IEICE TRANS. INF. & SYST., VOL.E92–D, NO.12 DECEMBER 2009

LETTER

RPP: Reference Pattern Based Kernel Prefetching Controller*

Hyo J. LEE[†], ***Nonmember***, **In Hwan DOH**[†], ***Member***, **Eunsam KIM**[†a)], ***and*** **Sam H. NOH**[†], ***Nonmembers***

SUMMARY Conventional kernel prefetching schemes have focused on taking advantage of sequential access patterns that are easy to detect. However, it is observed that, on random and even sequential references, they may cause performance degradation due to inaccurate pattern prediction and overshooting. To address these problems, we propose a novel approach to work with existing kernel prefetching schemes, called Reference Pattern based kernel Prefetching (RPP). The RPP can reduce negative effects of existing schemes by identifying one more reference pattern, i.e., looping, in addition to random and sequential patterns and delaying starting prefetching until patterns are confirmed to be sequential or looping.

key words: *kernel prefetching, reference pattern, read-ahead, overshooting*

1. Introduction

Prefetching is a technique to reduce disk access time by reading blocks contiguously stored together. Many research results have shown that prefetching can play a vital role in performance improvement if used properly. However, unwise prefetching may degrade the performance significantly. The blocks that are prefetched but not accessed later incur two kinds of penalties on the system. One is unnecessary I/O operations. The penalty for such I/O operations varies according to several factors such as how busy the disk is or where in disk requested blocks reside. The other is cache contamination. When prefetched blocks are uploaded to cache, they have to evict existing blocks. If the evicted block is the one to be accessed later, it must be re-fetched soon. If the prefetched block is never used, the cost will be greater.

To increase the prediction accuracy, previous kernel prefetching techniques have generally focused on taking advantage of sequential access patterns that are relatively easy to detect and quite effective [1]–[3]. However, it is observed that there are two major problems of these kernel prefetching techniques. One is that they may cause considerable performance degradation on random access patterns. To reduce this problem, most of them increase the number of blocks to be prefetched slowly until they become confident that access patterns are sequential [4]. Nevertheless, such prefetching schemes may make the system suffer because they always start prefetching even a small number of blocks by considering all access patterns as sequential initially. The other problem is that conventional schemes prefetch blocks too aggressively once workloads are recognized as sequential patterns [1], [4]. If the workloads do not have the sequences that are not long enough than expected, it may fetch too many blocks and eventually contaminate the cache space, which is called overshooting in this paper.

To address these problems, in this paper we adopt a novel approach to extend existing kernel prefetching schemes, which is called Reference Pattern based kernel Prefetching (RPP). The RPP can further improve the system performance by reducing negative effects of the conventional schemes. We add two important features to previous kernel prefetching schemes. First, we identify one more reference pattern, i.e., looping, to the existing patterns, i.e., sequential and random, which is the classification proposed for the caching scheme in UBM [7]. Thus, RPP can not only recognizes sequential patterns but also exploits repetitive occurrences of such sequential patterns. This can prevent repeated overshooting caused by aggressive prefetching. Second, unlike the conventional schemes, the reference pattern of each file is initially classified as random in RPP. This can eliminate the unnecessary prefetching of initial blocks of each file because we do not start prefetching until the pattern is confirmed to be sequential or looping. By extensive simulations, we show that RPP can eliminate negative effects of conventional kernel prefetching schemes on random and looping access patterns.

The remainder of the paper is organized as follows. Section 2 describes related work. In Sect. 3, we describe how RPP works. In Sect. 4, we present the experimental environment and the results. Finally, we conclude this paper in Sect. 5.

2. Related Work

To improve prediction accuracy of prefetching, there have been many attempts to obtain hints directly from applications [5], [6]. However, this kind of prefetching schemes have not been widely adopted by kernels due to their complexity and poor compatibility. Thus, most of kernel prefetching schemes have attempted to detect sequential reference patterns, which is effective but easy to detect. The linux kernel uses a prefetching scheme called read-ahead. The read-ahead schemes detect sequential pattern by keeping a read-ahead window. SARC partitions the cache space for the random and sequential references [2] and adapts prefetching degree and partition size according

Manuscript received July 17, 2009.

[†]The authors are with the Hongik University, Korea.

*This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MOST) (No.R0A-2007-000-20071-0).

a) E-mail: eskim@hongik.ac.kr

DOI: 10.1587/transinf.E92.D.2512

Copyright © 2009 The Institute of Electronics, Information and Communication Engineers

to the marginal utility. AMP is another kernel prefetching scheme to exploit sequential references, adapting the depth of prefetching depending on results of the previous prefetching effect [1]. Our work extends these existing schemes by decreasing negative effects of prefetching, rather than designs a new prefetching scheme.

On the other hand, in UBM (Unified Buffer Management) cache management scheme, an attempt have been made to exploit more elaborate access patterns, mimicking the Belady's optimal cache replacement policy. References are classified into three patterns, namely, sequential, looping, and random [7]. Even though prefetching techniques are highly related to cache management techniques, these kinds of patterns have not been exploited for kernel prefetching techniques. In this paper, we exploit the same patterns to further improve the performance of prefetching.

3. RPP Controller

In this section, we explain how RPP controller detects reference patterns and exploits the characteristics of the detected references.

3.1 Reference Patterns

Most previous kernel prefetching schemes have classified reference patterns as either sequential or random. In this paper, we add one more pattern, i.e., looping for effective prefetching. Note that sequential references are the sequence of consecutively accessed blocks that occur only once while looping references are sequential references that repeat during regular periods. Other references that are neither sequential nor looping are random.

3.2 Pattern Detection

We explain how references are detected. The pattern detector classifies the pattern for each file, maintaining a fileID, start block number, end block number, most recently accessed block number, last accessed time, and current detected pattern for each file. For a new file, the pattern detector adds an item for that file and marks the block number currently accessed. Note that the reference pattern of the file is initially classified as random.

On every access to each block, the pattern detector checks whether the block belongs to any file that have been accessed at least once before and is adjacent to the block accessed last in the file. If so, the pattern detector extends the corresponding sequence up to this block. When the length of the sequence exceeds a specific threshold, the pattern is classified as sequential. On the other hand, if the block turns out to be the start block of the sequence, the pattern classified as looping. If the block is neither sequential nor the start of a loop, the pattern remains random.

3.3 Additional Features

To reduce the negative effects of conventional kernel prefetching schemes, RPP add two important features to them. First, RPP delays triggering prefetching until the reference is determined as sequential or looping. In other words, it classifies every file as random when the file is accessed for the first time. Nevertheless, the decline in benefit from prefetching is not large because the number of blocks that are not prefetched is small and such small number of adjacent blocks can be also prefetched on the disk cache.

Second, RPP can avoid the overshooting of prefetching that are triggered by sequential references. Table 1 shows portions of references for typical workloads. We have observed from this table that sequential references form the majority of references and a major portion are looping references among sequential references. Thus, it is clear that we can reduce the overshooting of prefetching by reading ahead blocks only up until the end of the loop.

Table 1 Portion of patterns in access of various applications.

Trace	Sequential	Random	% of seq	% of loop
cscope	733457	385704	66	84
viewperf	300522	2601	99	58
tpc-h	641915	12827080	5	66
tpc-r	642782	8772745	7	67
multi2	1043321	537550	66	75
multi3	2546637	14024592	15	72

4. Experimental Evaluation

To evaluate the effectiveness of RPP, we have performed trace driven simulations.

4.1 Simulation Setup

The simulator used in our paper is based on Accusim [4] that faithfully implements the Linux kernel I/O clustering. We implemented RPP controller that manages kernel prefetching according to the type of detected references. As Accusim already supports the linux read-ahead prefetching, we added one more prefetching policy, i.e., AMP [1] for performance comparison. The Accusim simulator provides the traces pertaining to buffer cache management and prefetching [4], [9]. Note that LRU is used as a cache replacement policy.

RPP is a controller that can complement various kinds of kernel prefetching schemes. In this paper, we employ the Linux read-ahead [4] and the AMP scheme [1] as main kernel prefetching schemes, which are two different forms of kernel prefetching. Read-ahead is an aggressive prefetching scheme. Once it speculates that references are sequential, it starts to double up the number of blocks to be prefetched. On the other hand, AMP is rather a conservative approach to minimize mis-prefetching, dynamically adapting the the number of blocks to be prefetched and the trigger time of prefetching to optimize performance.

4.2 Simulation Results

In this section, five prefetching schemes are compared; no

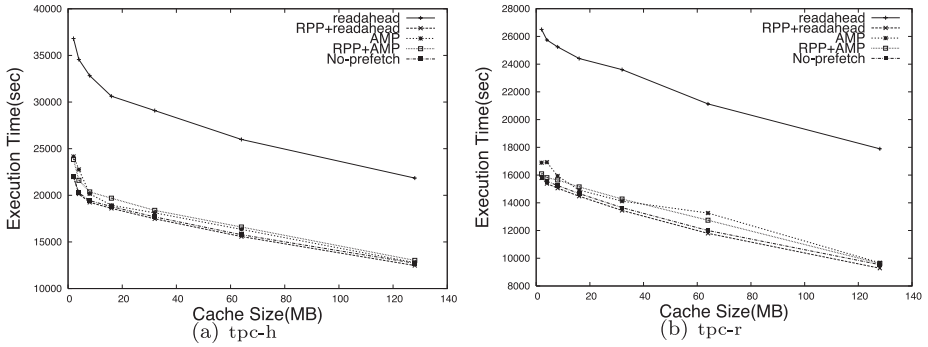

Fig. 1 Random.

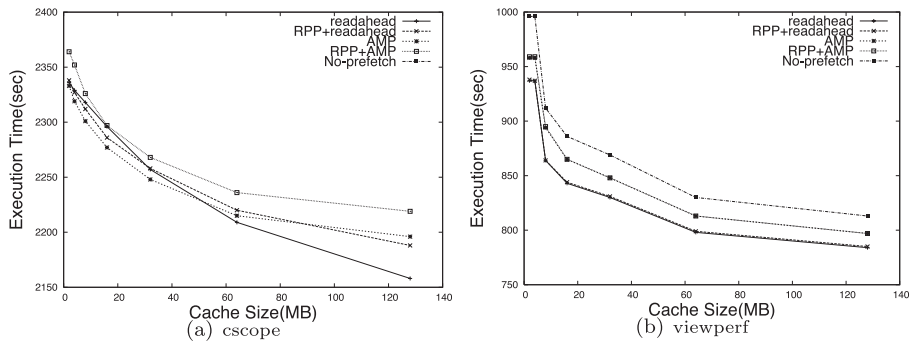

Fig. 2 Sequential.

prefetch (denoted as 'No-prefetch' in figures), the original Linux read-ahead scheme ('readahead'), the AMP scheme ('AMP'), RPP employing read-ahead as main scheme ('RPP+readahead'), and RPP employing AMP as main scheme ('RPP+AMP').

4.2.1 Effect on Random References

Figure 1 shows that, by reducing negative effects of existing kernel prefetching schemes, RPP can improve the prefetching performance in terms of execution time for random workloads. In this section, we employed *tpc-h* and *tpc-r* traces that comprise mainly of random accesses with only 5-7% sequential patterns.

We can see from Fig. 1 that read-ahead scheme performs worse than 'No-prefetch' by 67.3% and 71.7% on *tpc-h* and *tpc-r*, respectively. This implies that aggressive prefetching on random references degrades prefetching performance significantly. However, 'RPP+readahead' performs slightly better than 'No-prefetch'. This is because RPP decreases the negative effect of prefetching by postponing triggering prefetching until the pattern is confirmed to be sequential or looping.

It can be seen that RPP also improve the performance of AMP even though the improvement percentage in AMP is lower than that in read-ahead because AMP already takes a conservative approach for prefetching.

4.2.2 Effect on Sequential References

Figure 2 shows that, even on *Cscope* and *viewperf* that are typical sequential workloads, RPP can still performs better than both read-ahead and AMP even though improvement percentages become small.

As we can see from Fig. 2, prefetching schemes perform best on sequential workloads by taking full advantage of the sequentiality of patterns. In fact, RPP has a disadvantage on sequential patterns compared to conventional kernel prefetching schemes because it delays prefetching until the reference pattern is clearly recognized. Nevertheless, Fig. 2 shows that RPP achieved better performance than others. The reason for this is that it can avoid the overshooting of prefetching by exploiting looping patterns in addition to sequential ones.

4.2.3 Effect on Concurrent References

Figure 3 illustrates the simulation results of five schemes on concurrent execution of applications with different types of references. It is essential to examine the impact of concurrent execution of applications on overall performance because this may alter the reference characteristics of individual applications [3].

We employed the following two concurrent traces: *multi2* and *multi3*. *Multi2* represents the development environment consisting of concurrent executions of sequential references including *gcc*, *cscope*, and *viewperf*, while *multi3*

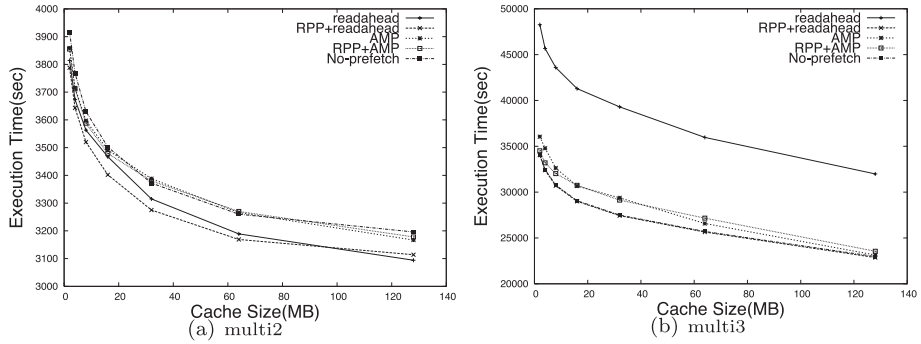

Fig. 3 Concurrent.

represents the server environment with a combination of sequential and random references including *glimpse* and *tpc-h*.

It can be seen from Fig. 3 that the performance trends of five schemes on *multi2* and *multi3* are similar to those of *cscope* and *tpc-h*, respectively, although their overall performances are worse due to relatively heavier concurrent workloads. This implies that RPP can work well on concurrent execution of applications with different access patterns.

5. Conclusions

In this paper, we proposed a new kernel prefetching approach to complement existing schemes, called RPP controller. The goal of RPP was to further improve the system performance by addressing the problems of existing kernel prefetching schemes. To do so, we first identified looping reference patterns additionally to prevent repeated overshooting caused by aggressive prefetching. Second, we did not trigger prefetching until the pattern is determined to be sequential or looping. Finally, we showed from extensive simulations that RPP could reduce negative effects of conventional kernel prefetching schemes on both random and looping access patterns.

References

[1] B.S. Gill and L.A.D. Bathen, "Amp: Adaptive multi-stream prefetching in a shared cache," FAST '07: Proc. 5th USENIX Conference on File and Storage Technologies, p.26, USENIX Association, Berkeley, CA, USA, 2007.

[2] B.S. Gill and D.S. Modha, "Sarc: Sequential prefetching in adaptive replacement cache," ATEC '05: Proc. Annual Conference on USENIX Annual Technical Conference, p.33, USENIX Association, Berkeley, CA, USA, 2005.

[3] C. Li, K. Shen, and A.E. Papathanasiou, "Competitive prefetching for concurrent sequential i/o," SIGOPS Oper. Syst. Rev., vol.41, no.3, pp.189–202, 2007.

[4] A.R. Butt, C. Gniady, and Y.C. Hu, "The performance impact of kernel prefetching on buffer cache replacement algorithms," SIGMETRICS Perform. Eval. Rev., vol.33, no.1, pp.157–168, 2005.

[5] P. Cao, E.W. Felten, A.R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," SIGMETRICS '95/PERFORMANCE '95: Proc. 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, pp.188–197, ACM, New York, NY, USA, 1995.

[6] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," Tech. Rep., Pittsburgh, PA, USA, 1995.

[7] J.M. Kim, J. Choi, J. Kim, S.H. Noh, S.L. Min, Y. Cho, and C.S. Kim, "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references," OSDI'00: Proc. 4th Conference on Symposium on Operating System Design & Implementation, p.9, USENIX Association, Berkeley, CA, USA, 2000.

[8] C. Gniady, A.R. Butt, and Y.C. Hu, "Program-counter-based pattern classification in buffer caching," OSDI'04: Proc. 6th Conference on Symposium on Opearting Systems Design & Implementation, p.27, USENIX Association, Berkeley, CA, USA, 2004.

[9] D. Lee, J. Choi, J.H. Kim, S.H. Noh, S.L. Min, Y. Cho, and C.S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," IEEE Trans. Comput., vol.50, no.12, pp.1352–1361, IEEE Computer Society, 2001.