Lynx: A Learning Linux Prefetching Mechanism For SSD Performance Model

Arezki Laga*†, Jalil Boukhobza†, Michel Koskas*, Frank Singhoff†
* Kode Software, Paris, France
† Univ. Bretagne Occidentale UMR 6285, Lab-STICC F-29200 Brest, France
Email: arezki@kodeSoftware.com, boukhobza@univ-brest.fr

***Abstract*—Traditional Linux prefetching algorithms were based on spatial locality of I/O workloads and performance model of hard disk drives. From the applicative point of view, current data-intensive applications I/O workloads are turning towards more random patterns while from the storage device perspective, flash based storage devices present a different performance model than HDDs. In this work, we present a new prefetching mechanism named Lynx. Lynx aims to adapt and/or complement the Linux read-ahead prefetching system for both SSD performance model and new applications needs. Lynx uses a simple machine learning system based on Markov chains. The learning phase detects I/O workload patterns and computes the transition probabilities between file pages. The prediction phase prefetchs predicted file pages with the resulting Markov state-machine. We have implemented our solution and integrated it into the Linux kernel. We experimented our solution using the TPC-H benchmark. The results show that Lynx divides the number of page cache misses (major page faults) by 2 on average and thus reduces TPC-H queries execution time by 50% as compared to traditional Linux read-ahead.**

I. Introduction

As the performance gap between the main memory and storage system is very wide and has been continuously growing [1], prefeching mechanisms at the operating system level were introduced in order to partly absorb performance difference between DRAM and Hard Disk Drives (HDD).

The Linux kernel includes, since version 2.6, a prefetching mechanism called Read-ahead that is integrated at the Virtual File System (VFS) level. The read-ahead prefetching mechanism loads data from disk into a buffer called the page cache. Read-ahead is mainly based on the assumption that most I/O workloads present a high spatial locality [2] and hence, when Linux detects a sequential pattern, it prefetches data from the disk to the page cache in DRAM before being requested by the application. The objective of doing so is both to hide and reduce I/O latencies. Indeed, read-ahead operates asynchronously during application I/O timeouts which allows prefetching data while processing application I/O requests and thus hiding latencies. Read-ahead allows also to reduce I/O latencies by insuring sequential access to HDDs while loading larger data volumes rather than issuing many small I/O requests which tend to increase HDD mechanical movements and thus I/O latencies. In fact, read-ahead mechanism was based on HDD performance model in which sequential and random operations are asymmetric and I/O latency is high due to those mechanical movements.

In the last decades, flash-based solid-state devices (SSDs) have emerged as the next-generation storage devices and are now an alternative to HDDs [3]. SSDs have been adopted in a wide range of areas thanks to better I/O characteristics such as; a high I/O bandwidth, a short access latency, and a better energy efficiency. SSDs offer a new performance model, different from HDD's, with symmetric random/sequential read performance and asymmetric read/write performance.

Many state-of-the-art studies tend to show that because of application disparity in data processing and data volume explosion, I/O workloads increasingly adopt a random pattern [4] [5]. This applicative trend, combined with the changing performance model of storage devices and the ever-growing memory/storage gap, pushes to revisit the original Linux read-ahead prefetching mechanism.

There have been prior work that aim to improve the kernel prefetching mechanism. We enumerated mainly two improvement paths. The first one [6] consists in tuning the actual read-ahead mechanism to prefetch more aggressively in order to reduce the I/O latency relying on higher DRAM volumes. The second optimization path [7] attempts to improve the prediction accuracy by proposing new prefetching algorithms.

Prefetching mechanisms in state-of-the-art work attempt mainly to improve the prefetching efficiency on HDDs storage devices. There have been few work focusing on SSD performance model. While [8] proposes a new prefetching mechanism mainly focused on application start up time optimization, the authors of [9] designed a framework that uses an I/O monitoring tool to detect I/O patterns and periodically prefetches data according to application patterns. Both mechanisms were not designed to be fully integrated into the Linux kernel and used continuously during runtime.

In this work, we propose a novel and yet simple prefetching mechanism fully integrated to the Linux kernel, named Lynx. Lynx looks farther than just ahead for sequential I/Os. It relies on SSD performance model to prefetch sequentially and/or randomly data according to application I/O patterns. It continuously uses a simple mechanism to learn I/O patterns from applications and stores those patterns under the form of Markov chain state-machines. Lynx improves the Linux read-ahead efficiency by 50% on execution times and page cache misses on average for the tested workloads. Lynx is intended to complement or replace the Linux read-ahead mechanism.

The rest of the paper is organized as follows: Section II

Authorized licensed use limited to: University of Illinois. Downloaded on September 25,2023 at 01:08:14 UTC from IEEE Xplore. Restrictions apply.

presents a background and related work about data prefetching. Section III describes our solution and our implementation approach. The evaluation is presented in Section IV. We conclude the paper in Section V.

II. Background and Related Work

In this section, we present the background knowledge on read-ahead prefetching in the Linux kernel. Next, we describe related work aimed to improve the prefetching mechanism.

A. Linux read-ahead, a sequential prefetching mechanism

Several data-centric applications were designed based on the performance model of HDDs. In order to avoid mechanical latencies, applications tend to sequentially store the file data on storage media. Hence, many common file operations such as copy, scan, backup and recovery may lead to sequential I/O patterns. This motivated the use of sequential prefetching [10].

We distinguish three design options for sequential prefetching [11], [12]: Prefetch Always (PA), Prefetch On a Miss (PoM) and Prefetch On a Hit (PoH). PA fetches contiguous data for all read requests when possible. It may achieve a high hit rate (provided we have a lot of memory) but with a low efficiency since many prefetched data will never be used. In contrast, PoM only prefetches data on a miss, and thereby its miss rate is higher. PoH is popular in practice because it achieves both high hit rate and cache size economy.

In order to perform sequential prefetching, one needs to answer two main questions: when to prefetch, and how much data to prefetch. Traditional Linux read-ahead mechanism answers the first question by associating PoM and PoH. A synchronous read-ahead operation is launched in the case of a page miss and an asynchronous one in case of a hit. For the second question, the read-ahead prefetches data within a window size of 32 sequential pages from the storage device. When the miss rate increases for a specific file, the read-ahead window is narrowed as the prefetching is evaluated to be inefficient. The data volume to prefetch is then dynamic.

Linux read-ahead behavior can be driven from the application level. In fact, the Linux kernel offers a system call named *madvise* to inform the read-ahead mechanism about the future access pattern on a specific file. Application can also ask for an aggressive read-ahead or completely avoid the read-ahead mechanism in case of random I/Os for instance. For example, the $MADV\_RANDOM$ is used for advising the system of the random pattern to decrease or stop the read-ahead prefetching, while the $MADV\_SEQUENTIAL$ increases the prefetching.

B. Related works

1) prefetching for HDDs: Data prefetching on HDDs has been largely studied in the literature. The most popular prefetching mechanism is the sequential one. It is based on spatial locality of I/O workloads. There have been several studies aiming to improve this type of prefetching. Authors of [6] propose to perform aggressive prefetching which is made possible thanks to the growing main memory capacities. In the same way, researchers in [7] mainly proposed a dynamic prefetching mechanism that adjusts the amount of prefetched data during runtime based on the incoming I/O operations. These two studies propose efficient prefetching policies that aggressively load data into memory for HDDs.

In [13], for instance, the prediction accuracy is improved by controlling the prefetching from the application layer.

2) Prefetching for SSD: There have been some attempts to improve the prefetching mechanisms relying on SSD performance model. FAST [8] focused on shortening the application launch time and uses prefetching on SSDs for a quick start up of various applications. It takes advantage of the nearly identical block-level I/O pattern from run to run. This prefetching mechanism is specific to application starting phase and is implemented at the application level.

Flashy prefetching [9] is another interesting work. It relies on flash based storage devices (SSDs). The proposed mechanism consists of four stages: (1) the trace collection stage which monitors the I/Os on stored data, (2) the pattern recognition stage that aims to understand the access patterns for a series of I/O requests, (3) the block prefetching stage which periodically moves data from the storage device to the cache, and finally, (4) the feedback monitoring stage that controls the efficiency of prefetching by comparing them with the real application request.

Our approach is different from the above in many aspects; previous state-of-the-art work mainly focused on prefetching at the applicative layer while Lynx is integrated at the kernel level. In addition our Lynx performs continuous prefetching with no specific monitoring phase. The proposed solution is less than 200 lines of code to be inserted in the Linux kernel to replace or complement current Linux read-ahead prefetching mechanism.

III. Lynx prefetching mechanism: design and implementation

A. Motivation

Data-intensive applications have increasing needs in I/O performance. They are continuously optimized to take advantage of new evolution of memory hierarchy. Traditional Linux read-ahead prefetching mechanism design is based on two main assumptions: the sequential/random performance asymmetry of HDDs due to mechanical latencies, and the spatial locality of I/O workloads [10]. There are two main reasons that motivate adapting the Linux read-ahead; first, as described earlier, random reads are used more and more in current data-intensive applications [5] [4] such as data analytics, modern database management systems [14] and second, SSDs have similar latencies for both sequential and random I/O operations.

Lynx design aims to take advantage of the similar random and sequential read bandwidth on SSDs in order to satisfy all I/O patterns of data-centric applications. Of course, predicting fully random workloads is not feasible, we assume that applications expose algorithmic locality [1] that may result in random or sequential I/O patterns.

Authorized licensed use limited to: University of Illinois. Downloaded on September 25,2023 at 01:08:14 UTC from IEEE Xplore. Restrictions apply.

B. The use of Markov chains

Machine learning is about making computers modify or adapt their actions so that these actions get more accurate [15]. The accuracy is measured by how well the chosen action reflects the correct one (occurred event). Our approach needs to be able to predict the next action based on previous events and to weight actions according to their occurrence number. The more an action occurs, the more it has chances to occur in the future, and the higher should be its weight.

Machine learning algorithms are very studied in the literature. For instance, decision tree methods [15] construct a model of decisions based on learned action while Bayesian methods are those that explicitly apply Bayes' Theorem [15] for problems such as classification and regression. Those machine learning algorithms are more suitable for classification problems than prediction. We found other algorithms in the literature but that are resource intensive (CPU and memory) which does not suit our problem.

For the sake of Lynx prefetching design, we chose to use a simple Markov chain as a probabilistic machine to learn and predict I/O access patterns with minimum CPU and main memory consumption. Markov chains have been already adopted for prefetching in a different context such as in [16].

A Markov chain is a sequence S of random variables $S = X_1, X_2, ...X_n$ with the Markov property, namely that the probability of moving to next state depends only on the present state and not on the previous ones.

Markov chains are often described as state machines where each state models one random variable of the set S. The directed edges between states are labeled with the probabilities of going from one state to the other.

A Markov chain can also be described as a transition matrix T where lines are labeled with states at time t and columns are labeled with states at time $t + 1$. The matrix T holds the occurrence number of a transition between two states.

We used Markov chains in our context to solve the problem of predicting I/O patterns:

- Each file is represented by an independent state-machine.
- Each node/state of a given state-machine represents uniquely a file page.
- An edge/transition between two states represents successively requested (read) file pages.

By using Markov learning machine, transition probabilities between pages are computed continuously. This enables Lynx to start the prediction phase as soon as the learning starts.

C. Learning and predicting I/O patterns

We describe our approach on a data set DS containing N file pages Fp_1 ... Fp_N, where all file pages have similar sizes (more precisely one Linux page).

The current implementation of Lynx answers the two questions concerning the prefetching as follows:

- When to prefetch: as soon as there is an outgoing edge from the actual node representing the accessed file page.
- How much to prefetch: this parameter is configurable in Lynx; in the performance evaluation part, it is set to be one file page considering the latency of accessing SSD is small enough.

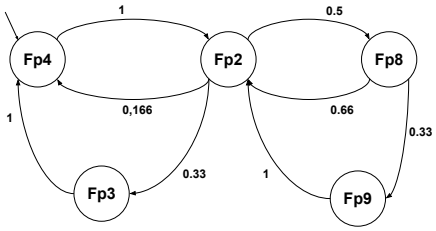

Fig. 1: Markov state-machine example

TABLE I: Transition table example

–	Fp_2	Fp_3	Fp_4	...	Fp_8	Fp_9
Fp_2	0	2	1	...	3	0
Fp_3	0	0	2	...	0	0
Fp_4	3	0	0	...	0	0
...				...		
Fp_8	2	0	0	...	0	1
Fp_9	1	0	0	...	0	0

Figure 1 shows an example of a Markov chain with its transition table in table I. This example is described further in the next section.

1) Learning phase: The learning phase captures incoming read operations and updates transitions between file pages to build up the state machine. For each transition, from Fp_i to Fp_j, Lynx continuously computes and updates the occurrence number and builds the transition table. The transition table T holds the number of occurrences $O_{i,j}$ for each transition from a file page Fp_i to the next file page Fp_j. The number of occurrences helps in calculating the probability of transition by dividing the number of occurrences on the total number of transitions.

2) Prediction phase: During the prediction phase, the transition table T is used to predict the next file page(s) accessed based on the current read operation to the file page Fp_i. The predicted access is given by retrieving, from T, all the possible transitions from the file page Fp_i and choosing the transition which has the highest probability. This probability is computed by dividing the number of occurrences of a transition on the total number of transitions on Fp_i.

D. Example

For a given file page Fp_i, we can have one or many possible transitions in case the file page has been accessed in different page file sequences. For instance, if we have the following sequence of read file page numbers (for a given file): {Fp_4, Fp_2, Fp_8, Fp_9, Fp_2, Fp_3, Fp_4, Fp_2 Fp_8, Fp_2, Fp_3, Fp_4, Fp_2, Fp_8, Fp_2, **Fp4**, (Fp_2), (Fp_8)}.The Fp_4 in bold is the last accessed page and parenthesized file pages are assumed to be the next read operations.

Based on the sequence of reads given above, we build the transition table in table I and the corresponding state machine in figure 1. For example, file page Fp_2 has three possible transitions with a number of occurrences equal to 2 for $O_{2,3}$,

Authorized licensed use limited to: University of Illinois. Downloaded on September 25,2023 at 01:08:14 UTC from IEEE Xplore. Restrictions apply.

1 for $O_{2,4}$ and 3 for $O_{2,8}$. As a consequence, probabilities are equal to $\frac{2}{6} = 0.333$, $\frac{1}{6} = 0.166$ and $\frac{3}{6} = 0.5$, respectively. Fp_4 in bold is the last accessed page which represents the current state of the Markov chain. Fp_4 state has only one out-going edge to Fp_2, then the predicted next element is Fp_2. The Markov chain next state is then Fp_2. From Fp_2, we have three out-going edges, the most accurate is the one going to Fp_8. Thus, Fp_8 is chosen as the predicted next access.

E. Implementation

We implemented Lynx on the Linux kernel version 3.2. We integrated the following parts to the traditional read-ahead system: the learning mechanism, the prediction mechanism, a prediction accuracy control and also a user-space control mechanism by system call.

1) Learning mechanism: The learning mechanism needs to intercept all read operations. We focused, for our implementation of the first version of Lynx, on memory file mapping operations. In fact, there are mainly two ways to access files: through file operation primitives, or through file memory mapping. File memory mapping is increasingly used by developers. In this case, Linux splits the file into page-sized chunks and maps them into virtual memory pages so that they can be made available in a process address space through load/store semantic.

In order to store information about the access pattern, we need to allocate a data structure which can be accessed by the prediction mechanism to prefetch the estimated future page.

We used a data structure for transition table named $transition\_table$ for each file concerned by the prediction process. We also declared a variable $last\_access$ on each processed file to hold the last access to the corresponding file. Incoming file I/O accesses to the file mapping are intercepted in the $filemap\_fault()$ routine within the $/mm/filemap.c$ kernel source file. The new incoming page access is associated with the last access (information stored in $last\_access$) to update the transition table. If the transition exists, the number of occurrences is updated, otherwise, a new transition is created and the number of occurrences initialized to one. In order to limit the transition table size, we limited the number of transition per file page. This limit can be configured by the user and was set to 20 in the experimental part.

Lynx learning mechanism can be adapted in the future to prefetch data for the file operation primitives, by intercepting the incoming page access into the files in the $vfs\_read()$ routine within $fs/readwrite.c$.

2) Prediction mechanism: When processing a new file read request, the $filemap\_fault()$ starts by trying to find the asked data in the page cache. If the requested file page is found (a minor page miss), the routine launches an asynchronous read-ahead to the storage device to prepare the data for the predicted next I/O, otherwise (if data is not available in the cache, a major page miss) a synchronous read-ahead is launched.

With the traditional read-ahead policy, synchronous / asynchronous read-ahead is not performed when the I/O access pattern is random. We adapted both synchronous and asynchronous read-ahead to take into account all I/O access patterns, random but also sequential according to the prediction mechanism. To do so, we created a new kernel routine named $\_do\_page\_cache\_lynx\_readahead()$ in $mm/readahead.c$ kernel source file, this routine predicts the next access on a specific file and prefetches the corresponding data to the page cache. Then, we modified the two kernel routines $do\_sync\_mmap\_readahead()$ and $do\_async\_mmap\_readahead()$ in $mm/filemap.c$ source file to call our newly created routine.

3) Controlling the prediction accuracy: We integrated a simple control of the prediction accuracy which checks continuously the efficiency of the prediction mechanism. It counts the number of misses (major page faults) on each accessed file. If the miss number for a file exceeds a predefined threshold, our mechanism considers that the corresponding state machine does not model well the current and future patterns. Then, the state-machine is simply reinitialized and will be populated again with the current I/Os to the file.

4) User space control mechanism: We needed a lightweight mechanism to control Lynx from the user space. To do so, the most convenient way we found is to modify the system call $madvise()$ source code (in $mm/readahead.c$). We needed to update one system flag to launch Lynx rather than the traditional read-ahead mechanism. From the existing advice argument flags, $MADV\_RANDOM$ is the one that reduces the use of read-ahead because of a random I/O pattern. So we updated $madvise()$ so that user-space applications can choose to use the proposed mechanism by using the $MADV\_RANDOM$ hint as argument. As the advise flag can be misleading, we recall that Lynx considers both random and sequential patterns, it can learn whatever I/O pattern provided there is an algorithmic locality.

IV. Experimental validation

A. Experimental methodology

In this section, we present an evaluation of Lynx in order to analyze the resulting performance in the light of its design goal: make good prediction on future I/O operations by exploiting the new performance model of SSDs. Our evaluation uses the following metrics:

- Total execution time: By measuring the total execution time a process spend to execute a given test, we evaluate the performance speedup using Lynx. The lower the execution time, the better the performance.
- Major page-faults: they occur when the requested page is not found in the page cache, this induces a physical read operation on the corresponding storage device.The lower this metric, the better the performance of Lynx.
- System time: The system time measures the time a process passes executing at the kernel level. The system time measure allows to evaluate the overhead of the Lynx mechanism. The lower the value is, the lower the overheads of Lynx, the better it is.
- Memory overhead: The designed read-ahead mechanism allocates memory to build the state machine, needed to

Authorized licensed use limited to: University of Illinois. Downloaded on September 25,2023 at 01:08:14 UTC from IEEE Xplore. Restrictions apply.

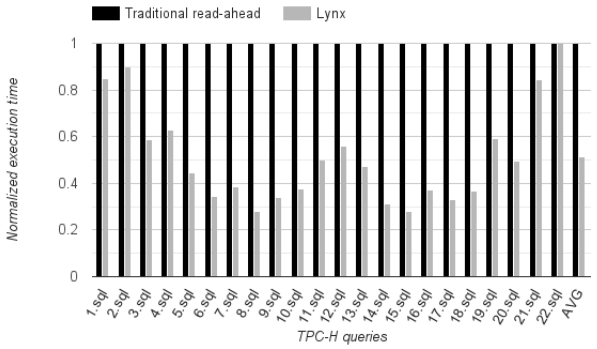

Fig. 2: The total execution time of TPC-H queries

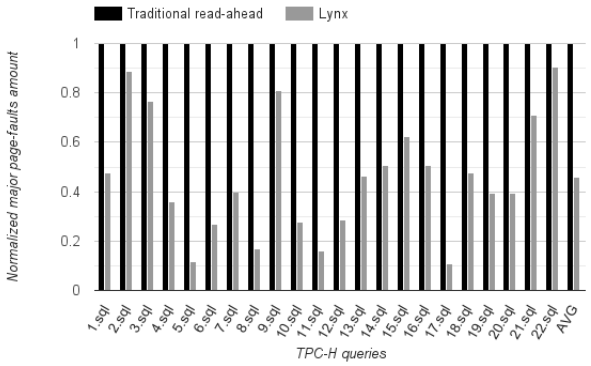

Fig. 3: Reduction of major page-faults for TPC-H queries

do the prediction. Thus, we need to measure the memory footprint of the proposed mechanism. The lower the memory footprint the better the prefetching mechanism.

We evaluated these metrics for both prefetching policies; the traditional policy and Lynx. All experiments were performed on memory mapped fils.

B. Evaluated workloads

In this study, we focused on one of the most data intensive applications, that is database management system. We experimented using the TPC-H benchmark [17] which offers a decisional database and 22 ad hoc queries. We created the database using an industrial high performance DBMS that is KoDe Server [18] and populated it with 10GB of data.

To avoid measuring transient regime for Lynx mechanism, we first ran a warm up phase for Lynx. This warm up consists in running the TPC-H benchmark once. After the warm up phase, we flushed the page cache before measuring the performance of the 22 tested queries. In order to isolate the effect of Lynx for each query, we also flushed the page cache before each query experimentation.

For experimentations, Lynx was configured to prefetch one page for future access on each miss/hit (PoM and PoH). We set the limit of transitions per state to 20. We also set the amount of major page-faults authorized per file before reinitializing the corresponding state machine to 100. All these values are configurable with Lynx.

C. Experimental environment

We ran the experimentations on a Linux virtual machine with kernel version 3.2. The virtual machine has 4GB of main memory and 64GB of storage space. The virtual machine is hosted on a Linux based machine equipped with 256GB Toshiba THNSNF SSD. This SSD presents a high bandwidth for both random and sequential reads (around 100 K IOPS for random reads and ∼500 MB/sec for sequential reads). We disabled the cache mechanism on the hypervisor kernel.

D. Results and Discussions

1) TPC-H queries execution time: Figure 2 shows the execution time for each TPC-H query normalized to the traditional read-ahead mechanism and compared to Lynx. This figure shows that Lynx read-ahead reduced the execution time for all TPC-H queries except the 22.sql one. Indeed, the 22.sql request uses the same table as previous requests with a different pattern, so Lynx applies its prediction based on previous request occurrences which interferes with the actual pattern. This issue is subject to optimization in future works.

Lynx reduces the execution time of queries by 50% on average (from 103.7 to 55.3 seconds) and up to 70%.

The resulting TPC-H queries execution time enhancement is correlated with the number of major page faults and system time reduction as observed in next sections.

2) Major page faults: Figure 3 presents the amount of major page faults for each TPC-H query normalized to the traditional read-ahead mechanism and compared to Lynx. The figure shows that Lynx reduces considerably the number of major page-faults which is reduced by 54% on average (from a total of 80203 major page-faults obtained with the traditional readahead to 29094 with Lynx). The decrease of major page-faults amount proves the prediction accuracy of Lynx.

Figure 4 shows the relation between execution time speedup and major page-faults enhancement when using Lynx. The figure shows that the speedup of queries execution times is mainly due to the decrease of major page-faults.

3) TPC-H queries system time: Figure 5 gives the time spend by the CPU for executing kernel instructions during the execution of the queries. We measured this metrics by

Fig. 4: Execution time speedup vs major page faults decrease

Authorized licensed use limited to: University of Illinois. Downloaded on September 25,2023 at 01:08:14 UTC from IEEE Xplore. Restrictions apply.

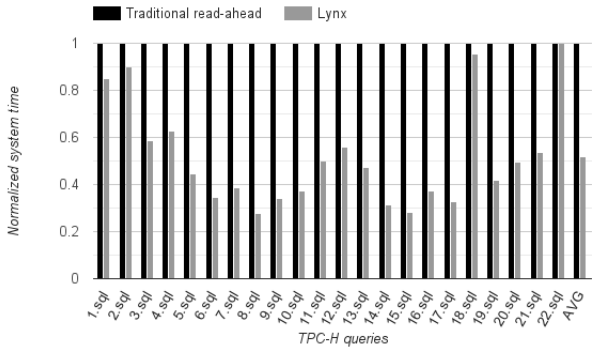

Fig. 5: The system time for TPC-H queries

the Linux $time$ command. It allows evaluating whereas Lynx incurs additional overheads for the Linux kernel.

The measured system time on the experimentation with Lynx was proved to be lower than the one with traditional read-ahead. This is due to the dynamic nature of data size prefetched in the traditional read-ahead while it is static (but configurable) in Lynx. In addition, by reducing the number of major page-faults, we also reduce the system time.

4) Memory overhead: The total main memory allocated to build the state machine for the TPC-H database files was measured to be equal to 94MB. This represents less than 1% of the TPC-H file data size (10GB), and less than 0.5% of the overall workload (18GB). This memory overhead scales with the number of file pages and the number of transitions between file pages in the state machine.

E. Limitations

While the achieved performance of Lynx is very good, there is still optimization room as some limitations need to be leveraged. One can cite the following:

- As each file is represented by a unique state machine, when two applications access the same file in an interleaved way, there will be only one pattern prefetched at the expense of the other. A way to leverage this issue would be to tag the state machine per process identifier or to consider previous file pages to detect the sequence.
- Another limitation is the size of meta data. Indeed, one needs to store the state machine in which each state represents a file page which can be very cumbersome for large data volume. One way to solve this issue is to consider blocks of pages per state rather than pages.

V. Conclusion

This paper describes a new prefetching mechanism named Lynx, to replace or complement the Linux read-ahead. Lynx design was motivated by two main observations, first I/O workloads for current data intensive applications are more and more random, and second, flash memory based devices have symmetric sequential and random performance with low latencies. Rather than focusing on spatial locality, Lynx focuses on algorithmic locality and prefetchs data whatever the pattern (sequential or random). Lynx relies on learning I/O patterns and representing them under the form of a Markov chain (state machine). Lynx was implemented and tested on a Linux kernel, and gave very good performance. We experimented it with TPC-H database benchmark on an industrial DBMS and our solution has shown 50% enhancement on the prefetching efficiency and the overall execution time as compared to traditional read-ahead. Lynx is also very light with about 200 lines of code. As discussed in the previous section, there is room for optimization that we plan to achieve. Lynx can also be extended for future storage systems with NVM.

Acknowledgment

This work is supported by the french National Association of Research and Technology (ANRT).

References

[1] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[2] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the design of a new linux readahead framework," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 75–84, 2008.

[3] J. Boukhobza, "Flashing in the Cloud: Shedding some Light on NAND Flash Memory Storage Systems," in *Data Intensive Storage Services for Cloud Environments*. IGI Global, 2013.

[4] A. Gulati, C. Kumar, and I. Ahmad, "Storage workload characterization and consolidation in virtualized environments," in *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.

[5] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, Sept 2008, pp. 119–128.

[6] S. Liang, S. Jiang, and X. Zhang, "Step: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers," in *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*. IEEE, 2007, pp. 64–64.

[7] B. S. Gill and L. A. D. Bathen, "Amp: Adaptive multi-stream prefetching in a shared cache." in *FAST*, vol. 7, no. 5, 2007, pp. 185–198.

[8] Y. Joo, J. Ryu, S. Park, and K. G. Shin, "Fast: Quick application launch on solid-state drives." in *FAST*, 2011, pp. 259–272.

[9] A. J. Uppal, R. C. Chiang, and H. H. Huang, "Flashy prefetching for high-performance flash drives," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–12.

[10] D. M. Huizinga and S. Desai, "Implementation of informed prefetching and caching in linux," in *Information Technology: Coding and Computing, 2000. Proceedings. International Conference on*, 2000, pp. 443–448.

[11] S. Bhatia, E. Varki, and A. Merchant, "Sequential prefetch cache sizing for maximal hit rate," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 89–98.

[12] M. Li, E. Varki, S. Bhatia, and A. Merchant, "Tap: Table-based prefetching for storage caches." in *FAST*, vol. 8, 2008, pp. 1–16.

[13] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 4, pp. 311–343, 1996.

[14] O. Rodeh, H. Helman, and D. Chambliss, "Visualizing block io workloads," *ACM Transactions on Storage (TOS)*, vol. 11, no. 2, p. 6, 2015.

[15] S. Marsland, *Machine learning: an algorithmic perspective*. CRC press, 2015.

[16] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *ACM SIGARCH Computer Architecture News*, vol. 25, no. 2. ACM, 1997, pp. 252–263.

[17] T. P. P. Council, "Tpc-h benchmark specification," *Published at http://www. tcp. org/hspec. html*, 2008.

[18] "Kode server." [Online]. Available: http://www.kodesoftware.com

Authorized licensed use limited to: University of Illinois. Downloaded on September 25,2023 at 01:08:14 UTC from IEEE Xplore. Restrictions apply.