IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 17, 2022, accepted May 31, 2022, date of publication June 6, 2022, date of current version June 9, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3180327

Efficient Stack Distance Approximation Based on Workload Characteristics

SOOYOUNG LIM AND DONGCHUL PARK, (Member, IEEE)
Department of Software, Sookmyung Women's University, Seoul 04310, South Korea

Corresponding author: Dongchul Park (dpark@sookmyung.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) Grant through the Korean Government [Ministry of Science and Information and Communication Technologies (ICT) (MSIT)] under Grant 2020R1F1A1048485, and in part by the Sookmyung Women's University Research under Grant 1-2203-2011.

ABSTRACT A stack distance of a reference is the depth from which the reference must be extracted from a stack. It has been widely applied to a variety of applications utilizing temporal locality information. However, calculating an exact stack distance requires significant computational complexity that is impractical for online production systems. This paper proposes an efficient algorithm that approximates a stack distance based on workload characteristics. The proposed algorithm utilizes two simple reference metrics: inter-reference distance and a quasi-unique reference count. More importantly, its approximation algorithm requires surprisingly simple calculation. Consequently, it significantly reduces computational overhead while exhibiting exceptional accuracy and runs extremely fast. Moreover, it allows configurable parameters for performance optimization by making optimal use of workload characteristics. Our extensive experiments with diverse realistic workloads demonstrate our stack distance approximation algorithm outperforms the current state-of-the-art algorithm with higher accuracy (3.7×), lower memory footprint (3.05×) and fast execution (less than 0.05 seconds) on average. In addition, with configuration parameters, our refined algorithm can achieve a 28% average additional accuracy improvement.

INDEX TERMS Stack distance, reuse distance, cache, inter reference distance, MRC, miss ratio curve.

I. INTRODUCTION

A significant performance gap between computer system CPUs and disks has introduced cache mechanisms to improve system performance by pre-storing data objects in the cache [1]. Access to the existing cache data objects is significantly faster than access to the disk data. Thus, effective cache design plays a pivotal role in notable system performance improvement. Designing an effective cache algorithm requires analyzing a representative workload (i.e., data access patterns) in order to characterize data re-access patterns. Here, a stack distance is a good measure of data reuse pattern [2].

A stack distance is literally the depth from which a reference must be extracted from a stack [3]. It corresponds to the number of unique accesses between two accesses to the same address in a trace [3]. Mattson *et al.* [4] originally proposed this stack distance in 1970, where they presented an evaluation of virtual memory page replacement strategies with a stack [1]. Their algorithm captured memory reference traces and constructed a stack by pushing and popping a reference from the stack. In other words, when any reference was accessed, the reference was pushed onto the stack. If any reference in the stack was accessed again, it was popped and moved to the top of the stack. This depth from which a certain reference needs extraction from the stack (due to re-access) is called the stack distance of the corresponding reference [1].

One of representative stack distance applications is a cache behavior simulation, which allows cache miss count estimation for any cache size in a single pass throughout the trace [1]. A miss ratio curve (MRC) can capture this cache miss. An MRC provides fundamental cache performance profiling information by presenting the cache miss rate as a function of the cache size [5]. In typical practice, an MRC exhibits the cache miss ratio or cache miss probability on the y-axis and cache size on the x-axis. A typical MRC tends to decrease as cache size increases (figure 1). To simulate a cache (e.g., LRU: Least Recently Used) behavior, the cache miss ratio with LRU cache size S corresponds to the fraction of references with stack distance larger than S.

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

In addition to this cache performance prediction, MRCs can also estimate the amount of data a workload uses [6]. Consequently, MRCs are useful to design efficient data placement policies in tiered storage systems consisting of different storage devices with different performance characteristics, such as high-performance persistent memories (e.g., Intel Optane persistent memory), Solid State Drives (SSDs) and conventional Hard Disk Drives (HDDs) [17]–[19]. MRCs can be helpful to determine different types of storage device sizes for effective hierarchical storage systems, capable of simultaneously considering both performance (i.e., data access latency) and storage cost [2].

A stack distance can be useful to hot data identification algorithms capturing both recency and frequency information [3]. The stack distance is also a good measure of recency. Recency is a valuable factor for temporally localized access patterns and the stack distance is a measure of this temporal locality. Small stack distances across all references indicates good temporal locality. This implies workloads with a small stack distance should consider recency a more crucial factor when classifying hot data and cold data [3].

Although a stack distance is very useful for various fields including planning and optimization, an exact stack distance calculation requires both high computational complexity ($O(NM)$, where N is the trace length and M is the number of distinct references in the trace) and high space consumption ($O(M)$) [7]. To resolve these limitations, Almasi *et al.* proposed a hole-based algorithm [1]. This is a new stack distance algorithm (not approximation algorithm) adopting a balanced binary tree (i.e., interval tree) to calculate stack distance more efficiently. Although it notably reduced stack distance calculation complexity ($O(Nlog(M))$), its complexity renders it impractical for production systems [7].

Recently, Zhang and Tay proposed PG2S, a stack distance approximation technique that is based on reference popularity and inter-reference distance (they called this gap distance) [2]. Popularity is the number of times an object appears in a trace. Inter-reference distance (IRD) is defined as the number of references between two consecutive references to the same object [2].

PG2S approximation technique reduced computational complexity to $O(M)$. However, as the authors mentioned, their stack distance approximation equation (please refer to equation 3) requires significant complicated mathematical calculations, exhibiting severely prolonged total execution times. More importantly, it requires whole trace analysis information before stack distance approximation, implying it is inappropriate for analyzing practical online systems.

To resolve these critical problems, this paper proposes a novel stack distance approximation algorithm based on workload characteristics. It adopts only one hash table that maintains simple information for each bucket, and performs very simple calculations to approximate stack distance. The proposed mechanism employs two simple workload-related factors for efficiency: inter-reference distance (IRD) and a quasi-unique reference count. IRD is the number of all references between two successive references to the same address in the trace. The quasi-unique reference count emulates the unique reference count between two identical references. Both values are produced by an extremely simple calculation that utilizes maintained hash table information. Consequently, the proposed algorithm dramatically reduces computational overheads.

Moreover, our extensive and comprehensive workload studies with various real workloads found some niches for specific access patterns the proposed algorithm can exploit. Though our proposed algorithm exhibits remarkable performance with a very high accuracy under most workloads, it also provides configuration parameters which can be exploited for further performance optimization. Consequently, it accommodates all workload access patterns more effectively.

The main contributions of this paper are as follows:

- **Efficient stack distance approximation:** For compactness, the proposed stack distance approximation algorithm requires surprisingly simple calculation such as just two value averages (i.e., average of both inter-reference distance and quasi-unique reference count). For efficient management and updating both counts, it employs one hash table. In addition, calculating both counts is also very simple (i.e., two index value subtraction). Though the proposed approximation adopts an surprisingly simple mechanism, it achieves excellent accuracy as well as low computational complexity, resulting in extremely fast total execution time.
  Summaries: The proposed algorithm significantly reduces computational overheads and takes only 0.046 seconds (*vs.* 2,581 seconds PG2S on average) to process 0.3 million Logical Block Address (LBA) requests. (Section IV).
- **Extensive performance evaluation:** To evaluate our proposed algorithm, we adopted 12 well-known real workloads and performed extensive performance evaluation with diverse aspects: accuracy, cache miss ratio, resource usage, execution time, etc. We compared our approximation to exact stack distance as well as the state-of-the-art approximation algorithm, PG2S.[1] In addition, our basic approximation algorithm employing default configurations is also compared to our refined algorithm with workload-optimized configurations to verify its effectiveness.(Section IV).
  Summaries: The proposed algorithm exhibits on average 3.7× higher accuracy and 3.05× less memory consumption than PG2S. Particularly, under some workloads with a large amount of unique LBA access patterns, our algorithm exhibits up to 282× higher accuracy.
- **Refinements with configurations:** It is almost impossible for approximation techniques to accommodate all workload access patterns perfectly. Our approximation

[1]The authors generously shared their PG2S codes with us.

algorithm performs very well under most workloads. However, the proposed algorithm additionally allows configuration parameters for further performance optimization to effectively address all access patterns. Moreover, our extensive and elaborate workload studies provide judicious suggestions (i.e., guideline) for an effective parameter configuration to make the best use of our algorithm. Consequently, the proposed approximation algorithm effectively accommodates various access patterns and further improves accuracy. Please note this configuration is totally optional (not required) for finer optimization for specific workloads.

Summaries: Our refined algorithm with parameter configurations improves its accuracy by an average of 1.28×. (Section III-C).

The remainder of this paper is organized as follows. Section II gives a locality and MRC overview, and presents related and previous research studies. Section III explains our stack distance approximation algorithm design and operations. Section IV provides a variety of experimental results and analyses. Section V discusses diverse challenging problems and suggestions. Section VI concludes this work.

II. BACKGROUND AND RELATED WORK

This section explains localities and miss ratio curves (MRCs), and discusses existing stack distance studies.

A. LOCALITY MEASURES

A stack distance can be calculated from the number of distinct references between two references to the same address in a trace. Thus, it describes when the last time an address was previously accessed. This stack distance has also been referred to as a reuse distance [7]. Given t_0, the stack distance definition, $SD(t) = |z|$, where z is the set of unique references between t_0 and t:

$$z = \{ref(\tau) | t_0 < \tau < t\} \tag{1}$$

This stack distance concept was originally designed for virtual page modeling [1], but has been primarily used for modeling cache behavior since it captures temporal locality information.

Inter-reference distance (IRD) is another locality measure [1]. It is the number of all references between two references to the same address in a trace. IRD looks similar to the stack distance. However, it counts *all* references between two identical address accesses. In contrast, stack distance only counts *distinct* accesses [2], [3], [5], [7].

When at time t, $ref(t) = x$, find t_0 which is the last previous x reference time,

$$t_0 = max\{\tau | 0 \leq \tau < t \wedge ref(\tau) = ref(t)\} \tag{2}$$

The IRD is defined as $IRD(t) = t - t_0$. While the exact stack distance calculation requires high computational complexity, an exact IRD calculation involves very low (i.e., almost negligible) complexity. Therefore, we adopt IRD information to design our approximation algorithm.

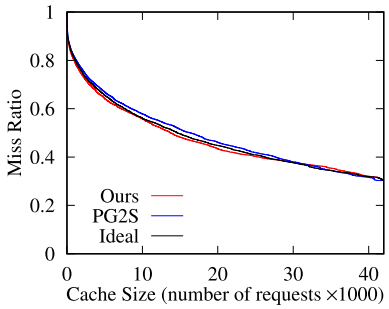


FIGURE 1. A miss ratio curve (MRC) example. An x-axis presents a cache size and y-axis shows a cache miss ratio.

B. MRC: MISS RATIO CURVE

Miss ratio curves (MRCs) are cache utility curves that effectively manage cache allocations [7]. MRCs plot the cache miss ratio or cache miss probability on the y-axis and cache size on the x-axis (figure 1). They provide fundamental information for cache performance profiling by presenting the cache miss rate as a function of the cache size [5]. That is, the higher the miss ratio, the lower the overall performance.

MRCs are very useful to plan or optimize cache utilization in computer systems [26], [27]. However, MRC construction is computationally intensive. Constructing an exact MRC requires stack distance observation over the workload access patterns. Data structures must maintain every access location information in a trace. Thus, MRC construction requires excessive computational complexity as well as space consumption [7]. Various MRC construction methods have been proposed according to different computational complexities and memory requirements [1], [4], [7]–[11].

The Mattson *et al.*'s algorithm [4] scans traces to collect stack distance histogram information. That is, it collects stack distance frequency information for each stack distance while scanning traces. Once the trace analysis completes, the histogram is normalized by dividing each value by the total number of requests. Adding the histogram values up to a given capacity provides the hit ratio. The corresponding miss ratio is obtained by its complement (i.e., $1 - hit\ ratio$). For a trace of length N with M unique references, this algorithm requires $O(NlogM)$ time complexity and $O(M)$ space cost [16], [25], [28]–[30].

To reduce these overheads, various MRC approximation or sampling techniques have been proposed. Berg and Hagersten [12] proposed a sampling algorithm, named StatCache. StatCache sampled every Nth reference for MRC construction to reduce complexity. Similarly, Eklov and Hagersten [16] proposed StatStack, which also adopted a sampling mechanism to reduce complexity overheads and employed both inter-reference distance (IRD) and forward IRD to approximate stack distance.

Zhong and Chang [13] proposed a novel sampling technique, named HRS, that samples the references whose last appearance is in the sampling interval in order to avoid StatCache's limitation: a biased sampling against longer stack distances. Waldspurger *et al.* [7] proposed SHARDS

(Spatially Hashed Approximate Reuse Distance Sampling), a hash-based sampling approach to enforce the property that the object for a sampled reference will be sampled again in the future. However, all sampling approaches cannot avoid their innate limitations: a trade-off between sampling ratio and accuracy.

In addition to these sampling algorithms, Tay and Zou [14] proposed an MRC approximation algorithm, named CME (Cache Miss Equation) which models cache behavior adopting 4 parameters. This CME requires several cache miss measurements for different cache sizes in order to calibrate those parameters. Since stack distance calculation requires high complexity, some MRC construction techniques employed a simpler metric, such as inter-reference distance (also called gap distance) which is defined as the number references between two identical references to the same object. Shen *et al.* [15] proposed RTH2CTH adopting the inter-reference distance (they called this IRD time distance) to approximate the stack distance distribution.

Recently, as a state-of-the-art stack distance approximation technique, Zhang and Tay [2] proposed PG2S (Popularity, Gap distance and Stack distance) which is based on reference popularity and IRD. They proved both popularity and IRD suffice for generating MRC under independent reference model and suggested an expected stack distance equation. Consider a string of independent reference $R =< r_1, r_2, \ldots >$ to M objects, and suppose the IRD $g(i) < \infty$. Further, let $p(x)$ be a reference popularity in R. Then, their expected (i.e., approximated) stack distance is calculated as follows;

$$E\left[s(i)\right] = M - 1 - \sum_{x \neq r_i} \left(1 - \frac{p(x)}{1 - p(r_i)}\right)^{g(i)} \tag{3}$$

PG2S exhibited good performance with respect to stack distance approximation accuracy. In addition, it reduced computational complexity to $O(M)$, where M is the number of unique references in a trace. However, as in the equation 3, it requires significant complicated mathematical calculation and suffers from a severely long execution time. More importantly, it requires whole trace analysis information in advance, implying it is not appropriate for online practical systems.

To overcome PG2S's critical limitations, this paper proposes an efficient stack distance approximation algorithm which achieves higher accuracy with lower computational overheads and lower memory footprint.

III. EFFICIENT STACK DISTANCE APPROXIMATION

This section describes our proposed stack distance approximation design with algorithm description.

A. MAIN DESIGN AND ARCHITECTURE

Figure 2 presents the proposed scheme's overall architecture. For efficiency, the proposed stack distance approximation mechanism adopts a hash table data structure. It employs a simple hash function (i.e., division method) to reduce

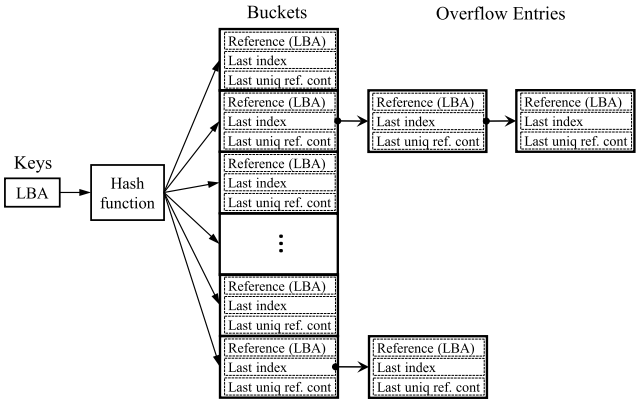


FIGURE 2. The overall architecture. Each hash bucket maintains three metrics: a reference, a last index and a last unique reference count. The proposed scheme adopts a simple hash function (i.e., division method) to reduce a hash computation overhead and a chaining technique to avoid a hash collision.

hash calculation complexity and a chaining mechanism to avoid hash collisions. Each hash bucket stores and updates three simple data values, *(1) a last unique reference count, (2) a reference and (3) a last index.*

As described, the stack distance is defined as the number of unique accesses between two consecutive accesses to the same address. This unique reference count is a key factor for stack distance calculation. Unfortunately, counting the exact number of distinct references between two accesses requires very high computational complexity (i.e., $O(NM)$, where N is the trace length and M is the number of distinct trace references).

To avoid this problem, using a last unique reference count, our proposed mechanism emulates an exact unique reference count between two identical references by storing a *quasi*-unique reference count when the corresponding reference was recently accessed. To update this last unique reference count, the proposed mechanism maintains another simple counter, named the global unique reference count. This global unique reference count increments by 1 whenever a unique trace reference appears by utilizing a reference information in the hash bucket.

The second information the hash bucket maintains is a reference (i.e., accessed LBA (Logical Block Address)). The aforementioned global unique reference counter efficiently utilizes this reference information to check if the reference is unique. In other words, for each trace reference, the proposed algorithm first checks the reference in the bucket. If it finds the identical reference in the hash bucket, meaning the reference has appeared before (i.e., not unique), the global unique reference count does not increase, otherwise the count increases by 1 (i.e., the reference is considered unique). For a quick reference check, a bloom filter can also be hash table alternative.

In addition to the quasi-unique reference count, our proposed algorithm adopts another metric: inter-reference distance (IRD). While the stack distance only considers unique references, IRD counts all intervening references between

two consecutive references. To calculate this IRD efficiently, the proposed mechanism maintains the last index in the hash table storing the index value whenever the corresponding reference is accessed. Aside from this last index, it simply manages a global index. This global index can be considered a time stamp that increments with each reference. Please note both the global index and the global unique reference counter are merely simple counters maintained outside of the hash table.

B. OPERATIONS

The proposed approximation algorithm operates as follows: once it processes a trace request with a reference (i.e., LBA), it first investigates the corresponding hash bucket and acquires the aforementioned three simple values (i.e., a last unique reference count, a last index and a reference). If it does not exist in the hash bucket (that is, the given LBA does not exist in the hash bucket), it simply returns infinity (∞) for the given LBA's stack distance since the accessed LBA has not been previously appeared. Then, it increases a global unique reference count because the given LBA is considered a unique LBA.

Alternately, if the accessed LBA exists in the hash bucket (i.e., the given LBA has been accessed before) the proposed mechanism approximates the stack distance as follows: first, it calculates the inter-reference distance (IRD) at a given time t, $IRD(t)$, by the following equation:

$$IRD(t) = current\ global\ index - last\ index \tag{4}$$

Second, it induces an approximated unique reference count at time t, $UDIFF(t)$, by subtracting the last unique reference count from a current global unique reference count:

$$UDIFF(t) = current\ global\ unique\ reference\ count - last\ unique\ reference\ count \tag{5}$$

Please note that since the accessed LBA is not unique, a global unique reference count does not increment in this case.

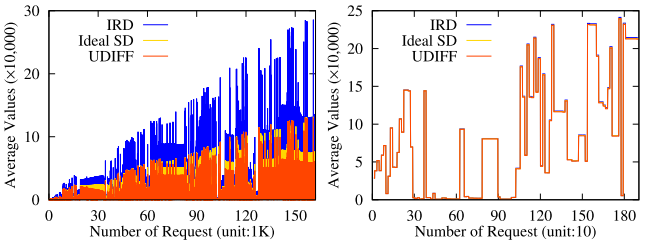
Now it is ready to approximate the stack distance of the given LBA at time t, $SD(t)$, by the following very simple equation:

$$SD(t) = round\left(\frac{IRD(t) + UDIFF(t)}{2}\right) \tag{6}$$

Finally, it updates both the last index and the last unique reference count in the hash bucket with the current global index and the current global unique reference count respectively. This proposed approximation algorithm significantly reduces the computational complexity. More importantly, this approximation exhibits surprisingly high accuracy (Please refer to our experiments in Section IV).

1) JUSTIFICATION

Our simple and accurate approximation equation (Equation 6) considers the following workload characteristics: generally, the difference between an IRD and a stack distance is very close to the difference between a stack distance and a UDIFF. Our comprehensive workload studies found that in most cases, the IRD is always greater than a stack distance. On the other hand, our UDIFF is always smaller than a stack distance (figure 3 (a)). However, under a specific workload pattern (figure 3 (b)), all three metrics exhibit very similar (almost identical) values.



(a) Financial1 (b) WebSearch3

FIGURE 3. Value characteristics among three metrics: an inter-reference distance (IRD), an ideal stack distance (Ideal SD), and an approximated unique reference count (UDIFF). While all three metric values are different under Financial1, they are almost identical under WebSearch3.

Figure 3 presents characteristics of three metrics under two representative workloads (Financial1 and WebSearch3) and justifies the aforementioned analyses. An IRD should be always greater than or equal to a stack distance regardless of workload access patterns because the stack distance only considers unique references. On the contrary, a UDIFF should be always smaller than or equal to a stack distance.

Financial1 has many re-accessed (i.e., not unique) references, implying temporal localities and lower cache miss rates. Many recurrent (re-accessed) references tend to contaminate (i.e., does not correctly increase) our global unique reference count since the proposed scheme adopts a hash table to check reference redundancy. Any re-accessed reference does not increase the global unique reference count. Since a stack distance counts unique accesses between two consecutive (i.e., from present to the very last) identical accesses, the global unique reference count tends not to address many recurrent access patterns effectively. Consequently, the UDIFF exhibits smaller (e.g., workloads with many re-accessed references) than, or equal (e.g., workloads with many unique references) to a stack distance. Please note our proposed algorithm provides a mechanism to resolve this problem. In our experiments, all Financial and Microsoft Research traces belong to Financial1's workload characteristic types. Thus, we omit other plots in the figure 3.

WebSearch3 traces have very different workload characteristics from Financial1. Our studies found WebSearch3 traces have an exceptionally large number of unique (i.e., single accessed) references, which implies significantly low (or almost no) temporal localities and very high cache miss rates. Under this workload type, both an IRD and a UDIFF are very close (nearly identical) to a stack distance (figure 3 (b)). WebSearch and Diskmon traces in our work belong to this access type.

Based on these observations and our extensive workload studies, we concluded that the average of two value (i.e., IRD and UDIFF) effectively approximates stack distance.

2) ALGORITHM

Algorithm 1 describes our proposed approximation algorithm. When any request with LBA is issued, the algorithm first checks if the given LBA has been accessed before. If the LBA has been previously appeared (Line 2), it calculates both IRD and UDIFF values. If the difference between IRD and UDIFF is greater than a predefined threshold value, it calculates a stack distance approximation value only after performing our stack distance correction mechanism (Line 7). Otherwise, it simply calculates our proposed stack distance value (Line 27).

The stack distance correction mechanism consists of a few steps as follows; if the trace suddenly re-accesses an LBA with a very long distance (i.e., accessed a long time ago), it increases an outlier count and resets (i.e., decreases) the IRD value (Line 9–14) because it considers such LBAs outliers. Thus, it does not employ the original IRD value to calculate our stack distance and, instead, reduces the outlier IRD value by half. If it keeps accessing (i.e., predefined times, e.g., 200) the very long distance LBAs, which implies such accesses are normal workload patterns, it does not correct the original IRD value. If the corrected IRD value is still exceptionally large, the predefined value is assigned to IRD (i.e., a smoothing effect) (Line 18–19) because our extensive workload analysis studies concluded this is a rare and exceptional access pattern. Consequently, smoothing out such an exceptional access pattern prevents severely skewing the approximation value. As a last step, if the segment number of the given LBA is greater than that of the previously accessed identical LBA, it increases a unique reference count (Line 23–24). This step corrects our quasi-unique reference count value because it is not an exact unique reference count between two identical references, but an emulated value to significantly reduce computational complexity.

If the given LBA has not been accessed, it initializes IRD, UDIFF, and stack distance (SD) values respectively (Line 29–32). We assign an infinite value (∞) to the stack distance for miss ratio curve construction. Periodically it resets the global unique reference count (Line 34–36) because our unique reference count has a tendency to show a lower value than the exact unique reference count as time goes on.

Algorithm 1 Stack Distance Approximation

```
1:  procedure SdApproximation(A, B, C, D)
2:      if preValue then                          ▷ The LBA has been accessed.
3:          // Calculate IRD and UDIFF values.
4:          IRD ← index − preValue.index
5:          UDIFF ← uniqCnt − preValue.uniqCnt
6:          // Perform our SD correction mechanism.
7:          if (IRD − UDIFF) > tValue then
8:              // If it suddenly accesses a far LBA.
9:              if IRD > avgIRD × A then
10:                 outlierCnt + +
11:                 // Reset (reduce) an IRD value.
12:                 if outlierCnt ≤ B then
13:                     IRD ← IRD/2
14:                 end if    ▷ If it keeps accessing far LBAs, do not reset IRD.
15:             else
16:                 outlierCnt ← 0
17:             end if
18:             if IRD ≥ C then      ▷ If the corrected IRD is still very large.
19:                 IRD ← C
20:             end if
21:         end if
22:         // Unique count correction mechanism
23:         if uniqSegment > preValue.uniqSegment then
24:             uniqCnt + +
25:         end if
26:         // Calculate our stack distance.
27:         SD = round((IRD + UDIFF)/2)
28:     else          ▷ The LBA has not been accessed before.
29:         IRD ← 0
30:         UDIFF ← 0
31:         uniqCnt + +
32:         SD ← ∞                              ▷ Initial stack distance
33:     end if
34:     if (totalReq%D) == 0) then
35:         // Reset the global unique count
36:         uniqSegment + +
37:     end if
38: end procedure
```

C. REFINEMENTS WITH CONFIGURATION PARAMETERS

Our extensive workload studies found some niches for specific access patterns the proposed algorithm can make best use of. The proposed approximation algorithm exhibits excellent performance under most workloads. It provides four configuration parameters which can be utilized to optimize performance. Some workload access patterns can utilize these parameters to improve their performance further. As mentioned, configuring these parameters in our algorithm is not required, but is an optional process.

The proposed algorithm allows the following four configuration parameters presented in Algorithm 1; an IRD threshold for checking outliers (A), an outlier count threshold (B), an IRD for smoothing effect (C), and unique count reset timing (D).

First, the IRD threshold value (A) checks if the current access belongs to typical trace access patterns. In other words, if any request suddenly accesses an LBA that was accessed a very long time ago (i.e., a very long IRD), the algorithm initially considers this access an outlier (i.e., abnormal) access pattern and reduces IRD by half. This workload type has very poor temporal locality and tends to cause a larger stack

distance approximation value than an accurate stack distance because of the unexpectedly large (i.e., contaminated) IRD value. Therefore, if a workload shows a low temporal locality, we suggest adopting a smaller A parameter value than our predefined default value (e.g., 4), which will help improve performance.

Second, if the algorithm considers the given LBA an outlier, as mentioned before, it does not employ the original IRD value since the outlier does not belong to a typical access pattern. Instead, it corrects (i.e., reduces) the calculated IRD value by half until the outlier count threshold value (B). However, if this unexpectedly large IRD appears B times in a row, implying the workload access pattern keeps changing (not temporarily), it subsequently accepts the original IRD without smoothing. Although a smaller B parameter value (default 200) is suggested for workloads with a high temporal locality, this is the least sensitive parameter for workload access patterns based on our observations.

Third, in the case of the aforementioned outlier access pattern, the proposed algorithm corrects the original IRD value. However, if this corrected IRD value is still exceptionally large (i.e., greater than C), it assigns a predefined IRD threshold value (C) to the IRD, instead of accepting the previously corrected IRD. This process smooths out exceptionally skewed effect of the IRD value. Based on our workload studies, a workload with a *very* low temporal locality can benefit from a larger IRD threshold (C) than our default value (i.e., 50,000) for a stronger smoothing effect.

Lastly, unlike the previous three parameters, the last is related to a unique reference count. The proposed approximation mechanism maintains a reference information in a hash table for an efficient unique reference check and increments a global unique reference count by 1 whenever a unique LBA appears in the trace. However, as traces come in, long distant and recurrent LBA accesses have an impact on our quasi-unique reference count value, causing a lower unique reference count than an exact value. Extensive investigation enabled us to determine that this problem occurs under a following specific workload access pattern: when many unique LBAs occur between the current access and the last access to the same LBA, where those unique LBAs were already repeatedly accessed before the last access. To resolve this issue, the proposed algorithm periodically (D) resets a global unique reference count. For this, we suggest a smaller unique count reset timing parameter (D) than the proposed default value (i.e., for each 90,000 request by default) for such a looping access pattern.

The suggested default parameter values of our approximation algorithm were judiciously selected by our extensive and comprehensive workload analysis studies with 12 well-known real workloads. Consequently, the proposed approximation mechanism adopting default parameter values achieves exceptional performance. However, our configuration parameters and advices can help further optimize performance and effectively accommodate all workload patterns.

IV. EXPERIMENTS

This section provides diverse experimental results and comparative analysis.

A. EVALUATION SETUP

To verify our approximation algorithm's effectiveness and efficiency, we compared the proposed mechanism to the state-of-the-art approximation scheme, PG2S which currently exhibits the best performance [2].

For more objective evaluation, 12 real workloads were employed. Financial 1,2 and WebSearch 1,2,3 traces are from the University of Massachusetts–Amherst Storage Repository [20], [21]. Financial1 and Financial2 trace files were collected from online transaction processing (OLTP) applications running at two large financial institutions [20]. Websearch1, Websearch2, and Websearch3 are read-dominant web search engine traces from Storage Performance Council (SPC) [21].

We also adopted Diskmon and Diskmon1 trace files that are solid state drive (SSD) trace files. They consist of one month block I/O traces of a desktop computer in the Center for Research in Intelligent Storage (CRIS) lab at the University of Minnesota–Twin Cities [22]. A well-known block I/O trace application from Microsoft, DiskMon for Windows [23], was installed to the computer and collected personal traces such as computer programming, running simulations, documentation work, web surfing, and watching movies, etc.

Lastly, we utilized various MSR trace files made up of one week block I/O traces of enterprise servers at Microsoft Research Cambridge Lab [24]. We select hm0, prn0, proj0, rsrch0, usr0, web0 traces for our evaluations.

Each trace consists of many read and write requests. These I/O requests are also subdivided into several or more subrequests. We employed this block-level request for our experiments. We conducted all experiments on a server with Intel Xeon 2nd generation Scalable Gold 5218 (16 physical cores, 2.3GHz), 64GB RAM, and Ubuntu 20.04 LTS.

We employed a hash table with 50K entries and a division method for our hash function. In addition, a chaining mechanism is adopted for hash collision avoidance. The proposed algorithm adopts default parameters described in section III-C.

B. PERFORMANCE METRICS

We measure an *average stack distance* to evaluate our proposed stack distance approximation (hereafter, referred to as Ours) performance by comparing it with both an exact average stack distance (referred to as Ideal) and a state-of-the-art scheme, PG2S (referred to as PG2S). This average stack distance is measured with 0.3 million requests for each trace and we plot it for each 1,000 request unit. An initial stack distance where the given LBA is accessed first time is typically assigned by infinity (∞). Thus, we remove

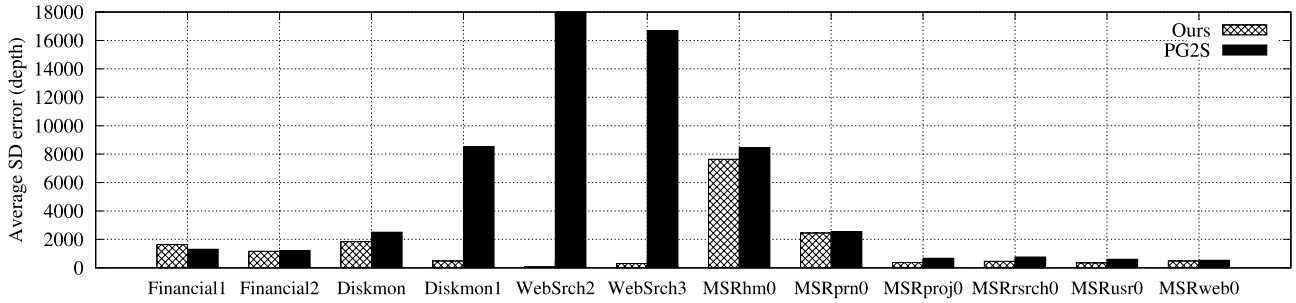


FIGURE 4. Average stack distance error of both the proposed approximation algorithm (Ours) and the state-of-the-art algorithm (PG2S) for each trace (the lower, the better). Here, since the average stack distance error of PG2S shows greater than 18,000, we limit its value for more effective presentation.

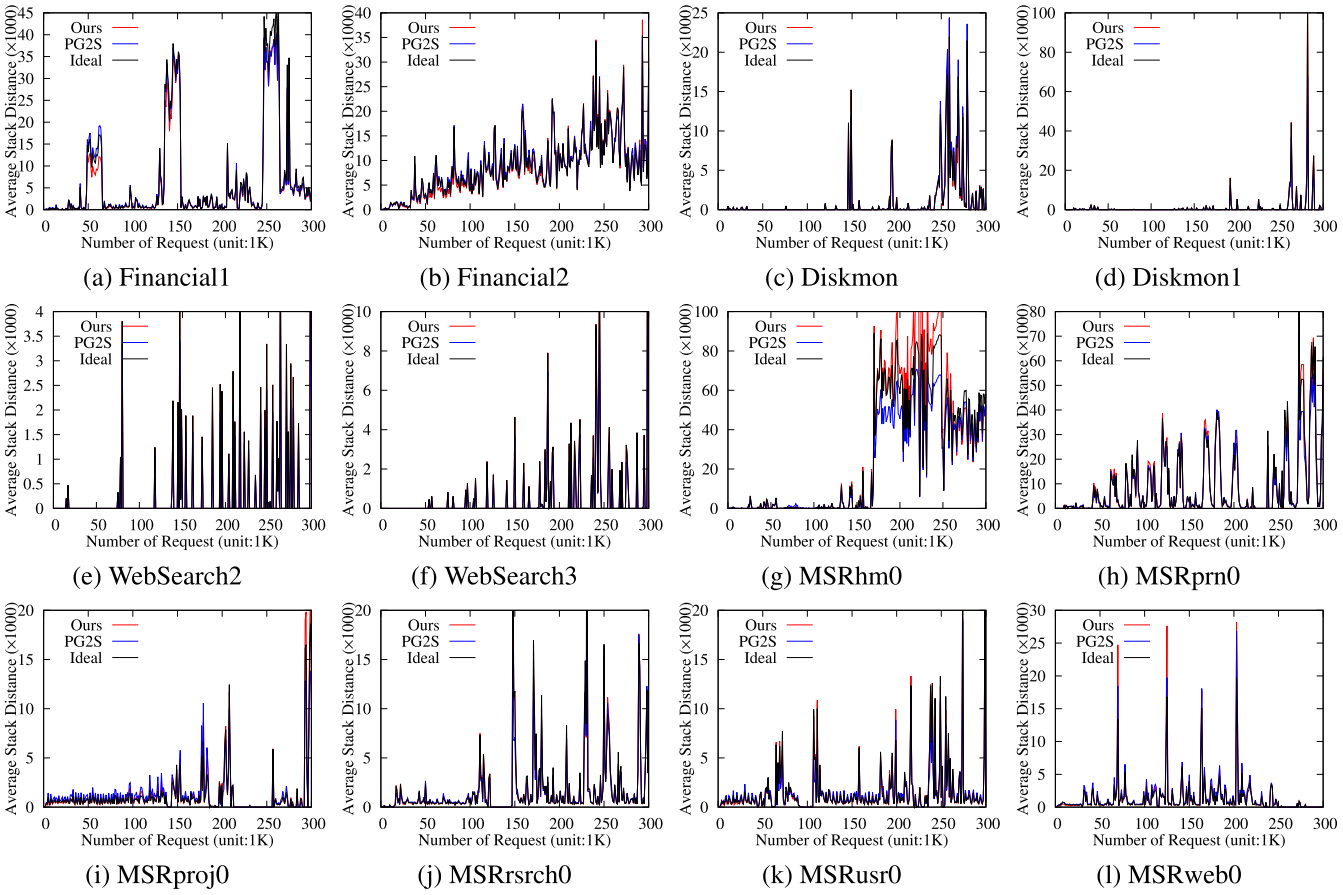
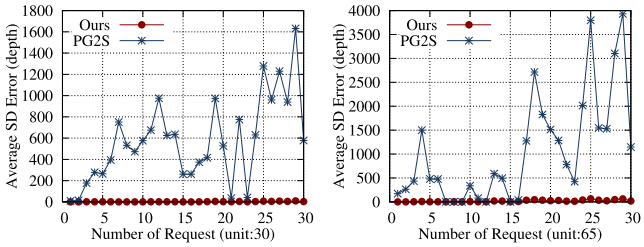


FIGURE 5. Average stack distance over time for each trace: the proposed approximation algorithm (Ours) *vs.* the state-of-the-art algorithm (PG2S) *vs.* the exact stack distance (Ideal). Here, each value corresponds to an average stack distance for each 1,000 request unit.

this initial stack distance from the average stack distance evaluation. Furthermore, we also measure an *average stack distance error* which is calculated by subtracting the approximation values of respective schemes from an exact stack distance value for each request, which provides more thorough comparison.

Cache miss ratio with LRU (Least Recently Used) is also a widely adopted performance metric for miss ratio curves (MRCs). To calculate this cache miss ratio, we first prepare a stack distance histogram with a stack distance on x-axis and frequency (i.e., count) on y-axis. Then, the cache miss ratio with a cache size S corresponds to the fraction of references with a stack distance greater than S. We also measure a *miss ratio error* (i.e., difference between miss ratio of each scheme and that of Ideal) and an *average miss ratio error* respectively.

A *total execution time* (in seconds) is another crucial factor since the computational complexity of a stack distance is severely high. *CPU and memory usage* are also considerable performance metrics to evaluate computing resource usage.



(a) WebSearch2 (b) WebSearch3

FIGURE 6. Average stack distance error comparison over time between the proposed algorithm and PG2S under WebSearch2 and WebSearch3 traces (the lower, the better). A zero on y-axis means an approximated stack distance is identical to an exact stack distance (i.e., 100% accuracy).

C. RESULTS AND ANALYSIS

We discuss our evaluation results from multiple and comprehensive perspectives.

1) AVERAGE STACK DISTANCE

Figure 4 presents an average stack distance error of both our proposed algorithm (referred to as Ours in the figure) and the PG2S for all 12 traces. For the average stack distance error calculation, we subtract the approximation values of both Ours and PG2S algorithms from an exact stack distance value for each request, and average them out. As in figure 4, the proposed algorithm outperforms PG2S by an average of 3.7×. Our algorithm dominates PG2S particularly under Diskmon1, WebSearch2, and WebSearch3 traces by up to 282× (WebSearch2). This results from the innate limitation of a complicated PG2S approximation equation (please refer to the equation 3). Our extensive experiments and investigation found that PG2S algorithm has a severe weak point under the workload characteristics with a significantly large amount of single accessed LBAs. This causes a significantly inaccurate IRD of PG2S unfavorably affecting its stack distance approximation. Diskmon1, WebSearch1, WebSearch2, and WebSearch3 traces belong to this workload type. Consequently, as in figure 4, PG2S exhibits a significantly low accuracy particularly under these workloads. Please note WebSearch1 trace shows a very similar workload patterns to WebSearch 2 and 3. Thus, we omit this WebSearch1 in the figure and our other experiments afterward.

Interestingly, PG2S exhibits 1.24× better accuracy than our proposed algorithm under Financial1. This results from workload characteristics of Financial1 trace and our basic algorithm with default parameters has a tendency not to effectively accommodate such trace. However, our refined algorithm with parameter configuration shows 1.67× better performance than PG2S. This will be discussed in subsection IV-C3 in more detail.

Figure 5 presents average stack distances of three (the proposed, PG2S, and exact stack distance) algorithms in more detail by plotting average stack distance values for each 1,000 request unit. Here, the nearer to the exact stack distance values (Ideal), the higher accuracy. As in the figure, both Ours and PG2S show very close patterns to Ideal, indicating PG2S also achieves a very good approximation accuracy. However, we cannot find significant performance difference between Ours and PG2S in figure 5 (e) with WebSearch2 and (f) with WebSearch3 even if there exist surprising performance gaps (by an average of 99.7×) between them in figure 4.

To investigate this cause, we thoroughly compare stack distance errors between Our and PG2S. Figure 6 presents this result. To calculate stack distance error, we subtract stack distance approximation values of each algorithm from the exact stack distance value. Therefore, a zero implies the algorithm's approximation achieves 100% accuracy. As in the figure, our approximated stack distance values exhibits an excellent accuracy (i.e., all approximated values are very close to zero), while PG2S shows a significantly lower accuracy under both traces. A thorough workload analysis found both (WebSearch 2 and 3) traces as well as Diskmon1 trace tend to have an exceptionally large number of unique (single accessed) LBAs (more than 90%). The stack distance for these uniquely accessed LBAs corresponds to infinity and they were removed for the average stack distance error comparison between Ours and PG2S in figure 6. Moreover, we also found PG2S approximates a significantly high stack distance value especially under these workload patterns with a very large volume of uniquely accessed LBAs. Since the Diskmon1 trace also shows a very similar workload characteristics to both WebSearch 2 and 3 traces, we omit the plot.

2) CACHE MISS RATIO

The cache miss ratio is closely related to a stack distance and figure 7 presents an average error of a miss ratio between the proposed algorithm and PG2S. This error is calculated by subtracting a miss ratio of each approximation algorithm from an exact miss ratio. As in the figure, our proposed algorithm shows a better performance than PG2S by an average of 1.51×. Like figure 4, the proposed algorithm (Ours) outperforms the competitor except Financial1 trace. Similarly to the average stack distance error evaluation, PG2S shows a 1.46× lower error rate than Ours under Financial1 trace. On the other hand, our refined algorithm exhibits 1.85× better performance than PG2S by judiciously configuring parameters, which more effectively accommodates Financial1 trace.

In figure 4, our approximation performance dominates PG2S particularly under Diskmon1, WebSearch 2 and 3 traces. However, such a dominating performance gap is not recognized in figure 7 because miss ratios of these three traces are exceptionally higher (larger than 0.99) than other traces (please refer to figure 8 (d), (e), (f)). Consequently, absolute error values are not noticeably large. To clarify this problem, figure 11 is presented. This figure 11 compares miss ratio errors of both approximation algorithms, and errors are normalized by Ours. As in the figure, Ours achieves a much lower error rate than PG2S by up to 6.66× (WebSearch2).

Figure 8 presents errors between approximate miss ratio curves (MRCs) and exact MRCs. Thus, the closer to exact

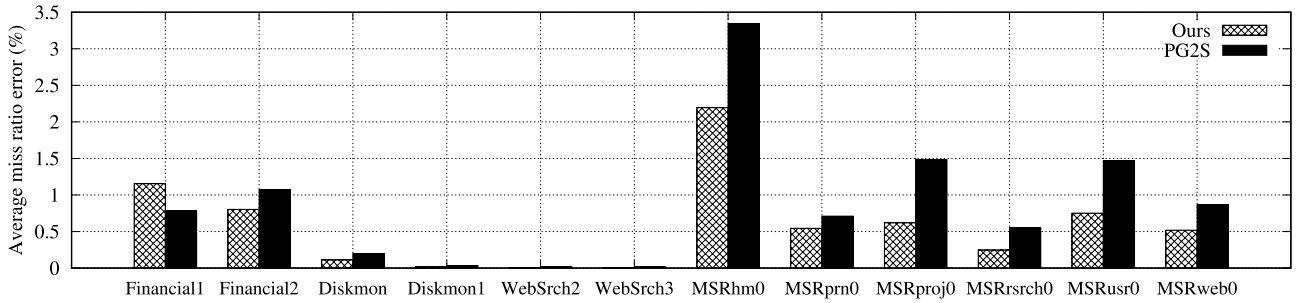


FIGURE 7. Average miss ratio error of both the proposed approximation algorithm (Ours) and the state-of-the-art algorithm (PG2S)for each trace (the lower, the better).

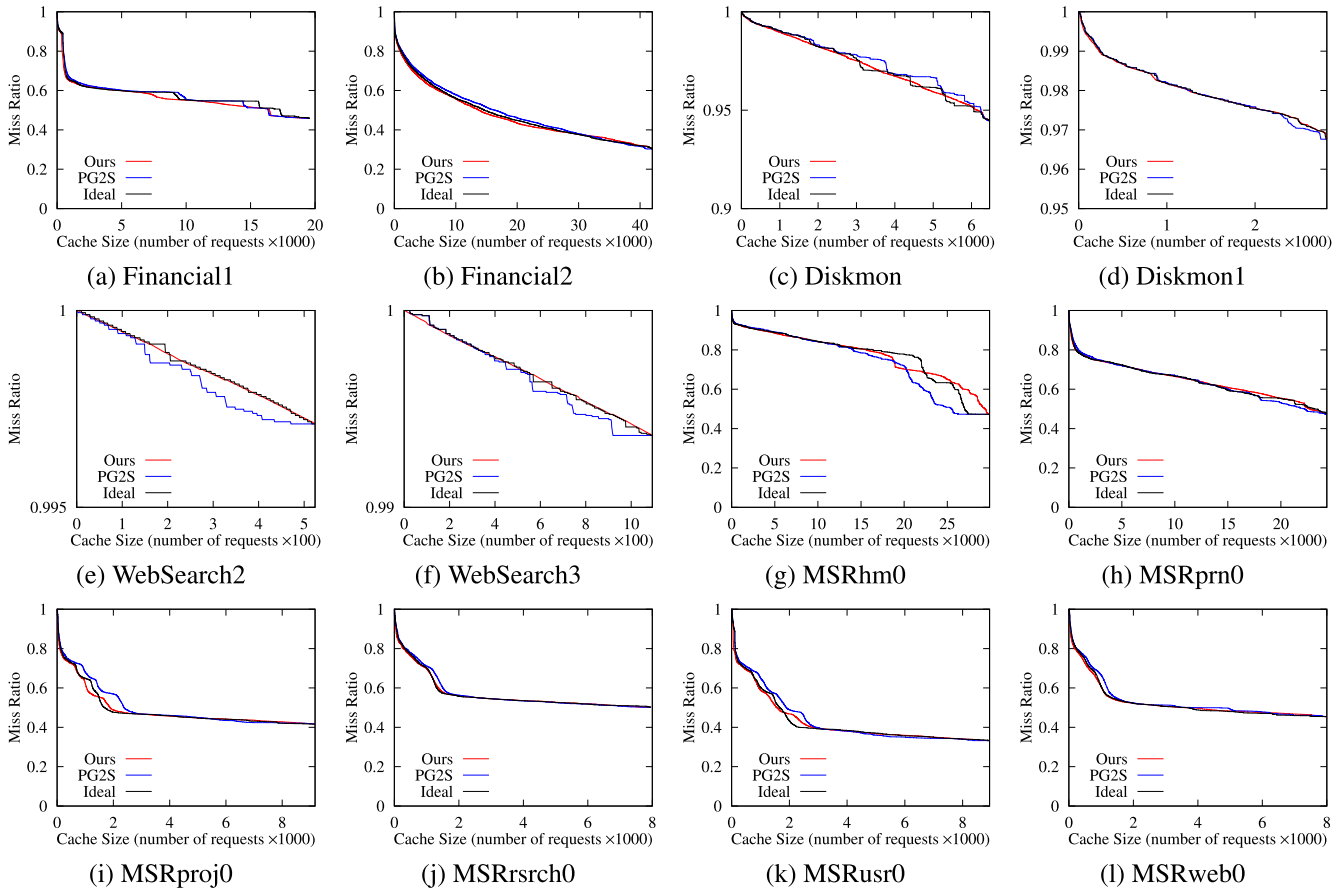


FIGURE 8. Error between approximate (Ours and PG2S) miss ratio curves (MRCs) and exact (Ideal) MRCs (the closer to Ideal, the better). Diskmon, Diskmon1, WebSearch 2 and 3 start with very high miss ratio values (not 0) on y-axis due to their noticeably high cache miss ratios.

MRCs (i.e., Ideal), the better performance. As shown in the figure, our MRCs approach exact MRCs closer than PG2S' MRCs.

3) REFINEMENT

Our proposed mechanism provides four configuration parameters to accommodate all workload patterns more effectively. This subsection evaluates performance improvement with parameter configurations. Figure 9 shows average stack distance errors of both our basic algorithm and a refined algorithm. We configured our algorithm for each trace with the help of our workload analysis studies. As in figure 9, our optimized algorithm improves its performance (i.e., lower error rates) by an average of 1.28×. Especially under Finaicial1 trace, our refined approximation finally exhibits 1.66× better performance than PG2S. Please note PG2S initially showed 1.24× better performance than our basic algorithm under Financial1 trace. For this refinement with Financial1, as shown in figure 10, our workload studies found this Financial1 trace, unlike other traces, has relatively short

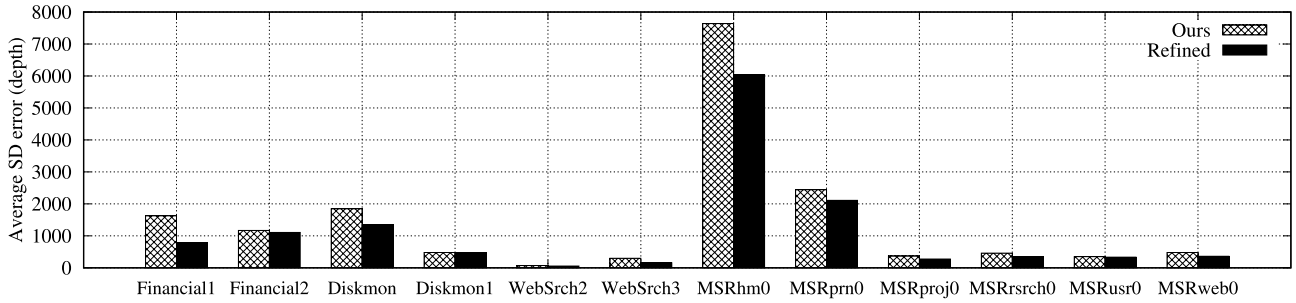


FIGURE 9. Performance improvement with parameter configurations (the lower, the better). Here, Refined stands for our refined approximation algorithm by configuring parameters for each trace. Please note this parameter configuration is an optional performance optimization process. Without this refinement, our proposed approximation algorithm already achieves excellent performance.

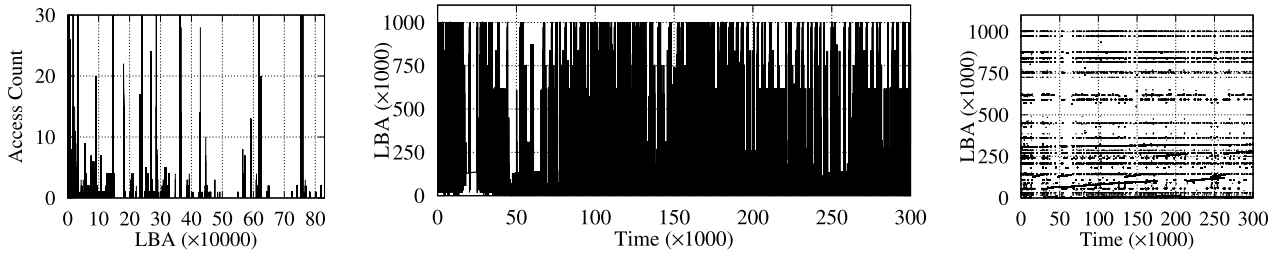


FIGURE 10. Financial1 trace analysis for parameter configurations. Financial1 tends to intensively access a very limited range of LBA (a). It also has a relatively short and recurrent LBA access pattern (b). It shows (not strong) temporal localities (c).

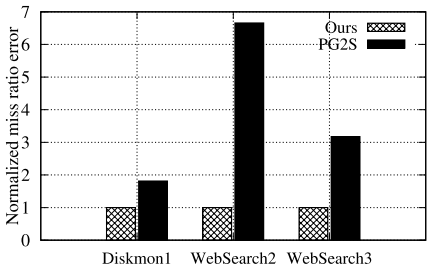


FIGURE 11. Average miss ratio error comparison between the proposed algorithm and PG2S under Diskmon1, WebSearch2 and WebSearch3 traces. Each performance is normalized by Ours.

recurrent LBAs, in which case, a shorter unique count reset timing (parameter D in algorithm 1) than our default value is suggested. Thus, we adopted 5,000 for this D parameter and achieved 1.66× performance improvement. We omit an average miss ratio error plots under all 12 traces since they also show a very similar performance improvement pattern to figure 9.

4) EXECUTION TIME AND RESOURCE USAGE

Figure 12 presents total execution time of our approximation algorithm and PG2S. For this execution time evaluation, unnecessary components or source codes of PG2S were removed (i.e., comment out). That is, only relevant PG2S source codes for stack distance approximation were executed for fair evaluation. As in figure 12, our proposed algorithm displays a surprisingly superior (by an average of 56, 108×) execution time (on average 0.046 seconds (Ours) *vs.* 2,581 seconds (PG2S)) to process 0.3 million requests with a hash table of 50K entries. As a hash table size increases, the average total execution time also decreases further (please refer to figure 14). Consequently, we can hardly find our total execution time bars in the plot. Please note Diskmon's execution time looks very similar to Diskmon1's, Financial2 is similar to Finaicial1, and WebSearch 1 and 2 also exhibit similar execution time patterns to WebSearch3. Thus, we omit them.

Interestingly, while our proposed algorithm consistently shows an excellent execution time regardless of workload types, PG2S' execution time is very sensitive to trace types. Deep PG2S source code analysis enables us to find the main reason: calculating PG2S' stack distance approximation (i.e., equation 3) has high computational complexity. WebSearch and Diskmon traces have an exceptionally large amount of unique (i.e., single access) LBAs, which skips stack distance calculation. Thus, they exhibit a relatively faster execution time. On the other hand, Financial and MSR traces retain many overlapped (i.e., accessed again) LBAs, requiring stack distance calculation. That is to say, the more times PG2S performs stack distance calculation, the longer its total execution time takes.

Memory consumption is also evaluated. Figure 13 presents both algorithms' total memory consumption. PG2S requires on average 3.05× higher memory footprint than the proposed

algorithm (43.3 MiB *vs.* 14.2 MiB). Interestingly, though total memory consumption of both is different, both consumption patterns are nearly identical. That is, while they consume more memory space under both Diskmon1 and WebSearch3 traces, they require less memory space under Financial1 and all MSR traces. This is because, similarly to the aforementioned execution time analysis, both Diskmon1 and WebSearch3 traces have a large number of unique LBAs, which requires more memory space for PG2S to build a larger dictionary map (storing unique key-value pairs) as well as for ours to generate more hash nodes (i.e., a larger hash table or a longer overflow chain). For instance, PG2S consumes 21.6 MiB for building two dictionaries and 38.1 MiB for processing stack distance approximation under Diskmon1. On the other hand, it requires 10.4 MiB for dictionaries and 24.9 MiB for approximation under MSRproj0. Similarly, while our algorithm produces 290,278 hash buckets (including both all hash table entries and chain nodes) under Diskmon1, it produces 124,867 hash buckets under MSRproj0.

We also measured CPU usage, but meaningful difference is negligible noticeable because parallel programming is not easily applied to each algorithm. Thus, each process is assigned to only single CPU core (out of 16 cores in our server), which seemingly consumes high CPU resource (actually, 3.1% average CPU usage with 16 cores). However, our proposed algorithm runs lightning fast so that its resource usage may not be a considerable issue.

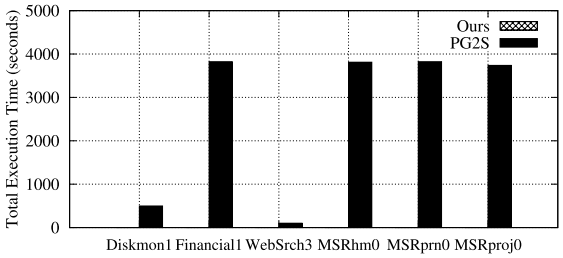


FIGURE 12. Total execution time of both the proposed mechanism (Ours) and PG2S (the lower, the better). We can barely find a total execution time of Ours in this plot because it takes a lot less than 0.1 second (on average 0.046 seconds). Please note Diskmon, WebSearch 1 and 2 show very similar execution time patterns to Diskmon1 and WebSearch3 respectively. So, we remove them.

V. DISCUSSION

The significant performance improvement of the proposed algorithm mainly results from our surprisingly simple stack distance approximation equation, significantly lowering mathematical computation overheads, compared to PG2S. Our hash table data structure also makes a contribution by efficiently checking if the given LBA is unique, which is judiciously utilized for our unique reference count approximation (i.e., quasi-unique reference count).

This section discusses hash table configurations and their impact on overall performance in more detail.

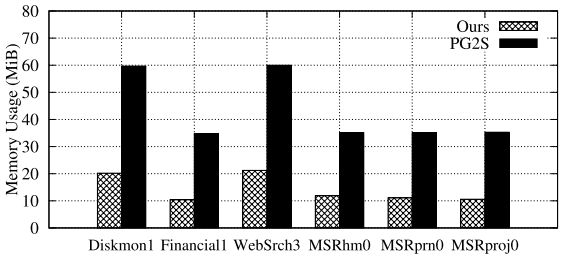
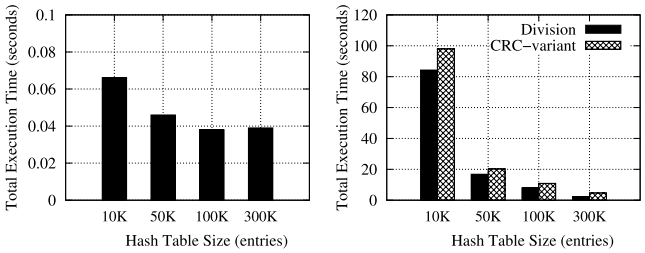


FIGURE 13. Total memory consumption of both the proposed mechanism (Ours) and PG2S (the lower, the better). Workloads retaining many unique LBAs such as Diskmon1 and WebSearch3 consume more memory space for both algorithms. Please note Diskmon, WebSearch 1 and 2 show very similar execution time patterns to Diskmon1 and WebSearch3 respectively. So, we remove them.

A. HASH TABLE SIZE

In our experiments, we employed a hash table of 50K entries because it exhibited well-balanced performance between total execution time and memory consumption. Figure 14 presents an average total execution time for various hash table size configurations under 12 traces. For a more objective comparison, heavy workloads (9 million requests) are employed. A hash table size varies from 10K to 300K and we average all 12 traces' total execution time under both 0.3 million requests and 9 million requests respectively. As in the figure, average total execution time intuitively decreases as a hash table size increases. Particularly under 9 million workloads, the total execution time noticeably decreases as a hash table size increases. This mainly results from linear search overhead reduction of hash overflow chains. The proposed scheme adopted a chaining mechanism for hash collision avoidance. This chain implements a linked list data structure. Thus, whenever a hash collision occurs, the proposed algorithm references the chain to check if an identical reference exists in the chain, requiring a linear search that generates overhead. The longer chain, the higher overhead. Under typical 0.3 million workloads, a significant performance improvement was not found as a hash table size increased because each hash bucket's chain length is short (e.g., less than 6 on average). On the other hand, a performance gain is significant under heavy workloads as in the figure 14 (b) because larger hash tables can noticeably reduce overflow chain lengths, reducing linear search overhead. For example, while an average overflow chain length of a 10K entry-hash table is 353.6, a 50K entry-hash table has on average 70.7 chain length. However, total memory consumption of both hash tables is nearly identical especially under heavy workloads, because no empty (i.e., wasted memory space) hash table buckets exist. Therefore, we suggest a larger hash table under heavy workloads. To further reduce linear search overhead, a bloom filter can efficiently check if a given LBA exists in the chain list at the cost of memory space. A bloom filter can reduce an unnecessary lookup overhead of the overflow chain lists by simply prepending a new node into the chain head without a list search overhead if a unique LBA caused a hash collision.



(a) 0.3 million workloads (b) 9 million workloads

FIGURE 14. Average total execution time for various hash table size configurations (the lower, the better). Heavy workloads (9 million) for all 12 traces is adopted for more objective comparison. A larger hash table exhibits a noticeable performance gain particularly under heavy workloads. Figure (b) also shows a total execution time of our algorithm adopting both a simple (Division) and a more complicated (CRC-variant) hash functions. The CRC-variant takes a longer time due to its higher computational overhead.

B. HASH FUNCTION

A hash function is another important hash table performance factor because it controls hash key distribution over the hash buckets. The proposed scheme adopted a simple hash function such as a division method (i.e., $h(k) = k \ \% \ M$, where M is a hash table size) to reduce hash calculation overhead. More complicated hash functions, including Knuth's Division variant, Multiplication methods, CRC variants, PJW hashes, etc. potentially provide more effective key distributions [31]. However, though these sophisticated hash functions may more effectively (i.e., more evenly) distribute hash key values over the hash table, they involve relatively higher calculation complexity. To verify these concerns, extensive experiments were performed.

Figure 15 presents hash key distribution of both a simple (i.e., Division method) hash function and a more complex (CRC-variant) hash function under MSRweb0 traces. As in the figure, even the Division hash function exhibits a good (actually, better) performance (i.e., very even key distribution). We also performed this experiment under all 12 traces and the Division hash function showed a very good performance (we omit them). Surprisingly, a CRC-variant hash function exhibited a wider distribution variation for each hash bucket.

As for hash calculation overhead, we measured a total execution time of our proposed algorithm by adopting both hash functions. Especially heavy workloads were employed in this evaluation because our algorithm exhibits extremely fast execution time (less than 0.05 seconds) under 0.3 million workloads. As in figure 14 (b), our algorithm adopting the CRC-variant hash function took on average 1.16× longer time than the one using Division hash function, implying CRC-variant causes 16% higher computational overheads than the simple Division hash function. Based on our extensive experiments, a simple Division hash function exhibited a competitive key distribution performance. Moreover, our proposed algorithm can benefit from its simplicity. Therefore, our approximation design employed the simple Division hash function.

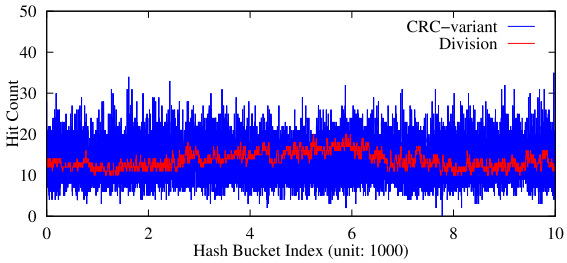


FIGURE 15. A hash key distribution of both simple (Division) and complicated (CRC-variant) hash functions. A Division hash function exhibits more even distribution than CRC-variant hash function.

VI. CONCLUSION

A stack distance is defined as the number of unique accesses between two identical accesses in a trace. Although this stack distance concept was originally suggested for virtual page modeling [1], it has been applied to a variety of applications utilizing temporal locality information. The stack distance applications include modeling cache behavior with miss ratio curves (MRCs) [6], data placement policies in tiered storage systems [2], and hot data classification algorithm design [3]. The stack distance is very useful for various planning and optimization fields. However, an exact stack distance calculation requires both high computational complexity and high space consumption.

To address these limitations, a new stack distance algorithm employing an interval tree has been proposed. Though it noticeably reduced stack distance calculation complexity to $O(NlogM)$, it is not useful for practical online use in production systems [7]. Recently, Zhang and Tay [2] proposed a stack distance approximation algorithm, named PG2S, that is based on reference popularity and inter-reference distance (IRD) information. As a result, it reduced computational complexity to $O(M)$ and achieved a high accuracy. However, because PG2S' stack distance approximation equation is so complicated, it results in an excessive total execution time. Moreover, it requires whole trace analysis information before running the approximation algorithm. These critical limitations prevent PG2S from being adopted in practical online production systems.

This paper proposed a more efficient stack distance approximation algorithm. The proposed algorithm utilizes simple two metrics such as inter-reference distance and a quasi-unique reference count. Further, our stack distance approximation equation is surprisingly simple (i.e., the two value average). Thus, our algorithm significantly reduces computational complexity. Moreover, the proposed algorithm allows configuration parameters for further performance optimization to make the best use of workload characteristics. We also provide some guidelines for this parameter configuration based on our extensive workload studies. The proposed algorithm also does not require any pre-processing or prerequisite information. Consequently, our algorithm is extremely fast (less than 0.05 second), consumes on average 3.05× less memory, and achieves on average 3.7× higher accuracy than PG2S.

Future work investigates applications for our efficient approximation technique. These include designing efficient tiered storage systems with different storage characteristics as well as efficient memory pooling designs utilizing next-generation, cache-coherent interconnects for processors, memory expansion, and accelerators such as Compute Express Link (CXL) [32].

ACKNOWLEDGMENT

The authors would like to thank David Schwaderer (ShapeShift Ciphers LLC, USA) for his valuable comments and proofreading. This work would not have been possible without initial research setup and priceless comments of both Daeun Shim (Samsung Electronics, South Korea) and Hyeonji Ha (Sookmyung Women's University). They are also grateful to Dr. Jiangwei Zhang and Dr. Yongchiang (Y. C.) Tay (PG2S+ paper authors, National University of Singapore) for their sharing PG2S source codes.

REFERENCES

[1] G. Almási, C. Caşcaval, and D. A. Padua, "Calculating stack distances efficiently," in *Proc. Workshop Memory Syst. Perform. (MSP)*, 2002, pp. 37–43.
[2] J. Zhang and Y. C. Tay, "PG2S+: Stack distance construction using popularity, gap and machine learning," in *Proc. Web Conf.*, Apr. 2020, pp. 973–983.
[3] H. Ha, D. Shim, H. Lee, and D. Park, "Dynamic hot data identification using a stack distance approximation," *IEEE Access*, vol. 9, pp. 79889–79903, 2021.
[4] R. L. Mattson, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.
[5] D. Carra and G. Neglia, "Efficient miss ratio curve computation for heterogeneous content popularity," in *Proc. USENIX ATC*, Jul. 2020, pp. 741–751.
[6] D. Byrne, "A survey of miss-ratio curve construction techniques," 2018, *arXiv:1804.01972*.
[7] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC construction with SHARDS," in *Proc. USENIX FAST*, Feb. 2015, pp. 95–110.
[8] B. T. Bennett and V. J. Kruskal, "LRU stack processing," *IBM J. Res. Develop.*, vol. 19, no. 4, pp. 353–357, Jul. 1975.
[9] Y. H. Kim, M. D. Hill, and D. A. Wood, "Implementing stack simulation for highly-associative memories," in *Proc. ACM SIGMETRICS Conf. Meas. Modeling Comput. Syst. (SIGMETRICS)*, 1991, pp. 212–213.
[10] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "PARDA: A fast parallel reuse distance analysis algorithm," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 1284–1294.
[11] D. Byrne, N. Onder, and Z. Wang, "MPart: Miss-ratio curve guided partitioning in key-value stores," in *Proc. ACM SIGPLAN Int. Symp. Memory Manage.*, Jun. 2018, pp. 84–95.
[12] E. Berg and E. Hagersten, "StatCache: A probabilistic approach to efficient and accurate data locality analysis," in *Proc. IEEE Int. Symp. ISPASS Perform. Anal. Syst. Softw.*, Mar. 2004, pp. 20–27.
[13] Y. Zhong and W. Chang, "Sampling-based program locality approximation," in *Proc. 7th Int. Symp. Memory Manage. (ISMM)*, 2008, pp. 91–100.
[14] Y. C. Tay and M. Zou, "A page fault equation for modeling the effect of memory size," *Perform. Eval.*, vol. 63, no. 2, pp. 99–130, Feb. 2006.
[15] X. Shen, J. Shaw, B. Meeker, and C. Ding, "Locality approximation using time," *ACM SIGPLAN Notices*, vol. 42, no. 1, pp. 55–61, Jan. 2007.
[16] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2010, pp. 55–65.
[17] H. Oh, "The performance analysis of big data processing platforms using next-generation large-capacity persistent memory," M.S. thesis, Comput. Sci., Sookmyung Women's Univ., Seoul, South Korea, Feb. 2022.
[18] Z. Fan and D. Park, "Extending SSD lifespan with comprehensive non-volatile memory-based write buffers," *J. Comput. Sci. Technol.*, vol. 34, no. 1, pp. 113–132, Jan. 2019.
[19] E. Lee, H. Oh, and D. Park, "Big data processing on single board computer clusters: Exploring challenges and possibilities," *IEEE Access*, vol. 9, pp. 142551–142565, 2021.
[20] UMass. (2022). *OLTP Application I/O*. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage
[21] UMass. (2022). *Search Engine I/O*. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage
[22] D. Park and D. H. C. Du, "Hot data identification for flash-based storage systems using multiple Bloom filters," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol. (MSST)*, May 2011, pp. 1–11.
[23] M. Russinovich. (2006). *DiskMon for Windows v2.01*. [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/downloads/diskmon
[24] D. Narayana, A. Donnelly, and A. Rowstron. (2007). *MSR Cambridge Traces (SNIA IOTTA) Trace Set)*. [Online]. Available: http://iotta.snia.org/traces/list/BlockIO
[25] C. CaBcaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proc. 17th Annu. Int. Conf. Supercomput. (ICS)*, 2003, pp. 150–159.
[26] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley, "Performance evaluation of hierarchical TTL-based cache networks," *Comput. Netw.*, vol. 65, pp. 212–231, Jun. 2014.
[27] V. Martina, M. Garetto, and E. Leonardi, "A unified approach to the performance analysis of caching systems," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2014, pp. 1–28.
[28] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *Proc. USENIX ATC*, 2016, pp. 351–364.
[29] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2014, pp. 1–14.
[30] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *Proc. USENIX ATC*, 2017, pp. 487–498.
[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2001, pp. 221–252.
[32] D. D. Sharma and S. Tavallaei. (2022). *Compute Express Link 2.0 White Paper*. [Online]. Available: https://www.computeexpresslink.org/

SOOYOUNG LIM received the bachelor's degree from the Department of Software, Sookmyung Women's University, Seoul, South Korea, in 2022, where she is currently pursuing the master's degree in computer science.

She is a member of the Big Data Storage Systems Laboratory under the advice of Prof. Dongchul Park. She is currently studying on an efficient stack distance approximation algorithm design. Her research interests include big data processing software platforms, next generation non-volatile memories (NVMs), such as Intel Optane persistent memory, heterogeneous computing memory sharing technologies and applications including Compute Express Link (CXL), and storage systems including NVM-based SSDs.

DONGCHUL PARK (Member, IEEE) received the Ph.D. degree in computer science and engineering from the University of Minnesota–Twin Cities, Minneapolis, USA, in 2012.

He was a member of the Center for Research in Intelligent Storage (CRIS) Group under the advice of Prof. David H. C. Du. From 2012 to 2016, he was a Senior Research Engineer with the Memory Solutions Laboratory (MSL), Samsung Semiconductor Inc., San Jose, CA, USA, and a Senior Staff Research Engineer with the Storage Technology Group (STG), Intel, Hillsboro, OR, USA, in 2017. From 2017 to 2019, he was an Assistant Professor in computer science and engineering at the Hankuk University of Foreign Studies (HUFS), Yongin, South Korea. Since 2019, he has been an Assistant Professor with the Department of Software, Sookmyung Women's University, Seoul, South Korea. His research interests include system design and applications including non-volatile memories (NVM), in-storage computing, data deduplication, key-value store, and shingled magnetic recording (SMR) technology. He is also interested in big data processing, Hadoop MapReduce, cloud computing, and next generation NVM, such as Intel Optane Memory.