Program-Counter-Based Pattern Classification in Buffer Caching

Chris Gniady Ali R. Butt Y. Charlie Hu
Purdue University
West Lafayette, IN 47907
{*gniady, butta, ychu*}*@purdue.edu*

Abstract

Program-counter-based (PC-based) prediction techniques have been shown to be highly effective and are widely used in computer architecture design. In this paper, we explore the opportunity and viability of applying PC-based prediction to operating systems design, in particular, to optimize buffer caching. We propose a Program-Counter-based Classification (PCC) technique for use in pattern-based buffer caching that allows the operating system to correlate the I/O operations with the program context in which they are issued via the program counters of the call instructions that trigger the I/O requests. This correlation allows the operating system to classify I/O access pattern on a per-PC basis which achieves significantly better accuracy than previous per-file or per-application classification techniques. PCC also performs classification more quickly as per-PC pattern just needs to be learned once. We evaluate PCC via trace-driven simulations and an implementation in Linux, and compare it to UBM, a state-of-the-art pattern-based buffer replacement scheme. The performance improvements are substantial: the hit ratio improves by as much as 29.3% (with an average of 13.8%), and the execution time is reduced by as much as 29.0% (with an average of 13.7%).

1 Introduction

One of the most effective optimization techniques in computer systems design is history-based prediction, based on the principle that most programs exhibit certain degrees of repetitive behavior. Such history-based prediction techniques include cache replacement and prefetching schemes for various caches inside a computer system such as hardware memory caches, TLBs, and buffer caches inside the operating system, as well as processor oriented optimizations such as branch prediction.

A key observation in the processor architecture is that program instructions (or their program counters) provide a highly effective means of recording the context of program behavior. Consequently, program-counter-based (PC-based) prediction techniques have been extensively studied and widely used in modern computer architectures. However, despite their tremendous success in architecture design, PC-based techniques have not been explored in operating systems design.

In this paper, we explore the opportunities and feasibility of PC-based techniques in operating systems design. In particular, we consider the application of PC-based prediction to the I/O management in operating systems. Since the main memory, like hardware caches, is just another level of the memory hierarchy, the I/O behavior of a program is expected to be similar to the data movement in the upper levels of the memory hierarchy in that there is a strong correlation between the I/O behavior and the program context in which the I/O operations are triggered. Furthermore, program counters are expected to remain as an effective means of recording such program context, similar to recording the context of data movement in the upper levels of memory hierarchy.

The specific PC-based prediction technique we propose, called Program-Counter-based Classification (PCC), identifies the access pattern among the blocks accessed by I/O operations triggered by a call instruction in the application. Such pattern classifications are then used by a pattern-based buffer cache to predict the access patterns of blocks accessed in the future by the same call instruction. A suitable replacement algorithm is then used to manage the accessed disk blocks belonging to each pattern as in previous pattern-based buffer cache schemes. Thus, PCC correlates the I/O access patterns observed by the operating system with the program context in which they are issued, i.e., via the PC of the call instruction that triggers the I/O requests. This correlation allows the operating system to classify block accesses on a *per-PC basis*, and distinguishes PCC from previous classification schemes in two fundamental aspects. First, if the same instruction is observed again for newly opened files, PCC can immediately predict the access pattern based on history. Second, PCC can differentiate multiple reference patterns in the same file if they are invoked by different instructions.

The design of PCC faces several challenges. First, retrieving the relevant program counter that is responsible for triggering an I/O operation can be tricky, as I/O operations in application programs are often abstracted into multiple, layered subroutines and different call sites may go through multiple wrappers before invoking some shared I/O system calls. Second, PCC requires a predefined set of access patterns and implements their detection algorithms. Third, the buffer cache needs to be partitioned into subcaches, one for each pattern, and the subcache sizes need to be dynamically adjusted according to the distribution of the blocks of different patterns.

This papers makes the following contributions.

- It is, to our knowledge, the first to apply a PC-based prediction in operating systems design;
- It presents the first technique that allows the operating system to correlate I/O operations with the program context in which they are triggered;
- It presents experimental results demonstrating that correlating I/O operations with the program context allows for a more accurate and adaptive prediction of I/O access patterns than previous classification schemes, and a pattern-based buffer caching scheme using PCC outperforms state-of-the-art recency/frequency-based schemes;
- It shows that exploiting the synergy between architecture and operating system techniques to solve problems of common characteristics, such as exploiting the memory hierarchy, is a promising research direction.

The rest of the paper is organized as follows. Section 2 briefly reviews the wide use of PC-based prediction techniques in computer architecture design. Section 3 motivates the pattern classification problem in buffer caching and discusses the design choices. Section 4 presents the PCC design and Section 5 presents the results of an experimental evaluation of PCC. Section 6 discusses additional related work in buffer caching. Finally, Section 7 concludes the paper.

2 PC-based techniques in architecture

History-based prediction techniques exploit the principle that most programs exhibit certain degrees of repetitive behavior. For example, subroutines in an application are called multiple times, and loops are written to process a large amount of data. The challenge in making an accurate prediction is to link the past behavior (event) to its future reoccurrence. In particular, predictors need the program context of past events so that future events about to occur in the same context can be identified. The more accurate context information the predictor has about the past and future events, the more accurate prediction it can make about future program behavior.

A key observation made in computer architecture is that a particular instruction usually performs a very unique task and seldom changes behavior, and thus program instructions provide a highly effective means of recording the context of program behavior. Since the instructions are uniquely described by their program counters (PCs) which specify the location of the instructions in memory, PCs offer a convenient way of recording the program context.

One of the earliest predictors to take advantage of the information provided by PCs is branch prediction [40]. In fact, branch prediction techniques have been so successful in eliminating latencies associated with branch resolution that they are implemented in every modern processor. The PC of the branch instruction uniquely identifies the branch in the program and is associated with a particular behavior, for example, to take or not to take the branch. Branch prediction techniques correlate the past behavior of a branch instruction and predict its future behavior upon encountering the same instruction.

The success in using the program counter in branch prediction was noticed and the PC information has been widely used in other predictor designs in computer architecture. Numerous PC-based predictors have been proposed to optimize energy [4, 35], cache management [21, 22], and memory prefetching [1, 6, 13, 19, 33, 38]. For example, PCs have been used to accurately predict the instruction behavior in the processor's pipeline which allows the hardware to apply power reduction techniques at the right time to minimize the impact on performance [4, 35]. In Last Touch Predictor [21, 22], PCs are used to predict which data will not be used by the processor again and free up the cache for storing or prefetching more relevant data. In PC-based prefetch predictors [1, 6, 13, 19, 33, 38], a set of memory addresses or patterns are linked to a particular PC and the next set of data is prefetched when that PC is encountered again.

PC-based techniques have also been used to improve processor performance by predicting instruction behavior in the processor pipeline [12, 36] for better utilization of resources with fewer conflicts, as well as to predict data movement in multiprocessors [21, 26] to reduce communication latencies in multiprocessor systems. Most recently, a PC-based technique was proposed to predict disk I/O activities for dynamic power management [15].

Despite their tremendous success in architecture design, PC-based techniques have not been explored in operating systems design. In this paper, we consider the opportunity and viability of PC-based prediction techniques in operating systems design. In particular, we consider the buffer cache management problem, which shares common characteristics with hardware cache management as they essentially deal with different levels of the memory hierarchy.

3 Pattern classification in buffer caching

In this section, we motivate the pattern classification problem in buffer caching and discuss various design options.

3.1 Motivation

One of the most important problems in improving file system performance is to design an effective block replacement scheme for the buffer cache. One of the oldest replacement schemes that is yet still widely used is the Least Recently Used (LRU) replacement policy [9]. The effectiveness of LRU comes from the simple yet powerful principle of locality: recently accessed blocks are likely to be accessed again in the near future. Numerous other block replacement schemes based on recency and/or frequency of accesses have been proposed [17, 18, 24, 27, 30, 37]. However, a main drawback of the LRU scheme and all other schemes based on recency and/or frequency of accesses is that they cannot exploit regularities in block accesses such as sequential and looping references [14, 39].

To overcome this drawback, pattern-based buffer replacement schemes [10, 11, 14, 20, 39] have been proposed to exploit the fact that different types of reference patterns usually have different optimal or best known replacement policies. A typical pattern-based buffer cache starts by identifying and classifying reference patterns among accessed disk blocks. It divides up the cache into subcaches, one for blocks belonging to each pattern. Based on the reference pattern, it then applies an optimal or best known replacement policy to each subcache so as to maximize the cache hit ratio for each subcache. In addition, a pattern-based buffer cache needs to dynamically adjust subcache sizes based on the distribution of different types of accesses with the goal of maximizing the overall cache hit ratio. Experimental results [10, 20] have shown that pattern-based schemes can achieve better hit ratios over pure recency/frequency schemes for a mixture of applications.

3.2 Design space

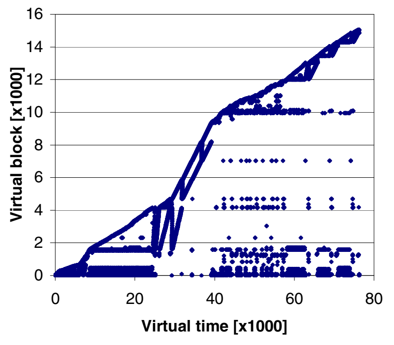
The design space for pattern classification centers around the granularity at which the classifications are being performed. In particular, the access patterns of an application can be classified on a per-application basis, a per-file basis, or a per-PC basis, i.e., for each program instruction that triggers any I/O operations.

Per-application classification In a per-application classification scheme such as DEAR [10], the pattern in accesses invoked by the same application is detected and classified. The patterns in DEAR include sequential, looping, temporally clustered, and probabilistic. The scheme periodically reevaluates and reclassifies the patterns and adapts the replacement policy according to the changing reference patterns in the application. However, as shown in [20] and later in this section, many applications access multiple files and exhibit multiple access patterns to different files or even to the same file. Thus the accuracy of per-application classification is limited by its application level granularity.

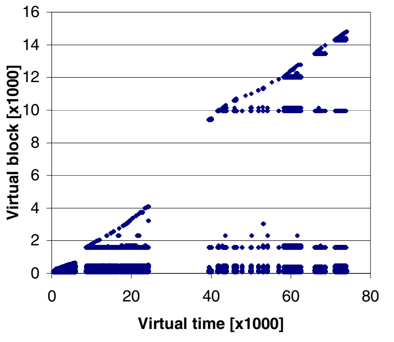
Per-file classification In a per-file classification scheme such as UBM [20], the access pattern in accesses to each file is dynamically classified. In other words, it distinguishes multiple concurrent patterns in the same application as long as they occur in different files. UBM classifies each file into one of the three possible access patterns: sequential, looping, or other. Each file can only have one classification at any given time, though it can be reclassified multiple times. A file receives a sequential classification when some predetermined number (threshold) of consecutive blocks is referenced. Once the file is classified as sequential, it can be reclassified as looping if the file is accessed again according to the sequence seen earlier. The file is classified as having the other reference type when the pattern is not sequential, or the sequence is shorter than the threshold. The Most Recently Used [29] replacement policy is used to manage the subcache for blocks that are accessed with a looping pattern, blocks with a sequential access pattern are not cached, and LRU is used to manage the subcache for blocks of the other reference type.

The classification on a per-file basis in UBM suffers from several drawbacks. First, the pattern detection has to be performed for each new file that the application accesses, resulting in high training overhead and delay in pattern classification when an application accesses a large number of files. Second, since the first iteration of a looping pattern is misclassified as sequential, pattern detection for every new file means such misclassification happens for every file with a looping pattern. Third, if an application performs accesses to a file with mixed access patterns, UBM will not be able to differentiate them. Lastly, the behavior of the threshold-based detection of sequential accesses is directly related to the file size, preventing proper classification of small files.

Per-PC classification In a per-PC classification such as PCC proposed in Section 4, the pattern in accesses in-

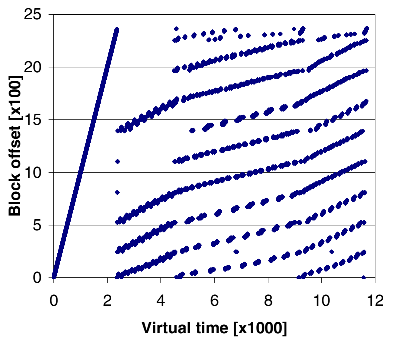


(a) All reference patterns

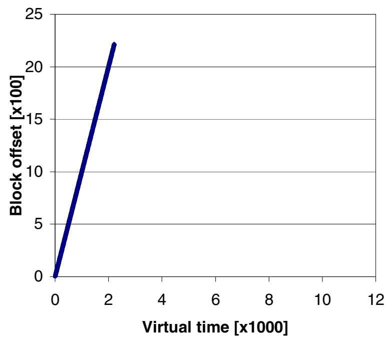


(b) References from reading header files by a single instruction

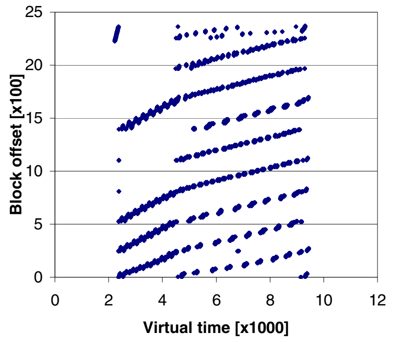
Figure 1: Reference patterns in *gcc*



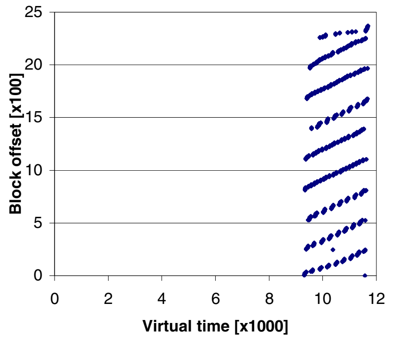
(a) Reference patterns in a single file



(b) Initial scan of the file by a single PC



(c) Processing of the file by multiple PCs



(d) Final references made by a single PC

Figure 2: Reference patterns in *tpc-h*

voked by a call instruction in the application executable is classified and correlated with future occurrences of the same call instruction (represented by its program counter). Compared to the per-application and per-file classification schemes, per-PC classification correlates the I/O behavior of a program with the program context in which the I/O operations are invoked and records the program context using appropriate PCs. Because of such correlations, per-PC classification is expected to be more accurate than classification schemes that do not exploit such correlations. We illustrate this potential advantage using two of the benchmarks studied in Section 5.

Figure 1(a) shows the space-time graphs of block references in *gcc* (details of which will be discussed in Section 5). The horizontal axis shows the virtual time which is incremented at every block access, and the vertical axis shows the corresponding block number at a given time. Figure 1(a) shows that there exists a mixture of sequential (a single slope) and multiple looping (repeated slopes) patterns in *gcc*. Figure 1(b) presents the reference pattern to blocks accessed by a single instruction in *gcc* responsible for accessing header files during the compilation. Accesses to header files correspond to 67% of the references in *gcc* of which 99% are reoccurring to header files already accessed once. The remaining 1% of the accesses are the first accesses to repeatedly accessed header files, or to header files that are accessed only once. Most importantly, all header files are accessed by the same single instruction. This observation suggests that the instruction triggering the I/O operations can be used to classify access patterns; the access pattern of the blocks accessed by it needs to be learned once and with high probability, the same pattern holds when it is used to access different files. In contrast, UBM needs to classify the access pattern for each header file, incurring a delay in the classification and, consequently, a missed opportunity in applying the best known replacement scheme.

Figure 2(a) shows the reference patterns to a single file in the *tpc-h* benchmark (details of which will be discussed in Section 5). The vertical axis shows the offset of the blocks in the accessed file. The reference pattern in Figure 2(a) shows a mixture of sequential accesses and looping accesses. To illustrate the use of PCs for pattern classification, we separate the patterns into three different components as shown in Figures 2(b)(c)(d). The accesses in Figure 2(b) are performed by a single PC which scans the entire file. The blocks read in Figure 2(b) are subsequently accessed again by multiple PCs as shown in Figure 2(c). Since the blocks accessed in Figure 2(b) are accessed again in Figure 2(c), the accessing PC in Figure 2(b) is classified as looping. Similarly, blocks in Figure 2(c) are accessed again in Figure 2(d), and therefore PCs in Figure 2(c) are also classified as looping. Finally, the blocks accessed by a single PC in Figure 2(d) are not accessed again and the accessing PC is classified as sequential.

PCC is able to separate multiple concurrent access patterns and make a proper prediction every time the file with multiple access patterns is referenced. In contrast, UBM classifies access patterns on a per-file basis, and therefore it will not be able to separate the sequential, looping, and other access patterns.

4 PCC Design

The key idea behind PCC design is that *there is a strong correlation between the program context from which I/O operations are invoked and the access pattern among the accessed blocks, and the call instruction that leads to the I/O operations provides an effective means of recording the program context*. Each instruction is uniquely described by its PC. Thus, once PCC has detected a reference pattern, it links the pattern to the PC of the I/O instruction that has performed the accesses. Such pattern classifications are then used by a pattern-based buffer cache to predict access patterns of blocks accessed in the future by the same call instruction.

4.1 Pattern classification

The main task of PCC is to classify the instructions (or their program counters) that invoke I/O operations into appropriate reference pattern categories. Once classified, the classification of the PC is used by the cache manager to manage future blocks accessed by that PC.

We define three basic reference pattern types:

Sequential references are sequences of distinct blocks that are never referenced again;

Looping references are sequential references occurring repeatedly with some interval;

Other references are references not detected as looping or sequential.

Figure 3 shows the two data structures used in PCC implementation. The PC hash table keeps track of how many blocks each PC accesses once (`Seq`) and how many are accessed more than once (`Loop`). The block hash table keeps records of M recently referenced blocks, each

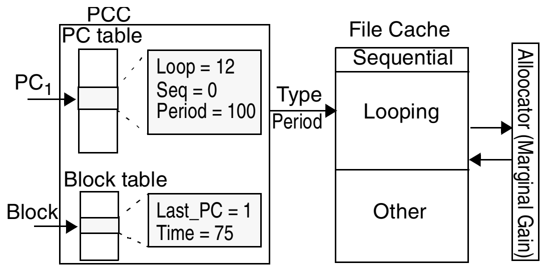


Figure 3: Data structures in PCC

containing the block address, the last accessing PC, and the access time. The choice of the M blocks is discussed in Section 4.2. The access time is simply a count of the number blocks accessed since the program started. PCC maintains the last accessing PC in the block table since a block may be accessed by different PCs.

To simplify the description, we first describe the PCC algorithm assuming all blocks accessed by each PC are monitored. The actual PCC uses a sampling technique to reduce the block hash table size and is discussed in Section 4.2. The pseudo code for PCC without sampling is shown in Figure 4. When a PC triggers an I/O to a block, PCC first looks up the block in the block table to retrieve the PC and the time of the last access to the block. The last accessing PC is used to retrieve the record of that last PC in the PC table. The record is updated by decreasing the `Seq` count and increasing the `Loop` count since the block is being accessed again. The exponential average of the period is then calculated based on the time difference recorded in the block entry. If the current PC differs from the last PC, PCC performs an additional lookup into the PC table to obtain the record of the current PC. At this time PCC classifies the reference pattern and returns both the type and the period from the PC entry.

PCC classifies accesses based on the `Loop` count, the `Seq` count, and a threshold variable. The threshold aids in the classification of sequential references made by a newly encountered PC. If there are fewer non-repeating (`Seq`) blocks than repeating (`Loop`) blocks, the PC is classified as looping, disregarding the threshold. Otherwise, the PC is classified as sequential if the `Seq` count is larger than or equal to the threshold, or other if the `Seq` count is smaller than the threshold. PCs that temporarily have comparable `Seq` and `Loop` counts and therefore do not fall clearly into sequential or looping categories are classified as other references. These PCs mostly have a combination of sequential references to some blocks and looping references to other blocks.

Similar to UBM, PCC will misclassify the first occurrence of a looping sequence as sequential, assuming it is longer than the threshold. However, once PCC assigns the looping pattern to a PC, the first references to any file by

```
PCC(PC, Block, currTime)
(1)  if ((currBlock = getBlockEntry(Block)) == NULL)
(2)       currBlock = NewBlockEntry(Block);
(3)  else { // the entry for last accessing PC must exist
(4)       currPC = getPCEntry(currBlock→Last_PC);
(5)       currPC→Seq- -;
(6)       currPC→Loop++;
(7)       currPC→Period = expAverage(currPC→Period,
(8)                 currTime - currBlock→Time);
(9)  }
(10) if (currBlock→Last_PC != PC)
(11)      currPC = getPCEntry(PC);
(12) if (currPC == NULL) {
(13)      currPC = NewPCEntry(PC);
(14)      currPC→Seq = 1;
(15)      currPC→Loop = currPC→Period = 0;
(16)      Type = "Other";
(17) } else {
(18)      currPC→Seq++;
(19)      if (currPC→Loop > currPC→Seq)
(20)           Type = "Looping";
(21)      else if (currPC→Seq >= Threshold)
(22)           Type = "Sequential";
(23)      else
(24)           Type = "Other";
(25)      Period = currPC→period;
(26) }
(27) currBlock→Time = currTime;
(29) currBlock→Last_PC = PC;
(30) return(Type, Period);
```

Figure 4: Pseudocode for PCC without sampling

the same PC will be labeled as looping right away.

We note the threshold value has different meaning and performance impact in PCC and UBM. In UBM, the threshold value is on a per-file basis and therefore files with sizes smaller than the threshold are not classified properly. In PCC, the threshold is set on a per-PC basis, and thus a PC that accesses enough small files will be properly classified.

Figure 5 shows some example reference patterns assuming that the threshold is set at three. When the application starts executing, PCC observes a set of sequential references by PC1. After three initial references are classified as other, the threshold is reached and PC1 is classified as sequential for the remaining references. When the sequence is accessed again, PCC reclassifies PC1 as looping, and future blocks referenced by PC1 are classified as looping. At that time, the loop period is calculated and recorded with PC1. In the meantime, PC2 is encountered and again classified after the initial three references as sequential. The classification is not changed for PC2 since no looping references are encountered. When PC3 accesses the same set of blocks that were accessed by PC1, PC3 is classified as sequential, since PC3 is observed for the first time. Note that the classification of PC1 remains

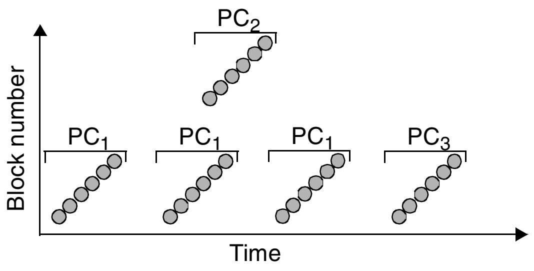


Figure 5: An example of reference patterns

the same, although the classification of the set of blocks it accessed has changed from looping to sequential.

4.2 PCC with block sampling

Maintaining the blocks in the block hash table is crucial to calculating the loop period. Keeping the information for every accessed block would be prohibitive for large data sets. Instead, PCC only keeps periodically sampled referenced blocks in the block hash table, and repeated accesses to the sampled blocks are used to calculate the loop period for each PC. Specifically, for each PC in the application, the block hash table maintains up to N blocks, each recorded at every T-th access by that PC. We discuss the choices of N and the sampling period T in Section 5.3.1. Note that sampling dictates which blocks will be inserted in the block table and thus used for calculating the average period for a PC. If the block in the block table is accessed again, the PC period calculation is performed as before. After the update, the block is discarded from the block table to free a slot for the next sample.

To further limit the block table size, when the block table is full, PCC uses the LRU policy to evict the least recently used PC from the PC table and the corresponding block table entries.

4.3 Obtaining signature PCs

Instead of obtaining a single PC of the function call from the application that invokes each I/O operation, PCC actually uses a *signature PC* which records the *call site* of the I/O operation by summing the sequence of PCs encountered in going through multiple levels of wrappers before reaching the actual system call. The wrapper functions are commonly used to simplify programming by abstracting the details of accessing a particular file structure. For example, in the call graph shown in Figure 6, Functions 2, 3, and 4 use the same PC in the wrapper function for I/O operations. Therefore, the PC that invokes the I/O system call within the wrapper cannot differentiate the behavior of different caller functions. To obtain a unique characterization of the accessing PC, PCC traverses multiple function stacks in the application. The PCs obtained during the stack frame traversal are summed

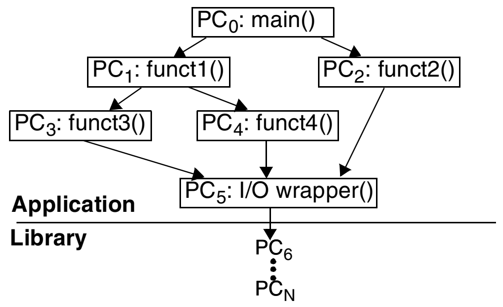


Figure 6: An example function call graph

together to obtain a unique identifier as the *signature PC* of the I/O operation. In the applications studied in Section 5, traversal of only two additional stack frames provided sufficient information to PCC. As the minimum number of stack frames needed to generate unique signature PCs varies from application to application, PCC always traverses all the function stacks in the application until reaching `main()`. We note that it is extremely unlikely that the signature PCs of different call sites will collide as each signature PC is 32-bit, while an application typically has up to a few hundred call sites. For simplicity, in the rest of the paper, we will refer to the signature PC of an I/O operation simply as its PC.

Compared to previous pattern classification schemes, a unique property of PCC is that the PCs that trigger I/O operations and their access patterns tend not to change in future invocations of the same application. Thus, PCC saves the per-PC pattern classifications in the PC table for potential future reuse. Similar to the block table, the size of the PC table can be easily managed using LRU. We note that the PC table is usually much smaller than the block table as most applications have a few call sites responsible for accessing a large number of blocks.

5 Performance evaluation

We compare the performance of UBM, PCC, ARC [27], and LRU via trace-driven simulations and an actual implementation in Linux. Simulated UBM results were obtained using the unmodified UBM simulator from the authors of UBM [20]. To obtain results for PCC, we modified the UBM simulator by replacing its classification routine with PCC; UBM's cache management based on marginal gain evaluations was kept without any modification. Using the same marginal gain in PCC and UBM isolates the performance difference due to different pattern classification schemes. We also implemented the ARC scheme [27] as presented in [28]. ARC is a state-of-the-art recency/frequency-based policy that offers comparable performance to the best online and off-line algorithms. It dynamically adjusts the balance between the LRU and LFU components for a changing workload by maintaining two LRU lists: one contains pages seen only once while the other contains pages seen more than once. At any given time, ARC selects top elements from both lists to reside in the cache. Finally, we implemented both PCC and UBM in Linux kernel 2.4.

5.1 Cache organization and management

Both the UBM-based and the PCC-based replacement schemes in our comparison study use the marginal gain in the original UBM [20] to manage the three partitions of the buffer cache, used to keep blocks with sequential, looping, and other references, respectively. Once the blocks are classified, they are stored in the appropriate subcache and managed with the corresponding replacement policy. Sequentially referenced blocks, as defined, are not accessed again, and therefore they can be discarded immediately after being accessed. Looping references are primarily managed based on the looping interval. Looping blocks with the largest interval will be replaced first since they will be used furthest in the future. If all blocks in the cache have the same detected interval, the replacement is made based on the MRU [29] replacement policy. References classified as other are managed by LRU as in the original UBM [20], but can be managed by other recency/frequency-based policies as well.

The cache management module uses marginal gain computation to dynamically allocate the cache space among the three reference types [29, 43]. As mentioned earlier, sequential references can be discarded immediately. Since there is no benefit from caching them, the marginal gain is zero, and the subcache for sequential references consists of only one block per application thread. The remaining portion of the cache is distributed dynamically between sequential and other references. The benefit of having an additional block is estimated for each subcache, and the block is removed from the subcache that would have gained less by having the additional block and given to the subcache that will benefit more.

5.2 Applications

Tables 1 and 2 show the five applications and three concurrent executions of the mixed applications used in this study. For each application, Table 1 lists the number of I/O references, the size of the I/O reference stream, the number of unique files accessed, and the number of unique signature PCs used for the I/O references. The selected applications and workload sizes are comparable to the workloads in recent studies [11, 17, 24, 27] and require cache sizes up to 1024MB.

Cscope [42] performs source code examination. The examined source code is Linux kernel 2.4.20. *Glimpse* [25] is an indexing and query system and is used

Appl.	Num. of references	Data size [MB]	Num. of files	Num. of PCs
cscope	1119161	260	10635	107
glimpse	519382	669	43649	25
gcc	158667	41	2098	334
viewperf	303123	495	289	179
tpc-h	121307	196	80	242
multi1	1278135	297	12246	442
multi2	1580908	792	12514	605
multi3	640467	865	43738	268

Table 1: Applications and trace statistics

Appl.	Applications executed concurrently
multi1	*cscope, gcc*
multi2	*cscope, gcc, viewperf*
multi3	*glimpse, tpc-h*

Table 2: Concurrent applications

to search for text strings in 550MB of text files under the `/usr` directory. In both *Cscope* and *Glimpse*, an index is first built, and single word queries are then issued. Only I/O operations during the query phases are used in the experiments. In both applications, looping references dominate sequential and other references. *Gcc* builds Linux kernel 2.4.20 and is one of the commonly used benchmarks. It shows both looping and sequential references, but looping references, i.e., repeated accesses to small header files, dominate sequential references. As a result, it has a very small working set; 4MB of buffer cache is enough to contain the header files. This constrains the buffer cache sizes used in the evaluation. *Viewperf* is a *SPEC* benchmark that measures the performance of a graphics workstation. The benchmark executes multiple tests to stress different capabilities of the system. The patterns are mostly regular loops as *viewperf* reads entire files to render images. The *Postgres* [34] database system from the University of California is used to run *TPC-H* (*tpc-h*) [44]. *Tpc-h* accesses a few large data files, some of which have multiple concurrent access patterns.

Multi1 consists of concurrent executions of *cscope* and *gcc*. It represents the workload in a code development environment. *Multi2* consists of concurrent executions of *cscope*, *gcc*, and *viewperf*. It represents the workload in a workstation environment used to develop graphical applications and simulations. *Multi3* consists of concurrent executions of *glimpse* and *tpc-h*. It represents the workload in a server environment running a database server and a web index server.

We briefly discuss the characteristics of the individual applications that affect the performance of the eviction policies. Figure 7 shows the cumulative distribution of the references to the files in these applications. We observe that the number of files contributing to the number

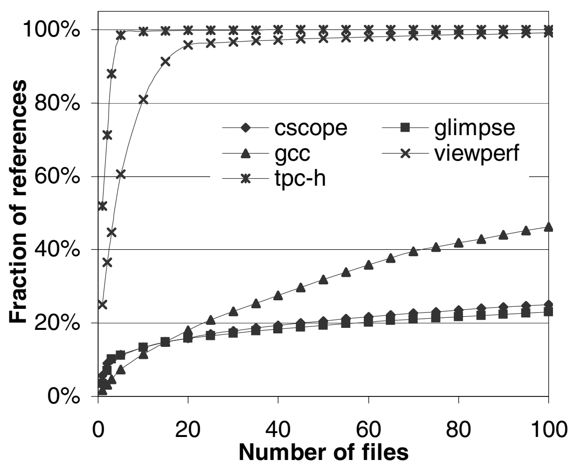


Figure 7: Cumulative distribution of file accesses

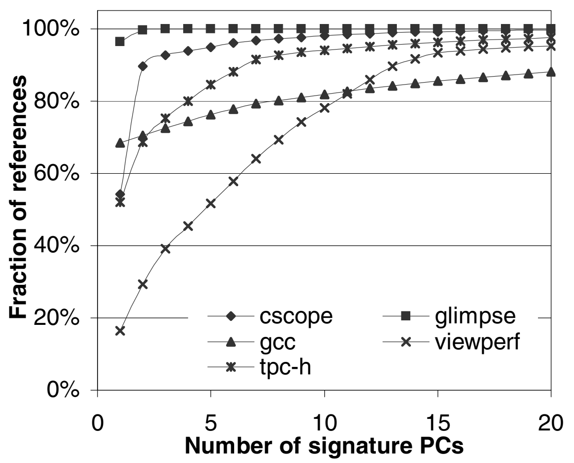


Figure 8: Cumulative distribution of signature PCs

of references spans a wide range. The number of files affects the performance of UBM. Since UBM trains on each new file to detect the looping and sequential references, the amount of training in UBM is proportional to the number of files that an application accesses. The number of I/O call sites (signature PCs) has a similar impact on training in PCC. Figure 8 shows the cumulative distribution of the references triggered by signature PCs in the applications. Compared to UBM, in PCC, fewer than 30 PCs are responsible for almost all references in all applications, resulting in shorter training in PCC than in UBM. Lastly, the average number of application function stack frame traversals to reach `main()` for the eight application versions is 6.45.

5.3 Simulation results

The detailed traces of the applications were obtained by modifying the *strace* Linux utility. *Strace* intercepts the system calls of the traced process and is modified to

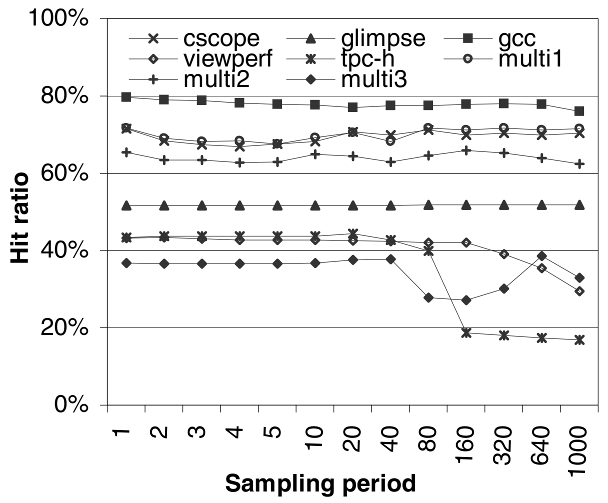


Figure 9: Impact of sampling period on PCC's hit ratio

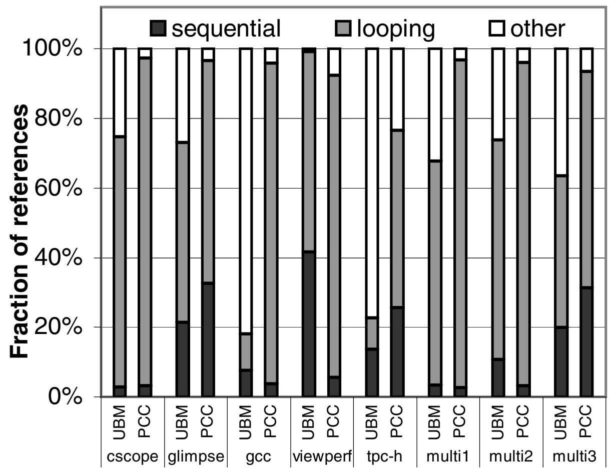


Figure 10: Reference classification in UBM and PCC

record the following information about the I/O operations: PC of the calling instruction, access type, file identifier (`inode`), and I/O size.

5.3.1 Impact of sampling frequency in PCC

Figure 9 shows the impact of block sampling on the hit ratio in PCC. We selected a single cache size for each benchmark – 128MB for *cscope*, *tpc-h*, *multi1*, and *multi2*, 512MB for *glimpse* and *multi3*, 2MB for *gcc*, and 32MB for *viewperf*. The same cache sizes are used in Section 5.4 to compare the performance of LRU, UBM, and PCC in our Linux implementation. No limit is imposed on the block table size, and the threshold value is set to 100. Figure 9 shows that increasing the sampling period to 20 barely affects the hit ratio for all applications. However, as the sampling period continues to increase, PCC may not capture changes in the loop periods and result in reduced hit ratios. We also performed sensitivity analysis on the threshold value of PCC. The results show that varying the threshold between 5 and 400 results in less than a 2% variation in hit ratios for the eight application versions. Thus, we chose a sampling period of 20 and a threshold of 100 in our simulation and implementation experiments below.

Using sampling significantly reduces the storage overhead of the block table. Recording all blocks accessed by an application require as much as 220K entries in the block table for the eight application versions in Table 1. Using a sampling period of 20, the maximum number of entries in the block table is less than 9K entries for all applications. Since the memory overhead of a block table with 9K entries is comparable to that of the per-file pattern table in UBM (up to 20K entries for these applications), we do not explore the impact of the LRU replacement in the PC table in this paper.

5.3.2 Pattern classification

To help understand the measured hit ratios, we first present the classification results for PCC and UBM in Figure 10. The classification results are a characteristic of the applications and are independent of the cache size. The accuracies of the classification schemes will dictate how often the cache manager can apply the appropriate eviction policies for the accessed blocks.

UBM and PCC on average have 15% and 13% sequential references, respectively. PCC has an advantage in detecting looping references when the same PC is used to access multiple files. As a result, PCC has an average of 80% looping references and only 7% other references. UBM classifies many more references as other, resulting in an average 46% looping and 39% other references. In the cases of *gcc* and *tpc-h*, UBM classifies a limited fraction of references, and thus the performance of UBM is expected to degrade to that of LRU. In these cases, ARC and PCC will provide improved performance over UBM.

5.3.3 Cache hit ratios

Figure 11 shows the hit ratios for the studied applications under different cache sizes in Megabytes. We plot the results for five replacement policies for each application: OPT, PCC, UBM, ARC, and LRU. The optimal replacement policy (OPT) assumes future knowledge and selects the cache block for eviction that is accessed furthest in the future [3]. PCC performs block sampling as described in Section 4.2. The threshold value of 3 was found to be optimal for UBM for the studied applications and was used in the experiments.

Overall, compared to UBM, PCC improves the absolute hit ratio by as much as 29.3% with an average of 13.8% maximum improvement over the eight application versions. Compared to ARC, PCC improves the absolute

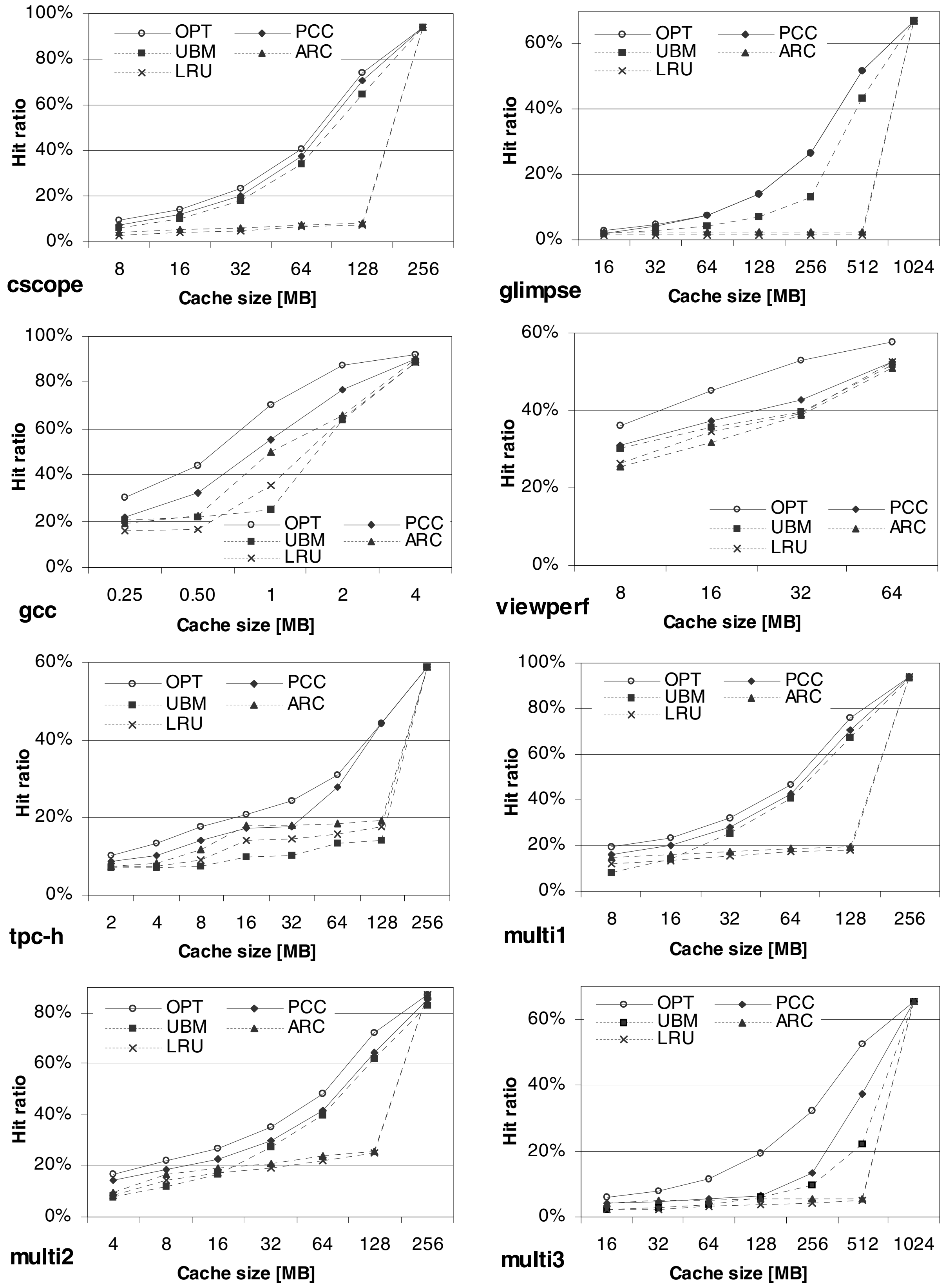


Figure 11: Comparison of cache replacement schemes

hit ratio by as much as 63.4% with an average of 35.2% maximum improvement.

Cscope *Cscope* scans an index file and many text files during source code examination and shows the looping behavior for a large number of files. This is a pathological case for LRU when the blocks accessed in the loop cannot fit in the cache. Similar behavior is present in ARC as all blocks are accessed more than once and the frequency list is also managed by LRU. The benefit of LRU and ARC caching is only observed when the entire looping file set fits in the cache. Both UBM and PCC show significant gains over LRU and ARC. UBM is able to classify the references and take advantage of the MRU replacement policy, achieving a maximum of 57.3% improvement in hit ratio over LRU. In contrast, PCC achieves as much as a 64.4% higher hit ratio than LRU and 7.1% higher than UBM. Two factors contribute to the hit ratio difference between PCC and UBM. First, UBM is unable to classify small files which are inside a loop and account for 10% of the references. Second, *Cscope* periodically calls the seek function to adjust the file position during sequential reading of the index file. The adjustment, although very small, disrupts the looping classification in UBM, causing the the classification to switch to other and subsequently to sequential. In contrast, PCC only observes small fluctuations in the resulting loop period but maintains the looping classification.

Glimpse *Glimpse* is similar to *cscope* as it scans both an index file and many text files during string search. UBM has to train for each of the 43649 files it accesses. It classifies 56% of these files as other since they have sizes below the threshold of three blocks. Each of the remaining files is first classified as other during the initial two references, and then as sequential upon the third reference, and finally looping once a loop is detected. Since all of these files are accessed by a single PC, PCC will train much more quickly, resulting in the accurate classification of looping. Figure 10 shows that PCC detects 12% more looping references than UBM. Figure 11 shows that PCC closely matches the hit ratio of OPT across different cache sizes as it is able to classify all references as looping and apply the MRU policy to the accessed blocks.

Gcc In *Gcc*, 50% of references are to files shorter than the threshold. As discussed in Section 3.2, accesses to all the headers files are triggered by a single PC, and thus PCC is able to learn the looping access pattern of the PC once and makes a correct looping prediction for all the remaining accesses to the header files. In contrast, files shorter than the threshold will be classified by UBM as having other references. Figure 10 shows PCC is able to detect more looping references as compared to UBM, and Figure 11 shows PCC achieves a significantly higher hit ratio than UBM. When the cache size is 1MB, UBM achieves a lower hit ratios than LRU. This is due to accesses to temporary files that are accessed only twice. First-time accesses to a temporary file are misclassified by UBM as sequential and discarded from the cache. Second-time accesses to the temporary file are again misclassified by UBM as looping and the blocks are placed in the loop cache, taking resources away from the LRU cache. These misclassifications in UBM hurt the hit ratio the most when the cache size is 1MB.

Viewperf *Viewperf* is driven by five different datasets, each having several files describing the data. Over 95% of the references when driven by the five datasets are to 8, 4, 1, 4, and 3 different files, respectively. Each file is reused as input to several invocations of the viewperf application with different parameters and is then discarded. A total of four files in *viewperf* are accessed only twice, which results in wrong classifications in UBM for both passes, similarly as in *gcc*. As in *gcc*, PCC uses only the first file to learn and correctly classifies accesses to the remaining files, resulting in higher hit ratios than UBM.

Tpc-h *Tpc-h* has three files that account for 88% of all accesses, and therefore the training overhead for new files is not a major issue for UBM. The main difference between PCC and UBM is due to accesses to one of the three files which accounts for 19% of the references. As explained in Section 3, accesses to the file contain multiple concurrent looping references, and UBM will misclassify all references to the file. Accesses in Figures 2(b) and 2(c) should be classified as looping, and accesses in Figure 2(d) sequential. However, irregularities and multiple concurrent looping references will cause UBM to classify accesses in Figures 2(b)(c)(d) as other. Because of the high percentage of other references in UBM as shown in Figure 10, UBM performs similarly as LRU as shown in Figure 11.

Multi1, multi2 and multi3 The traces for *multi1*, *multi2*, and *multi3* contain the references of the individual applications and thus inherit their characteristics. The pattern detection is performed as in the case of individual applications, since there is no interference among different applications due to their distinct PCs.

The resulting cache hit ratio curves are dominated by the applications that perform accesses more frequently, i.e., *cscope* in *multi1* and *multi2* and *glimpse* in *multi3*. In *multi1* and *multi2*, *gcc* is able to cache the looping header files in under 4MB, while for larger buffer sizes, the access pattern is primarily dominated by the remaining applications. Interleaved references from *glimpse* and *tpc-h* in *multi3* affect the period calculation, resulting in lower hit ratios for PCC than expected from individual executions of the two applications as shown in Figure 11.

5.4 Implementation in Linux

We implemented both UBM-based and PCC-based replacement schemes in Linux kernel 2.4.20 running on a 2Ghz Intel Pentium 4 PC with 2GB RAM and a 40GB Seagate Barracuda hard disk. We modified the kernel to obtain the signature PCs. Specifically, we modified the `read` and `write` system calls such that upon each access to these calls, PCC traverses the call stack to retrieve the relevant program counters. Since multiple levels of stacks have to be traversed to determine the signature PC, the traversal involves repeatedly accessing the user space from the kernel space.

We also modified the cache in Linux to allow setting a fixed cache size. This modification allows us to vary the cache size and study the effectiveness of the classification schemes for different cache sizes while keeping all other system parameters unchanged. One limitation of our implementation is that it cannot retrieve the PCs for references to files mapped into memory by *mmap*. Such references are classified as other references. Out of the applications we studied, only *gcc* has a few memory mapped file accesses.

Based one the working set sizes, we selected the cache size to be 128MB for *cscope*, *tpc-h*, *multi1*, and *multi2*, 512MB for *glimpse* and *multi3*, 2MB for *gcc*, and 32MB for *viewperf*. Figure 12 shows the number of disk I/Os and the execution time for each application under UBM and PCC normalized to those under the basic LRU scheme. For all three schemes, the standard file system prefetching of Linux (`generic_file_readahead`) was enabled, which prefetches up to 32 blocks from the disk for sequential accesses. The execution times reported here are the average of three separate executions. Each execution was run after the system was rebooted to eliminate any effects from prior buffer cache content.

Overall, compared to the basic LRU, UBM reduces the average number of disk I/Os by 20.7%, while better classification in PCC results in an average of 41.5% reduction in the number of disk I/Os, or 20.8% over UBM. The reduction in disk I/Os leads to a reduction in the execution time; UBM reduces the average execution time of LRU by 6.8%, while PCC reduces the average execution time of LRU by 20.5%, or 13.7% compared to UBM. We notice that in *viewperf* the small decrease in I/Os does not translate into saving in execution time because *viewperf* is a CPU-bound application. All other application are I/O-bound and the reduction in execution time follows the reduction in the number of I/Os.

5.4.1 Runtime overhead of PCC

To measure the overhead of PCC, we used a microbenchmark that repeatedly reads the same block of a file which results in hits under all of the LRU, UBM, and PCC schemes. Since the average number of application stack frame traversals in our experiments is 6.45, we set our microbenchmark to go through 7 subroutine calls before invoking the actual system call. The time to service a hit in the buffer cache in the unmodified kernel is 2.99 microseconds. In our implementation, this time increases to 3.52 microseconds for LRU because of code changes we incorporated to support a fixed cache size and different replacement schemes. The time to service a hit in UBM is 3.56 microseconds due to the marginal gain cache management and pattern classification overhead. This time increases to 3.75 microseconds in PCC. The additional overhead of 0.19 microsecond is due to obtaining the signature PC of the I/O operation. The additional overhead of 0.23 microsecond compared to LRU is promising: one saved cache miss which would cost 5-10 milliseconds is equivalent to the PCC overhead in about 20000 – 40000 cache accesses (hits or misses), and thus the overhead is expected to be overshadowed by the gain from the improved cache hit ratio.

Finally, we measured the impact of PCC overhead when the applications do not benefit from PCC. Using a 1GB cache, which is large enough that all replacement schemes result in identical cache hit ratios for all five individual applications. The average running time of the five applications using PCC is 0.65% longer than that using LRU.

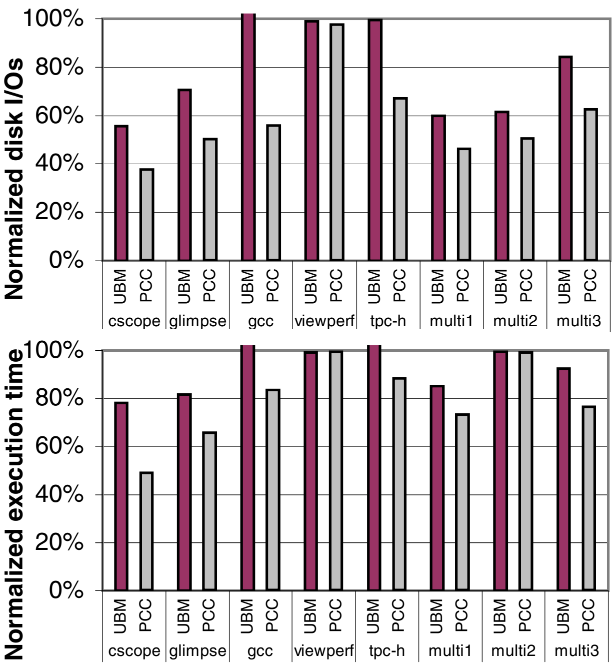


Figure 12: Performance of UBM and PCC integrated into the Linux kernel (Normalized to the performance of the basic LRU scheme)

6 Related work

The vast amount of work on PC-based techniques in computer architecture design have been discussed in Section 2. Here we briefly review previous buffer caching replacement policies which fall into three categories: recency/frequency-based, hint-based, and pattern-based.

Frequency/Recency-based policies Despite its simplicity, LRU can suffer from its pathological case when the working set size is larger than the cache and the application has looping access pattern. Similarly, LFU suffers from its pathological case when initially popular cache blocks are subsequently never used. Many policies have been proposed to avoid the pathological cases of LRU and LFU. LRU-K [30, 31] is related to LFU and replaces a block based on the Kth-to-the-last reference. LRU-K is more adaptable to the changing behavior but it still requires the logarithmic complexity of LFU to manage the priority queue. To eliminate the logarithmic complexity of LRU-K, 2Q [18] maintains two queues: one queue for blocks referenced only once, and another for reoccurring references. If a block in the first queue is referenced again, it is moved to the second queue. This simple algorithm results in constant complexity per access; however, it requires two tunable parameters. Low Inter-reference Recency Set (LIRS) [17] maintains a complexity similar to that of LRU by using the distance between the last and second-to-the-last references to estimate the likelihood of the block being referenced again.

Many policies have been proposed to combine recency and frequency. The first policy to combine LRU and LFU is Frequency-Based Replacement (FBR) [37]. It combines the access frequency with the block age by maintaining an LRU queue divided into three sections: new, middle, and old. Least Recently/Frequently Used (LRFU) [24] provides a continuous range of policies between LRU and LFU. A parameter λ is used to control the amount of recency and frequency that is included in the value used for replacement. Adaptive LRFU (ALRFU) [23] dynamically adjusts λ, eliminating the need to properly set λ for a particular workload. The most recent additions to the LFU/LRU policies are Adaptive Replacement Cache (ARC) [27] and its variant CAR [2]. The basic idea of ARC/CAR is to partition the cache into two queues, each managed using either LRU (ARC) or CLOCK (CAR): one contains pages accessed only once, while the other contains pages accessed more than once. Like LRU, ARC/CAR has constant complexity per request.

Hint-based policies In application controlled cache management [8, 32], the programmer is responsible for inserting hints into the application which indicate to OS what data will or will not be accessed in the future and when. The OS then takes advantage of these hints to decide what cached data to discard and when. This can be a difficult task as the programmer has to carefully consider the access patterns of the application so that the resulting hints do not degrade the performance. To eliminate the burden on the programmers, compiler inserted hints were proposed [7]. These methods provide the benefits of user inserted hints for existing applications that can be simply recompiled with the proposed compiler. However, more complicated access patterns or input dependent patterns may be difficult for the compiler to characterize.

Pattern-based policies Dynamically adaptable pattern detection not only eliminates the burden on the programmer but also adapts to the user behavior. SEQ [14] detects sequential page fault patterns and applies the Most Recently Used (MRU) policy to those pages. For other pages, the LRU replacement is applied. However, SEQ does not distinguish sequential and looping references. EELRU [39] detects looping references by examining aggregate recency distribution of referenced pages and changes the eviction point using a simple cost/benefit analysis. As discussed in Section 3, DEAR and UBM [10, 11, 20] are closely related to PCC in that they are pattern-based buffer cache replacement schemes and explicitly separate and manage blocks that belong to different reference patterns. They differ from PCC in the granularity of classification; classification is on a per-application basis in DEAR, a per-file basis in UBM, and a per-call-site basis in PCC.

7 Conclusions

This paper presents the first design that applies PC-based prediction to the I/O management in operating systems. The proposed PCC pattern classification scheme allows the operating system to correlate I/O operations with the program context in which they are triggered, and thus has the potential to predict access patterns much more accurately than previous schemes. Compared to the per-file access pattern classification scheme in UBM, PCC offers several advantages: (1) it can accurately predict the reference patterns of new files before any access is performed, eliminating the training delay; (2) it can differentiate multiple concurrent access patterns in a single file.

Our evaluation using a range of benchmarks shows that, compared to UBM, PCC achieves on average a 13.8% higher maximum hit ratio in simulations with varying cache sizes, and reduces the average number of disk I/Os by 20.8% and the average execution time by 13.7% in our Linux implementation. These results demonstrate that exploiting the synergy between architecture and operating system techniques to solve problems of common characteristics, such as exploiting the memory hierarchy, is a promising research direction.

Acknowledgments

We wish to thank Mendel Rosenblum, our shepherd, and the anonymous reviewers for their constructive feedback on this work. We would also like to thank Jongmoo Choi for giving us access to the UBM simulator code.

References

[1] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. ICS*, June 1991.

[2] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. *Proc. FAST*, March 2004.

[3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] N. Bellas, I. Hajj, and C. Polychronopoulos. Using dynamic cache management techniques to reduce energy in a high-performance processor. In *Proc. ISLPED*, August 1999.

[5] D. Bitton, D. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *Proc. VLDB*, October 1983.

[6] B. Black, B. Mueller, S. Postal, R. Rakvic, N. Utamaphethai, and J. P. Shen. Load execution latency reduction. In *Proc. ICS*, July 1998.

[7] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM TOCS*, 19(2):111–170, 2001.

[8] 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 TOCS*, 14(4):311–343, 1996.

[9] R. W. Carr and J. L. Hennessy. WSCLOCK – a simple and effective algorithm for virtual memory management. In *Proc. SOSP-8*, December 1981.

[10] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. An Implementation study of a detection-based adaptive block replacement scheme. In *Proc. 1999 USENIX ATC*, June 1999.

[11] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proc. ACM SIGMETRICS*, June 2000.

[12] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. ISCA*, June 1998.

[13] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proc. ISCA*, June 1997.

[14] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *Proc. ACM SIGMETRICS*, June 1997.

[15] C. Gniady, Y. C. Hu, and Y.-H. Lu. Program counter based techniques for dynamic power management. In *Proc. HPCA-10*, February 2004.

[16] R. J. Hanson. TPC Benchmark B - What it means and how to use it. In *Transaction Processing Performance Council. TPC-B standard specification, revision 2.0, 1994*.

[17] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. ACM SIGMETRICS*, June 2002.

[18] T. Johnson and D. Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proc. VLDB-20*, September 1994.

[19] Y. Jgou and O. Temam. Speculative prefetching. In *Proc. ICS-7*, July 1993.

[20] 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. In *Proc. OSDI*, October 2000.

[21] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proc. ISCA*, June 2000.

[22] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proc. ISCA*, June 2001.

[23] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proc. ACM SIGMETRICS*, May 1999.

[24] 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 Transactions on Computers*, 50(12):1352–1360, 2001.

[25] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Winter 1994 Technical Conference*, January 1994.

[26] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proc. ISCA*, June 2003.

[27] N. Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proc. FAST*, March 2003.

[28] N. Megiddo and D. S. Modha. One up on LRU. *;login: - The Magazine of the USENIX Association*, 18(4):7–11, 2003.

[29] R. Ng, C. Faloutsos, and T. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers*, 44(4):546–560, 1995.

[30] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. ACM SIGMOD*, May 1993.

[31] E. J. O'Neil, P. E. O'Neil, and G. Weikum. An optimality proof of the LRU-K page replacement algorithm. *J. ACM*, 46(1):92–112, 1999.

[32] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. SOSP-15*, December 1995.

[33] S. S. Pinter and A. Yoaz. Tango: a hardware-based data prefetching technique for superscalar processors. In *Proc. MICRO-29*, December 1996.

[34] Postgres. http://www.postgresql.org/.

[35] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proc. MICRO-34*, Dec. 2001.

[36] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *Proc. MICRO-31*, December 1998.

[37] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. ACM SIGMETRICS*, May 1990.

[38] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proc. ISCA*, June 2000.

[39] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: simple and effective adaptive page replacement. In *Proc. ACM SIGMETRICS*, May 1999.

[40] J. E. Smith. A study of branch prediction strategies. In *Proc. ISCA*, May 1981.

[41] SPEC. http://www.spec.org/gpc/opc.static/viewperf711info.html.

[42] J. Steffen. Interactive examination of a C program with Cscope. In *Proc. USENIX Winter 1985 Technical Conference*, 1985.

[43] D. Thibaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6):665–676, 1992.

[44] TPC. Transaction Processing Council. http://www.tpc.org.