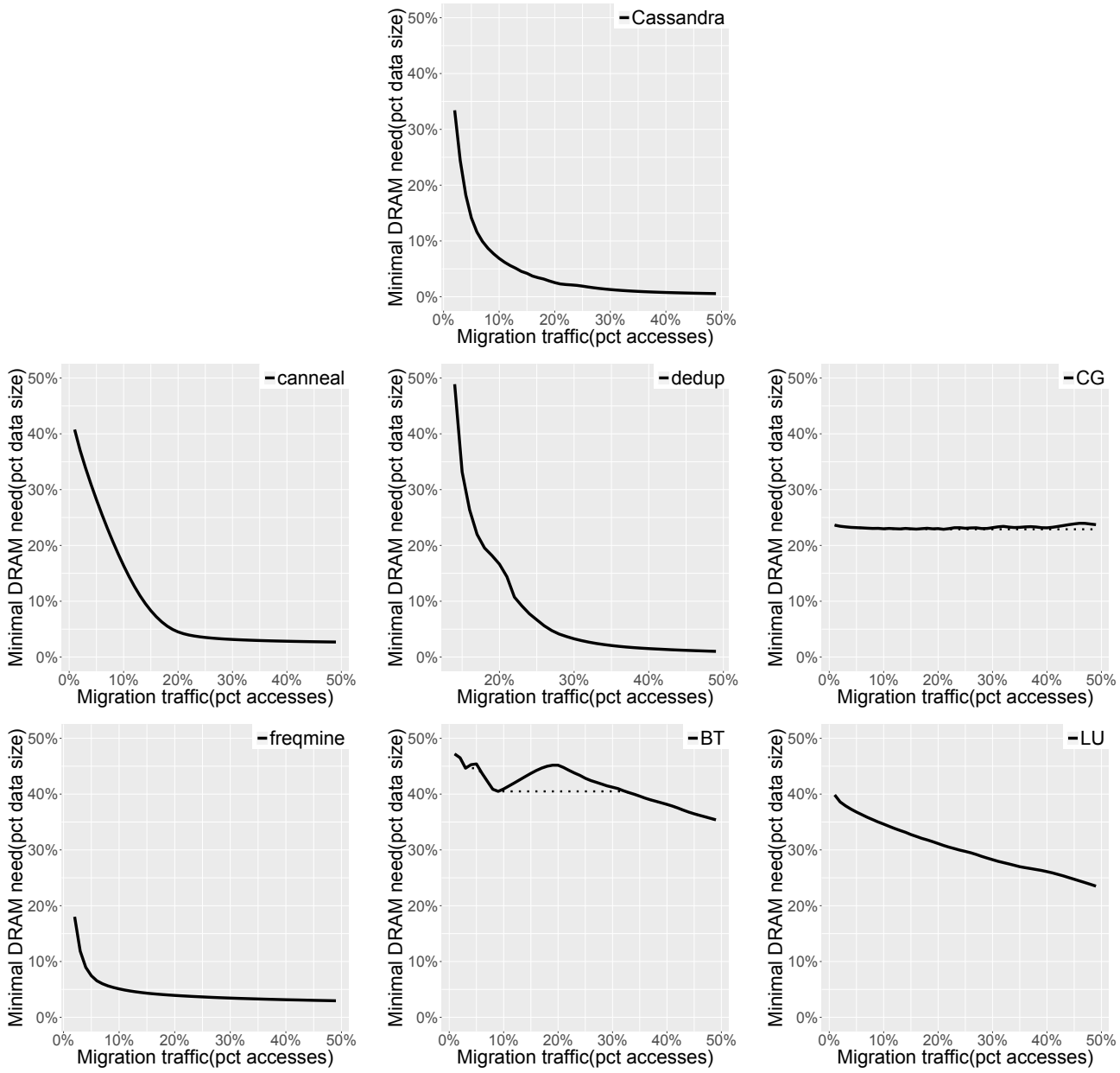

Figure 6. Minimal DRAM portion (y-axis) computed by deductive optimization for *exact* migration cost (x-axis) ranging from 0% to 50% of accesses. In *CG,BT*, the minimal DRAM portion differs for *bounded* migration cost and is shown by dotted lines.

by the y-axis value. The figure shows the cached fraction (selected by deductive optimization) indirectly — it is 0.5 plus the migration traffic (the x-axis value).

The baseline DRAM need is 50% of data size. Fraction cache achieves significant DRAM savings in most of the tests, obtaining DRAM needs below 3% for 4 programs including 0.55% for the one with the largest data size *Cassandra*. Compared to the baseline, the DRAM size reduction is over 16 times (90 times for *Cassandra*). When the migration is within 20%, the DRAM need can be reduced to as low as 3.8% of the program data size and 24.7% on average. If the migration is within 10% of accesses, the DRAM need is 16% for *canneal*, 5% for *freqmine*, and 7% for *Cassandra*, which means a reduction by a factor of 3, 10, and 7 respectively.

Most graphs show a monotone drop of DRAM portion as more migration is permitted. Four programs have a clear inflection point where the rate of DRAM portion reduction is steep before the inflection and flat after the inflection.

Different DRAM portion reductions are caused by different program locality. Figure 7 shows the miss ratio curve for three programs. *Cassandra* has good locality, which means that the miss ratio drops precipitously to near zero when the

cache size increases. *BT* has poor locality, and its miss ratio decreases somewhat linearly when the cache size increases. The case of *LU* is in between. In *Cassandra*, the good locality makes the fraction cache effective at reducing the DRAM portion. The inflection point in the miss ratio curve leads to the inflection point in the minimal DRAM portion. *Cassandra* is representative of the group of programs that also include the three PARSEC benchmarks.

In two programs, *CG,BT*, the minimal DRAM portion is not monotone. At some amounts of migration traffic, it needs to store more data in DRAM to increase the migration traffic. For the optimization, we desire a bound, not an exact amount of the migration traffic. The smallest DRAM portion is always used by the optimization. This result is shown by the dotted line in Figure 6.

4 Related Work

The problem of memory management dates back to the beginning of first general-purpose computers. Some early systems (before 1970) have multiple types of memories. Denning categorized the solutions into two approaches: slave memory and distributed memory [7, pp. 184]. The memories are hierarchical in both approaches. The difference is that slave memory permits direct access only at the top level, while distributed memory allows the access at all levels. With just two memories, the vertical organization is a type of slave memory, and the lateral organization is distributed memory. With three or more memories, lateral connections may form a network more complex than that of a hierarchy, and this complexity adds to the challenges of memory management.

A number of recent studies considered either vertical and lateral organization of DRAM and PCM. Su et al. developed HpMC which selects the best organization based on application locality, both temporal and spatial, for both performance and energy [18]. In separate studies, Liu et al. developed a system called Memos for lateral memory management [13]. Memos builds on an operating system kernel extension, which supports online program monitoring [12], and uses the program characteristics to guide the page placement on DRAM and PCM and dynamic page migration between them. The evaluation showed broad improvements, not just throughput but also QoS, NVM latency, energy consumption, and NVM lifetime [13].

The studies by Su et al. and Liu et al. show convincingly the merit of lateral organization of memory and the need to consider multiple objectives. Inspired by the previous work, the fraction cache aims to broaden the consideration to include the design space between the vertical and lateral solutions. Our vision of the memory equalizer hinges on this broad solution space to satisfy multiple objectives and on deductive optimization to search this space efficiently without exhaustive testing.

For example, lateral solutions must solve the problem of data migration. In both HRank and Memos, pages with more frequent accesses are moved from PCM to DRAM. In a fraction cache, similar data migration happens as a result of the caching mechanism and therefore can be controlled and tuned using existing performance models. The large solution space and deductive optimization may generalize previous work to support more memory types and different objectives.

Many studies have designed cache for multiple objectives such as performance, fairness and QoS. For example, the concept of *sharing incentive* means that the participants would pool their resource if the per-participant performance with sharing is at least the performance without sharing [25]. Baseline optimization is developed to optimize cache partition-sharing to maximize the throughput given the sharing incentive as the constraint [24]. Similar to these techniques, DRAM portion reduction in Section 2.6 treats bandwidth and migration requirements as constrains and then minimizes the DRAM size.

While the fraction cache is a technique for memory management, it is similar to two recent techniques of cache management. For LRU cache and given the MRC, Talus divides data and partitions the cache and by cleverly choosing their parameters, Talus ensures that the new MRC lies within the convex hull of the original [2]. Another system, SLIDE, removes miss ratio "cliffs" for both LRU and non-LRU policies including ARC, 2Q and LIRS [21]. The convex transformation by Talus and SLIDE has at least two benefits. One is miss-ratio reduction by removing the "cliffs", and the other benefit optimal cache partitioning in multi-programming, which can be done by a linear-time greedy algorithm [17, 19] (otherwise the cost is number of programs times cache size square as solved in optimal cache partition-sharing [4, 24]).

Talus and SLIDE divide both program data and its cache into two partitions and use one cache partition for each data partition. Logically, the fraction cache is more general. Talus and SLIDE are equivalent to a three-fraction cache, with the size of the third fraction set to zero. In this difference we see the distinction between traditional hierarchical and current lateral memory management. A lateral structure means direct memory access not through a cache, which is irrelevant to the purposes of Talus and SLIDE. In mechanisms, however, the fraction cache solves similar problems, including partitioning data and modeling MRC. This paper assumes randomly partitioned data (in LRU caches) and uses the HOTL model. With HOTL, the fraction cache may partition data based on its footprint and perform better than random partitioning, in either memory management or caching.

Finally, modern processors may use multiple memories even when they are of the same type. IBM POWER8 has 8 DRAM channels, and each may be connected a Centaur chip [16]. A Centaur chip has 16MB eDRAM. Therefore, the processor has up to 128MB memory separate from DRAM. Intel Knights Landing (KNL) has 16GB Multi Channel DRAM (MCDRAM) on its CPU package [15]. On POWER8, eDRAM

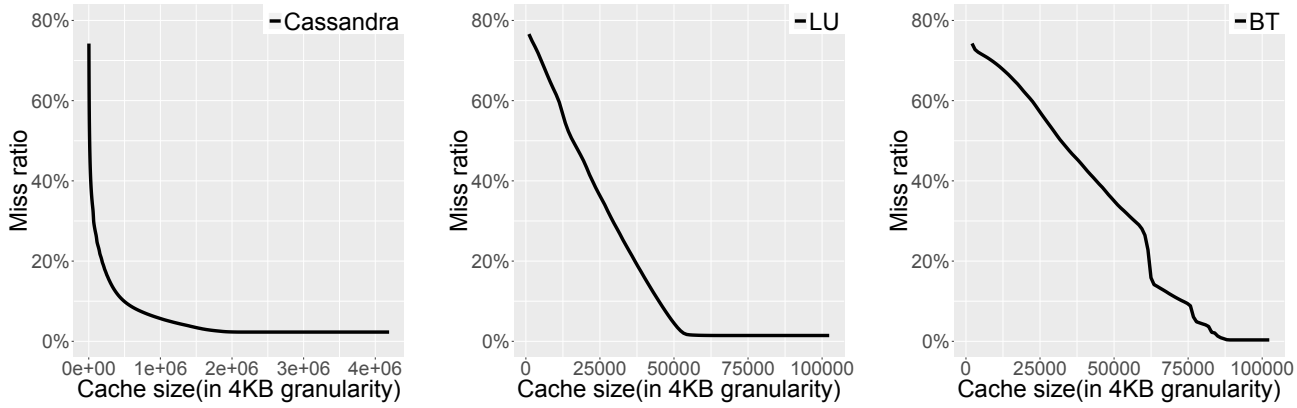

Figure 7. Miss ratio curve of Cassandra, LU and BT, assuming 50% data is cached.

is used as L4 buffer cache (directory in SRAM) to reduce the latency and energy cost of L3 misses. On KNL, MCDRAM has three memory modes: cache, flat, and hybrid. Data access in MCDRAM has a similar latency but much greater bandwidth compared to DRAM. Both eDRAM and MCDRAM are aimed to augment DRAM and represent new system opportunities and challenges in the study of lateral memory management.

5 Summary

This paper has presented a vision of memory equalizer for lateral memory management. The key is an abstract design called *fraction cache* which is organized as a two level exclusive cache. Its parameters, the fraction, the size of the memory, the traffic of processor and inter-memory communication, encompasses a large solution space. In practice, a user may specify performance objectives and leave the tuning and optimization to an automatic tool. As a demonstration, we use fraction cache to minimize the need for DRAM while limiting the cost of data migration. When the migration is bounded to no more than 20%, the DRAM size can be reduced to as low as 3.8% of the program data size and 24.7% on average.

We believe that our technique is the first to observe and characterize the inflection points in the trade off between DRAM size and migration traffic. We expect that similar patterns happen in other problems of lateral memory management, and the fraction cache and its deductive optimization will be a useful tool.

Acknowledgments

The authors would like to express sincere thanks to Peter Denning for his insightful criticism and the abbreviations SP and EQ and to Michael Scott and Lei Liu for the technical discussion on related ideas. The authors would also like to thank Alex Benishek, Zhizhou Zhang and the anonymous referees of MEMSYS 2017 for their careful corrections and suggestions. The image of the equalizer in Figure 1 is a screen copy of the software created by Jingbing Yuan. The work is supported by the National Science Foundation (Contract No. CCF-1717877, CCF-1629376, CNS-1319617), an IBM CAS Faculty Fellowship, the National High-tech Research and Development Program of China and the National Natural Science Foundation of China (Contract No. 2015AA015303, No. 61433019). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

References

[1] David H. Bailey. 2011. NAS Parallel Benchmarks. In *Encyclopedia of Parallel Computing*. 1254–1259.

[2] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A simple way to remove cliffs in cache performance. In *Proceedings of HPCA*. 64–75. https://doi.org/10.1109/HPCA.2015.7056022

[3] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of PACT*. 72–81.

[4] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal Cache Partition-Sharing. In *Proceedings of ICPP*.

[5] Dong Chen, Chencheng Ye, and Chen Ding. 2016. Write Locality and Optimization for Persistent Memory. In *Proceedings of the International Symposium on Memory Systems*. 77–87. https://doi.org/10.1145/2989081.2989119

[6] Peter J. Denning. 1968. The working set model for program behaviour. *Commun. ACM* 11, 5 (1968), 323–333.

[7] Peter J. Denning. 1970. Virtual Memory. *Comput. Surveys* 2, 3 (1970), 153–189. https://doi.org/10.1145/356571.356573

[8] Peter J. Denning. 1980. Working sets past and present. *IEEE Transactions on Software Engineering* SE-6, 1 (Jan. 1980).

[9] Peter J. Denning and Stuart C. Schwartz. 1972. Properties of the working set model. *Commun. ACM* 15, 3 (1972), 191–198.

[10] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic Modeling of Data Eviction in Cache. In *Proceedings of USENIX ATC*. 351–364. https://www.usenix.org/conference/atc16/technical-sessions/presentation/hu

[11] Edward G. Coffman Jr. and Peter J. Denning. 1973. *Operating Systems Theory*. Prentice-Hall.

[12] Lei Liu, Yong Li, Chen Ding, Hao Yang, and Chengyong Wu. 2016. Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random? *IEEE Trans. Comput.* 65, 6 (2016), 1921–1935. https://doi.org/10.1109/TC.2015.2462813

[13] Lei Liu, Hao Yang, Yong Li, Mengyao Xie, Lian Li, and Chenggang Wu. 2016. Memos: A full hierarchy hybrid memory management framework. In *IEEE International Conference on Computer Design (ICCD)*.

[14] Sabela Ramos and Torsten Hoefler. 2017. Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL. In *Proceedings of IPDPS*. 297–306. https://doi.org/10.1109/IPDPS.2017.30

[15] Avinash Sodani, Roger Gramunt, Jesús Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* 36, 2 (2016), 34–46. https://doi.org/10.1109/MM.2016.25

[16] William J. Starke, Jeffrey Stuecheli, David Daly, J. S. Dodson, Florian Auernhammer, Patricia Sagmeister, G. L. Guthrie, C. F. Marino, M. S. Siegel, and Bart Blaner. 2015. The cache and memory subsystems of the IBM POWER8 processor. *IBM Journal of Research and Development* 59, 1 (2015). https://doi.org/10.1147/JRD.2014.2376131

[17] Harold S. Stone, John Turek, and Joel L. Wolf. 1992. Optimal Partitioning of Cache Memory. *IEEE Trans. Comput.* 41, 9 (1992), 1054–1068. https://doi.org/10.1109/12.165388

[18] Chun-Yi Su, David Roberts, Edgar A. León, Kirk W. Cameron, Bronis R. de Supinski, Gabriel H. Loh, and Dimitrios S. Nikolopoulos. 2015. HpMC: An Energy-aware Management System of Multi-level Memory Architectures. In *Proceedings of the International Symposium on Memory Systems*. 167–178. https://doi.org/10.1145/2818950.2818974

[19] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. 2004. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing* 28, 1 (2004), 7–26.

[20] Carl A. Waldspurger, Nohhyun Park, Alexander T. Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 95–110. https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger

[21] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *Proceedings of USENIX ATC*. 487–498. https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger

[22] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. 2011. Linear-time Modeling of Program Working Set in Shared Cache. In *Proceedings of PACT*. 350–360.

[23] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: a higher order theory of locality. In *Proceedings of ASPLOS*. 343–356.

[24] Chencheng Ye, Jacob Brock, Chen Ding, and Hai Jin. 2017. Rochester Elastic Cache Utility (RECU): Unequal Cache Sharing is Good Economics. *International Journal of Parallel Programming* 45, 1 (2017), 30–44. https://doi.org/10.1007/s10766-015-0384-3

[25] Seyed Majid Zahedi and Benjamin C. Lee. 2014. REF: resource elasticity fairness with sharing incentives for multiprocessors. In *Proceedings of ASPLOS*. 145–160.