electronics

MDPI

Article

CluMP: Clustered Markov Chain for Storage I/O Prefetch

Sungmin Jung [1], Hyeonmyeong Lee [2] and Heeseung Jo [1,*]

[1] Department of Computer Science, Chungbuk National University, Chungju 28644, Republic of Korea; jjsm9595@cbnu.ac.kr
[2] Department of Software, School of Software, Sungkyunkwan University, Suwon-si 16419, Republic of Korea; myeong58@g.skku.edu
* Correspondence: heesn@cbnu.ac.kr

Abstract: Due to advancements in CPU and storage technologies, the processing speed of tasks has been increasing. However, there has been a relative slowdown in the data transfer speeds between disks and memory. Consequently, the issue of I/O processing speed has become a significant concern in I/O-intensive tasks. This research paper proposes CluMP, which predicts the next block to be requested within a process using a clustered Markov chain. Compared to the simple read-ahead approach commonly used in Linux systems, CluMP can predict prefetching more accurately and requires less memory for the prediction process. CluMP demonstrated a maximum memory hit ratio improvement of 191.41% in the KVM workload compared to read-ahead, as well as a maximum improvement of 130.81% in the Linux kernel build workload. Additionally, CluMP provides the advantage of adaptability to user objectives and utilized workloads by incorporating several parameters, thereby allowing for optimal performance across various workload patterns.

Keywords: prefetch; Markov chain; data prediction; disk I/O

1. Introduction

The capacity of storage devices and the processing capability of CPUs have been steadily improving. However, the performance in terms of bandwidth and access time of disk I/O systems has not seen significant advancements. As a result, the performance gap between CPUs and disk I/O systems is widening. Disk arrays such as RAID can enhance overall I/O throughput, but they still suffer from long random access latency due to mechanical operations. Disk buffers and cache layers utilizing main memory can reduce the waiting time, but for workloads with frequent cache misses, the improvement in access time is limited [1]. Consequently, extensive research has been conducted to provide operating-system-level support for preventing bottlenecks and improving performance when handling disk I/O-intensive tasks.

The Linux kernel supports disk I/O operations through techniques such as prefetching and paging, which manages memory as a cache for disk data. Prefetching refers to the technique of loading disk data into the memory cache in advance to reduce the waiting time for disk access. When a cache miss occurs in the memory, it takes a considerable amount of time to fetch the data from the disk. Prefetching aims to predict the data that will be requested and proactively loads it into the memory before the actual access occurs, thereby reducing the I/O time. For prefetching to be effective, it is crucial to accurately predict the data to be requested with minimal overhead at the appropriate timing. Prefetching techniques need to consider secondary effects such as cache pollution and increased memory bandwidth requirements. Despite these challenges, prefetching can significantly improve the overall program execution time by overlapping computation and I/O access. Numerous studies have proposed algorithms to perform prefetching more effectively [2–4].

Citation: Jung, S.; Lee, H.; Jo, H. CluMP: Clustered Markov Chain for Storage I/O Prefetch. *Electronics* **2023**, *12*, 3293. https://doi.org/10.3390/electronics12153293

Received: 26 June 2023
Revised: 26 July 2023
Accepted: 29 July 2023
Published: 31 July 2023

In the Linux kernel, a simple sequential read-ahead algorithm is used as one of the prefetching techniques. The Linux read-ahead algorithm demonstrates significant performance improvement for operations that access blocks sequentially. However, the Linux read-ahead algorithm can actually degrade performance for workloads that access data nonsequentially. It has a vulnerability where it reads unnecessary data into memory, thus resulting in memory wastage.

Indeed, the kernel of an operating system needs to handle exceptions and interruptions swiftly. Due to this requirement, it is not feasible to use complex prediction models such as deep learning for I/O processing support. The focus of the kernel is on maintaining efficient and responsive handling of exceptions and interruptions, rather than deploying computationally intensive prediction models. Therefore, the use of simpler and lightweight prediction models, such as Markov chains becomes more suitable for I/O-related tasks in the operating system kernel. The overhead for prediction within the operating system kernel needs to be minimal to be practically feasible. In this paper, a lightweight prediction model, the Markov chain, was employed to predict the next requested block. The paper proposes the Clustered Markov Chain for Prefetch (CluMP), which minimizes memory overhead by adjusting the size of the Markov chain and enables accurate predictions with reasonable time overhead. CluMP adjusts the number and size of Markov chains to suit the user's workload, thus resulting in a new prefetch technique that achieves a reasonable prefetch hit ratio, even for operations that access blocks randomly.

The remaining sections of the paper are organized as follows. Section 2 provides background knowledge on the prefetch algorithms of the Linux and Markov chain. In Section 3, the design, data structures, and operational algorithm of the CluMP proposed in this paper are described, and Section 4 presents a performance evaluation that analyzes the effectiveness and overhead of the CluMP. Furthermore, Section 5 discusses related research, while Section 6 concludes the paper by summarizing the key findings and contributions.

2. Background Knowledge

2.1. The Linux Read-Ahead Algorithm

The Linux kernel introduced the Linux read-ahead algorithm from version 2.4.13 onwards to enhance disk I/O performance [5]. The Linux read-ahead algorithm considers the sequentiality of data and preloads a sequence of blocks, including the requested block, from the disk to memory when the I/O operations access consecutive blocks in a sequential pattern.

In Linux kernel 5.14.0, the prefetch algorithm utilizes file descriptors (FDs) to identify the start and end of the file for the I/O operation. When the requested block is not present in the memory and the blocks being requested from the disk are contiguous, prefetching is performed. The kernel initially loads a prefetch window size of 128 KB of future contiguous data into the memory. If subsequent requests for sequential blocks persist, the prefetch window size is doubled incrementally, thereby allowing for more efficient prefetching. However, if nonsequential reads commence thereafter, the read-ahead algorithm resets the prefetch window size to reduce memory wastage.

2.2. Markov Chain

A Markov chain (MC) is a discrete-time stochastic process. A Markov chain represents the changes in the states of a system over time. At each time step, the system either transitions to a new state or remains in the same state. These state transitions are known as transitions. The Markov property states that the conditional probability distribution of future states, given the past and current state, is determined solely by the current state and is independent of the past states.

Figure 1 shows the transition counts from each current state *(t)* to each of the states *A*, *B*, and *C* in the next state *(t')*. For example, if the transition occurs from state *A* to state *B* five times and from state *A* to state *C* five times as well, the probabilities *B(t')* and *C(t')* would be 50% each. Table 1 presents the transition probabilities for all cases, thus depicting

the Markov state transitions for states *A*, *B*, and *C*, which constitute the Markov chain (MC). Each transition is determined by the accumulated probabilities of the previous transitions and is independent of past actions. The Markov state transition probabilities dictate the next state without being influenced by previous actions. As transitions accumulate over time, the impact of an individual transition on the overall probability diminishes.

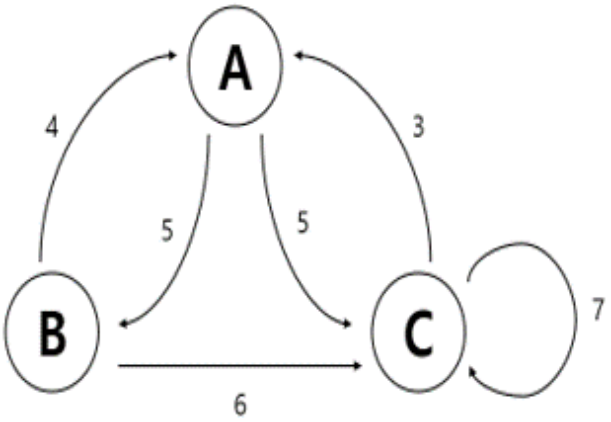

Figure 1. Example of a Markov state transition diagram. *A*, *B*, and *C* denote states, and the numbers show the number of transition.

Table 1. Example of a Markov state transition table.

	A (t′)	**B (t′)**	**C (t′)**
A (t)	0	0.5	0.5
B (t)	0.4	0	0.6
C (t)	0.3	0	0.7

Based on the probability distribution of the Markov chain, it can be effective to predict the next block of disk I/O for prefetch algorithms. For example, assuming *A*, *B*, and *C* in Table 1 represent blocks of the disk, if block *B* is requested, the probability of accessing block *C* next is 60%, while the probability of accessing block *A* is 40%. Therefore, prefetching block *C*, rather than block *A*, would be a preferable choice.

3. Design

3.1. Overview

The existing Linux read-ahead algorithm in the Linux kernel suffers from a critical limitation, as it operates only effectively for sequential access patterns. This means that prefetching is not triggered at all when I/O requests exhibit nonsequential behavior, thereby leading to a significant performance degradation for workloads with random access patterns. Additionally, another constraint is the fixed prefetch window size of 128 KB, with adjustments limited to multiples of this size. A notable drawback arises when the prefetch window size increases due to sustained sequential accesses, but, subsequently, nonsequential access occurs. In such cases, a substantial amount of prefetched data becomes unnecessary, thereby resulting in significant memory waste. These limitations in the Linux read-ahead algorithm hinder its ability to efficiently handle diverse I/O access patterns. As a result, alternative approaches or enhancements are required to address these challenges and optimize prefetching performance in scenarios involving both sequential and nonsequential access patterns.

In general, considering the widely used prediction method called Least Recently Used (LRU), it is possible to predict that disk I/O requests will follow the sequence of past requests. However, maintaining LRU for the entire disk incurs a significant overhead, thus making it impractical. The CluMP is designed to utilize the MC to predict the next

requested block. Therefore, when a block is read, the CluMP predicts that the next requested block will be the one most frequently requested based on the MC.

One notable problem is the need to maintain a significant amount of data in the memory if we were to construct the MC for the entire disk. For example, assuming a 4 GB disk with 4 KB blocks, it would necessitate a 1 million $\times$ 1 million array. Such a requirement is deemed impractical. Moreover, as the disk size increases, the size of the MC needed would proportionally grow.

In reality, not all blocks on a disk contain valid data, and even among the valid blocks; only a small portion is actively used by the workload. Therefore, constructing and maintaining an MC for all blocks in advance would be inefficient. In the CluMP, we aim to address this issue by dividing the MC into multiple segments, thereby targeting the entire disk. These segmented MCs can be created and managed dynamically when necessary, similar to the multilevel page table technique, thus significantly reducing overhead.

3.2. Structure

Figure 2 illustrates the structure of the MC employed in the CluMP. To dynamically manage the necessary MC fragments, the concept of chunks and clusters was introduced. In a conventional MC model, for a disk with N blocks, it requires a 2-dimensional data space of $N \times N$. However, for sparse MCs such as in disks, a significant portion of the $N \times N$ space remains unused. Therefore, the CluMP utilizes chunk and cluster to efficiently manage and maintain only the required portion of the $N \times N$ space by slicing it accordingly.

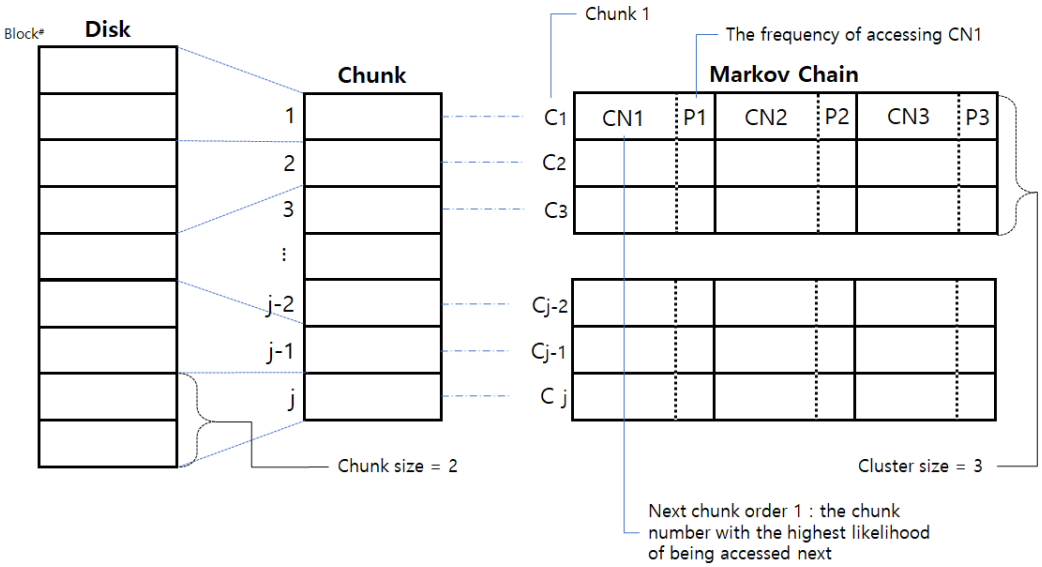

Figure 2. Markov chain structure of CluMP.

A chunk represents a set of disk blocks. In Figure 2, an example is shown where a chunk contains two blocks. The CluMP utilizes chunks to efficiently manage a large number of blocks and prevent memory wastage. The chunk size refers to the number of disk blocks included within a single chunk.

A cluster represents a set of MC fragments, with each cluster corresponding to multiple chunks. The cluster size denotes the number of chunks associated with a single cluster. In order to maintain the Markov chain for predicting the requested data, clustered MCs are instantiated in the memory selectively, thus targeting a specific moment and involving a number of chunks equal to the cluster size. This approach enables efficient memory utilization by keeping only the necessary clustered MCs in the memory.

3.3. Management of MC Cluster

The traditional MC requires data proportional to the number of blocks squared, with each row requiring a number of columns equal to the number of blocks. To minimize this

overhead, the CluMP employs a reduced representation by structuring each row of the MC, which predicts the next chunk with six fields. This design choice enables the CluMP to maintain and manage only the most probable next block for predicting the upcoming disk I/O requests.

If we consider the total number of blocks on the disk as B_{total}, the size of a chunk as CH_{size}, and the size of a cluster as CL_{size}, then the total required memory size can be calculated as follows:

$$CH_{total}\,(Total\ number\ of\ Chunks) = B_{total}/CH_{size}.$$

$$CL_{total}\,(Total\ number\ of\ Clusters) = CH_{total}/CL_{size}.$$

$$Mem_{required}\,(Maximum\ memory\ usage) = CL_{total} \times 24B \times CL_{size}.$$

Here, *24B* represents the size required to store the information for each cluster line (*six field* $\times$ *4B*). Indeed, in the CluMP, not all clusters are preallocated, so the actual memory usage is much smaller than $Mem_{required}$. Clusters are allocated and maintained only when they are needed, which means that the CluMP dynamically allocates memory as required, thereby making its memory usage much more efficient. The actual memory utilization will vary depending on the characteristics of the workload being processed by the CluMP.

As depicted in Figure 2, each row of the Markov chain represents the probabilities of accessing the next chunk, where *C1* to *CN1, CN2, and CN3* (referred to as *CNx*) denote the chunk numbers with the highest likelihood of being accessed next. *P1, P2, and P3* (referred to as *Px*) indicate the frequencies of accessing the corresponding chunks *CNx*. *CN1* holds the chunk number that has been accessed the most, *CN2* represents the second most-accessed chunk number, and *CN3* stores the most recently accessed block number. Additionally, *CN3* serves as a buffer for sorting purposes.

With each I/O access, the values of *Px* are updated, thus leading to the reordering of *CNx* values. When multiple *Px* values are equal, the most recently updated value is considered to have a higher probability of being accessed next. For instance, in Figure 3, when row *C1* holds the information and the subsequent access is to *C2*, the value of *P2* is incremented by 1, resulting in a change to 3. In this scenario, although *P1* and *P2* have the same value, the chunk value stored in *CN2*, which was most recently updated, is swapped with the chunk value in *CN1*, while the previous value of *CN1* is assigned to *CN2*.

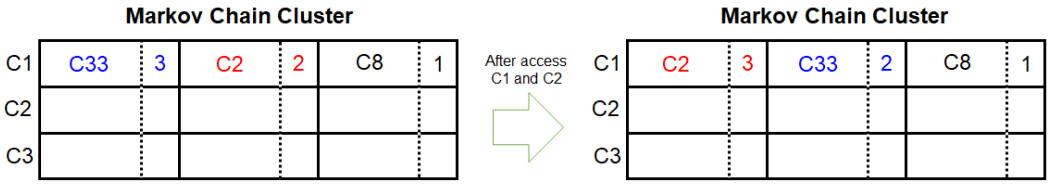

Figure 3. Example of MC management for an I/O access.

The MC row is updated with each occurrence of a block access request. The *CNx* and *Px* values contain records for previously requested chunks, and if there is a new I/O request for a chunk that is not already in *CNx*, the existing *CN3* and *P3* are respectively initialized with the recently accessed chunk number and 1. If there are existing records for a chunk that include the requested block within the MC, the corresponding *Px* value is incremented by 1. Through sorting with each request, *CN1* consistently stores the block number with the highest probability of being accessed next within that chunk. For prefetching purposes, the CluMP always refers to *CN1* and uses it to predict the next I/O request.

Once the predicted chunk is determined in the CluMP, prefetching is performed with a prefetch window size that can be defined by the user. One notable feature of the CluMP, which utilizes the MC, is its ability to efficiently perform prefetching without the need for dynamic adjustment of the prefetch window size, unlike the read-ahead mechanism in Linux.

Figure 4 shows the operation sequence of the CluMP for one read operation, and the detailed explanation is as follows:

1. A disk I/O read operation is requested.
2. The operation checks if the requested disk block is present in the memory. If the requested data is in the memory, it verifies the existence of the Markov chain and either creates or updates the information.
3. If the requested data is not present in the memory, it requests a read from the disk.
4. It retrieves the corresponding data from the disk and loads it into the memory.
5. It checks if there is an existing Markov chain for the data.
6. If a predicted Markov chain exists, it updates the information for the corresponding chunk number.
7. Using the updated Markov chain's predictions, it performs a prefetch read of the prefetch window size.
8. If there is no Markov chain, it creates a new one using the available information.

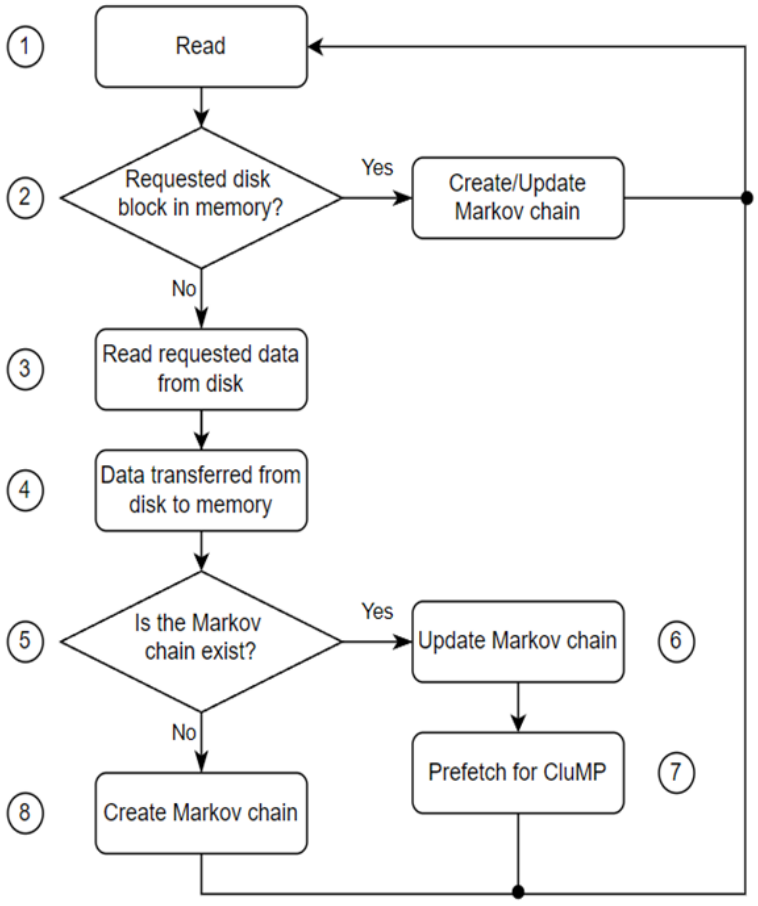

Figure 4. Operation sequence of CluMP.

4. Evaluation

In this section, we aim to evaluate the performance of the CluMP in comparison to the existing Linux read-ahead algorithm. The Linux kernel version used for the comparison is 5.4.0. The main objectives of this performance evaluation are as follows:

- Hit Rate of Prefetching: We compared the improvement in the cache hit rate achieved by the CluMP over the Linux read-ahead algorithm. Real workloads, such as booting a KVM virtual machine and building the Linux kernel, were used for analysis. We evaluated the performance by comparing the cache hit rates between the CluMP and the Linux read-ahead algorithm.

- Analysis of CluMP Overhead: The CluMP introduces overhead due to the maintenance and update of the MC. We analyzed the overhead incurred by the CluMP in relation to the performance gains it offered. This analysis helped us understand the trade-off between performance improvement and the additional overhead introduced by the CluMP.
- Performance Variation with Parameters: We analyzed the performance variation and its impact on the miss prefetch and memory efficiency by tuning different parameters such as the chunk size, cluster size, and prefetch window size. Through this analysis, we aimed to identify the optimal parameter values that yielded improved performance and memory efficiency.

By conducting this performance evaluation, we sought to gain insights into the cache hit rate improvement achieved by the CluMP, analyze the overhead associated with the CluMP, and confirm the effect of different parameter configurations on the performance of the CluMP compared to the Linux read-ahead algorithm.

In order to obtain accurate prefetch hit/miss ratios, we utilized disk I/O logs collected during the execution of workloads in an SSD environment. These logs were utilized for performance evaluation purposes. Through simulation based on the logs, we were able to accurately determine the cache hit rates and memory usage. Furthermore, we were able to precisely measure the memory overhead associated with the CluMP.

4.1. Workloads

In order to evaluate the performance, we constructed workloads that reflected real-world scenarios. The following workloads were used:

- Kernel-based Virtual Machine (KVM) Boot: During the boot process of a virtual machine, we traced the requested block numbers using iosnoop and recorded the logs. The CluMP utilized these logs to create Markov chains (MC) and performed prefetching of the blocks into the memory based on the generated block sequences. We measured the hit and miss rates and compared them with the results of Linux's read-ahead algorithm. The disk usage requested by the KVM workload was 42.53 MB.
- Linux Kernel Build: We recorded block logs during the building of the Linux kernel using the make tool. To introduced load and accessed nonconsecutive blocks; we simultaneously built two different Linux kernels. We compared the hit and miss rates of the CluMP using the MC created from the log files with the read-ahead algorithm of the traditional Linux implementation. The workload size for building the two Linux kernels was 7.96 GB.

4.2. Hit Rate of Prefetch

Figures 5 and 6 illustrate the comparison of the memory hit rates between the CluMP and the Linux read-ahead algorithm (default) using the KVM booting workload and the Linux kernel build workload. The memory hit rate of the Linux read-ahead algorithm was 41.39% and 59% for each workload, respectively. In contrast, the CluMP achieved maximum memory hit rates of 79.224% and 77.252%, respectively, which represent improvements of 1.91 times and 1.31 times, respectively. While the hit rate in the CluMP showed some variation depending on the chunk size and cluster size, it consistently outperformed the Linux read-ahead algorithm overall.

The hit rate is influenced more by the chunk size than the cluster size. The cluster size determines the size of the MC fragments loaded into the memory, thus having a trivial impact on the hit rate. On the other hand, the chunk size, which corresponds to the size of consecutive disk block sets, directly affects the hit rate.

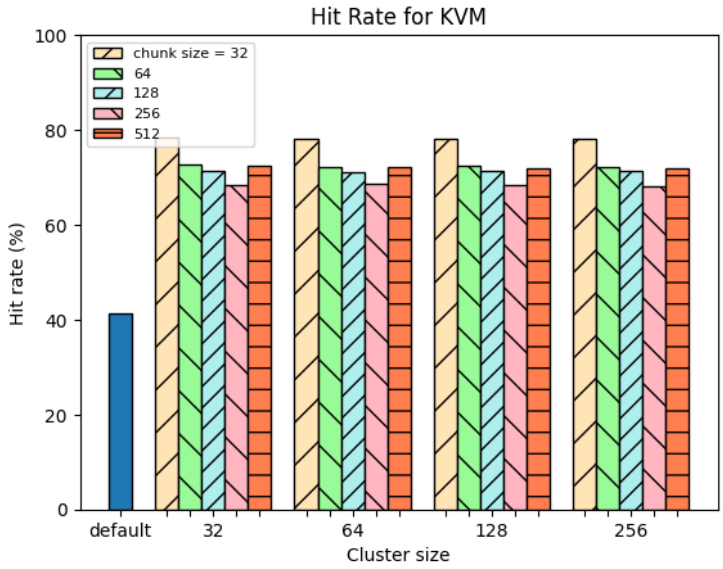

Figure 5. Hit rate of Linux read-ahead (default) and CluMP for KVM workload.

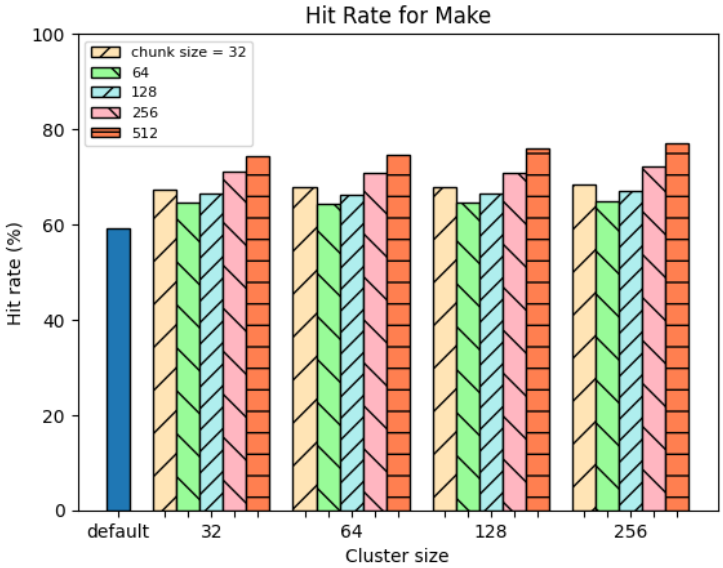

Figure 6. Hit rate of Linux read-ahead (default) and CluMP for Linux kernel build workload.

4.3. Comparison of Miss Prefetch

A missed prefetch refers to the blocks that were prefetched from the disk to the memory based on the prefetch algorithm and mechanism but that were not actually utilized. Figures 7 and 8 demonstrate the quantity of unused blocks that were prefetched using the Linux read-ahead and the CluMP algorithms. With an increase in chunk size, a larger amount of data was prefetched at once, which could result in a higher occurrence of including blocks that were not originally requested by the workload.

When excluding chunk sizes of 256 and 512, the CluMP exhibited significantly lower levels of missed prefetches. Although chunk sizes of 256 and 512 were included for evaluation purposes, they are not considered suitable for practical use. Missed prefetch data were loaded into memory but remained unused, thus resulting in unnecessary memory consumption. By generating significantly fewer missed prefetches compared to the Linux read-ahead, the CluMP demonstrated its superiority. The CluMP allowed for a reduction in the missed prefetch quantity compared to the traditional readahead approach.

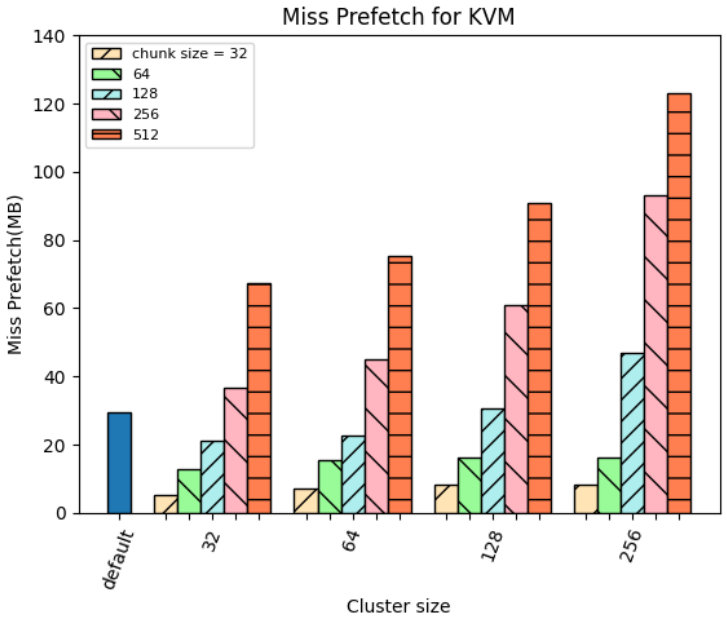

Figure 7. Missed prefetch amount of Linux read-ahead (default) and CluMP for KVM workload.

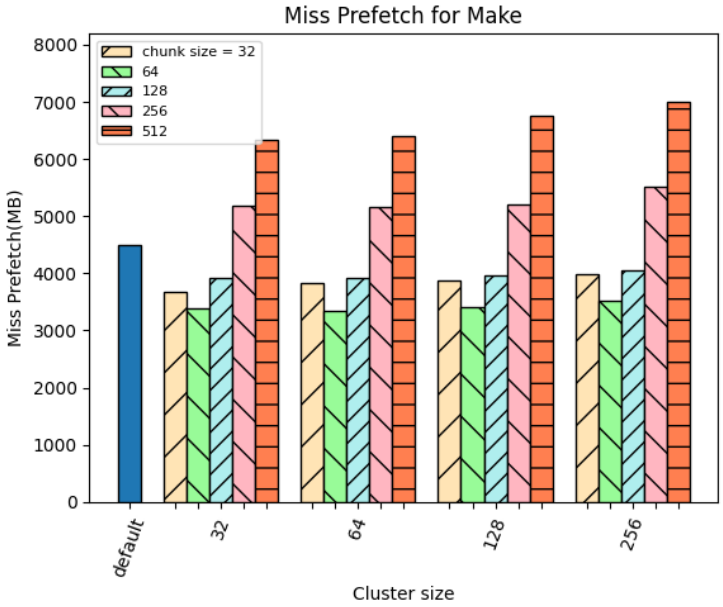

Figure 8. Missed prefetch amount of Linux read-ahead (default) and CluMP for Linux kernel build workload.

4.4. Memory Overhead for MC

Figures 9 and 10 depict the cumulative sizes of the MC used during workload execution while varying the cluster and chunk sizes. When the CluMP performs prefetching, it needs to maintain the MC fragments in memory to predict the blocks that will be requested. This size is directly proportional to the size of the working set of blocks utilized by the workload.

As the chunk size increases, the accuracy of the MC in predicting the next block to be accessed becomes less precise. This leads to an increase in the number of MC fragments being used. However, since each MC fragment has a relatively small size, the overall memory consumption of the MC remains modest compared to the total workload size. Despite the increased number of MC fragments, the effectiveness of the CluMP prefetching in improving cache hit rates was substantial. Therefore, by carefully adjusting the chunk size, it was possible to achieve efficient memory utilization while still benefiting from the significant improvement in prefetch hit rates provided by the MC.

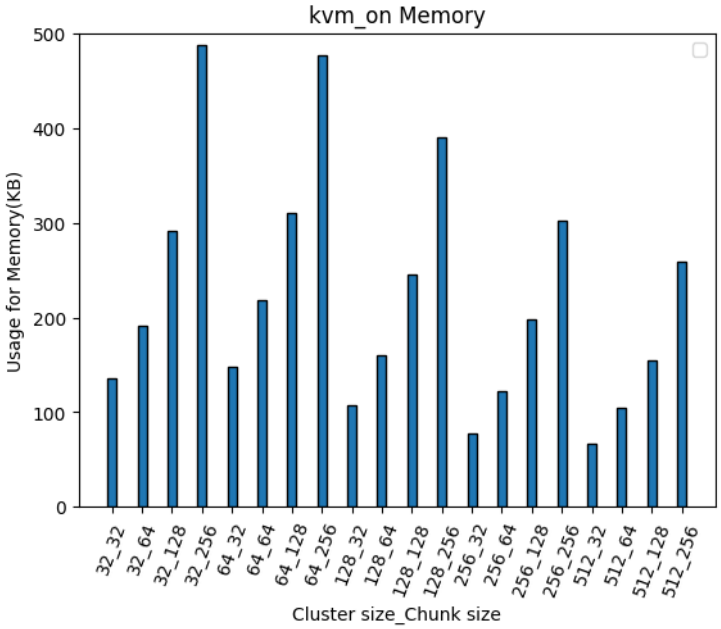

Figure 9. Memory overhead of Linux read-ahead (default) and CluMP for KVM workload.

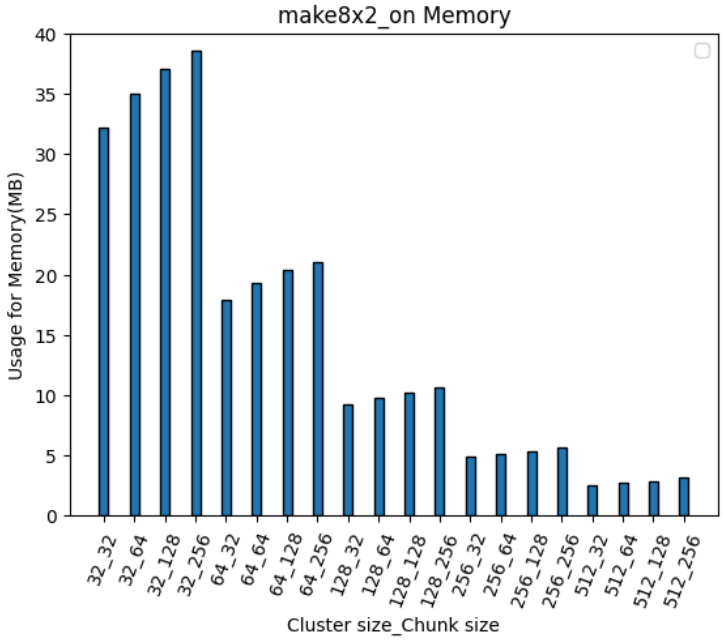

Figure 10. Memory overhead of Linux read-ahead (default) and CluMP for Linux kernel build workload.

Moreover, as the cluster size in the CluMP increased, the number of chunks included in a single clustered MC also increased. This resulted in a reduction in the cost of maintaining the MC in memory. Therefore, considering prefetch hit rate, missed prefetch rate, and memory overhead, it can be reasoned that using a smaller chunk size and a larger cluster size is more reasonable and beneficial in practice.

5. Discussion

The CluMP may introduce overhead in two aspects. Firstly, it requires some memory space to manage the MC fragments in the memory. However, this overhead is relatively small and can be considered negligible compared to traditional MC models, as evidenced by the experimental results presented in Section 4.4. The second overhead is related to

the computation needed to manage and update the MC fragments. Nevertheless, the computational overhead for managing the MC fragments in the CluMP is minimal, as it involves a single computation to update data in the MC for each disk I/O request. The computational complexity was $O(1)$, thereby indicating that the actual computation overhead is insignificant and can be ignored.

In this paper, the experimental results primarily focused on the analysis of short-term data for the CluMP. However, it is reasonable to anticipate that the CluMP will also exhibit robust performance with long-term workloads. This expectation stems from the inherent characteristics of the MC, as it continuously records and retains long-term workload patterns within its structure. Consequently, the accumulated information allows the CluMP to make more accurate predictions for future accesses.

Although the results for long-term data and workload patterns were not presented in this paper due to the time-consuming nature of long-term experiments, it can be acknowledged as a limitation of this study. To address this limitation and provide further insights into the behavior of the CluMP with long-term workloads, future work endeavors will involve conducting extended experiments and analysis. By doing so, we aim to validate and establish the effectiveness of the CluMP in handling long-term workloads effectively.

6. Related Work

Modern processors are equipped with various hardware prefetchers, including CPU caches, to enhance memory hierarchy. These hardware prefetchers work in conjunction with software-based prefetch algorithms to target different levels of the memory hierarchy. Several prefetching studies have been conducted to accelerate disk I/O operations, which are the focus of this paper [6–9]. By leveraging these prefetching techniques, the aim is to provide faster support for disk I/O operations and improve the overall system performance.

Zhang et al. proposed the iTransformer [10], a technique aimed at alleviating the memory constraints of the DRAM while improving the processing speed of the I/O operations through large-scale prefetching. By utilizing small SSDs as nonvolatile memory, the concerns associated with dirty data could be mitigated while maintaining high disk efficiency, even under concurrent requests. Additionally, iTransformer employs background request scheduling to mask the costs associated with disk access.

Bhatia et al discussed an extended prefetching technique [11] that enhanced the effectiveness of the baseline prefetching approach. The proposed technique, called Perceptron-Based Prefetch Filtering (PPF), expanded the scope of prefetching without negatively impacting accuracy. It achieved this by filtering out inaccurate prefetches. PPF trains the perceptron layer to identify inaccurate prefetches, thereby improving the accuracy of the filtering process.

There have been previous attempts to utilize Markov chains for prefetching [6,12–15]. Laga et al., for instance, created a Markov state machine for every page and used the Markov chain to predict the next page requested by the process. They maintained Markov state values for all page–page pairs and updated the Markov chain when the state values reached a certain threshold to address overfitting issues. However, this approach could result in significant memory wastage due to the need to maintain Markov state for every page.

In contrast, the CluMP leverages a chunk and cluster structure that groups pages or blocks, thereby allowing for the more efficient utilization of the memory space required to maintain the Markov chain. By organizing the data into chunks and clusters, the CluMP optimizes the memory usage for Markov chain maintenance.

7. Conclusions

The present paper proposed the CluMP as a novel solution for enhancing the efficiency of memory utilization and accelerating the processing of I/O operations. The CluMP adopted a methodology wherein the MC that captured the block requests made by processes were organized into clusters, thereby enabling effective prefetching. Notably, the CluMP exhibited noteworthy improvements in the memory hit rates, which ranged

from approximately 8% to 91%, when compared to the conventional Linux read-ahead algorithm. Although the size of the MC utilized for block prediction may increase with larger workloads, the memory space required for prefetching will remain significantly diminutive in relation to the overall workload. Thus, the CluMP highlights the substantial advantages conferred by prefetching, thus outweighing the associated minimal memory overhead. Additionally, the utilization of chunk and cluster structures within the CluMP substantially mitigates the memory footprint necessary for the MC, thus constituting a prominent advantage of the CluMP.

As a future research direction, the plan is to implement an enhanced version of the CluMP in the actual kernel, thereby utilizing file descriptors to determine the start and end points of the files. This improved CluMP implementation will facilitate prefetching for only those blocks that fall within the range of the workload, thus avoiding unnecessary prefetching. By enhancing the prefetch algorithm in the Linux kernel through this approach, it is expected to further reduce the I/O processing time effectively.

Author Contributions: Conceptualization, S.J. and H.J.; software, S.J.; experiments, S.J. and H.L.; writing—original draft preparation, S.J. and H.J.; writing—review and editing, S.J., H.L. and H.J.; supervision, H.J.; project administration, H.J. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2020R1F1A1075941), and by the National Security Research Institute (No. 2023-094).

Data Availability Statement: The data presented in this study are available on request from the first author.

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

References

1. Yang, Q.; Ren, J. I-CASH: Intelligently coupled array of SSD and HDD. In Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture, San Antonio, TX, USA, 12–16 February 2011; pp. 278–289.
2. Vanderwiel, S.P.; Lilja, D.J. Data prefetch mechanisms. *ACM Comput. Surv. (CSUR)* **2000**, *32*, 174–199. [CrossRef]
3. Kang, H.; Wong, J.L. To hardware prefetch or not to prefetch? a virtualized environment study and core binding approach. *ACM Sigplan Not.* **2013**, *48*, 357–368. [CrossRef]
4. He, J.; Sun, X.H.; Thakur, R. Knowac: I/o prefetch via accumulated knowledge. In Proceedings of the 2012 IEEE International Conference on Cluster Computing, Beijing, China, 24–28 September 2012; pp. 429–437.
5. Wu, F.; Xi, H.; Li, J.; Zou, N. Linux readahead: Less tricks for more. In Proceedings of the Linux Symposium, Ottawa, ON, Canada, 27–30 June 2007; Volume 2, pp. 273–284.
6. Laga, A.; Boukhobza, J.; Koskas, M.; Singhoff, F. Lynx: A learning linux prefetching mechanism for ssd performance model. In Proceedings of the 2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA), Daegu, Republic of Korea, 17–19 August 2016; pp. 1–6.
7. Lee, C.J.; Mutlu, O.; Narasiman, V.; Patt, Y.N. Prefetch-aware memory controllers. *IEEE Trans. Comput.* **2011**, *60*, 1406–1430. [CrossRef]
8. Nesbit, K.J.; Smith, J.E. Data cache prefetching using a global history buffer. In Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA'04), Madrid, Spain, 14–18 February 2004; p. 96.
9. Kallahalla, M.; Varman, P.J. PC-OPT: Optimal offline prefetching and caching for parallel I/O systems. *IEEE Trans. Comput.* **2002**, *51*, 1333–1344. [CrossRef]
10. Zhang, X.; Davis, K.; Jiang, S. iTransformer: Using SSD to improve disk scheduling for high-performance I/O. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, Shanghai, China, 21–25 May 2012; pp. 715–726.
11. Bhatia, E.; Chacon, G.; Pugsley, S.; Teran, E.; Gratz, P.V.; Jiménez, D.A. Perceptron-based prefetch filtering. In Proceedings of the 46th International Symposium on Computer Architecture, Phoenix, AZ, USA, 22–26 June 2019; pp. 1–13.
12. Joseph, D.; Grunwald, D. Prefetching using markov predictors. *IEEE Trans. Comput.* **1999**, *48*, 121–133. [CrossRef]
13. Peled, L.; Mannor, S.; Weiser, U.; Etsion, Y. Semantic locality and context-based prefetching using reinforcement learning. In Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, 13–17 June 2015; pp. 285–297.

14. Chen, H.; Zhou, E.; Liu, J.; Zhang, Z. An rnn based mechanism for file prefetching. In Proceedings of the 2019 18th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES), Wuhan, China, 8–10 November 2019; pp. 13–16.
15. Hashemi, M.; Swersky, K.; Smith, J.; Ayers, G.; Litz, H.; Chang, J.; Kozyrakis, C.; Ranganathan, P. Learning memory access patterns. In Proceedings of the International Conference on Machine Learning, Stockholm, Sweden, 10–15 July 2018; pp. 1919–1928.