An Automatic Prefetching and Caching System

Joshua Lewis, Mohammed Alghamdi†, Maen Al Assaf, Xiaojun Ruan, Zhiyang Ding, Xiao Qin
Department of Computer Science and Software Engineering
Auburn University
Auburn, Alabama 36849
Email: {jtl0003, mma0005, xzr0001, dingzhi, xzq0001}@auburn.edu
†Computer Science Department
Al-Baha University
Al-Baha City, Kingdom of Saudi Arabia

***Abstract*—Steady improvements in storage capacities and CPU clock speeds intensify the performance bottleneck at the I/O subsystem of modern computers. Caching data can efficiently short circuit costly delays associated with disk accesses. Recent studies have shown that disk I/O performance gains provided by a cache buffer do not scale with cache size. Therefore, new algorithms have to be investigated to better utilize cache buffer space. Predictive prefetching and caching solutions have been shown to improve I/O performance in an efficient and scalable manner in simulation experiments. However, most predictive prefetching algorithms have not yet been implemented in real-world storage systems due to two main limitations: first, the existing prefetching solutions are unable to self regulate based on changing I/O workload; second, excessive number of unneeded blocks are prefetched. Combined, these drawbacks make predictive prefetching and caching a less attractive solution than the simple LRU management. To address these problems, in this paper we propose an automatic prefetching and caching system (or APACS for short), which mitigates all of these shortcomings through three unique techniques, namely: (1) dynamic cache partitioning, (2) prefetch pipelining, and (3) prefetch buffer management. APACS dynamically partitions the buffer cache memory, used for prefetched and cached blocks, by automatically changing buffer/cache sizes in accordance to global I/O performance. The adaptive partitioning scheme implemented in APACS optimizes cache hit ratios, which subsequently accelerates application execution speeds. Experimental results obtained from trace-driven simulations show that APACS outperforms the LRU cache management and existing prefetching algorithms by an average of over 50%.**

I. Introduction

A. Prefetching

Prefetching is a technique that proactively constitutes reading data blocks from secondary storage to main memory in order to avoid disk access delays. Most existing prefetching schemes fall into two categories: predictive or instructive prefetching. The former infers likely block accesses based on past I/O system call patterns. The latter uses hints provided by applications to inform file systems of future data block requirements. Both types of prefetching schemes provide a simple mechanism to mask the I/O latency associated with multiple disk accesses by parallelizing CPU and disk read activities.

Because the application requirements are known *a priori* in informed prefetching, a prefetching pipeline [9] can further boost performance. The pitfall of the informed approach lies in the lengthy static compile-time analysis that must accompany writing applications. Before applying the informed prefetching strategy, each application must be analyzed for specific I/O patterns to determine the eligibility for prefetching. Applications that make sparse sequential I/O requests would not benefit from prefetching as much as applications that make frequent nonsequential requests. The added time and effort required to identify likely candidates and insert hints in the proper locations make informed prefetching a less attractive solution to I/O performance bottlenecks.

Alternatively, predictive prefetching, implemented at the file system level, maintains a mechanism that can predict future read/write accesses based on recent I/O system calls. While predictive prefetching may not provide the same accuracy as informed prefetching, predictive approaches remain application independent. Informed prefetching must be implemented into each application, while the predictive prefetching module works from the system kernel. Furthermore, fetching multiple data blocks from disks simultaneously reduces cumulative seek times associated with several serial I/O accesses. Pipelining prefetches allow data blocks to be accessed in parallel, which condenses stall times of multiple block accesses to a single stall period. Informed prefetching can seamlessly include pipelining (see, for example, [9]) as the prefetches are known during the implementation phase of applications. Predictive prefetching can not guarantee accurate prefetches consistently which makes pipelining less feasible. With a limited cache space, pipelining inaccurate prefetches could easily overload the memory buffer. In this study, we demonstrate a novel approach to pipelining predictive prefetches based on an analysis of the *probability graph* structure to find common sequences of I/O system calls that can be prefetched in parallel. The concept of the *probability graph* structure was introduced by Griffioen and Appleton [5]. With a correct implementation, predictive prefetching exhibits similar performance improvements to that of informed prefetching.

B. Cache Management

Assuming a partitioned cache, prefetching can drastically improve cache hit ratios for any size cache [5] by coupling the Least Recently Used (LRU) cache management algorithm

978-1-4244-9328-9/10/$26.00 ©2010 IEEE

with a prefetch buffer management algorithm. The net effect of a higher cache hit ratio includes, but is not limited to, smaller cache sizes, increased execution speeds, and reduced energy consumption.

By implementing a cost-benefit analysis of buffer allocations, with a logic similar to that employed by Patterson *et al.* [9], the proposed prefetching and caching system or APACS dynamically allocates and replaces buffer and cache blocks for both prefetching and caching in accordance to global system performance impacts. Prefetch replacements occur by measuring the elapsed time from when a prefetched block has been loaded from secondary storage to the current time. If this time exceeds the window in which the block was most likely to be cached, a page replacement occurs. Indeed, with only a fraction of cache space dedicated to prefetching blocks, this replacement policy ensures maximal use of prefetched buffer space. Data intensive applications benefit from minimized disk accesses, especially with secondary or tertiary storage devices. Dynamic partitioning optimizes cache hit ratios, which subsequently accelerates application execution speeds.

C. Contributions

The main contributions of this study are summarized as follows:

- We developed a novel pipelining mechanism for a predictive prefetching and caching environment, thereby effectively reducing disk I/O stall periods.
- We designed an effective optimization algorithm for dynamic buffer/cache partitioning. The algorithm partitions the buffer cache memory used for prefetched and cached blocks by automatically changing buffer/cache sizes in accordance to global I/O performance.
- We implemented a new buffer management subsystem supporting predictive prefetches. We seamlessly integrated the dynamic cache partitioning module, the prefetch pipelining module, and the buffer management module in the APACS (automatic prefetching and caching) system.

D. The Organization of this Paper

The remainder of this paper details the design and implementation of the APACS system. Section 2 describes related work. Sections 3 and 4 describe the designs and implementation of APACS. Section 5 provides a detailed analysis and comparison of the APACS system with other prefetching solutions. Finally, Section 6 outlines our conclusions and direction for future work.

II. Related Work

Cache Management. Cache hit ratio is the most important performance factor in any I/O-intensive application. For this reason, many approaches have been developed for cache management. Page replacement algorithms decide a way of managing a full undivided memory cache space. The most prevalent page replacement policies include the Least Recently Used (LRU) policy, the Most Recently Used (MRU) policy, the Least Frequently Used (LFU) policy, and the First In First Out (FIFO) policy [1], [7]. For example, the current GNU/Linux kernel uses a simple LRU replacement policy to manage its cache. Advantages for employing these simple management schemes include minimal cache management overhead and reduced complexity. These algorithms depend on the size of applications' working sets being proximal to cache sizes. The performance degrades when an application's working set has weak locality and is larger than the cache capacity [3].

Prefetching. To reduce the naïveté of the caching algorithms, predictive prefetching provides insight to the nature of I/O access patterns and greatly complements the aforementioned cache management policies when implemented in tandem. To accomplish this goal, one can divide the memory cache space into a prefetch buffer and a cache buffer. The prefetch buffer and cache buffer for a storage system are using different hypotheses about I/O access patterns. Many management schemes have been developed to control these two different types of buffer-cache partitions [1], [2], [6], [9], [10], [11], [16], [17].

Another issue to consider when prefetching is how aggressively to prefetch. In most cases, a threshold of probability must determine whether a particular prefetch is to take place. This throttling of prefetch quality reduces the number of pointless I/O accesses.

When dealing with networks like the Internet, Yang and Zhang showed that correctly prefetching and caching web content reduces network traffic and wasted bandwidth [15]. Web prefetching, microprocessor instruction prefetching, and file system prefetching are a few among many forms of prefetching that aid in hiding disk latencies by parallelizing computation and the I/O accesses.

Buffer-Cache Partitioning. For instance, to reduce prefetched blocks' consumption of cache buffers, Reungsang *et al.* made use of a *Fixed Prefetch Block* (FPB) to limit the prefetch buffer size [11]. This static buffer partitioning proposed by Reungsang *et al.* reduces the potential performance gains of prefetching when cache hit ratios are minimal. A low cache hit ratio indicates that the cache buffer size is too small, file access patterns are not redundant, or files are large enough to flush the cache frequently. FPB does not attempt to compensate for these conditions. If the prefetch buffer space is large, then the cache hit ratio could deteriorate due to less cache buffer memory. Static cache partitioning does not account for changing I/O access patterns, which stunts optimal prefetching and caching performance.

Compared with static buffer-cache partitioning, dynamic partitioning provides more flexibility to take full advantage of cache memory. The dynamic cache management approach used in the APACS system can be considered as an extension of the SA-W^{2}R scheme proposed by Jeon *et al.* [6]. The LRU One Block Lookahead (LRU-OBL) approach developed by Jeon *et al.*, however, only works well for specific access patterns and their approach does not take into account any type of parallel prefetching. Moreover, the resizing operations

of each cache partition occur independently versus cooperatively, a slight divergence from the global cost-benefit analysis implemented in our APACS system. Cao *et al.* [2] used the LRU-OBL aggressive approach to show that prefetching can be integrated with LRU to increase cache performance in a single disk system.

Cost-Benefit Analysis. Vellanki *et al.* (1999) [13] used a similar cost-benefit analysis as Patterson *et al.* [9] except modified the algorithm for use with predictive prefetching. This adds the quality of application independence that was lacking in the informative approach [13]. The cost-benefit analysis investigated by Vellanki and Chervenak [13] dynamically partitions an aggregate cache buffer according to weighing the costs of ejecting blocks of memory from the cache partition, versus allocating blocks to the cache partition based on probabilities of the block's future accesses. The APACS system presented here differs from their approach, in that APACS relies on a more general way of using cache and prefetch buffer hit ratios as an indicator for buffer/cache adjustment. Importantly, this method takes into account the full context of the current access patterns versus a single access.

Prediction of I/O Requests. Several models for encoding past access patterns and future probabilities have been developed by researchers [5], [12], [13]. The directed weighted graph, originally proposed by Griffioen *et al.* [5], is employed to estimate access probabilities. The Markov predictor is another model used by many prediction algorithms [3], [8], [13], [14]. The Markov predictor uses prediction by partial match (PPM) to find recurring sequences of events. Many proposals for web prefetching use PPM based on Markov predictors because of the hypertextual nature of the Internet and also the variability of access sequences [8], [4], [7]. Vellanki *et al.* [13] use a Lempel-Ziv (LZ) scheme that builds a directed tree structure based on temporal evaluation of disk accesses. This tree structure represents a method of pipelining prefetches alternative to the one described in this paper. The fundamental issue with using the LZ data compression algorithm and other Markov predictor based models lies in scalability. The APACS prefetching graph size increases in a linear pattern with the number of unique I/O requests, whereas the LV tree increases at an exponential rate. Memory and computation requirements increase proportionally with the latter. Wang *et al.* proposed a solution to this problem using parallel processes on separate machines for offloading predictive computation [14]. This offloading solution limits the number of platforms that the LZ scheme may benefit. The APACS maintains an ordered set of encountered system calls in a file access graph that can be modified and searched with little overhead in terms of memory space and execution time.

III. Automatic Prefetching and Caching

A. System Model

Similar to the model described in [13], we assume a single processor system running a modern operating system. The system includes a file cache that is divided into a *cache buffer* (C_{buff}) and a *prefetch buffer* (P_{buff}). The total file cache size and the disk access delay, T_{disk}, are static constants. A Samsung Serial ATA hard drive's average seek time, 8.9 ms, is used for T_{disk} in all simulations. C_{buff} stores recently accessed files and uses an LRU page replacement policy. P_{buff} stores blocks of memory that have been prefetched based on access probability and that do not already reside in C_{buff}. In line with [13], we assume that enough disk parallelism is present to diffuse any disk congestion. However, we make no assumption about the application's I/O requests or access patterns. The simulator sends requests to the "file system" at the same intervals provided by the real-world application traces.

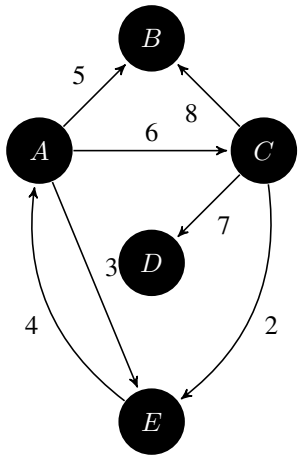


Fig. 1. The above figure shows a directed weighted graph that represents the probability of sequential file accesses based on temporal relation. For instance, the probability of file C being accessed after A is $\frac{6}{6+5+3}$ = 42.9%.

B. Predictive Prefetching

1) Probability Graph: The APACS predicts future file I/O activities based on the probability graph model presented in [5]. Fig. 1 shows a representative probability graph where each shaded circle represents a file and each edge represents a temporal association based on past file accesses. The set of files pointed to by a given file f is called f's *association window*, denoted W. File g appears in W if and only if g is accessed within a *lookahead window* time period, λ, after f. Additionally, a *minimum chance* variable, denoted δ, sets the minimum probably threshold for prefetching any file in W. The probability of accessing file g in W is calculated by dividing the incoming edge weight from $f \rightarrow g$ by the sum of f's outgoing edge weights. The lookahead window and minimum chance values throttle the degree of prefetching, ultimately reducing false predictions and buffer overflows.

2) Prefetch Pipelining: Systems that lack prefetching mechanisms experience stalls (T_{stall}) in which the CPU sits idle while data blocks are read from disks. Pipelining file prefetches minimizes the stall time present between I/O requests that occurs due to slow disk speed. To guarantee zero stall time, Patterson *et al.* [9] defined a term called the *prefetch*

TABLE I
SUMMARY OF PREFETCHING TIMING EXPRESSIONS

Expression	Definition
x	The number of accesses prefetched into the future
T_{cpu}	Appliation cpu time between I/O requests
T_{hit}	Time to read data from cache memory
T_{driver}	Time to allocate buffers in cache memory
T_{disk}	Disk access latency
T_{miss}	$T_{hit} + T_{driver} + T_{disk}$
$T_{stall}(x)$	$T_{disk} - x(T_{cpu} + T_{hit} + T_{driver})$

horizon, denoted Λ, as follows:

$$\Lambda = \frac{T_{disk}}{(T_{cpu} + T_{hit} + T_{driver})}$$

Prefetch horizon defines the degree of pipelining, or quantity of simultaneous prefetches, required to guarantee zero stall time between prefetch requests. Table I defines the meaning of all of the time variables used in this paper. Λ - a function of T_{cpu} - remains application dependent, thereby defeating the main benefits of predictive prefetching over informed prefetching. To overcome this problem, we enable APACS to dynamically adjust the value of Λ by maintaining a weighted moving average of T_{cpu}. Doing so allows for maintained performance through fluctuating application access patterns. One remaining issue involves the accuracy of the prefetches. Specifically, if prefetched files are pipelined, but not cached almost immediately, stalls in the pipeline will still occur. Thus, APACS attempts to dynamically locate sequences of commonly occuring system calls to pipeline.

Assume f is the most recent accessed file and for any given temporally ordered sequence of files f'_n, where $f'_i \in W$ and $1 \leq i < n \leq |W|$, f'_i has a lookahead window W'_i, such that the following equation holds true for all i in the sequence:

$$|W'_i \cap W'_{i+1}| = |W \cap W'_i| - 1$$

If such a sequence exists in W, then an accurate prefetching pipeline can be started provided a high enough probability exists between files in the sequence. If these probability values are relatively low and λ is high, then false positives may arise causing multiple stalls between prefetches and inefficient buffer usage.

Using a binary matrix with dimensions $(n-1)$ and $max_{i=1}^{n}(|W'_i|)$ for rows and columns, a '1' placed at the i^{th} row and j^{th} column represents a commonality between the sets W'_i and W when $0 \leq i \leq (n-1)$ and $0 \leq j \leq max_{i=1}^{n}(|W'_i|)$. Assuming a strict temporal ordering, an upper triangular matrix signifies a sequence of system calls that appear together freqently and can, thus, be prefetched in a pipeline. If the sequence length is greater than $\Lambda - 1$ and the predictions are accurate, no stalls will occur between prefetches. By identifying potential areas for pipelining, APACS drastically mitigates the performance difference between optimal and predictive prefetching. Fig. 2 illustrates the predictive prefetch pipeline (PPP) idea where each subsequent concentric circle represents a successive W'_i in the sequence. Fig. 3 shows the pseudo code of the PPP algorithm implemented in APACS.

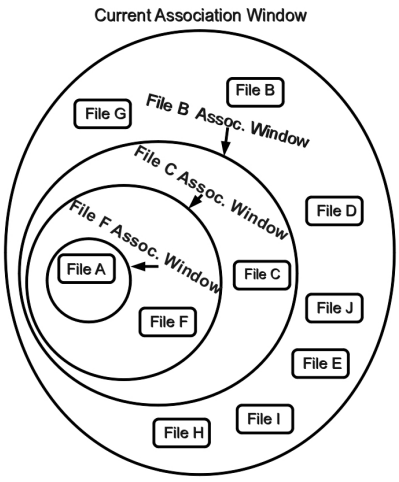


Fig. 2. The idea of the Predictive Prefetching Pipeline (PPP). In this case W'_1 is the current association window, W'_2 is file B's association window, W'_3 is file C's association window, and W'_4 is file F's association window.

The Predictive Prefetching Pipeline (PPP) Algorithm

```
for i = 0 and i < |W| do
  if call i.strength > min recurrence threshold then
    pipeline start = i, pipeline end = i
    //find sequence of Λ files that share same strength
    for j = i and j < i + Λ do
      if call j.strength == call i.strength then
        pipeline end++
      end if
    end for
    if pipeline end - pipeline start >= Λ then
      if total sequence strength > 1/2 Σ_{i=1}^{|W|} f'_i.strength
      then
        if triangular matrix form then
          perform pipelining on Λ calls
        end if
      end if
    end if
  end if
end for
```

Fig. 3. Two-step algorithm for Predictive Prefetching Pipeline (PPP). Step 1 ensures that there exists a sequence of system calls that have the same strength within W. Step 2 makes sure that the upper triangular matrix pattern occurs in the binary matrix.

Certain subsets (say, for example, S) of W may contain elements of the same probability of next access and appear, thus, to be eligible for pipelining. However, there is a likelihood that one or more subsets of S, are temporally separated sequences in W, with probability of this occuring increasing proportionally with $|W|$. Therefore, the triangular matrix algorithm differentiates these subsets to maintain better precision and to avoid false predictions.

C. Dynamic Cache Partitioning

To optimize a finite cache space, dynamic partitioning is required. Two partitions are needed: one for prefetching and one for caching. Recall that P_{buff} is prefetch buffer and C_{buff}

TABLE II
THE DYNAMIC CACHE PARTITIONING POLICY

Condition	$\|C_{buff}\|$	$\|P_{buff}\|$
$\Delta = \Theta = 0$	$Total_{buff} - P_{buff}$	$\Lambda * \|W_{avg}\| * \gamma$
$\Delta > \Theta$	increase	decrease
$\Delta < \Theta$	decreaes	increase

is cache buffer. Let $|W_{avg}|$ denote the average association window size, γ be the current hit ratio of P_{buff}, Δ denote the recent change in the hit ratio of C_{buff}, and Θ be the recent change in the hit ratio of P_{buff}.

The dynamic partitioning behavior optimizes the cache space according to the conditions outlined in Table II. The following guidelines were followed:

- If Δ is greater than Θ, increase cache buffer C_{buff}.
- If Θ is greater than Δ, increase prefetch buffer P_{buff}
- If Δ and Θ are both zero, reset partitions to current estimated optimal sizes.
- Else, do nothing.

Using the above principles, APACS dynamically optimizes the buffer utilization based on current changes in hit ratios of both prefetch and cache buffers. In a graphical sense, if the P_{buff} and C_{buff} hit ratios were plotted over time t, Δ and Θ represent the slope at any particular t. The idea is to maximize the combined area under the curves. In a cost-benefit sense, the buffer with the steepest positive slope, or rate of change, provides the most benefit to performance and vice versa.

When the size of P_{buff} exceeds the needed capacity for pipelining, Λ prefetches, a *prefetch time to live* (PTTL) policy is invoked for the least recent prefetches. PTTL measures the period of time a prefetched block must remain in P_{buff} before being ejected.

$$(T_{stall}(x) + \Lambda)(T_{cpu} + T_{hit} + T_{driver})$$

Here, the first factor, $(T_{stall}(x)+\Lambda)$, quantifies the window of time, in units of $(T_{cpu}+T_{hit}+T_{driver})$, a prefetch will likely be consumed by the application upon first arrival to P_{buff}. Since $T_{stall}(x)$ measures the time between prefetches and Λ prefetches may occur per prefetching pipeline, this window guarantees a time cushion for each prefetched block to be consumed by the application. When PTTL for a prefetch has elapsed, APACS ejects the prefetched block from P_{buff} and either allocates the freed pages to C_{buff} or P_{buff}.

After considering the PTTL policy, P_{buff} uses the standard LRU replacement policy because the most context relative blocks are those with the highest locality of references. Indeed, the more recent prefetched blocks are most likely to be needed by the application because prefetching occurs based on temporal associations.

The minimum chance value increases or decreases with respect to many parameters, not solely the frequency of appearance with any give association window. Thus, APACS dynamically adjusts the minimum chance parameter based on a comparison of Δ and Θ. The principle logic is as follows:

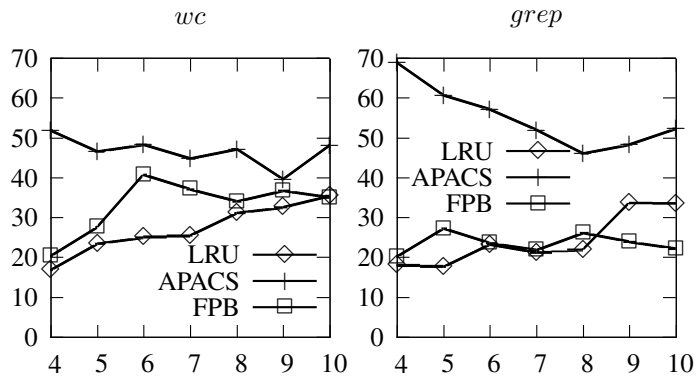


Fig. 4. Impacts of cache size on the cache hit ratios of standard LRU, APACS, and FPB buffer management policies with I/O traces of two common Linux commands. The cache size varies from 4 MB to 10 MB. The results indicate that APACS shows the greatest cache hit ratio average taken over all trials.

- If Δ is greater than Θ increase minimum chance
- If Δ is less than Θ decrease minimum chance
- Else, do nothing

In the first case, because $|P_{buff}|$ is decreasing, APACS requires prefetched files to have a higher probability of next access. In the second case, a less stringent minimum chance allows more prefetching opportunities when the prefetch hit ratio is rising.

IV. EXPERIMENTS

A. Benchmarks - Linux Commands

To measure the performance of APACS in which the new predictive prefetching and cache management mechanisms were integrated, we gathered I/O traces from two common linux commands (i.e., grep and wc) with a certain amount of redundancy. The commands were run on a standard, 2.0 GHz dual-core HP laptop notebook with 1 gigabyte of RAM running Linux Mint 8. Because prediction requires a degree of historical reference, the trace redundancy allowed APACS to utilize the predictive prefetching approach. Traced file system calls were passed to a file system simulator within which the cache manager performed dynamic cache partitioning and predictive prefetching. Keeping the same lookahead window, experiments were also conducted using LRU and Fixed Prefetch Block (FPB) algorithms.

Fig. 4 illustrates the cache hit ratios for LRU, FPB, and APACS with cache sizes ranging from 4 to 10 MBytes. Compared with LRU and FPB, APACS improved the performance of the *grep* and *wc* commands by an average of 145% and 83%, respectively. Several conclusions can be drawn from the results plotted in Fig. 4. First, in every case, the predictive prefetching algorithm implemented in APACS performs considerably better than LRU and FPB when the cache size is small. Indeed, when the cache hit ratio remains low, more opportunities arise for improving the cache hit ratio with prefetching. Second, the dynamic cache partitioning mechanism in APACS always performs significantly better than FPB. In contrast to FPB, APACS flexibility optimizes

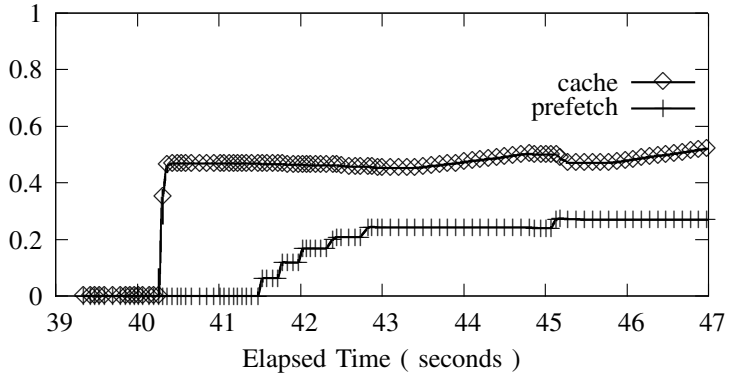


Fig. 5. P_{buff} and C_{buff} Hit Ratio Comparison. Hit ratios of cache buffer C_{buff} and prefetch buffer P_{buff} over the period of eight seconds using 1 MB of cache memory.

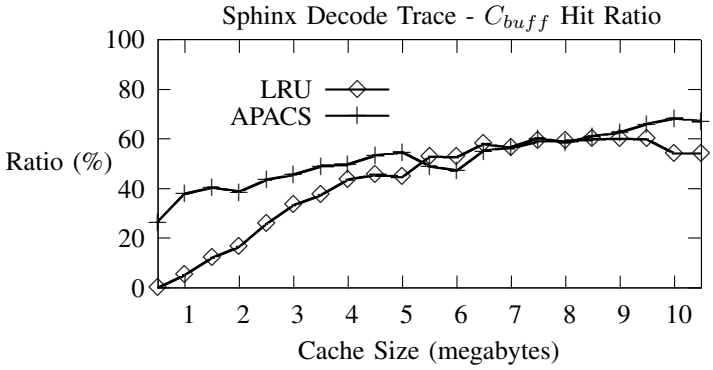


Fig. 6. Impacts of the cache size on the hit ratio of cache buffer C_{buff}. The Sphinx application is used as a benchmark. The best cache hit ratios occur when the lookahead window is small and the cache size is large.

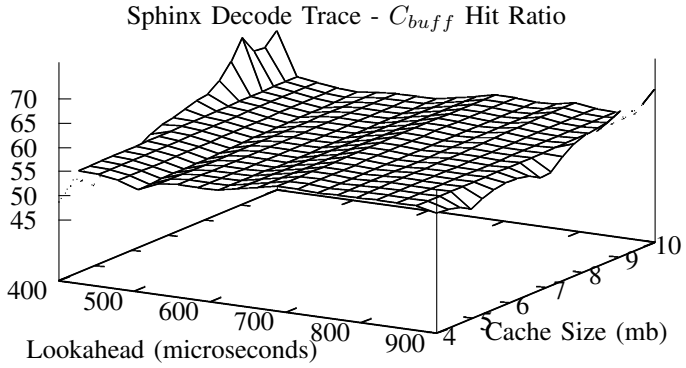


Fig. 7. Impacts of the lookahead window and cache size on the hit ratio of cache buffer C_{buff}. The Sphinx application is used as a benchmark.

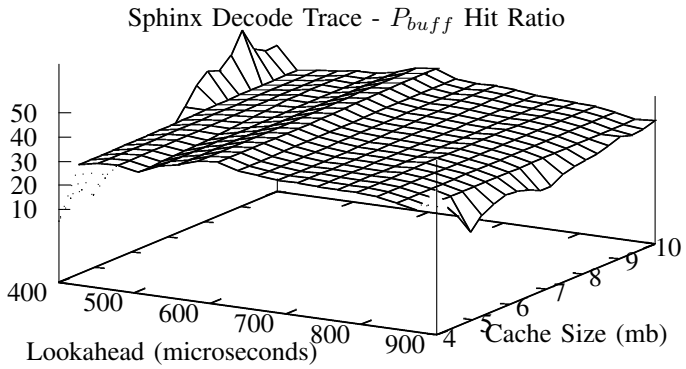


Fig. 8. Impacts of the lookahead window and cache size on the hit ratio of prefetch buffer P_{buff}. The Sphinx application is used as a benchmark.

possible performance gains over varying I/O workloads and access patterns. In this way, APACS overcomes the rigid LRU policy and FPB size requirements with a dynamic optimization approach.

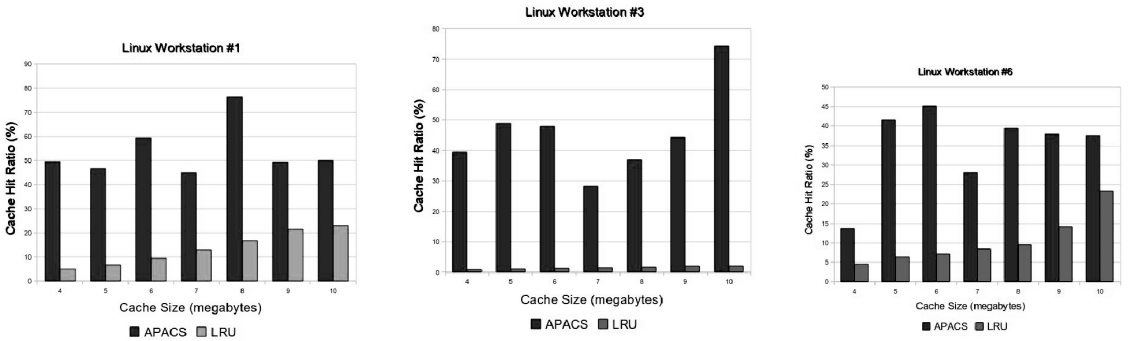
Fig. 5 shows the relationship between the cache hit ratio and prefetch hit ratio. When the cache hit ratio has a negative slope, APACS attempts to increase the prefetch hit ratio by allocating more space to prefetch buffer P_{buff} while increasing the minimum chance value. In doing so, when one ratio is falling, another is generally rising to compensate and curb the decline in performance. Functionally, APACS makes an effort to maximize the combined area under both curves at any point in time.

B. A Real-World Application - Sphinx Speech Recognition

In the second experiment, we evaluated I/O workload using real-world traces representing the Sphinx speech recognition engine. Sphinx-3, specifically, is a Hidden Markov Model (HMM) based speech recognition system that was designed at Carnegie Mellon University. Sphinx operates by first learning the attributes of different sound units and then uses this knowledge to determine the correct sequence of sound units for a given speech signal. These two processes are called training and decoding, respectively. In this experiment, I/O traces were collected from Sphinx-3, which is the same application Patterson *et al.* chose for testing Transparent Informed Prefetching and Caching (TIP) [9]. Because the Sphinx-3 system uses a database of audio files to train, many of the system calls are known *a priori*. This makes Sphinx a perfect candidate for informed prefetching. This fact does not exclude Sphinx and similar applications from the performance benefits of APACS.

Fig. 6 shows the simulation results for the Sphinx application when the file system is supported by LRU and APACS. When the cache size is small, APACS performs magnitudes better than LRU. When the cache size is large, APACS still outperforms LRU. It is interesting to note that the prefetch hit ratio remained below 10% when less than 6 MB of cache was used in the system. The blocks APACS P_{buff} saves from LRU replacement occuring in the C_{buff} accounts for most of the performance benefits when smaller caches were used and prefetching is nominally beneficial.

Figs. 7 and 8 show the relationship between the lookahead window size and the cache and prefetch hit ratios over an increasing buffer size. It is obvious that the cache size has more significant impact on the cache hit ratio than the lookahead window. However, with a larger cache size, smaller lookahead windows are more beneficial because access probabilities are not as diluted by larger association window sizes.



(a) I/O Trace has 49,030 system calls. (b) I/O Trace has 343,022 system calls. (c) I/O Trace has 50,972 system calls.

Fig. 9. Results for the three real-world I/O traces from the Linux workstations. APACS outperforms LRU in each experiment.

C. Real-World Workstation Traces

Real-world traces for this set of experiments were downloaded from http://iotta.snia.org/traces/list/SystemCall. The I/O traces covered 13 machines that were used by computer science researchers for a security related software development project. The traces span an entire year, from 2000 to 2001, and include 3.2 GB of compressed system-call traces. To produce accurate measures, the APACS file system simulator attempts to send I/O requests at the same intervals provided in the traces. The accurate control of arrival times for I/O requests is important, because if the requests were sent in sequence and without pause, the access patterns would not accurately represent the inter-access CPU time between successive file system requests. A consequence of this would be negligible prefetching performance gains because of the T_{stall} delay required to prefetch needed blocks from secondary storage devices. Simulations were run at the Alabama Supercomputer Center to guarantee dedicated resources. The available hardware consisted of a cluster of SGI Altix 350 and 450 nodes using a shared memory architecture. Each SGI Altix 350 node includes 16 1.5 GHz Intel Itanium2 processors and 32 GB of shared memory. The SGI Altix 450 nodes include 32 1.67 Ghz dual-core Intel Itanium2 processors with 464 GB of shared memory.

Fig. 9 shows testing results for I/O traces recorded from three of the Linux workstations. In all three cases, the LRU cache managment gradually improves the cache hit ratio as the cache size increases. APACS improves LRU's cache hit ratio in all cases by more than 75%, albeit with no visibly consistent trend. This latter property results from the dynamic adjustments that take place from fluctuating hit ratios and subsequent cache repartitioning.

D. Linux Kernel Compile Traces

The latest stable linux kernel release, 2.6.34, was downloaded and compiled on the same HP notebook as used in the first experiment. Using the *strace* command and some useful parameters, the compilation process was traced for I/O system calls. The kernel compilation traces measure 205 megabytes and span 29.5 minutes, the time it took to compile

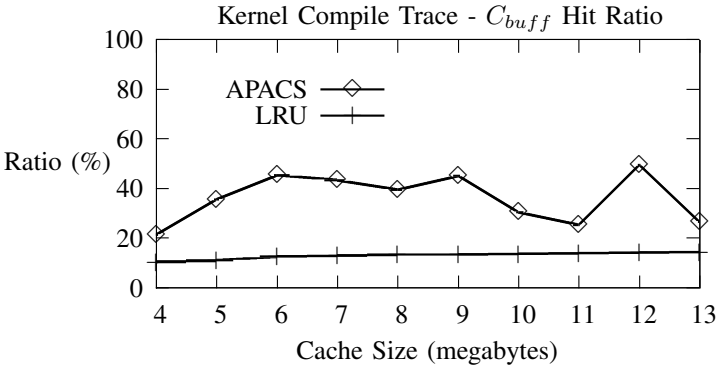


Fig. 10. Results for the Linux kernel 2.6.34 compilation process traces.

the disk image. The two main programs running during this process were the linux linker (ld) and C compiler (cc). In the first series of simulations, cache hit ratios over varying cache sizes were measured to compare LRU, FPB, and APACS cache managment strategies. Fig. 10 shows that in every case, APACS outperforms LRU.

V. Results Analysis

Table III shows the prefetch hit ratios recorded for each trace during simulations to show, to some extent, the degree of system call redundancy. Alternatively, when applications exhibit infrequent system call recurrence, APACS does not mask LRU performance benefits because dynamic cache partitioning inhibits performance degradation caused by an ineffective P_{buff}. Also, as the size of P_{buff} decreases, the minimum chance threshold increases to ensure the best possible use of the limited prefetch buffer space. These reasons combined make APACS a reliable alternative to LRU because it is well suited for changing I/O workloads.

In most cases, when cache sizes are smaller, a larger disparity exists between LRU and APACS performance. LRU is less effective when a small number of files can be cached. Prefetching files makes the most sense in these situations because a more informed hypothesis about future file requirements allows the limited file cache to be used more effectively. Using Sphinx-3 as an example, 0.5 MB of cache was too

TABLE III
TRACE DATA SIZES AND PREFETCH HIT RATIOS

Trace	P_{buff} Hit Ratio with Varying Cache Sizes (mb)					
	4	5	6	7	8	9
wc	35.5	45.6	45.6	45.6	45.6	45.6
grep	9.52	9.62	9.72	10.1	10.5	10.5
Sphinx	49.8	54.6	47.2	56.8	58.5	62.7
Linux Kernel	47.2	43.2	46.5	46.6	48.9	44.6
Workstation 1	54.5	54.4	69.6	71.3	76.9	70.3
Workstation 2	49.8	54.6	47.2	56.8	58.5	62.7
Workstation 3	62.7	61.2	83.6	69.8	67.2	71.2
Workstation 6	57.8	57.4	56.4	40.8	60.2	59.3

small for LRU because the application's working set was larger than the cache size. Every time a recurring block of system calls appeared, these files were already replaced by new file requests. APACS predicted the recurrence of these requests and consequently, drastically improved efficiency of cache memory.

To the contrary, when large cache sizes were used, APACS still surpassed the LRU policy. A diverse application working set with large files incapacitates LRU because the working set may be too large for any size LRU cache to be effective.

VI. CONCLUSIONS AND FUTURE WORK

Extensive experiments were conducted to measure the performance improvements gained by the proposed APACS compared to the traditional LRU and prefetching methods. In order to validate APACS efficacy over a multitude of I/O access patterns, several real-world traces were used to represent real I/O benchmarks and applications. In all experiments, APACS performed measurably better than the alternative strategies. The prefetch pipelining module of APACS reduces the stall time associated with each disk access by parallelizing the prefetching task at times that are most beneficial. Prefetch buffers are allocated dynamically according to an optimization analysis using the cost-benefit model. Interestingly, in many cases when the prefetch hit ratio is negligible, APACS still outperforms LRU. This happens because many of the files that LRU flushes from the cache remain in the prefetch buffer.

In the case of the Fixed Prefetch Block scheme or FPB, APACS does not assume LRU or prefetch cache replacement policy is more effective at any point in time. APACS, instead, uses the hit ratios of both buffers to determine which replacement algorithm in which to give more weight. Fig. 5 illustrates how APACS attempts to adjust the hit ratio of prefetch buffer P_{buff} dynamically to compensate for a descending hit ratio of cache buffer C_{buff}.

In the future, we will implement a mechanism to dynamically optimize the lookahead window based on the hit ratio of P_{buff} and the average inter-access CPU time. In this way, arbitrary values are not set based on simulation results. Rather, all the parameters will self regulate based on I/O workload conditions at hand. Even further, we plan to integrate machine learning algorithms with predictive prefetching. This integration approach would provide a more general approach, instead of interlinking several distinct optimization algorithms.

ACKNOWLEDGMENT

This work was supported by the U.S. National Science Foundation under Grants CCF-0845257 (CAREER), CNS-0917137 (CSR), CNS-0757778 (CSR), CCF-0742187 (CPA), CNS-0831502 (CyberTrust), CNS-0851960(REU), CNS-0855251 (CRI), OCI-0753305 (CI-TEAM), DUE-0837341 (CCLI), and DUE-0830831 (SFS), as well as Auburn University under a startup grant and a gift (Number 2005-04-070) from the Intel Corporation.

REFERENCES

[1] Butt, A.R., Gniady, C., Hu, Y.C., *The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms*, IEEE Transactions on Computers, Jul. 2007, vol. 56, no. 7., pp. 889-908.

[2] Cao, P., Felten, E.W., Karlin, A.R., Li, K., *A Study of Integrated Prefetching and Caching Strategies*, *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, Ontario, Canada, May 1995, vol. 23, no. 1, pp. 188-197.

[3] Chen, Y., Byna, S., Sun, X., *Data Access History Cache and Associated Data Prefetching Mechanisms*, *Proceedings fo the AMC/IEEE Conference on Supercomputing*, Reno, NV, Nov, 2007, pp. 1-12.

[4] Domenech, J., Sahuquillo, J., Gil, J.A., Pont, A., *The Impact of the Web Prefetching Architecture on the Limits of Reducing Userâs Perceived Latency*, IEEE/WIC/ACM International Conference on Web Intelligence, Hong Kong, China, Dec. 2006, 740-744.

[5] Griffioen, J., Appleton, R., *Reducing File System Latency using a Predictive Approach*, Proc. of the USENIX Summer 1994 Technical Conference, vol. 1, 1994.

[6] Jeon, H.S., *Practical Buffer Cache Management Scheme Based on Simple Prefetching*, IEEE Transactions on Consumer Electronics, Aug. 2006, vol. 52, no. 3.

[7] Jeon, J., Lee, G., Cho, H., Ahn, B., *A Prefetching Web Caching Method Using Adaptive Search Patterns*, 2003 IEEE Pacific Rim Conference On Communications, Computers, And Signal Processing, Aug. 2003, vol. 1, pp. 37-40.

[8] Nanopoulos, A., Katsaros, D., Manolopoulos, Y., *A Data Mining Algorithm for Generalized Web Prefetching*, IEEE Transactions on Knowledge and Data Engineering, Sep. 2003, vol. 15, no. 5.

[9] Patterson, H.R., Gibson, G.A., Ginting, E., Stodolsky, D., Zelenka, J., *Informed Prefetching and Caching*, Proc. of the 15th ACM Symp. on Operating System Principles, Copper Mountain Resort, CO, Dec. 3-6, 1995, pp. 79-95.

[10] Phalke, V., Gopinath, B., *Compression-Based Program Characterization for Improving Cache Memory Performance*, IEEE Transactions on Computers, Nov. 1997, vol. 46, no. 11., pp. 1174-1186.

[11] Reungsang, P., Park, S.K., Jeong, S., Roh, H., Lee G., *Reducing Cache Pollution of Prefetching in a Small Data Cache*, *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, Austin, TX, Sep 23-26, 2001, pp. 0530.

[12] Varadan, S.V., Vaishnav, K.R., *Application of Neural Networks in Predictive Prefetching*, Networking, Sensing, and Controlling, Mar. 2005, pp. 252-255.

[13] Vellanki, V., Chervenak, A.L., *A Cost-Benefit Scheme for High Performance Predictive Prefetching*, *Proceedings of the AMC/IEEE SC99 Conference*, 1999.

[14] Wang, J.Y.Q, Ong, J.S., Coady,Y., Feeley, M.J., *Using Idle Workstations to Implement Predictive Prefetching*, *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, PA, Aug. 2000, pp. 87-94.

[15] Yang, Q., Zhang, Z., *Model based Predictive Prefetching*, *Proceedings of the 12th International Workshop on Database and Expert Systems Applications*, Munich, Germany, Aug. 2002, pp. 291-295.

[16] Yeh, T., Long, D.D.E., Brandt, B., *Increasing Predictive Accuracy by Prefetching Multiple Program and User Specific Files*, *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications*, 2002, pp. 12-19.

[17] Zhuang, X., Lee, H.S., *Reducing Cache Pollution via Dynamic Data Prefetch Filtering*, IEEE Transactions on Computers, Baltimore, MD, Jan. 2007, vol. 56, no. 1, pp. 18-31.