Allocation Policies for Application-Controlled File Cache Management

Pei Cao
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706
cao@cs.wisc.edu

Abstract

Application control on file cache management promises to improve file I/O performance dramatically by exploiting user-level knowledge. A file cache allocation policy is essential in order for kernel to support application control. This paper presents the design, validation and refinement of the LRU-SP allocation policy, which guarantees fair usage of resources and protects system performance. The paper highlights three key aspects in successfully applying theoretical analysis to system problems: proper abstraction, experimental validation, and simple solutions.

1 Introduction

As disk performance continues to lag behind microprocessor performance, file cache management becomes increasingly important for file system performance. Application-controlled cache management promises to improve file caching and prefetching through policies tailored to individual applications' needs [7, 6]. In order to allow applications to control file cache, operating systems must have an allocation policy to decide how the physical memory is allocated to competing processes.

The complication in designing an allocation policy is that applications may be either "smart" or "stupid". In other words, compared with the default replacement policy that the kernel chooses for every application (which is usually LRU), an application might use a policy that performs much better, or it might make mistakes about its own file access behavior and use a policy that performs much poorer. The operating system must be prepared to deal with either situation. The memory allocation algorithm not only needs to allow "smart" applications to improve their performance, but also needs to protect system performance from "stupid" applications.

This paper describes the design and evolution of one allocation algorithm, LRU with Swapping and Placeholder (LRU-SP). The design of LRU-SP started from an abstract model capturing the desired properties of allocation algorithms [3], proceeded with an algorithm that is proven to satisfy the properties, and continued through implementation and experimentation [4]. The experiments revealed that although theoretical predictions do match most experimental results, there are important aspects of system behaviors that are missing in the abstract model. The discovery then led to a refinement of both the model and the algorithm. Partly because the interaction between theoretical modelling and system experiments has enabled a much better understanding of the behavior of LRU-SP. LRU-SP has been employed in a variety of environments, including application-directed prefetching in single disk [2] and multi-disk systems [8], and adaptive physical memory management in modern virtual memory systems [5].

2 Establishing a Model

How do we characterize the above vaguely stated properties that an allocation algorithm should satisfy? We need a model of behavior for multiple processes' file accesses. To start, let us assume that there is a sequence $\omega = (r_1, r_2, \ldots, r_n)$ of accesses to file blocks (files are most commonly cached by blocks, i.e. 8K byte of file data), and the sequence is an interleaving of accesses from p processes $P_1, P_2, \ldots, P_p$,. Furthermore, let us assume that each block appears in exactly one process's accesses, and each file block belongs to exactly one process and is accessed only by that process. Shared files are considered belonging to a process that represents the collection of shared file accesses from the sharing processes. We assume that the sequence is given *a priori* and is not affected by the replacement decisions made. The file cache can hold a total of k file blocks $((k < n)$.

In a traditional operating system, which does not allow applications to control their own file cache, the file cache is often managed by applying global LRU replacement on the whole access sequence $\omega = (r_1, r_2, \ldots, r_n)$. This means that each application's cache is managed by an LRU policy.

In a system that supports application control on file cache management, the paradigm is that the kernel keeps certain global data structures about cache blocks, and decides on which process to ask for a cache block when a replacement is necessary. The process then decides on which of its block to replace. The kernel gives the process a default choice for replacement, but the process is free to make its own decision. Therefore, there are three types of processes: the "oblivious" ones which let the operating system use the default LRU policy to manage the memory for them, the "smart" ones that use policies better than LRU, and the "stupid" ones that use policies worse than LRU.

To capture the concept of "better" or "worse", we introduce the following notion: whenever a process P makes a replacement, if its replacement choice would be referenced later in the future than the default choice by the operating system, process P is said to have made a wise decision; if its replacement choice would be referenced earlier than the default choice by the operating system, it is said to have made a foolish decision. The default choice is essentially the replacement choice that would be made by LRU if the process always uses LRU as the replacement policy.

Equipped with the above model, we can define the fairness conditions that an allocation algorithm should satisfy:

1. *A process that always replaces the default choice made by the kernel does not suffer more file cache misses than it would under global LRU.*

2. *A foolish decision by one process never causes another process to suffer more misses than it would suffer under global LRU.*

3. *A wise decision by one process never causes any process, including itself, to suffer more misses than it would suffer under global LRU.*

Essentially, the three conditions reflect the requirement that application-controlled file cache management should not make system performance worse than traditional operating systems, unless applications make mistakes themselves.

3 Designing an Algorithm

In the above model, the allocation algorithm is essentially the algorithm with which the kernel decides on which process to replace a block when some block needs to be replaced. Our goal is to find an algorithm that can satisfy the fairness conditions.

We start from the simplest solution, that is, letting kernel use the global LRU list to decide which process to replace a block. For example, if block a is the least recently used block, and a belongs to process P, then the kernel asks process P to evict a block. However, examination of this simple algorithm quickly reveals that it does not satisfy the fairness condition. Rather, two modifications to the global LRU list, Swapping and Placeholders, are necessary.

Swapping means that if process P decides to replace block b, the kernel swaps the positions of a and b in the global "LRU" list. A placeholder is a record $\{b, a\}$, used to remember the fact that block b is replaced when block a is at the least-recently-used end of the global list. A placeholder $\{b, a\}$ is "built for b and pointing to a".

The rule of maintaining placeholders is the following. Suppose reference r to block a hits in the file cache, then a is moved to the mostly-recently-used end of the global "LRU" list. If there is a placeholder pointing to a, for example, $\{b, a\}$, the placeholder is deleted.

If the reference misses, then there are two cases. In the first case, there is a place-holder for a, pointing to t; in this case t is replaced and its page is given to a.

In the second case, there is no place-holder for a. In this case, the kernel finds the block at the least-recently-used end of the global "LRU" list. Say that block c, belonging to process P, is at the least-recently-used end of the LRU list. The kernel tells P to choose a block to replace, and tells P that its default choice is c. Suppose P's choice is to replace block x. If c and x are different, then the kernel swaps x and c in the LRU list. If there is place-holder pointing to x, it is changed to point to c; otherwise a place-holder is built for x, pointing to c (i.e. $\{x, c\}$). Finally, x's page is given to a, and x is written to disk if it is dirty.

In both cases, a moves to the most recently used end of the global "LRU" list after the reference.

Theorem 1 *Both swapping and placeholders are necessary for LRU-SP to meet the fairness conditions.*

Proof. The proof is by examples showing when the fairness conditions are violated if swapping or placeholders are not employed. Details can be found in [3]. ■

The following theorem shows that swapping and placeholders are also sufficient for LRU-SP to meet the fairness conditions.

Let $F_{global-LRU}(P_i, t)$ be the number of cache misses suffered by process P_i during $(r_1, r_2, \ldots, r_t)$, under global LRU replacement policy, and let $F_{LRU-SP}(P_i, Alg_i, t)$ be P_i's misses under the system with LRU-SP as allocation policy and Alg_i as P_i's replacement policy. In addition, let $M(P_i, Alg_i, t)$ be the number of mistakes Alg_i made that are caught by placeholders during $(r_1, r_2, \ldots, r_t)$, in other words, the number of times when it was discovered that Alg_i's replacement choice was referenced earlier than the default choice by the operating system.

Theorem 2 *Given any reference sequence $\omega = (r_1, r_2, \ldots, r_n)$, for every process P_i and any replacement algorithm Alg_i it uses, its number of misses under application-controlled cache management with LRU-SP allocation is no more than the misses under the global LRU replacement policy plus Alg_i's mistakes. In other words, for any $\omega = (r_1, r_2, \ldots, r_n)$,*

$$F_{LRU-SP}(P_i, Alg_i, n) \leq F_{global-LRU}(P_i, n) + M(P_i, Alg_i, n).$$

It is trivial to see that it is a corollary of Theorem 2 that LRU-SP satisfies the fairness conditions.

The intuition behind the proof of the theorem is that the LRU-SP list can be augmented to establish a one-to-one correspondence with the LRU list. Let $[g_1(t), g_2(t), \ldots, g_k(t)]$ be the global LRU policy's LRU list of blocks, at r_t of ω. In other words, $g_i(t)$ is the block in the ith position in the LRU list, with $g_1(t)$ being the most recently used block, and $g_k(t)$ being the least recently used.

Similarly, let $[a_1(t), a_2(t), \ldots, a_k(t)]$ be the global list under LRU-SP, as described in the previous section. If $a_i(t)$ has a placeholder pointing at it for another block $h_i(t)$, we mark $a_i(t)$ as $a_i(t)\langle h_i(t)\rangle$. Thus, the list may look like $[a_1(t), \ldots, a_{i-1}(t), a_i(t)\langle h_i(t)\rangle, a_{i+1}(t), \ldots, a_k(t)]$. Let's call it the LRU-SP list.

For the proof we need to augment the LRU-SP list. Instead of deleting the placeholder $\langle h_i(t)\rangle$ when $a_i(t)$ is accessed, we leave $\langle h_i(t)\rangle$ in the same position in the list, and label it as a virtual block. In other words, if the list is

$$[a_1(t), \ldots, a_{i-1}(t), a_i(t)\langle h_i(t)\rangle, a_{i+1}(t), \ldots, a_k(t)],$$

and $r_t = a_i(t)$, then the new list becomes

$$[a_i(t), a_1(t), \ldots, a_{i-1}(t), \langle h_i(t)\rangle, a_{i+1}(t), \ldots, a_k(t)].$$

Thus, virtual blocks are old placeholders that no longer point at any real block but stay in the list. The list can now grow beyond k entries.

As the list changes according to the LRU-SP rule, a virtual block stays in the same position as if it were a real block. It is deleted from the list either when it is hit by a reference (i.e., $r_t = h_i(t)$ and $\langle h_i(t)\rangle$ is a virtual block), or when it reaches beyond the first k positions of the list (i.e., $\langle h_i(t)\rangle$ with $i > k$). Obviously, if at reference r_t there are $N_v(t)$ virtual blocks, then the augmented list has $k + N_v(t)$ elements.

Note that the block at the LRU (right) end of the augmented LRU-SP list and that at the LRU end of the original LRU-SP list are always the same, because a virtual block can never reach the LRU end without being deleted first. Therefore, the LRU end of the augmented list is always the candidate replacement block in LRU-SP.

The following lemma shows that at any reference r_t of ω, if we take the first k elements from the augmented list, and substitute every $a_i(t)\langle h_i(t)\rangle$ by $h_i(t)$, and every $\langle h_j(t)\rangle$ by $h_j(t)$, then we obtain exactly the global LRU policy's list $[g_1(t), g_2(t), \ldots, g_k(t)]$.

Lemma 1 (Correspondence)
Given $\omega = (r_1, r_2, \ldots, r_n)$, at reference r_t, let $G(t) = [g_1(t), g_2(t), \ldots, g_k(t)]$ be the list of the global LRU replacement policy, and let

$$A(t) = [a_1(t), \ldots, a_i(t)\langle h_i(t)\rangle, \ldots, h_j(t), \ldots, a_m(t)], m \geq k$$

be the augmented LRU-SP list. Let $A_{1-k}(t)$ be the first k elements of $A(t)$, and let H be the operation that substitutes every $a_i(t)\langle h_i(t)\rangle$ by $h_i(t)$, and every $\langle h_j(t)\rangle$ by $h_j(t)$. Then at every r_t, $1 \leq t \leq n$,

$$H(A_{1-k}(t)) = G(t). \tag{1}$$

The proof is essentially by case study and inductions. The details are in [1]. The techniques used in proving Lemma 1 also prove the following lemma.

Lemma 2 *At every reference r_t of ω, let $N_v(P_i, Alg_i, t)$ be the number of virtual blocks in $A(t)$ that are in P_i's access sequence. Then for all i,*

$$F_{LRU-SP}(P_i, Alg_i, t) + N_v(P_i, Alg_i, t)$$
$$\leq F_{global-LRU}(P_i, n) + M(P_i, Alg_i, n).$$

Since $N_v(P_i, Alg_i, t) \geq 0$ for every t, this completes the proof of Theorem 2. ∎

4 Experimental Validation

To test whether LRU-SP indeed performs well in real systems, we implemented the algorithm in the Ultrix 4.3 kernel and designed a series of experiments using a collection of I/O intensive applications [4]. The experiments showed that a simple-minded allocation policy, which does not satisfy the fairness conditions, can indeed hurt system performance even when all applications use smart replacement policies. In contrast, LRU-SP improves system performance by up to 45% when applications use good replacement policies.

However, in our test LRU-SP failed to prevent foolish processes from degrading the performance of other processes. The test run a source-analysis tool (cs2), a text-retrieval application (gli), the link-editor (ldk), and a hardware cache simulator (din), each with a program read300, both when read300 is oblivious (using an LRU replacement policy) and when read300 is foolish (using an MRU replacement policy). The characteristics of read300 is such that MRU replacement policy will generate a lot more cache misses than LRU. Table 1 shows the effect of read300 on both the elapsed time and number of block I/Os of the applications.

The data show that there are degradations in both the number of block I/Os and the elapsed time. This contradicts results from the theoretical analysis. The reasons include the following:

- Foolish processes generate more I/O requests, and thus put a heavier load on the disk. This leads to longer queueing time for the disk, so non-foolish processes' I/O requests wait longer to be served.
- Foolish processes take longer to finish and therefore occupy cache blocks for a longer time. Processes that run concurrently with them and finish after them normally get an increase in available cache blocks after they are finished. If they finish later, this increase comes later.

Essentially, the experiments point out two aspects of real systems that the theoretical model fails to take into account. First, the model only considers numbers of file cache misses, while the running time of an application also depends on how long a cache miss can take, which includes the queueing delay at the disk. Second, a major assumption in the model is that the interleave of references from processes is unaffected by file cache misses or replacement algorithms. In practice, operating systems perform context switch at disk I/Os, which means that a process that often misses in file cache lags more and more behind other processes. The second observation above illustrates this effect. The foolish process is effectively slowed down by its replacement policy, while LRU-SP still allocates its due share of memory. Thus, compared to when the read300 process is not foolish, other processes get their memory increase at a later time.

To accommodate these facts, we refined our model to include queueing delays at the disk, and the context switches at the time when a process misses in its file cache and when a process's disk I/O finishes. Doing so significantly complicates the model and the analysis. In fact, it is not even clear whether there exists an allocation algorithm that satisfies the fairness conditions under the new model.

Instead of trying to design an algorithm that can satisfy the new fairness conditions, we took a simpler approach. Noticing that these aspects only took effect for foolish processes, we changed the role of placeholders to be detectors of foolish processes: if a process makes mistakes more than 50% of the time, the kernel simply declares the process unfit to manage its own cache and takes over the control. The simple solution effectively limits the damage an errant application can do to system performance.

5 Discussions

Swapping and placeholder techniques are not limited to LRU-based systems. In fact, they can be generalized to any default replacement algorithm and turn it into an allocation algorithm satisfying the fairness conditions. In particular, the framework applies to various algorithms that approximate LRU, such as those used in virtual memory systems. Thus, the techniques presented here solve the allocation problem in application-controlled physical memory management as well. Placeholders are also used in adaptive cache management, in which the kernel observes application behavior and adapts the replacement policy. Placeholders provide a mechanism to detect mistakes and back off wrong adaptations.

The design process of LRU-SP demonstrates that theoretical studies can help system design significantly, and experimental validation is necessary to explore both the benefits and limitations of theoretical studies. In particular, three elements are most important in a successful application of theoretical analysis to system problems:

- *proper abstraction*: understanding the first-order goals and first-order factors in the system, and coming up with a simplification that makes the problem tractable. The initial abstract model used in the design of LRU-SP is simplistic. How-

"Read300"	Elapsed Time				Block I/Os			
Policy	din	cs2	gli	ldk	din	cs2	gli	ldk
Oblivious	155	225	156	112	3067	9760	9086	5201
Foolish	202	339	261	208	3495	10542	9759	5374

Table 1: Effect of a foolish process on other applications.

ever, it does capture the potential effects of individual processes' replacement decisions on allocation. The simple model made the allocation policy problem tractable: we found a solution that satisfies the fairness conditions. Had we started with the refined model, we would not be able to find a solution.

- *experimental validations* are extremely important in discovering hidden assumptions in a model and refining it to more accurately reflect real systems. This often leads to new research problems, and enables better understanding of the behavior of target algorithms.

- *simple solutions.* Operating system software must have very low overhead and very high reliability. These requirements mandate simple solutions to any design question; complicated solutions rarely get implemented or deployed in practice. For example, though the refined LRU-SP algorithm does not strictly satisfy the new fairness conditions, its simplicity is attractive and the algorithm suffices in practice.

Theoretical analysis on traditionally system problems is an exciting new area. System problems need more theoretical modelling and analyzing, and successful studies do influence system design. We hope that this paper, by providing a case study, encourages more efforts in this area.

References

[1] Pei Cao. *Application-Controlled File Caching and Prefetching.* PhD thesis, Princeton University, December 1995. Also published as technical report CS TR-522-95.

[2] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching and disk scheduling. In *TOCS*, November 1996.

[3] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In *Proc. USENIX Summer 1994 Technical Conference*, pages 171–182, June 1994.

[4] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Proc. First USENIX Symposium on Operating Systems Design and Implementation*, pages 165–178, November 1994.

[5] Gideon Glass. Adaptive page replacement. Technical report, Master's Thesis, University of Wisconsin-Madison, 1997, May 1997.

[6] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page-cache management. In *The 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[7] Michael Stonebraker. Virtual memory transaction processing. *Operating Systems Review*, 18(2):8–16, April 1984.

[8] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. In *SIGMETRICS'97*, June 1997.