PIBUS: A Network Memory-based Peer-to-Peer IO Buffering Service*

Yiming Zhang, Dongsheng Li, Rui Chu, Nong Xiao, Xicheng Lu

National Laboratory for Parallel and Distributed Processing, NUDT,
Changsha 410073, Hunan, China
ymzhang@nudt.edu.cn

Abstract. This paper proposes a network memory-based P2P IO BUffering Service (PIBUS), which buffers blocks for IO-intensive applications in P2P network memory like a 2-level disk cache. PIBUS reduces the IO overhead when local cache is missed due to speed advantage of network memory over disks, and improves hit ratio based on accurate classification of IO behaviors.

1 Introduction

Under the limit in size of disk cache, the performance of IO-intensive applications is determined by the hit ratio of disk cache and IO overhead when the cache is missed [1]. However, under the limit of local cache neither of them can improve significantly. Recently we proposed RAM-Grid [2], which proved the feasibility of using Internet memory for performance purpose. This motivates us to the idea of using Internet memory for IO-intensive applications. However, the assumption of unlimited network memory by RAM-Grid doesn't work here [3], thus indiscriminately buffering all blocks in network memory (like what RAM-Grid does) is impractical and prohibitive.

In this paper we propose a network memory-based P2P IO BUffering Service (PIBUS), in which each node views the free memory of other nodes in P2P overlays as a 2-level disk cache. PIBUS is built on top of Armada DHT [4], and includes two basic services, namely, caching service to reduce the latency of IO operation, and policy service to improve the hit ratio of local cache. Based on the speed advantage of network memory over local disks, PIBUS reduces the IO latency when the local cache is missed. Furthermore, PIBUS helps to manage local cache through fine-grained classification of IO behaviors, and then improves hit ratio effectively.

2 Block Caching Service

Due to lack of space, all detailed analysis is omitted here and can be found at [5].

* The work was partially supported by the National Natural Science Foundation of China under Grant No. 60673167 and No. 90412011, and the National Basic Research Program of China (973) under Grant No. 2005CB321801.

The blocks are sent to caching service when replaced out of local cache. Network IO is usually faster than local disk IO due to the speed advantage of network memory over local disks. And these blocks are fetched back through networks simultaneously with local disk file reads when re-accessed. There are 4 sets of nodes in PIBUS, namely, *Provider*, *Consumer*, *Potential Provider*, and *Potential Consumer*. 1) When a *Potential Provider* finds its local memory utilization exceeds some threshold, it turns into a *Potential Consumer*, and a *Cancel-Publication* and a *Make-Subscription* occur. 2) When a *Potential Consumer* finds its own memory utilization is less than some certain threshold, it becomes a *Potential Provider*, and a *Cancel-Subscription* and a *Make-Publication* occur. 3) When a *Potential Consumer* starts its IO-intensive applications and begins buffering its obsolete blocks in subscribed caching services, it becomes a *Consumer*. 4) When a *Potential Provider* begins its caching service, it becomes a *Provider* and a *Cancel-Publication* occurs. 5) When a *Consumer* ends its IO-intensive applications it becomes a *Potential Consumer*. 6) When a *Provider*'s caching service is no longer used, it becomes a *Potential Provider* and a *Make-Publication* occurs. 7) When a *Provider* finds its utilization exceeds some threshold, it handovers its buffered blocks to others and turns into a *Potential Consumer.*

3 Replacement Policy Service

Similarly to other policies, In PIBUS the *Consumer* records accesses that hit its local cache and makes a coarse-grained preliminary classification as *Unknown*, *One-off*, and, *Repeating.* These three patterns have their counterparts in recent proposals [1]. However, due to lack of records of each block's access history, in practice a large fraction of so-called *Repeating* blocks is not really accessed periodically.

To account for this imprecision, policy service takes an accurate record of the accesses to blocks buffered in its memory and further classifies the *Repeating* pattern as *Regular-Looping*, *Single-Clustered*, and *Multi-Clustered*, as shown in figure 1.

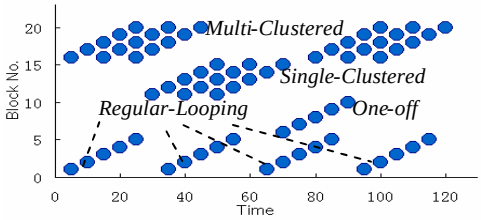

Fig. 1. Examples of IO patterns.

As shown in figure 2, each *Consumer* maintains a *File* table, in which each entry represents the IO record of a file, including the file ID, the count of *One-off* accesses, the count of *Repeating* accesses, the current IO pattern, and the period of *Repeating* accesses. For each file at the *Consumer*, the *Provider* maintains a *Block* table, in which each entry records the block number, the current access pattern, $Period_1$, $Period_2$ and the last access time. $Period_1$ is used to record the period of *Regular-Looping*

accesses, or the intra-cluster period of *Single-Clustered* and *Multi-Clustered* accesses; $Period_2$ is used to record the inter-cluster period of *Multi-Clustered* accesses.

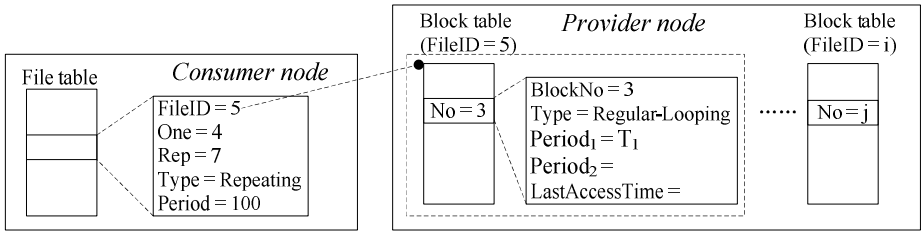

Fig. 2. Data structures of *policy service*.

When a block is accessed we first look up the block table. If the block is not in the table the *One* count is increased; otherwise the *Rep* count is increased and the *One* count is decreased. If the block has been accessed it will be further identified by the *Provider*. Otherwise the block takes the current pattern of its file as its pattern. Files are classified based on *Rep*, *One* and *Threshold.* If the *Rep* count is greater than the *One* count, the file is classified as *Repeating.* Otherwise the file is classified as *One-off* if *One* is greater than *Threshold*, or *Unknown* if *One* is less than *Threshold*.

```
FurtherIdentify (Block, LastTime, NoAccess)
01 if (NoAccess == 0) { // no access occurs.
02   deltaTime = currunt_Time -
       Block->LastAccessTime;}
03   switch (Block->Type) {
04   case Regular-Looping: { // n > 1
05     if (deltaTime > n * Block->Period1) {
06       Block->Type=Single-Clustered;
07       Block->Period1= deltaTime; } }
08   case Single-Clustered: {
09     Block->Period1= deltaTime; }
10   case Multi-Clustered: { //inter
11     if (deltaTime > n*Block->Period1) {
12       Block->Period2=deltaTime;}}}
13 else { // some accesses to this block.
14   curPeriod = LastTime -
       Block->LastAccessTime;
15   curPeriod = curPeriod / NoAccess;}
16   switch (Block->Type) {
17   case Regular-Looping: {
18     if (curPeriod < n*Block->Period1) {//n > 1
19       Block->Period1=expAvr(Block->Period1,
           curPeriod);
20     }else { // turn into Multi-Clustered pattern.
21       Block->Type= Multi-Clustered;
22       Block->Period2 = curPeriod; } }
23   case Single-Clustered: {
24     if (curPeriod < n*Block->Period1) {
25       Block->Period1=expAvr(Block->Period1,
           curPeriod);
26     }else { // turn into Multi-Clustered pattern.
27       Block->Type= Multi-Clustered;
28       Block->Period2= curPeriod; } }
29   case Multi-Clustered: {
30     if (curPeriod < n*Block->Period1) { //intra
31       Block->Period1=expAvr(Block->Period1,
           curPeriod);
32     }else{Block->Period2=curPeriod;}}}//inter
```

Fig. 3. Further identification algorithm.

As shown in figure 3, policy service further identifies *Repeating* blocks. If the access intervals of a *Repeating* block differ within a range, it is classified as *Regular-Looping* and uses $Period_1$ to record the period. If a block has been classified as *Regular-Looping* but won't be accessed any more, it is classified as *Single-Clustered* and uses $Period_1$ to record the intra-cluster period. Otherwise it is classified as *Multi-Clustered*, and uses $Period_1$ for intra-cluster period and $Period_2$ for inter-cluster one.

Blocks are placed in the corresponding subcache. 1) The *Unknown* blocks in *Unknown* subcache are managed by LRU. 2) The *One-off* blocks are discarded because they won't be accessed any more. 3) All the *Repeating* blocks are buffered in *Repeating* subcache. The blocks are ranked according to the estimated next access time (*ENAT*) and the last one will be replaced out. *ENAT* is estimated as follows: *Regular-Looping* blocks use (*Block->Period*$_1$ - *LastAccessTime*). *Single-Clustered* blocks use (*Block->Period*$_1$ - *LastAccessTime*). *Multi-Clustered* blocks use (*Block->Period*$_1$ - *LastAccessTime*) if the interval between the current time and *LastAccessTime* is less than $n \times$ *Block->Period*$_1$; otherwise use (*Block –>Period*$_2$ - *LastAccessTime*).

4 Evaluation

We trace an actual meteorological application by modifying the Linux kernel to record all the IO operations, including the access time, file ID and offset in the file. We have built a simulation environment for RAM-Grid in our previous work [2]. Based on this, we simulate PIBUS by using Armada for resource discovery. The underlying network is composed of 1,000 nodes with different memory capacity, CPU frequency, bandwidth, etc. The influence of PIBUS on system IO performance is shown in table 1.

TABLE 1. PIBUS VS. LRU under different local cache sizes: completion time/hit ratio(%).

	32 MB	64 MB	128 MB	256 MB	512 MB	1024 MB
LRU	56.5s/5	53.2s/10	51.4s/13	50.8s/15	41.2s/33	10.1s/83
PIBUS	29.2s/24	25.5s/28	22.3s/35	14.5s/68	11.6s/79	10.1s/83

5 Conclusion

PIBUS uses *Armada* protocol as its underlying resource discovery infrastructure and is composed of *caching service* and *policy service*. Trace driven simulation shows that PIBUS improves the performance of IO-intensive applications efficiently.

References

1. C. Gniady, A.R. Butt, and Y.C. Hu, Program-Counter-Based Pattern Classification in Buffer Caching, in: OSDI 2004.
2. R. Chu, N. Xiao, Y. Zhuang, Y. Liu, and X. Lu, A Distributed Paging RAM Grid System for Wide-area Memory Sharing, in: IPDPS 2006.
3. S. Ghemawat, H. Gobioff, and S.T. Leung, The Google File System, In: SOSP 2003,
4. D. Li, J. Cao, X. Lu, K.C.C. Chan, B. Wang, J. Su, H. Leong, A.T.S. Chan, Delay-Bounded Range Queries in DHT-based Peer-to-Peer Systems, in: ICDCS 2006.
5. http://www.kylinx.com/Papers/PIBUS_Networking_1208.pdf