Statistical Caching for Near Memory Management

Dong Chen*
jameschennerd@gmail.com
National University of Defense Technology
Changsha, China

Fangzhou Liu
fliu14@cs.rochester.edu
University of Rochester
Rochester, NY

Mingyang Jiao
jiao@rochester.edu
University of Rochester
Rochester, NY

Chen Ding
cding@cs.rochester.edu
University of Rochester
Rochester, NY

Sreepathi Pai
sree@cs.rochester.edu
University of Rochester
Rochester, NY

ABSTRACT

Modern GPUs often use near memory or high-bandwidth memory, which may be managed as cache when the application data is too large to fit in the near memory. Unlike CPU caches, the near memory cache has a much larger size. A recent approach is statistical caching, which shows near optimal results when managing large memory for file caching.

The prior work is ideal and not practical. This paper outlines two extensions. It first formulates a new caching algorithm called least expected use (LEU) replacement and shows, through examples, that the statistical solution automatically integrates two otherwise disparate policies. Then the paper describes a system design to implement LEU. To position the new design for discussion, the paper draws parallels with two familiar ideas, branch prediction and spectral analysis, and considers a set of opportunities and challenges of achieving statistical caching in near memory.

CCS CONCEPTS

• **Computer systems organization** → **Heterogeneous (hybrid) systems**; • **Software and its engineering** → **Memory management**.

ACM Reference Format:
Dong Chen, Fangzhou Liu, Mingyang Jiao, Chen Ding, and Sreepathi Pai. 2019. Statistical Caching for Near Memory Management. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3357526.3357557

1 INTRODUCTION

Memory technology is a field with rapid changes, including the new organization through stacking, e.g. hybrid memory cube (HMC), and new silicon interconnect, i.e. high bandwidth memory (HBM). Recent studies in MEMSYS have extended DRAMSim to model its performance [15] and used the new types of memory to complement traditional DRAM, for example, by adding memory bandwidth as shown by BATMAN [4] and enabling fast data processing shown by Lloyd and Gokhale [8].

Modern GPUs possess tens of gigabytes of high-bandwidth onboard memory that is used to store data for GPU programs (i.e. kernels). The limited size of this memory has prevented GPU programs from working on large datasets. Traditionally, all data needed by the GPU kernel must be copied into GPU memory before the kernel executes. After the kernel executes and produces results – which are also stored in GPU memory – the CPU can copy the results to its memory. Recently, however, GPU memory is now unified with CPU memory [12], enabling on-demand migration of pages between CPU and GPU during runtime. Since moving these pages to the GPU over the PCIe bus is still slow, we ask: when a page miss occurs in the GPU and must be handled by the GPU Memory Management Unit (GMMU), is there a way to select the victim page to optimally reduce CPU–GPU communication? In this on-demand scenario, GPU memory is actually acting as a page cache for memory backed by the system memory. Thus, this is the traditional cache replacement problem.

To manage GPU memory, we propose a new page replacement policy called Least Expected Use (LEU). LEU is a new run-time approach based on statistical caching. In this position paper, we describe the policy and outline how it can possibly be implemented.

LEU extends the recent work by Li et al. [7] of a statistical caching policy called OSL. Unlike existing policies such as LRU and its variants, and even the ideal OPT which are based on predicting the exact reuse, OSL is statistical caching based on a distribution of possible reuses in the future, which we will describe more in Section 2. Li et al. [7] evaluated its performance. We reproduce one of their graphs in Figure 1, showing that OSL manages a large amount of cache space, up to 60GB in this test, and significantly outperforms LRU and achieves the same performance as OPT (or better because OSL is a variable-size cache policy[1]).

Although promising in its potential, OSL is impractical for two reasons. The first is the large space overhead needed to store the statistics. Secondly, OSL is for variable-sized caches, so the cache size changes dynamically. The new LEU policy solves these two problems. Its relation with OSL is shown in Figure 2. Unlike OSL, LEU is based on program code and can bound the space overhead

*This work was done when the author was a graduate student at University of Rochester.

[1]The optimal variable size policy is VMIN, compared also in Figure 1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MEMSYS '19, September 30-October 3, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7206-0/19/09…$15.00
https://doi.org/10.1145/3357526.3357557

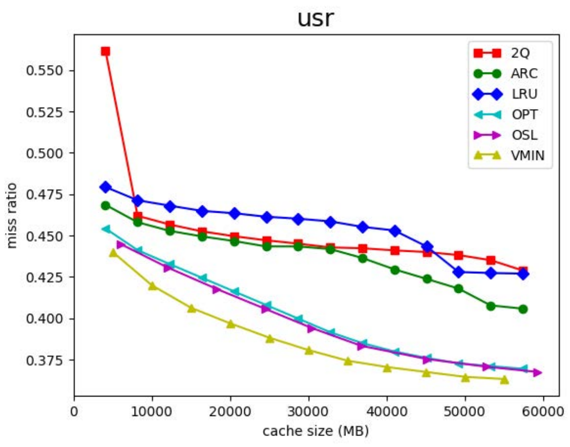


Figure 1: Performance of statistical caching: the cache miss ratio curves collected from `usr` benchmark shows that OSL could outperform existing policies LRU, 2G and ARC and achieve the optimal performance of OPT, shown by Li et al. [7]

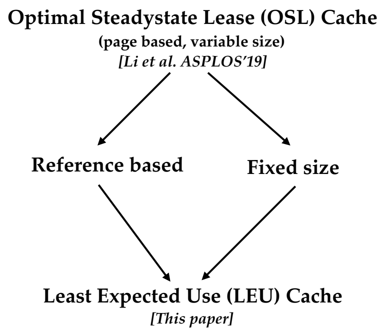

Optimal Steadystate Lease (OSL) Cache
(page based, variable size)
[Li et al. ASPLOS'19]
Reference based
Fixed size
Least Expected Use (LEU) Cache
[This paper]

Figure 2: The two pronged improvement of statistical caching [7]: using program information (left) and managing fixed size cache (right).

of storing statistics, and it manages fixed-size cache rather than variable-size cache by OSL.

Using statistics, LEU subsumes past policies and integrates them automatically. We will show through an example that LEU automatically achieves the equivalent effect of combining LRU and MRU.

There is significant time overhead in statistical optimization. Fortunately, most LEU operations, especially those computing the priority for each page, are massively parallel, so there is potential synergy in using statistical caching on GPUs.

The rest of the position paper is organized as follows: Section 2 will briefly introduce the lease cache technique. We will discuss our new approach at Section 3. To help considering the new proposal, in Section 4, we point out remarkable similarities with two successful techniques: branch prediction and spectral analysis. Section 5 and Section 6 will list several potential directions of our new techniques and summarize.

2 BACKGROUND: STATISTICAL CACHING

Recently, Li et al. [7] developed statistical caching. The key idea is that while obtaining oracular knowledge is in general unrealistic, clairvoyance can still be approached through probabilistic prediction.

Given a data access trace, the *forward reuse interval* is defined for each access as the number of accesses between this current access and the next access to the same data block [2]. In this paper, we shorten it to *reuse interval (RI)*.

Let data item a be reused 4 times in "*aabb aabb* ..." An ideal solution requires knowing the next reuse each time a is accessed [1]. A probabilistic prediction states that the reuse interval is 1 for half of the a reuses and 3 for the other half. Statistical caching uses statistics, rather than exact knowledge of future accesses. Li et al. [7] developed a technique to select the best lease *per data block*, called Optimal Steadystate Lease (OSL).

Unlike traditional caches such as the LRU cache, the lease cache has a simple control. Each access comes with a lease, and the cache stores the data block for the length of the lease. Either the data block is accessed again and the lease is renewed or it is evicted when the original lease expires. In OSL, the lease is measured by "program" time, i.e. the number of accesses.

It helps to draw an analogy with an automatic water faucet. When a faucet detects a user's hand, it discharges water for a period of time. This time can be viewed as a lease. If a hand is detected again, the lease is renewed; otherwise, the lease expires and the water valve is closed. If a lease is too short, water stops while a user is still washing, but if it is too long, it wastes water. Essentially, statistical caching finds the lease that statistically maximizes the benefit over the cost.

3 LEU CACHE REPLACEMENT

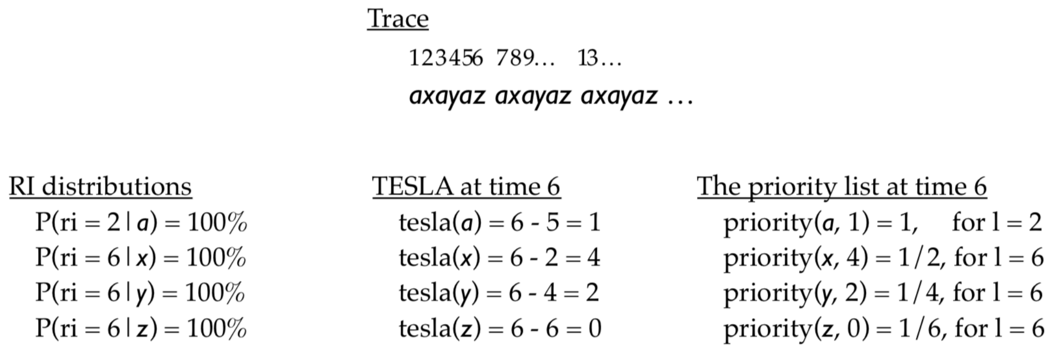
In this section, we talk about the page replacement policy we designed for GMMU, which is called Least Expected Use (LEU). The LEU algorithm is parameterized by data block granularity. In this paper, we treat (virtual memory) pages as data blocks.

Unlike statistical caching, fixed size caches cannot use *lease* anymore, as using leases to guide fixed size cache replacement is incomplete – a discussion we defer to Section 4.2.

3.1 PPUC Based Priority

LEU is a new cache replacement policy, which uses a metric called *profit per unit of cost (*PPUC*)*. In *PPUC*, the profit is the expected hit ratio, and the cost is the amount of cache occupied over time. Their ratio, *PPUC*, is the hit rate per unit of cache space and per unit of time. For simplicity of presentation, we first compute the *PPUC* for each data block. By ranking a data block by its *PPUC*, keeping high-*PPUC* blocks while evicting low-*PPUC* blocks, LEU maximizes the overall hit ratio. The actual LEU is based on references – i.e. memory access instructions – and we show this at the end of the section.

Figure 3 gives the running example which we use to show the *PPUC* calculation and ranking first for data blocks and then later extend to references. The example is a trace with frequent reuses of block a and repeated iterations of blocks x, y, z. Jiang and Zhang [6] characterized individual and cyclic patterns as strong and weak



Trace

123456 789... 13...

axayaz axayaz axayaz ...

RI distributions	TESLA at time 6	The priority list at time 6
P(ri = 2 | *a*) = 100%	tesla(*a*) = 6 - 5 = 1	priority(*a*, 1) = 1, for l = 2
P(ri = 6 | *x*) = 100%	tesla(*x*) = 6 - 2 = 4	priority(*x*, 4) = 1/2, for l = 6
P(ri = 6 | *y*) = 100%	tesla(*y*) = 6 - 4 = 2	priority(*y*, 2) = 1/4, for l = 6
P(ri = 6 | *z*) = 100%	tesla(*z*) = 6 - 6 = 0	priority(*z*, 0) = 1/6, for l = 6

Figure 3: An example trace, its RI distributions and the LEU priority values at time 6.

locality. Naturally, strong locality data a should be given a higher priority over weak locality data x, y, z.

Let the RI distribution for block b be $P(ri|b)$, i.e. the percentage of accesses to b with the reuse interval ri. Given an access trace, at the tth access, let t_b be the last access time to b. We define *time elapsed since last access* (TESLA) as $t - t_b$. For the running example, Figure 3 shows the RI distributions and the TESLAs at a particular time in the trace.

We compute the profit for each block b, i.e. the hit ratio, given how long it stays in cache. Following the OSL terminology, we call this duration the *lease*. A lease l means to store b for l accesses after its last access, i.e. staying in cache till time $t_b + l$. The hit ratio is computed from the RI distribution $P(ri|b)$ as a function of l as follows:

$$hr(b, tesla, l) = \frac{P(ri \le l|b) - P(ri \le tesla|b)}{1 - P(ri \le tesla|b)} \quad \text{for } l > tesla \qquad (1)$$

The equation computes $hr(b, tesla, l)$, which is the expected hit ratio of b when keeping it in cache for time l after b's last access. The expectation is adjusted based on $tesla$. Consider the case $tesla = 0$, that is, the time when b is accessed. The hit ratio is $P(ri \le l|b)$, i.e. the portion of reuses whose RI is no more than the lease. When $tesla > 0$, Eq. 1 calculates the conditional expectation, since now we know that during the elapsed time, there is no access to b. The probability of $P(ri \le tesla|b)$ is known to be 0.

While the hit ratio is the profit, the PPUC of b (after elapsed time $tesla$) is the hit ratio divided by the cost, which is the cache occupancy of b for lease l. The cost is the incremental, i.e. the additional time needed to reach l.

$$PPUC(b, tesla, l) = \frac{hr(b, tesla, l)}{l - tesla} \qquad l > tesla \qquad (2)$$

The priority of block b in the LEU cache is its largest PPUC. The next equation computes the PPUC for all $l > tesla$ and chooses the largest.

$$priority(b, tesla) = \max_{l > tesla} PPUC(b, tesla, l)$$

When evaluating data blocks for cache replacement, we compare their LEU priority values. For the running example, Figure 3 shows the priority at time 6 of the trace, where the strong locality data a has the highest priority, and the low locality data is also ordered. A reader may notice that the latter ordering is the same as the one that would have been used by the most recently used (MRU) replacement policy, and MRU is the best policy for cyclic accesses.

To further demonstrate how LEU effectively integrates LRU and MRU, consider the trace "*aaxyz aaxyz ...*" At time $t = 5$, LRU would fail to cache a unless the cache size is 4 or more. If we compute the priority values, we would obtain 1 for a and 1/2, 1/3, 1/4 respectively for x, y, z. Hence in LEU, a is always cached first, and the rest are ordered by MRU, same as the case in Figure 3.

3.2 Reference-based LEU

LEU has two key ideas. First, it manages data blocks based on which instruction accessed it last, referred to as the reference (instruction) r. Second, it ranks these references by *PPUC*. So far, we have computed the *PPUC* for each data block rather than for each reference. Note that in the special case where each reference only accesses a single data block, LEU would be identical to our previous description.

While LEU takes the input of an RI distribution, it actually does not matter whether the distribution comes from accesses to one data block or many. Therefore, it requires no change to the equations when computing the priority for a reference. In the running example, we may consider the trace as one generated by a program with two references r_1, r_2, where all a accesses are from r_1 and x, y, z accesses from r_2. Consider r_2. If LEU was based on data blocks, it would need a RI distribution for each of x, y, z, as shown in Section 3 in Figure 3. Instead, LEU is based on references. It has just one RI distribution for all accesses of x, y, z. The following table shows the RI distribution for r_2:

r_2 RI distribution	r_2 priorities at $t = 6$
$P(ri = 6\|r_2) = 100\%$	$priority(r_2, x, 4) = 1/2$
	$priority(r_2, y, 2) = 1/4$
	$priority(r_2, z, 0) = 1/6$

Although r_2 has just one RI distribution, it computes different priorities for x, y, z, because they have different *tesla*.

Reference based LEU handles general data references. First, a reference may access many data blocks, which are individually ranked by LEU. Second, multiple references may access the same data block, and the LEU ranking is based on the reference that makes the last access. Third, at every point in a program execution, LEU assigns a priority value to all its data blocks, however, the priority does not depend on the cache size. We will state without proof that LEU is a stack algorithm and has all its theoretical properties.[2] There exists an LEU stack distance, similar to the LRU stack distance (often called the reuse distance). However, we do not explore the LEU stack distance in this paper.

3.3 Near Memory Management

In this section, we show how GPU coordinates with CPU to implement LEU. We design a new module called Reuse Interval Memory Manager (RIMM) and integrate it on both host and device side, we name these two parts *RIMM host* and *RIMM device* respectively. RIMM host contains a fixed size Reference Reuse Interval Table (Ref RIT) which maps each reference to its reuse interval distribution. Then the algorithm we mentioned in previous 2 sections computes *PPUC* for each page and its priority.

Figure 4 demonstrates how a CPU and a GPU coordinate to implement LEU. An RIMM device will communicate with an RIMM host under two conditions: 1) the Trace Sampler in RIMM finds a reuse or 2) the Computing Unit (CU) requests data that is not present in the Translate Lookaside Buffer (TLB) or in the GMMU. When there forms a reuse, the RIMM device will send a reference id and its reuse interval to the RIMM host and the latter one updates its Ref RIT. Note that this communication is batched, each communication is not just for one page request.

Now let's consider the second condition, when the CU incurs a page fault in the GPU. The RIMM device will send the reference id, its TESLA value and the demand page id to the host. The RIMM host, when receiving this message, will pass the missing reference's reuse interval distribution to the *PPUC* computing algorithm and derive the *PPUC* of the requested page. The missing page id *pid* together with its *PPUC* will then be sent to the priority list, which orders each demand page by its *PPUC* and sends back the replacement decision to device.

LEU can be either a software or a hardware solution. Next, we'd like to envision one pure software implementation of LEU. To integrate LEU in GPU, two extra components should be added: 1) a trace sampler, which should trigger a signal to update RIMM when there is a reuse, and 2) a GMMU page fault handler, which guides the page replacement based on the priority values. All these two parts could be implemented without extra hardware support. The trace sampler could be implemented by GPU hardware performance counters (e.g., the *CUDA Profiling Tools Interface (CUPTI)* [10] for NVIDIA GPU), which allows developers to do event samplings and customize its call back. To handle page fault, Power et al. [11] gave one hardware solution, which modified CPU interrupt return microcode and added one extra CPU register to handle GPU page fault. But extra hardware support is not necessary, Tanasic et al. [14] gave 3 possible design of GPU page handler (*Wrap Disabling*, *Replay Queue* or *Operand log*), which could remove CPU involvement in GPU page fault.

[2]As shown in the seminal paper by Mattson et al., a stack algorithm observes the inclusion property, has monotone miss ratios (no Belady anomaly), and can be evaluated for all cache sizes by one-pass simulation [9].

4 ANALOGY WITH FAMILIAR IDEAS

This section illustrates the new idea by drawing an analogy with two familiar ideas, branch prediction and spectral analysis. As pointed out by Grossman [5], an analogy "sparks discussion and is easy to remember [and] can inspire new research ideas and let one adapt terminology from one side of the analogy for use in the other." Still, it is important "to understand that an analogy ... is not a complete argument; it is an introductory remark."

4.1 Analogy with Branch Prediction

We show first how caching and branch prediction are both prediction problems and then how LEU is based on program code, as is the case for modern branch predictors.

Branch prediction reduces the latency of branching by predicting the most likely outcome of a branch and speculatively executing along the predicted path. A success is a prediction hit, and a failure is a prediction miss. Caching reduces the latency of memory access by predicting the data blocks with the most likely access and speculatively storing them in the cache. A success is a cache hit, and a failure is a cache miss.

Caching and branch prediction are both statistical optimization. A program still runs correctly at a miss. Its speed is maximized if the miss ratio is minimized. Both must maximize profit over cost. Profit comes from speculation based on correct prediction, i.e. speculative execution at a branch target or caching data blocks before next use. Cost comes from waste of the speculative work and lost opportunity at mis-prediction.

The difference, however, is how they collect and use statistics. Previous caching solutions collect information from memory access but not from program code. Branch prediction is based on program code, in particular, branch instructions. It gathers dynamic information on a static branch.

Like branch prediction but unlike traditional caching, LEU collects information on memory instructions. Branch prediction analyzes the history of a branch, while LEU gathers the access information of a memory load or store. Both require and target program code. LEU finds most likely data usage by a memory instruction, just as branch prediction finds the most likely direction of a branch.

In implementation, branch prediction uses a predictor table. The table is a fixed size and indexed by the addresses of branch instructions. The space cost is bounded, and space utilization is maximized by evicting infrequently executed branch instructions from the table. The table used by LEU, Ref RIT, is fixed size and indexed by the addresses of memory instructions. The space cost is controlled and utilization maximized by evicting infrequently executed memory instructions.

Prediction is based on program behavior. Branch predictors use taken or not taken branches. LEU uses data reuse behavior, which has no direct parallel with the branching behavior. Next we explain it using a different analogy.

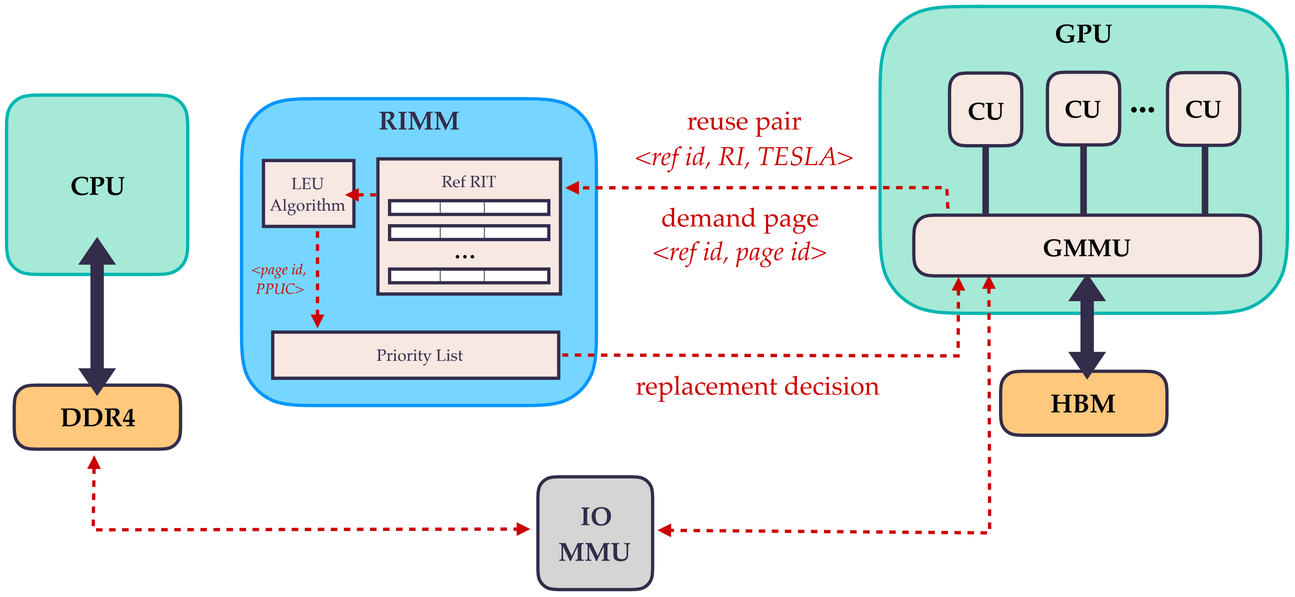


Figure 4: Overview of LEU-based GPU page migration management.

.

4.2 Analogy with Spectral Analysis

There is a long history in understanding a program execution as a signal [13]. Cache behavior is periodical, and the periodicity is measured by miss frequency. The miss frequency depends on two factors: data reuse in a program and the management of the cache (including its size). LEU is designed to minimize the miss frequency for a given program and cache size. Each reuse in a program is quantified by an RI, while different reuses have different RIs. The spectrum of reuse frequencies is shown by the RI distribution. Based on the RI distribution, LEU analyzes the spectrum of reuses to manage the cache.

The techniques of RI sampling and the table of RI distributions are analogous to those used by a spectrum analyzer, in particular, frequency sampling and the limits on the frequency range and precision which affect the cost and the effectiveness of the instrument.

OSL manages variable size caches. For fixed size caching, a naive approach is to use the distribution to compute the average RI as the expected reuse and rank data blocks by it. It is easy to see that this solution, although simple, is incomplete in that it does not optimize for all cache sizes. The expected RI is shorter than the longest RI. Once the cache is sufficiently large to hold all data blocks for their expected RI, we can no longer rank data blocks to utilize additional cache space. This is a problem regardless of whether TESLA is used with the expected RI. Similar incompleteness happens if we invoke OSL to compute the best lease at each access and use it as priority. In comparison, LEU computes the most profitable RI as the yardstick for ranking. As the cache size increases, shorter reuses are all hits, and the LEU ranking is made on long RIs. The ranking stops only when all RIs are cached, and there is no capacity misses. LEU is therefore a complete solution.

5 OPPORTUNITIES AND CHALLENGES

Caching requires joint optimization, i.e. orchestration among all memory accesses. It differs from problems where individual optimization suffices. For example, in branch prediction the program performance is maximized if each branch is predicted with its best possible accuracy.[3] In caching, increasing the hit frequency for a load instruction would mostly likely cause more misses elsewhere.

We may call this phenomenon *optimization interference*. In branch prediction, optimizing one branch does not negatively affect the optimization at other branches. The optimization of different branches does not interfere. For caching, if without interference, we would be able to trivially optimize for any single memory instruction by allocating the whole cache to store its data. Because all memory instructions share the same cache, the optimization problem in caching is optimization interference.

To achieve joint optimization, LEU considers the full spectrum of reuses by using the RI distribution, and it considers the RI distribution of all memory instructions. It computes the PPUC for each instruction, ranks them, and stores in the cache the data with the highest PPUC.

Spectral analysis enable joint optimization not attempted by previous policies such as LRU and its variants. A problem, however, is the overhead. In OSL, the space overhead is proportional to the size of data. LEU is reference based, so its overhead is proportional to the number of memory instructions. Using an RRI table, this cost can be bounded by a constant, in a way analogous to the predictor table used in branch prediction, as discussed in Section 4.1.

[3]Sometimes two branches correlate, and considering multi-branch correlation adds accuracy. Even in this case of correlated prediction, the benefits are still independent when being counted into the total.

LEU may be extended to differentiate data blocks by the predominant access type, whether a block is mostly read only and whether it has frequent writes. In MEMSYS 2016, Brock et al. [3] extended the working set model and the ideal VMIN model to consider different read/write time-costs. Both models require variable size caches. The working set policy is based on the last access time (similar to LRU). LEU may improve the prior policy by managing the fixed cache and by using spectral analysis.

An alternative design is to move PPUC calculation to RIMM device. As PPUC calculation for different pages and different *tesla*s are independent, moving to device side can take advantage of massive parallel processing. PPUC calculation can also be implemented by adding specialized hardware components as Lloyd and Gokhale did for near memory key/value lookup acceleration. Their results show 2.9-12.8X speedups compared to CPU lookup [8].

6 SUMMARY

In this paper, we have presented the LEU replacement algorithm which uses the RI distribution of each reference to assign priority to manage a fixed size cache of all sizes. Based on statistics, it alone is able to achieve the effect of LRU and MRU, which would have required special selection in past solutions. Looking forward, the LEU system design can follow a practical framework similar to that of branch prediction, and its performance can be studied and improved through spectral analysis.

ACKNOWLEDGMENTS

The authors would like to thank Shawn Maag and Wesley Smith for doing the proof-reading, and the comments and suggestions of the anonymous reviewers. This research was supported in part by the National Science Foundation (Contract No. CCF-1909099, CCF-1717877, CCF-1629376) and an IBM CAS Faculty Fellowship.

REFERENCES

[1] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101.
[2] Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture* 51, 4 (2005), 223–250.
[3] Jacob Brock, Chencheng Ye, and Chen Ding. 2016. Replacement Policies for Heterogeneous Memories. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 232–237. https://doi.org/10.1145/2989081.2989123
[4] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2017. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with stacked-DRAM. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '17)*. ACM, New York, NY, USA, 268–280. https://doi.org/10.1145/3132402.3132404
[5] Dan Grossman. 2007. The transactional memory / garbage collection analogy. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*. 695–706.
[6] Song Jiang and Xiaodong Zhang. 2005. Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance. *IEEE Trans. Computers* 54, 8 (2005), 939–952.
[7] Pengcheng Li, Colin Pronovost, Benjamin Tait, William Wilson, Jie Zhou, Chen Ding, and John Criswell. 2019. Beating OPT with Statistical Clairvoyance and Variable Size Caching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems. accepted, to appear*.
[8] G. Scott Lloyd and Maya Gokhale. 2017. Near memory key/value lookup acceleration. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 26–33. https://doi.org/10.1145/3132402.3132434
[9] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM System Journal* 9, 2 (1970), 78–117.
[10] NVIDIA. [n. d.]. CUPTI: CUPTI Documentation, v10.1, May 2019. https://docs.nvidia.com/cupti/Cupti/r_main.html.
[11] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (2014), 568–578.
[12] Nikolay Sakharnykh. 2016. Beyond GPU Memory Limits with Unified Memory on Pascal. *https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/*.
[13] Jeffrey R. Spirn. 1977. *Program behavior : models and measurements*. Elsevier New York. x, 277 p. : pages.
[14] I. Tanasic, I. Gelado, M. Jorda, E. Ayguade, and N. Navarro. 2017. Efficient Exception Handling Support for GPUs. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 109–122.
[15] Rommel Sánchez Verdejo, Kazi Asifuzzaman, Milan Radulovic, Petar Radojkovic, Eduard Ayguadé, and Bruce Jacob. 2018. Main memory latency simulation: the missing link. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 107–116. https://doi.org/10.1145/3240302.3240317