Article

AOIO: Application Oriented I/O Optimization for Buffer Management

Xiaochang Li ^{1,*}, Zhengjun Zhai ¹ and Xin Ye ²
¹ School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an 710072, China; zhaizjun@nwpu.edu.cn

² School of Computer Science and Engineering, Xi'an Technological University, Xi'an 710021, China; yexin@xatu.edu.cn

* Correspondence: xcli@mail.nwpu.edu.cn

Abstract: Emerging scale-out I/O intensive applications are broadly used now, which process a large amount of data in buffer/cache for reorganization or analysis and their performances are greatly affected by the speed of the I/O system. Efficient management scheme of the limited kernel buffer plays a key role in improving I/O system performance, such as caching hinted data for reuse in future, prefetching hinted data, and expelling data not to be accessed again from a buffer, which are called proactive mechanisms in buffer management. However, most of the existing buffer management schemes cannot identify data reference regularities (i.e., sequential or looping patterns) that can benefit proactive mechanisms, and they also cannot perform in the application level for managing specified applications. In this paper, we present an Application Oriented I/O Optimization (AOIO) technique automatically benefiting the kernel buffer/cache by exploring the I/O regularities of applications based on program counter technique. In our design, the input/output data and the looping pattern are in strict symmetry. According to AOIO, each application can provide more appropriate predictions to operating system which achieve significantly better accuracy than other buffer management schemes. The trace-driven simulation experiment results show that the hit ratios are improved by an average of 25.9% and the execution times are reduced by as much as 20.2% compared to other schemes for the workloads we used.

Keywords: I/O management; page cache; program counters; operation system

Citation: Li, X.; Zhai, Z.; Ye, X. AOIO: Application Oriented I/O Optimization for Buffer Management. *Symmetry* **2021**, *13*, 573. <https://doi.org/10.3390/sym13040573>

Academic Editors: Jan Awrejcewicz and Sergei D. Odintsov

Received: 11 February 2021

Accepted: 23 March 2021

Published: 31 March 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The scale of I/O intensive workloads generated and shared by enterprises, scientific research, and databases has increased immeasurably. For example, the *GraphChi* application structures a large amount of data in the buffer for calculation. *GraphChi* is an application for large-scale graph computation on a computer. It is used to replace distributed computing cluster handling very large real-world problems, such as analysis on social networks or the web graph. The popular big data training applications analyze data in kernel buffer as large as they can narrow the speed gap between CPU and storage devices. Some of the massive data even need to be analyzed and completed in a time-sensitive manner [1,2]. However, under scale-out workloads, the modern computer buffer hardware faces physical constraints in capacity. Then the management for size-limited buffer plays a critical role in systems, which provides a way to store data in storage devices (e.g., HDD, SSD) by bringing them into kernel buffer when they are needed. The reason is that the speed of DRAM is about 5–10 times that of the storage device [3]. Generally speaking, only the frequently accessed data are suitable for caching, so as to improve storage efficiency. Furthermore, treat-ahead support in the buffer also prompts the performance of the page cache. Given a series of continuous data, the experiments of 8 KB and 32 KB prefetching units could outperform that of the 4 KB prefetching unit by almost 60% and 71%, respectively. Therefore, how to identify accurate data patterns and exploit the prefetching length are the key concerns in buffer management schemes, called proactive abilities.

It is lucky that some I/O intensive applications offer clear regularities about their future I/O requirements [4]. However, observing traditional Least Recently Used (LRU) like strategies [5–7], they do not support the length parameter of prefetching, and they have no way to exploit regularities in data access, such as looping or sequential patterns. The refined data classification in cache can be regarded as working in different zones [8], and each data pattern has its corresponding optimal management policy. Furthermore, LRU-based schemes cannot discover these hints from application level as well, because there is no way to classify which application the buffer data come from. In conclusion, traditional kernel buffer management schemes are reactive, or only have coarse and general kernel fetching support from the operating system, which makes the proactive I/O management desirable.

How to exhibit or record certain degrees of repetitive behavior in detail in application level is the main concern of the proactive buffer management. The program instructions (calculated or represented by their program counters) have been proven that they provide a highly effective method of recording the context of program repetitive behavior [9] which has been extensively used in modern computer architectures. The success of using PC information was first proposed in branch prediction area [10], and then PC has been widely used in other study areas. For instance, in energy optimization studies [11,12], program instructions are used to predict the regularities of processor pipeline to achieve power reduction of the hardware. PCs also can predict which data will not be used by processor again and free up the memory caches or TLBs for storing or prefetching more relevant data [13,14]. Except architecture designs or the energy efficiency optimizations of the processors, in operation systems, there are also many history-based prediction techniques are based program instructions, such as page cache optimizations in file systems [9,15], or storage device designs [16,17].

Despite that the great success of PC techniques has been explored, the PC technique has not been used for page cache management beneficial to specified applications. In this paper, we explore the feasibility of PC-based technique by making statistics analysis between logical block numbers and files originated from different applications, and propose the I/O optimization technique oriented towards application level, called AOIO. AOIO identifies the data access pattern among the file pages accessed by I/O operations triggered by call instructions in the application. According to the pattern classifications, a suitable replacement or a prefetching prediction can be used to manage the file pages belonging to each pattern. In addition, if the same PC is observed again for incoming file pages, AOIO can immediately appoint its related pattern based on history prediction records. In the page cache layer, all the data patterns are used by the kernel system call function, `fcntl()` interface [18]. The `fcntl()` interface helps to deploy AOIO in operating systems in a simple way, and tells the system how it expects to use a file handle, so that the system can select the appropriate read-ahead and caching techniques to access the corresponding file. Compared to the *posix_fadvise* interface in user space used by the developer manually, AOIO provides an automatic and intelligent I/O management.

As this article classifies the data patterns for I/O optimization, we can observe that the input data and output directions are in symmetry. Furthermore, among our patterns, there is a ‘loop’ pattern. All the data in the ‘loop’ pattern are repeated in the ‘period’ distance. The repeated data in the ‘loop’ data are in symmetry.

In summary, we make the following contributions:

- New finer-grained I/O patterns are proposed in our article, and therefore, the kernel can deal with different kinds of data patterns by corresponding optimal policies.
- Basing on the hints from patterns, we design and implement a new buffer manager, AOIO, co-existing with Linux buffer manager.
- When applications are managed by AOIO, the managements for target applications are independent of the original buffer management scheme in operating systems, which can insulate and significantly improve the performance of selected applications.

- The experimental results prove that AOIO provides a more accurate prediction of I/O patterns than traditional buffer schemes, and AOIO can outperform state-of-the-art schemes.

The rest of this paper is organized as follows: Section 2 describes the background. Section 3 discusses the motivation. Section 4 presents the design and implementation of AOIO. The experimental results are shown in Section 5. Finally, we conclude the article in Section 6.

2. Background

In this section, we classify the existing buffer data management schemes into three categories and in turn, discuss them in each category.

- Frequency or recency-based buffer management schemes, called LRU-based schemes.
- User-level buffer management schemes based on application hints.
- Program counters based buffer management by making blocks statistics in file systems.

For the first category, LRU-based schemes with data frequency or recency factors are still widely used in kernel buffer management due to their simplicity. The frequency-based replacement (FBR) [19] scheme selects victim blocks to be replaced based on the frequency factor. The LRU-K scheme [20] uses blocks' Kth-to-last references to make replacement decisions. The LRFU scheme [21] makes use of both the frequency and recency factors. Although they are simple, a main drawback of the LRU like policies, however, is they cannot exploit detailed data access patterns [4].

In the second category, application-controlled cache management designs [4,22,23] are schemes based on user-level hints. These schemes need the hints provided by developers manually according to application reference characteristics, which allow different applications to use different buffer management policies. The challenge is developers should be familiar with the inner context of applications and accurately understand the characteristics of block regularities of applications.

The third category of schemes aim to explore regularities of data references. As mentioned earlier, the PC techniques have been proved to be effective. Two existing buffer managements with PC techniques, PCC [9] and AMP [15] will be discussed and compared to our AOIO in this article. Both PCC and AMP are specially designed for the operating system in the buffer and file systems layer to identify various block access patterns such as sequential or looping references. According to different data patterns, operating systems use appropriate replacement schemes based on detected patterns. However, their designs are complex and their statistics of blocks are unstable which will be described in Section 3.

The aforementioned techniques utilize different strategies. However, according to our detailed analysis, LRU-based policies usually fail to capture detailed I/O behaviors in applications level, therefore causing many cache misses. In the second category, application hints provided by developers from user space are not easy to be identified manually. Furthermore, only file page statistics are not enough to express accurate data regularities. For the third category, the existing PC-based buffer management designs perform unsteadily and their data patterns are not detailed enough to separate data. In the following section, we analyze limitations of buffer management schemes from three categories above.

3. Motivation

The key concern of buffer management is to design an effective block replacement scheme. According to three categories classified in Section 2, we discuss and illustrate their limitations.

LRU-based schemes. LRU-based schemes are widely used replacement schemes. It is based on the assumption that recently accessed blocks are likely to be accessed again in the near future. Figure 1 exhibits a buffer example. Given that the capacity of the cache in Figure 1 is 3 requests, then the 'Loop' request (request1) should be expelled by LRU policy when the sequential ('Seq') request (request4) comes. However, this is not a reasonable

strategy, because we know the 'Loop' request1 will be accessed later and 'Loop' request1 has priority over 'Rand' and 'Seq' requests to stay in the cache. Therefore, it is necessary to identify all I/O reference patterns and design optimum cache management strategies for different patterns. Additionally, the 'Seq' and 'Loop' data patterns provide prefetching ability for AOIO. Therefore, we define that the buffer management optimization in this article is to accurately discover patterns and to design appropriate buffer management strategies for them. These patterns should be indicated by exact parameters which promises our optimization design is repeatable and reliable [24,25].

Figure 1. An example of the buffer management.

Application hint-based schemes. As aforementioned, the I/O pattern recognition plays a key role in buffer management. There exist two kinds of identification techniques based on the files. The first is to classify the files in user space manually, AFC, two-level cache [22,23]. The second is to make file statistics to classify the I/O access patterns, UBM [4]. However, they all have instinctive drawbacks shown below.

For the first identification technique, file hints made in user space, the management granularity is each file as it is hard to separate individual file in a fine-grained manner, such as a file page unit. In our test, a single file has several data patterns. Figures 2 and 3 show the address-time graph of I/O workloads collected from two applications, cscope and glimpse, respectively. In the *Cscope* and *Glimpse* workloads, we enclose those I/Os from the same program counter. In Figure 2, data enclosed by the red and blue ovals belong to the same LBN scope from a single file. That is to say, data in a single file could be indexed by various PCs (various PCs represent different data patterns). In Figure 3, there is the same situation that a single file has at least two kinds of patterns enclosed by red and blue ovals. Therefore, the manual file hints in user space are not reasonable.

For the second technique, it needs to classify the access patterns for each file, resulting in a delay in the classification process and a missed opportunity to apply the most appropriate replacement scheme for each pattern.

Figure 2. I/O workload from cscope.

Figure 3. I/O workload from glimpse.

PC based schemes. The existing PC-based schemes (AMP and PCC) are based on logical block numbers (LBNs) from the file system layer. However, data patterns relying on LBNs are vulnerable to the file modification. All the LBNs are controlled by file systems which will frequently change the logical block numbers when applications issue I/O operations. In many cases, file modification actions distort the LBNs position in file systems, resulting in the former data patterns altering. The data patterns will not be changed from the perspective of file pages. The second drawback is that the LBN statistics cannot be used for an appointed application. Conventional PC-based schemes only focus on the entire buffer management, instead of the application level. At last, AOIO proposes a new kind of data pattern, ‘short-consecutive loops’ pattern to describe a group of hot data in a short consecutive length. In addition, AOIO can be easily deployed in the module state. Note that, the ‘short-consecutive loops’ pattern is in strict symmetry.

4. Design of AOIO

In Figure 4, we describe the detailed design of AOIO which is comprised of two steps and three main modules: *Pattern Calculator*, *Pattern Classifier*, and *Buffer Commander*. The first step is that *Pattern Calculator* and *Pattern Classifier* modules work out a *PC Manager Table* storing the regularities of data patterns, which is shown in Figure 5.

The second step is that *Buffer Commander* makes use of the recorded regularities in *PC Manager Table* to guidance buffer management.

Figure 4. Overall architecture of AOIO.

PC value	Pattern	Len	Period
PC0	Sequential	9	∞
PC1	Loop	10	90
PC2	Loop	110	150
PC3	Short-seq	4	120
⋮			
PC _x	Sequential	30	∞

Figure 5. PC manager table.

4.1. PC Calculator

As shown in Figure 6, one application can be interpreted as a function call tree. Among it, two function call routines have been marked by red and blue colors. Each routine is composed of several functions. In addition, the address values of these functions can be searched in virtual address space. For example, the red function routine contains a set of functions, 'funct5()-funct3()-funct1()-main()', and its address list is 'addr5-addr3-addr1-addr0'. Each routine can be uniquely identified and represented by the sum of function addresses, called a PC value in this article. During the process of PC exploration, there is a key insight that in most applications, a few dominant I/O activities exist and most dominant I/O activities have definite data access patterns [26]. These dominant I/O activities are represented by the PC values. In conclusion, working out a PC indicator for each request invoked by a 'SYSCALL' is the task of the *PC Calculator* module.

Furthermore, as full-scanning in the virtual address space for every function costs too much, *PC Calculator* module needs to find a suitable number of function candidates. Based on our empirical tests, about 5–10 address candidates are enough to identify different calling routes and enough time-saving. In our experimental tests, we use a 3.4 GHZ CPU machine running Linux operating system. Then each system call will only take 300–400 mnsec. Furthermore, the function address searching number in our design is limited to 5. With these two premises, the calculation overhead can be negligible.

Figure 6. An example of function routines.

4.2. Pattern Classifier

As described above, every I/O request has been assigned with a PC indicator by the *PC Calculator* module. The next step is to distinguish a pattern for each PC which is the task of our *Patterns Classifier* module. We can take the routine 'funct5()-funct3()-funct1()-main()' as an example. Providing a succession of requests from this example routine occur repeatedly, this routine represented by a PC value will be tagged with a looping pattern. Once one PC value is assigned a fixed pattern, the next requests coming with the same PC will be treated as the same pattern directly.

In this design, we focus on three kinds of patterns: loop, sequential, and short sequential patterns, called as 'Loop', 'Seq', and 'Short-seq loops' patterns. These data patterns contains different parameters and are managed by different buffer policies [24,25,27]. For automatic online detection of these three references, the necessary elements are shown in Figure 7. Note that a large file can be divided into several groups with different patterns, and each group contains 4 records, a file descriptor, a start file page number, an end file page number, and an interval length, such as '(2, 2, 11, 90)' in Figure 7. The period of sequential pattern is infinite, ' ∞ '.

The elements of file groups mentioned in Figure 7 are concretely organized as a two-level list in Figure 8. Each PC value representing a function routine is the list head of a file list, and each fileID indicating a file is the list head of a file page list. The two-level list promises a fast locating for a upcoming file page accompanying a PC value. When a new page access is coming, if it is already organized by other PCs, AOIO will migrate the corresponding entry to its newest location directly. For example, in Figure 8, if the Node2 shown in FileID7 (listed by PC16) is accessed by PC1 value, then the Node2 in FileID7 will be moved from the PC16 branch to the PC1 branch. The two-level list is updated in real time.

PC1 represents a sequential pattern, in which consecutive block references only occur once. PC2 indicates a loop pattern, according to which sequential references occur repeatedly with a regular interval, named as *Period*. We use the 'virtual time' to describe the 'period' field. The 'virtual time' is not real time, but the page sequence. The 'period' field implies the average of the intervals between the same data accesses. PC3 describes a short sequential pattern. The *len* parameter is to describe the sequential length for loop and sequential patterns. Except for the three detected patterns, other patterns are not our goals which have no clear repeated regularities, named as 'Rand' patterns. The 'Pattern Classifier' module keeps updating the 'PC Manager Table' while AOIO is running. If the pattern of a file has been changed by another application, then 'Pattern Classifier' module will update and modify the changed pattern in 'PC Manager Table' directly.

The first step shown in Figure 4 will create a *PC Manager Table* for each target application. Tables for different applications are stored as configuration files in a fixed directory, in which the *Buffer Commander* module can select related tables designed by developers.

Figure 7. Examples of data patterns.

4.3. Buffer Commander

In Figure 4, *Buffer Commander* conducts the buffer according to *PC Manager Table*, in which all the related parameters of PCs trained by *PC Classifier* are listed. PC0 is a sequential reference as it has ' ∞ ' as its loop period. PC1 and PC2 are loop references with loop periods of 90 and 150, respectively. PC3 has a loop period of 120. In our second step of Figure 4, after calculating a PC value, if this PC value hits in *PC Manager Table*, this table can directly provide PC features for *Buffer Commander* without training overhead.

Proactive policies. The proactive operations contain prefetching, eviction (data will be used or not in the future) actions. All the proactive operations should be done

according to the reference features of each pattern. The sequential files will never be accessed again, while it is worth prefetching. Hence, sequential files need to be assigned with both prefetching and eviction operations. The looping and short sequential files should be considered carefully. Due to their re-access probabilities, they should be kept in the page cache, not immediately evicted. In addition, the prefetching operation should be considered for data in looping pattern. For data in 'Rand' pattern, they are not suitable for proactive policies as it has no clear repeated data or consecutive data regularities. Note that the *len* parameter shown in Figure 5 could help *Buffer Commander* to make rational decisions to make enough room for coming data and then preload predicted length, which is appropriate for sequential or loop files.

Compared to conventional *posix_fadvise()* functions in user space, AOIO conveys hints from Sys_calls in kernel space. As hints are not in real time, the relationship between our hints and the original kernel cache manager is the subordinate relationship. That is to say, all hints will be coordinated by the kernel. Only when the kernel can get around to handling hints, then these hints can be executed.

Figure 8. Data structure in PC classifier.

For a cache management policy, how to select a victim file to be deported is an important decision. After making conclusion at the proactive policies above, we can get the eviction sequence in AOIO, 'Seq' $\geq$ 'Rand' $\geq$ 'Short-seq' $\geq$ 'Loop'. However, the eviction process is not only executed by AOIO. Our entire cache workflow of a target application is that AOIO issues hints to the original kernel page cache manager firstly, and then the kernel page cache manager administers page cache according to various hints from AOIO. How to complete the entire cache workflow is described below.

Implementation in page cache of a kernel. There are several existing realistic implementations in cache source allocation, such as virtual box [28], cgroups [29], or commands in user space, *posix_fadvise()* [30,31]. However, existing techniques are complex and non-automated. For example, the virtual box and cgroups techniques need to be evaluated and set up in advance. Furthermore, their resources cannot be dynamically adjusted during operation. Developers have to understand their applications well before adding the *posix_fadvise()* function. As the *posix_fadvise()* function has its corresponding function in kernel, AOIO utilizes the kernel functions directly instead of the *posix_fadvise()* function in the user-space. The results created by kernel functions or *posix_fadvise()* function are totally the same. For example, in our code design, we use the 'invalidate_mapping_pages' kernel function to discard pages, which corresponds to the 'POSIX_FADV_DONTNEED' variable of *posix_fadvise()* function. Note that, the kernel variable like 'POSIX_FADV_DONTNEED' will be discussed later. After deploying AOIO, it can automatically run without knowing details of applications. As a result, our proactive hints in AOIO have been listed in Figure 5.

The *posix_fadvise()* function contains four variables: file descriptor (*fd*), start page position (*offset*), predicted length (*len*), and the future advice (*advice*). For the *Buffer Commander* module of AOIO, the *fd*, *offset* variables are fixed, and another two parameters, *len* and *advice*, should be predicted by rule and line. This function has six advice variables, 'POSIX_FADV_NORMAL', 'POSIX_FADV_SEQUENTIAL', 'POSIX_FADV_RANDOM', 'POSIX_FADV_NOREUSE', 'POSIX_FADV_WILLNEED', and 'POSIX_FADV_DONTNEED'. We can infer the six variables by their definitions. For example, 'POSIX_FADV_NORMAL' option means that no further suggestion, and the preloading capacity is the default value. 'POSIX_FADV_SEQUENTIAL' option will preload a sequential length guided by the *len* parameter in Figure 5. 'POSIX_FADV_RANDOM' option will give up its default preloading length. Data indicated by 'POSIX_FADV_NOREUSE' mean on time access. 'POSIX_FADV_WILLNEED' and 'POSIX_FADV_DONTNEED' options will preload and deport specified file pages at once. In conclusion, according to Figure 5, *Buffer Commander* will conduct page cache by the *posix_fadvise()* function with different data patterns, 'Loop', 'Seq', 'Short-seq loops', and 'Rand'.

5. Performance Evaluation

In this section, we exhibit the performance of the AOIO scheme, and compare it with those of the original page cache scheme (named as ORIG), LRU, UBM, PCC, AMP schemes through trace-driven simulations. We have implemented kernel modules, *PC Calculator*, *Pattern Classifier*, and *Buffer Commander* modules, discussed in Figure 4 in the Linux kernel 4.15.

5.1. Experimental Setup

Since AOIO gets pattern prediction results from the host, we collect workloads from real-world applications. All the workloads used in our simulations are obtained by currently executing diverse applications under Linux operating system. Here, we describe the features of applications used in our experimental tests.

The gcc is the GNU C compiler, in which we can observe sequential data patterns. The *postgres* application is a relational database system. We run it by join queries in two settings, named as 'hundredthoustup', 'twohundredthoustup'. The *postgres* provides sequential references and several looping references in different looping period. The *RocksDB* is a persistent key-value store for fast storage. In this article, we choose *readwhilewrite* workbench to test *RocksDB* creating sequential references. *MySQL* application is also a popular database. By running sysbench with *oltp_insert* script, we can get a random-dominant workload. The *mpeg-play* is a small video player, in which, the data pattern is dominated by strong sequential references as the video works in a stream style. The *gunplot* is a basic plotting program, which creates looping references with similar periods. The *cscope* is an application used to check C source. It exhibits looping references. Note that all the applications above will create some 'Rand' references.

In this experiment, we run several applications at the same time. They are named as *Multi1*, *Multi2*, *Multi3*, and *Multi4*, respectively. As the *MySQL* application creates 'Rand' workload, we use it as random noise and mix it with other applications. Their characteristics are shown in Table 1. *Multi1* and *Multi2* are sequential-dominant workloads with some random noise. *Multi3* workload is mainly composed of looping data with random noise. *Multi4* workload is a mixed workload with sequential, looping and random patterns.

Table 1. Specification of workloads.

Workloads	Applications Run Concurrently	#.of References	# of Unique Pages
<i>Multi1</i>	gcc, MySQL	84,109	51,687
<i>Multi2</i>	gcc, RocksDB, mpeg-play, MySQL	300,016	162,746
<i>Multi3</i>	gunplot, cscope, MySQL	203,321	40,293
<i>Multi4</i>	postgres, gcc, gunplot, MySQL GraphChi	3,202,321	810,714

5.2. Performance Comparison Results

To prove the advantages of AOIO, we compare its buffer hit ratio, disk I/O number, and average execution time with that of other buffer managements. For the cache size, we increase kernel memory size gradually to do our experimental evaluations.

Buffer hit ratio. Figure 9 shows the buffer hit ratio comparisons of four workloads under the increasing buffer size on the X-axis. The X-axis represents the entire memory size in MB. As we all know, the original kernel cache manager has its primitive proactive mechanism. Then the LRU policy might only outperform ORIG for those temporal-locality-dominant workloads. For the basic LRU policy, it has no prediction ability, and it performs the worst among remaining four schemes listed in this article. The UBM policy exploits file regularities, such as looping and sequential references which provide proactive hints leading to a better performance than that of LRU policy. However, when the workload has a large amount of small-size files, UBM will have no clear sequential or looping regularities which results in a similar performance compared with LRU policy. For PCC, it is a pattern-based buffer caching policy as well. Comparing to UBM, it classifies I/O access on a per-PC instead of per-file in UBM. Under the PC characters discussion in Section 4.2, PCC can discover longer sequential/looping length striding across several files and thus get better prefetching support than UBM. PCC can also alleviate pattern discovery overhead in UBM since the features represented by a PC value are fixed and used for next access. In AMP design, it classifies refined data patterns including a ‘non-consecutive loops’ pattern that is not mentioned in PCC. In *multi3* and *multi4* workloads, due to existing ‘non-consecutive loops’ data, AMP always outperforms PCC since AMP can distinguish and keep that ‘non-consecutive loops’ data for future access. However, in the PCC scheme, those data in ‘non-consecutive loops’ pattern are regarded as random data which provide no hints for buffer management.

Figure 9. Hit ratio comparison. (a) Multi1 workload. (b) Multi2 workload. (c) Multi3 workload. (d) Multi4 workload.

In most cases, AOIO performs the best comparing with other policies. Although AMP can recognize the ‘non-consecutive loops’ pattern, it stops to exploit further details of this pattern. AOIO continues to exploit the prefetching length and period value for a cluster of hot data, named as a ‘Short-seq loops’ pattern. The parameters, prefetching length and period value, will help AOIO to outperform AMP policy.

At last, as AMP has the best performance except AOIO, we choose the points on the X-axis where the performance gaps between AMP and AOIO are the biggest in our tests as

our comparison points. For example, by observing the test results in Figure 9c, at about 97 MB point of X-axis, the performance of AOIO increases by 29%, 27%, 26%, 21%, and 14%, compared with LRU, ORIG, UBM, PCC, and AMP, respectively. From the Figure 9d results, under about 7.8 MB cache, we can find the performance of AOIO has been improved by 46%, 45%, 37%, 17%, and 14%, compared with LRU, ORIG, UBM, PCC, and AMP. In the view of AOIO, compared with other policies, the average hit ratio improvement is 25.9%.

We know that the proactive mechanism of original kernel cache manager is suitable for sequential data, but it is not good at discrete data. As we described above, *Multi1* and *Multi2* are sequential-dominant workloads. *Multi3* workload is in looping pattern. *Multi4* workload is a mixed workload with a relative weak sequential character. Then, for the first three workloads (in sequential or looping characters), ORIG is able to make full use of continuous data to perform like the UBM scheme. However, for the last workload, *Multi4*, ORIG can not handle discontinuous data well, just performing like the LRU policy, not the UBM policy, which is shown in Figure 9d.

Average Disk I/O Numbers. Figure 10 shows the normalized disk I/O access numbers comparisons of collected workloads. In this article, we select 39 MB, 7.8 MB, 97 MB, 78 MB memory sizes for testing in *Multi1*, *Multi2*, *Multi3*, and *Multi4*, respectively. Under the cache sizes mentioned above, the performance gaps between AMP and AOIO are the biggest. The y-axis represents normalized disk I/O access numbers. Each disk I/O access number is normalized to that of LRU policy. The disk I/O access numbers are in proportion to hit ratios. The higher the hit ratio, the lower the disk I/O access number could be. For example, in Figure 10c, under the 97 MB cache size, normalized disk I/O number of AOIO decreased by 70%, 61%, 54%, 43%, and 29%, compared with that of LRU, ORIG, UBM, PCC, and AMP. Compared with other policies, the disk I/O access numbers of AOIO is reduced by about 23.2% on average.

Figure 10. Normalized disk ios performance. (a) Multi1 workload. (b) Multi2 workload. (c) Multi3 workload. (d) Multi4 workload.

Average Execution Time. Figure 11 describes the average execution time comparisons of four collected workloads under the same cache sizes, and 39 MB, 7.8 MB, 97 MB, 78 MB memory sizes mentioned before. The y -axis represents normalized execution time. Each execution time is normalized to that of LRU. Request execution time mainly depends on buffer hit ratio, probability of replacing a dirty page, page flush back cost. In conclusion, the execution time cost is comprised of two main parts, ‘disk I/O cost’ and ‘page cache cost’. In Figure 11b, the performance comparisons show the response time gaps are a little different from cache hit ratio gaps in Figure 10b. That is to say, an experimental test with a higher hit ratio might still suffer a serious delay. In Figure 10c, under the 25,000 pages size, normalized execution time of AOIO is shorter than that of LRU, ORIG, UBM, PCC, and AMP, and their differences are about 31%, 29%, 27%, 22%, and 15%, respectively. Compared with other policies, the average reduction of disk I/O access numbers of AOIO is about 20.2%.

Figure 11. Normalized execution time performance. (a) Multi1 workload. (b) Multi2 workload. (c) Multi3 workload. (d) Multi4 workload.

6. Conclusions

In this paper, we propose a new cache management algorithm called AOIO, which is based on program counter technique. Its automatic patterns identification ability promises that AOIO outperforms traditional existing cache management policies in hit ratio, disk I/O access numbers, and response time. In addition, the cache management in application level is also a flexible and efficient strategy. We try out best to reduce the PC calculation and PC classifier cost; however, it still occupies the precious software calculation resources. Therefore, to achieve better optimization, in the future, it is our goal to accelerate the pattern identification processes by hardware (FPGA or ARM chips).

Author Contributions: Conceptualization, X.L. and Z.Z.; methodology, X.L.; software, X.L. and X.Y.; validation, X.L., X.Y.; investigation, X.L. and Z.Z.; writing—original draft preparation, X.L.; writing—review and editing, X.L. and X.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Advance Research Project on Information System Equipment for the PLA during the 13th five-year plan period (No. 31511030103).

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Sivarajah, U.; Kamal, M.M.; Irani, Z.; Weerakkody, V. Critical analysis of Big Data challenges and analytical methods. *J. Bus. Res.* **2017**, *70*, 263–286. [CrossRef]
2. Chen, C.P.; Zhang, C.Y. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Inf. Sci.* **2014**, *275*, 314–347. [CrossRef]
3. Zhang, Y.; Swanson, S. A study of application performance with non-volatile main memory. In Proceedings of the 2015 31st IEEE Symposium on Mass Storage Systems and Technologies (MSST), Santa Clara, CA, USA, 30 May–5 June 2015; pp. 1–10.
4. Kim, J.M.; Choi, J.; Kim, J.; Noh, S.H.; Min, S.L.; Cho, Y.; Kim, C.S. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation, San Diego, CA, USA, 22–25 October 2000; Volume 4. Available online: <https://dl.acm.org/doi/10.5555/1251229.1251238> (accessed on 29 March 2021).
5. Jiang, S.; Zhang, X. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Perform. Eval. Rev.* **2002**, *30*, 31–42. [CrossRef]
6. Lee, D.; Choi, J.; Kim, J.H.; Noh, S.H.; Min, S.L.; Cho, Y.; Kim, C.S. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.* **2001**, *50*, 1352–1361.
7. Megiddo, N.; Modha, D.S. ARC: A Self-Tuning, Low Overhead Replacement Cache. *Fast* **2003**, *3*, 115–130.
8. Ye, X.; Zhai, Z.; Li, X. ZDC: A Zone Data Compression Method for Solid State Drive Based Flash Memory. *Symmetry* **2020**, *12*, 623. [CrossRef]
9. Gniady, C.; Butt, A.R.; Hu, Y.C. Program-counter-based pattern classification in buffer caching. *Osdi* **2004**, *4*, 27.
10. Smith, J.E. A study of branch prediction strategies. In Proceedings of the 25 Years of the International Symposia on Computer Architecture (Selected Papers), Saint-Malo, France, 19–23 June 1998; pp. 202–215.
11. Bellas, N.; Hajj, I.; Polychronopoulos, C. Using dynamic cache management techniques to reduce energy in a high-performance processor. In Proceedings of the 1999 International Symposium on Low Power Electronics and Design, San Diego, CA, USA, 16–17 August 1999; pp. 64–69.
12. Powell, M.D.; Agarwal, A.; Vijaykumar, T.; Falsafi, B.; Roy, K. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture, MICRO-34, Austin, TX, USA, 1–5 December 2001; pp. 54–65.
13. Lai, A.C.; Falsafi, B. Selective, accurate, and timely self-invalidation using last-touch prediction. In Proceedings of the 27th IEEE International Symposium on Computer Architecture (IEEE Cat. No. RS00201), Vancouver, BC, Canada, 14 June 2000; pp. 139–148.
14. Lai, A.C.; Fide, C.; Falsafi, B. Dead-block prediction & dead-block correlating prefetchers. In Proceedings of the 28th IEEE Annual International Symposium on Computer Architecture, Goteborg, Sweden, 30 June–4 July 2001; pp. 144–154.
15. Zhou, F.; von Behren, J.R.; Brewer, E.A. AMP: Program Context Specific Buffer Caching. In Proceedings of the USENIX Annual Technical Conference, General Track, Anaheim, CA, USA, 10–15 April 2005; pp. 371–374.
16. Ha, K.; Kim, J. A program context-aware data separation technique for reducing garbage collection overhead in NAND flash memory. In Proceedings of the 7th IEEE SNAPI, Denver, CO, USA, 25 May 2011.
17. Kim, T.; Hong, D.; Hahn, S.S.; Chun, M.; Lee, S.; Hwang, J.; Lee, J.; Kim, J. Fully automatic stream management for multi-streamed ssds using program contexts. In Proceedings of the 17th {USENIX} Conference on File and Storage Technologies ({FAST} 19), Boston, MA, USA, 25–28 February 2019; pp. 295–308.
18. Pages, L.M. Posix_fadvise (2)-Linux Man Page. Available online: https://pubs.opengroup.org/onlinepubs/009695399/functions/posix_fadvise.html (accessed on 29 January 2021).
19. Robinson, J.T.; Devarakonda, M.V. Data cache management using frequency-based replacement. In Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Boulder CO, USA, 21–24 May 1990; pp. 134–142.
20. O’neil, E.J.; O’neil, P.E.; Weikum, G. The LRU-K page replacement algorithm for database disk buffering. *ACM Sigmod Rec.* **1993**, *22*, 297–306. [CrossRef]
21. Lee, D.; Choi, J.; Kim, J.H.; Noh, S.H.; Min, S.L.; Cho, Y.; Kim, C.S. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Atlanta, GA, USA, 1–4 May 1999; pp. 134–143.

22. Cao, P.; Felten, E.W.; Li, K. Application-Controlled File Caching Policies. In Proceedings of the USENIX Summer, Boston, MA, USA, 6–10 June 1994; pp. 171–182.
23. Patterson, R.H.; Gibson, G.A.; Ginting, E.; Stodolsky, D.; Zelenka, J. Informed prefetching and caching. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, 1995; pp. 79–95.
24. Zapata, H.; Perozo, N.; Angulo, W.; Contreras, J. A Hybrid Swarm Algorithm for Collective Construction of 3D Structures. *Int. J. Artif. Intell.* **2020**, *18*, 1–18.
25. Soares, A.; Râbelo, R.; Delbem, A. Optimization based on phylogram analysis. *Expert Syst. Appl.* **2017**, *78*, 32–50. [[CrossRef](#)]
26. Choi, J.; Noh, S.H.; Min, S.L.; Cho, Y. Towards application/file-level characterization of block references: A case for fine-grained buffer management. In Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Santa Clara, CA, USA, 18–21 June 2000; pp. 286–295.
27. Precup, R.E.; David, R.C.; Petriu, E.M.; Szedlak-Stinean, A.I.; Bojan-Dragos, C.A. Grey Wolf Optimizer-Based Approach to the Tuning of Pi-Fuzzy Controllers with a Reduced Process Parametric Sensitivity. *IFAC PapersOnLine* **2016**, *49*, 55–60. [[CrossRef](#)]
28. Li, P. Selecting and using virtualization solutions: Our experiences with VMware and VirtualBox. *J. Comput. Sci. Coll.* **2010**, *25*, 11–17.
29. Oh, K.; Park, J.; Eom, Y.I. Weight-Based Page Cache Management Scheme for Enhancing I/O Proportionality of Cgroups. In Proceedings of the 2019 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 11–13 January 2019.
30. Plonka, D.; Gupta, A.; Carder, D. Application Buffer-Cache Management for Performance: Running the World's Largest MRTG. In Proceedings of the Large Installation System Administration Conference, Dallas, TX, USA, 11–16 November 2007.
31. Congiu, G.; Grawinkel, M.; Padua, F.; Morse, J.; Brinkmann, A. MERCURY: A Transparent Guided I/O Framework for High Performance I/O Stacks. In Proceedings of the 2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), St. Petersburg, Russia, 6–8 March 2017.