Application-Specific File Caching for Unmodified Applications and Kernels

Omri Traub, Alex Lloyd, Jonathan Ledlie

Abstract

The policies chosen for the kernel's file cache (eviction, read ahead, write behind) are crucial for the performance of many modern applications. But as we show, the kernel's generic policies do not perform well for a compiler's pattern of phases of reads, computation and then writes, leading to excessive blocking. Short of making the kernel extensible by accepting application-specific policies, no quick fix would allow the kernel to address the needs of a variety of applications.

We propose and implement a File System Agent, containing a shared file cache and pluggable policy component that interposes between an application and the operating system, intercepting file I/O operations. This agent is dynamically linked into participating applications through the use of a binary rewriting toolkit. We present our FS Agent with a policy module aimed at the compilation domain.

1 Introduction

Balancing the often-contradictory needs of particular applications while at the same time keeping the programming interface simple has been an overarching challenge in operating system design. Stonebraker, for example, provided evidence that operating systems provide poor support for database applications [12]. A number of systems such as Scheduler Activations [1], application hints into a modified kernel [6], and compiler-driven solutions [4] have tried to address this problem by modifying the operating system, the compiler, the application, or all of the above.

A recent development in this arena is the gray-box approach which treats operating systems neither as closed and impenetrable "black-boxes" nor as open and malleable "white-boxes" [2, 5, 8]. Instead, this approach shows that individual applications can discover which algorithms are in use and leverage that knowledge. By modifying applications to selectively probe and guess operating system-specific characteristics like buffer cache organization and file system layout, Arpaci-Dusseau *et al.* have demonstrated that significant application performance gains are achievable simply by reacting to what the operating system is already doing and assuming this is indicative of future behavior. One implementation involved modifying `grep` to use a file content cache detector to re-order the files scanned in a multi-file search and achieved a three-fold speedup on a scan of 100 10-MB files [2]. Other work modified a web server to probe memory contents to re-order requests [5]. To use a gray-box layer, applications are re-written to query a special linked library.

Gray-box techniques have several significant deficiencies. First, they require modifications to each application incorporating gray-box supplied guesses. Second, although they may allow applications to gain an approximate picture of the algorithms the operating system is using, the data currently in memory and the disk-block organization, they provide no mechanism to change undesired policies. Third, gray-box techniques focus on a single process learning the state of the operating system and do not support groups of cooperating processes, as should be the case in server systems.

We propose an alternative to gray-box techniques for addressing application-specific needs. By capturing an application's system calls at runtime, we introduce an intelligent layer between an application and the operating system and provide per-application and per-OS specialization without modifying either the applications or the operating system. This application-level agent contains a policy tailored to the specific needs of the application. Our goal in this paper is to examine the design and performance of this alternative architecture.

We focus on the compilation domain in our current implementation. Compilers involve several cooperating processes, follow unusual patterns of being alternately CPU-bound and I/O-bound, write some short-lived and some long-lived files, and read many short files (headers) that are often scattered over the disk. Compilers have well-defined and repeatable behavior, much of which is bungled by the operating system and little of which could be ameliorated by probing the operating system using gray-box techniques. Short of making the kernel extensible by accepting application-specific policies, no quick fix to the kernel would allow it to address the needs of a variety of applications. The specific case of a compiler could be generalized to many complex applications with predictable behavior ill-served by kernel caching policies.

This paper examines the introduction of a file system agent between the processes that constitute a compiler application and the operating system in the Win32 environment. The runtime binary rewriting library toolkit that redirects system calls to the agent was developed in previous work, but a description is included for completeness. We primarily focus on the use of the agent as an application-driven shared cache. We discuss a distinct "policy" component which directs cache management and which is replaceable for use with other applications. We examine a "lazy writer" thread that cooperates with the policy component to asynchronously clean the cache in an application-specific way. Experimentally, we compare the relative performance of compilation with and without the interposed file system agent.

The rest of the paper is organized as follows. Section 2 describes the problems with existing operating system policies for the compilation process and makes the case for an agent in a compiler application. Section 3 gives a high-level overview of the design and implementation and includes a discussion of some of the more interesting details. Section 4 presents our experiments and results. Section 5 surveys related work and Section 6 concludes.

2 A Case for Compiler-Specific File Caching Policies

This section highlights some of the shortcomings of running compilation of large software projects on general-purpose operating systems. On many operating systems, the buffer cache attempts to minimize application-time waiting for I/O while simultaneously pursuing an equitable memory management policy. To this end, the operating system engages in read-ahead and write-behind of file data. As we will show, the generalized implementations of these policies are often ill-suited to the compiler application domain.

2.1 Intra-file Read Prediction

Though some compiler I/O is composed of large private meta-data files that benefit enormously from linear intra-file read-ahead, the majority of compiler reads consist of small and numerous header files, both in terms of bytes and number of accesses. Especially in C++ applications using external frameworks with thousands of headers, a large portion of I/O operations is reads that will be unaided by standard read-ahead. When the compiler is invoked on a source file with many header dependencies, the system will need to wait for one disk access per header. (Subsequent reads of the header will be successfully served from the cache.)[1] A system that monitors and learns the compiler's access patterns can issue speculative reads for a large group of related input files, dramatically increasing the chance the files will already be in the cache upon the compiler's first access.

2.2 Globally-optimized Write Behind

Under normal circumstances, the system as a whole experiences a continuous mixture of file reads, writes, and other memory allocation. Writes are usually appended to a "lazy-writer" queue and

[1]We have observed one compiler under Windows 2000 waiting from 5 to 30 milliseconds on the first access of each header file, and on the order of only 500 microseconds for accesses thereafter.

the application is allowed to proceed immediately. If an application issues a large number of writes, many operating systems will limit the number of dirty pages the cache is allowed to contain. If one application was allowed to consume the entire cache with its dirty data, other applications would be forced to block on any memory allocation while that data was flushed to disk. To prevent this, many operating systems (*e.g.,* Windows) will steadily flush lazy-writes in the background but prevent any further writes from proceeding until the dirty pages have been trimmed to an acceptable level.

2.3 A Bottleneck

Typical compiler behavior consists of reading a large set of input files, performing a period of CPU-bound computation, and then writing out object files and any compiler-specific meta-data. In the compilation of a large project, successive invocations of a compiler for each source file perform this pattern such that the last set of writes is being lazy-written while the next set of reads is being attempted. These factors combine to produce a disk bottleneck, where the CPU is idle and the compiler is waiting for the disk.

Figure 1 illustrates two segments of a compilation of a large commercial software application dependent on several C++ toolkits, compiled using the Microsoft 32-bit Optimizing Compiler Version 12.00. We chose to test-compile a large commercial software application dependent on several C++ toolkits to demonstrate access patterns common to those of modern Windows development. The two graphs show two instances of the compiler blocking (once on reads and once on writes) due to poor policy decisions by the operating system. An application-specific policy, such as the one we implement, should prevent these operations from blocking.

The blocking reads graph shows a period of 100% CPU utilization while the compiler completes one source file. The then CPU hovers around only 40% while the compiler spends time blocked in the Win32 `ReadFile` function. Here the lack of inter-file read prediction prevents the predictive execution of many file page-ins, while the continued execution

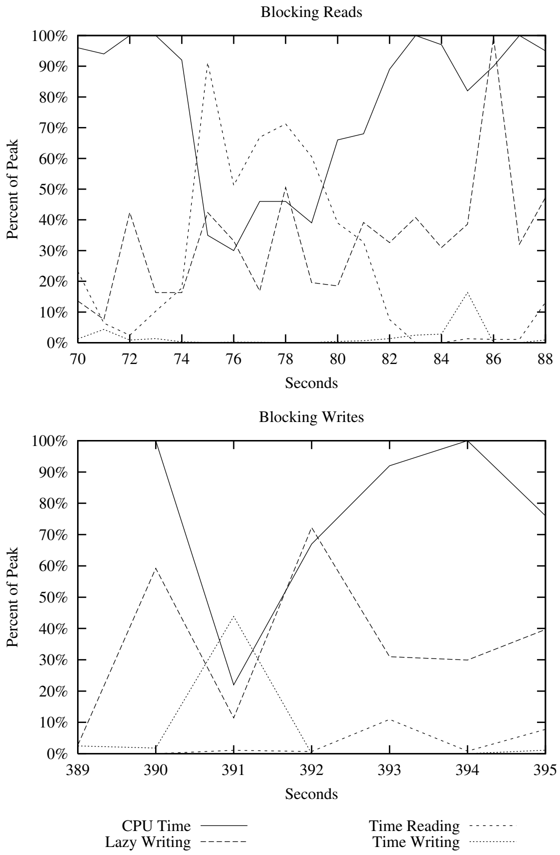

Figure 1: A graph showing CPU and I/O activity during a subset of a large compilation. Time reading and writing indicates time spent by the application in the `ReadFile` and `WriteFile` system calls. Lazy writes are those performed in the background.

of the lazy writer adds unnecessary competition for disk I/O.

The blocking write graph shows again a period of 100% CPU utilization while the compiler completes its previous compilation. The CPU immediately drops to 20% utilization while time spent in the Win32 `WriteFile` API spikes dramatically. Here, the cache's dirty threshold has been reached, and the compiler is forced to block while dirty pages are flushed. Note that an increase in CPU utilization and decrease in time blocking in the WriteFile API follow the brief spike in lazy-writer activity that is freeing up the cache. A compilation-aware cache policy that allows dirty pages to accumulate beyond the standard threshold can avoid this wasted

CPU time and again safely assume that the compiler would soon idle the disk long enough to aggressively flush dirty pages.

The graphs zoom in on two phases during a longer run of `make` during most of which CPU utilization was kept at around 100%. CPU time and lazy writing activity were sampled with Windows 2000's performance counters. The read and write activity was computed from our file I/O hooks (described in the following section) and used a high resolution timer available on Windows.

This section pointed out a bottleneck that could be fixed with application-specific information but is too specific to be repaired by changing the operating system's default behavior. Changing the dirty threshold would assist compilation but would hurt other applications, like file-system based databases, or cases when many applications were trying to share the kernel cache. Also, no change to the operating system's intra-file read ahead would help it prefetch the inter-file dependent header files extant in compilation. To achieve these behaviors without modifying the application, we advocate the use of a File System Agent.

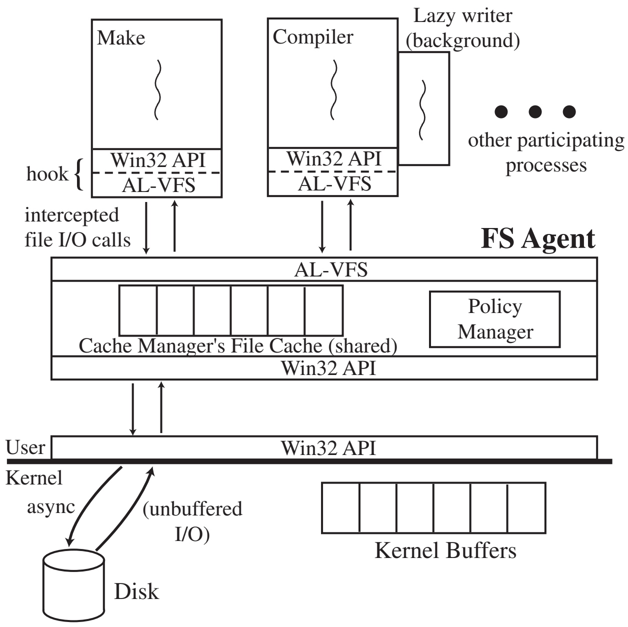

Figure 2: Architecture overview. Participating applications have been rewritten to call into the FS Agent, which is mapped into process memory. Squiggly lines denote threads of control. The Lazy Writer runs in the background and is a thread of the "dominant" process (see Section 3.1.2). The AL-VFS interface is discussed below.

3 File System Agent Design

This section describes the design and implementation of the FS Agent. At a high level, the FS Agent intercepts file I/O system calls and serves them using a shared-memory cache, all within the address space of the running application. Figure 2 illustrates the main components. Section 3.1 describes the architecture design in more detail while Section 3.2 describes some relevant implementation details.

3.1 Architecture Design

Two design decisions result from of having an application-level architecture for a file cache: execution occurs mainly in application threads and shared state is maintained in user-level shared memory.

Unlike a traditional approach where applications trap into the kernel to execute file system code, in our system application threads are the direct clients of the file system API. Through the use of binary rewriting, an application is dynamically linked to a code library that satisfies file system requests. As a result, file-system code is run in the context of each active application thread. Shared state is locked to prevent race conditions in the file-system library. The single non-application thread is the Lazy Writer, which manages the background flushing of dirty cache content to disk.

The state maintained by the FS Agent, both for file contents and other data structures, resides in a shared region of memory that is mapped into the address spaces of participating processes (*e.g.,* shell, make, compiler, linker, editor). This state is initialized when the first participating process starts and uninitialized when the last participating process exits. By using a participating shell process to run the compilation tasks, the FS Agent persists between multiple builds.

To connect file I/O requests made through the Win32 system call API to the FS Agent's cache, we use *hook* functions to intercept the appropriate system call. These hooks translate the system calls into calls on the AL-VFS (Application Level Virtual File System) interfaces, modeled after the VFS layer in the Unix kernel. The Cache Manager, discussed below, implements the AL-VFS interfaces, thus providing a clean separation between caching functionality and the use interposing hooks.

3.1.1 Cache Manager

The Cache Manager is the component in charge of managing the shared cache and serving application requests out of it. Optimistic, lightweight locking is used to guard against multiple application threads trying to manipulate shared state concurrently.

Figure 3 shows the data structures used to implement the shared file cache. The process-specific data structures are allocated by the FS Agent out of the application's default heap using `malloc` and `free`. The shared data structures are allocated out of the shared memory region by a custom allocator (locking and memory allocation are further discussed in Section 3.2). We illustrate the function of the various data structures by walking through the sequence of steps used to satisfy a read request.

The application records descriptors supplied by the file-system library when files are opened. (in this example, the file descriptor is the integer value `0xbeef`). When the application passes this descriptor to the read call, the file I/O hooks recognize it as belonging to a reserved range. The file descriptor table is then consulted to resolve the file descriptor value to an AL-VFS object for the file or directory in question.

The AL-VFS objects for files and directories implement a uniform interface that allows the hooks to be isolated from the details of the cache. Internally, the caching AL-VFS objects maintain the path in the file system to the file or directory they represent. File objects also keep track of the current offset in the file. There is a one-to-one correspondence between AL-VFS objects and open file descriptors.

The code in the AL-VFS file object's `read` method uses the stored path to the file to make file data requests from the Cache Manager. The Cache Manager keeps a hash table mapping file paths to `FileInfo` objects. This hash table is kept in shared memory and acts as an index into the contents of the cache. If an entry for a given file does not exist (because of an eviction for example), the Cache Manager creates one and adds it to the hash table. It then reads the data from disk. The `FileInfo` class contains various meta-data about the file (size, modification times), its current state in the cache (empty, read, or dirty), and a pointer to a chain of buffer descriptors for its contents.

The contents of a file can be separated across multiple variable-sized buffers. Each buffer descriptor contains information about the file extent (starting position and size) contained in a single data buffer as well as a pointer to the data buffer itself. To satisfy a read request, the Cache Manager traverses the chain of buffer descriptors to find the one or more extents that contain the data that was requested.

In order to reduce the complexity of the file cache, we chose to manage buffers at the granularity of whole files. This means that when a file is read, its entire content is brought into the cache, and on eviction, the entire content is flushed. This is appropriate for our first target applications, compilers, which typically read and write entire files. Due to this focus on whole files, we use the variable-sized buffer model similar to program heaps instead of the fixed-size buffer model used in disks and traditional kernel buffer caches. When a file is read, we try to find a buffer large enough for the entire file using a best-fit algorithm. If the cache becomes fragmented, the file may still fit in the cache without evictions but will require multiple buffers to be stored.

While many requests can be served from the cache, the Cache Manager will need to eventually contact the operating system for file I/O. We use asynchronous I/O calls exclusively in the Cache Manager. Since buffers are kept aligned to page boundaries, we are able to use asynchronous scatter/gather I/O calls to read/write entire files, even when a file is spread across multiple buffers. The asynchronous calls allow the Cache Manager to block only if the application needs the data immedi-

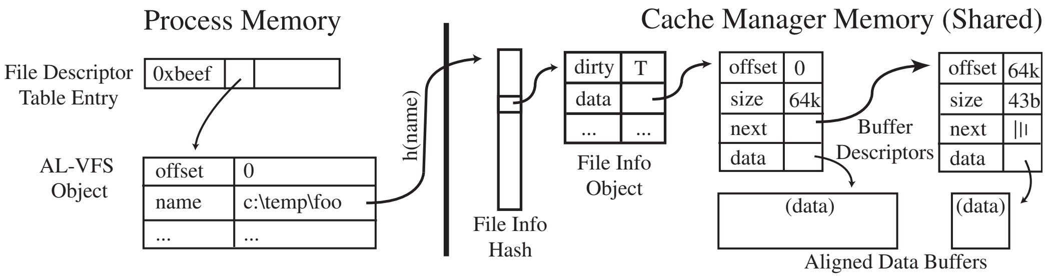

Figure 3: Data structures used to access a file's data.

ately. This is the case for read requests for files that are not in the cache. For all other file I/O (write and read-ahead operations), the asynchronous call sets a per-file flag so that future access to that file will first trigger a check for the completion of the I/O operation, blocking if necessary.

3.1.2 Lazy Writer

Modifications to file contents in the cache are never written through to the operating system immediately. Instead, another thread called the *Lazy Writer* issues writes of dirty file data in the background. To achieve high performance, we had to carefully plan both the scheduling of and the execution context for the Lazy Writer.

In order to work in the background and independently of application threads, we rely on an additional kernel-scheduled thread to issue lazy write operations. Rather than create a separate process for this thread, we decided to have the Lazy Writer thread run in the same address space as the dominant application. We loosely define the dominant application to mean the application making the greatest number of requests to the file cache in a recent time period. This optimization prevents a context switch to a separate process when a thread switch would suffice. We use a simple handoff algorithm to make sure that the Lazy Writer runs in the context of the process making the last five (a tunable parameter) requests.

The proper timing of lazy writes is important for the high performance of the application. Should these writes be scheduled too close to certain read operations, the application could be forced to block. But should we delay these writes for too long, an unacceptable proportion of the cache could be dirty. This would stall the application because no clean buffers exist. The Lazy Writer contacts the Policy Manager to make this decision.

3.1.3 Policy Manager

One goal in our application-level architecture was to allow the policies used in the FS Agent to be easily replaced depending on the particular application. We have attempted, as much as possible, to construct the Cache Manager and the Lazy Writer as a general mechanisms and created the Policy Manager as the component responsible for making application domain-specific decisions. The one notable exception is the simplifying assumptions that the cache reads, writes, and evicts whole files at a time. In the future we can implement a more general mechanism in the cache and let the Policy Manager make those decisions as well.

The Policy Manager is responsible for implementing three policies for a target application group: eviction, read ahead, and write behind. To that end, the Cache Manager notifies the Policy Manager about application file I/O events (open, close, read, write) and consults the Policy Manager when it needs to make an eviction decision or when it is appropriate to read ahead or write behind.

Eviction: The eviction policy implemented by the Policy Manager is a variant on LRU replacement on a whole-file granularity. Files are considered in LRU order, but with clean files evicted before dirty files. Some files are referenced so often in the compilation process that the Policy Manager never evicts them. These include precompiled headers, debugging symbol files and compiler meta data files.

Read Ahead: While it may be hard to devise a general read-ahead policy that adapts to applications with random access or predicts which other files an application will try to read, in the case of a compiler access patterns are predictable. When compiling a source file, a compiler will read that file and all of the header files it includes sequentially. In addition, the compiler may update any number of temporary files and meta-data files (debugging symbol files, incremental compilation files, etc).

As we discussed, the Cache Manager already takes care of intra-file read ahead by reading entire files into the cache. In addition, the Policy Manager implements a read-ahead policy and tries to make sure that when a source file is first accessed, all headers it will eventually need are present in the cache by the time they are needed. To learn the correlation between sources and headers, we run a simple offline analysis from a trace of file accesses. This tracing and analysis can be enabled the first time a project is built with the results available to future compilations. An online scheme is certainly possible, but was beyond the scope of this work. The offline analysis tool generates a *header pack* file for each source file in the project. This file is a collection of all headers needed by a given source file, all packed in one place with individual headers aligned to page boundaries to make for efficient retrieval later.

When the Policy Manager is initialized, it reads information about available header packs. When it receives notification that a source file is opened, it allocates cache buffers for the relevant headers and issues asynchronous read-gather calls to bring them into the cache. The header packs contain only *non-volatile* header files, headers that are supplied by the operating system or third-party libraries and are therefore not expected to change. The Policy Manager also knows of the set of volatile headers used by a given source file and issues read ahead requests for them from their regular location on disk.

Write Behind: The user of a compiler (and in general of the entire software build process) usually cares more about reducing the latency of the process than losing data since the data the compiler writes can be easily regenerated. We would therefore like to implement a write-behind policy that minimizes blocking while still taking advantage of unused I/O cycles to synchronize the cache with the disk.

As Section 2 shows, a potential pitfall in background writing for the compilation process is scheduling the writes from the compilation of one file at the same time as the reads from the compilation of the next file in the build. The Policy Manager tries to optimize this process by setting up the Lazy Writer thread to wake up after 100 milliseconds. This interval is configurable and reflects the elapsed time since the last file I/O request was made by a participating application. Like all other I/O in our system, the Lazy Writer issues asynchronous write-scatter operations to write whole files at a time to disk, marking them as clean in the cache on completion. To avoid overly-eager writes, the Lazy Writer issues writes for one file at a time but continues to write additional files if no I/O activity has been observed since the last one. Our simple policy is to write the largest dirty file first.

In addition to timing the lazy writes, the Policy Manager blocks lazy writing of some files. Before writing a file, the Policy Manager waits until there are no current writers. Finally, some compiler-metadata files (debugging symbol files, and incremental compilation files) are written frequently by the application throughout the build process and are therefore never lazily written by the FS Agent. When the last participating process exists and the cache is shut down, we first commit any pending dirty files to disk.

3.2 Architecture Implementation

This section describes some of the details of the implementation of the FS Agent on the Windows platform.

Shared Memory and Allocation: Windows contains a flexible mechanism for sharing memory between processes, similar to the `mmap` facility on Unix. We allocate the cache as a large shared memory region and pin it in memory to avoid having it paged out. Windows does not provide a way of using a shared memory segment as a heap for allocating data structures. We therefore chose to write our own two-level allocator for managing both the buffers and the data structures for our shared cache.

At the lowest level, we wrote a simple coalescing, best-fit allocator that handles requests for page-sized chunks. The Cache Manager uses this allocator directly for its data buffers, which have sizes that are always multiples of the page size. For all other purposes, we use pooling sub-allocator which we modified to use our low-level allocator. The sub-allocator manages two pools: one for small objects and one for all others. The sub-allocator also serves as an STL allocator, letting us use STL data structures through our implementation.

Locking: For optimistic, lightweight locking we use the Optex class written by Jeffrey Richter. The Optex uses very efficient interlocked memory instructions to try to acquire a lock that resides in the shared memory region. If it discovers that the lock is held by another thread or process, the Optex falls back on a kernel-implemented locking facility and sleeps until the lock is released.

Limitations: Due to time constraints, we do not currently support memory-mapped access to files that reside in the shared cache. To do so, we would have to intercept the system calls for setting up a memory-mapped file and return a pointer to the buffer in the cache that contains the file. Such files would have to reside in a single buffer in the cache and be marked as locked for as long as an application has the memory-mapped objects open. An alternative approach would allow evicting such files by resetting the protection bits on the affected pages and handling the page fault signals like a DSM system would. The implication of this constraint is that we do not test linking, as the linker makes heavy use of memory-mapped files. The compiler we use typically employs traditional read/write system calls. It does attempt to memory-map a small number of temporary files which we do not cache but rather pass through to the operating system. Finally, also due to time constraints we chose not to cache directory contents and just let the operating system handle them.

4 Experimental Evaluation

In this section we evaluate the performance of the FS Agent when used to compile a large software application and compare it to doing the same compilation using the operating system's cache.

All experiments were run using a 750MHz Pentium III machine with 384 MB of SDRAM and a 5400 RPM hard drive running Windows 2000 Professional. We compare the speeds of compiling one component of a large commercial Windows program containing 98 source files and depending on hundreds of header files on the two caches.

Table 1 shows the compile times for various cache configurations. OS refers to compilation using the operating system's cache. OS + hooks refers to compilation using the operating system's cache with a compiler process that has the file I/O hooks installed, but not pointing to the FS Agent's cache. Finally, Agent refers to compilation using the FS Agent's shared cache through the file I/O hooks. For all configurations, the working set of the compiler fit entirely within the cache it was using.

We see that despite its compiler-specific policies, the FS Agent cache makes the compiler almost half a minute slower than the operating system cache does. In order to investigate the reason for this slowdown we created a configuration that still installs file I/O hooks but routes file I/O requests to the operating system cache instead of the FS Agent's cache. The resulting configuration (OS + hooks) runs about a minute slower than the configuration that uses the FS Agent. This results suggests that the file I/O hooks, doing extra processing such as parsing file paths and tracking which handles are managed, incur a significant overhead. The numbers also suggest, however, that holding the hook cost fixed, the FS Agent cache provides a significant benefit over the operating system's cache. We plan to optimize the hook implementations to minimize this overhead.

In order to better understand the breakdown of costs with the different caches, we used the file I/O hooks to measure the time spent on varying I/O op-

Cache Location	Time to Completion
OS	8:51 min
OS + hooks	10:20 min
Agent	9:18 min

Table 1: Compile times for various cache configurations.

System Call	OS Time	Agent Time	Function Calls
Create/Open File	19.5%	23.2%	257215
Delete File	0.2%	0.3%	688
Read File	65.9%	42.4%	95590
Write File	6.0%	8.4%	45731
Find File	3.1%	10.1%	32624
Get/Set Offset	3.4%	3.3%	1678
Close File	0.7%	9.8%	34225

Table 2: A breakdown of the time spent in each file I/O operation as a percentage of time spent on all file I/O operations as measured using the file I/O hooks. The remainder of time is spent on a few other small routines. The operating system spends 92 seconds in read file. The FS Agent 36 seconds in read file.

erations. We used a high-resolution timer available on Windows and report the time spent in each file I/O function as a percentage of the total time spent in file I/O functions. For the OS cache, we used the OS + hooks configuration to allow us to collect this information. Table 2 summarized the results.

The results suggest a significant difference in the breakdown of costs. The OS spends 65% of its file I/O time serving reads, while the FS Agent serves reads only 42% of the time. In contrast, the FS Agent spends quite a bit more time on operations that manipulate the cache data structures, such as file open or close, and on operations that stress meta data such as find file. This is an encouraging result since it suggests that optimizing the way the FS Agent manipulates its data structures and manages file meta data would result in significant overall improvements. In absolute terms, the operating system is spending 92 seconds in read file while the FS Agent only spends 36 seconds in reading, indicating that the FS Agent is succeeding in its main goal of optimizing access times for file data.

5 Related Work

Previous work has frequently focused on providing greater *information and control* to applications. In 1981, Stonebraker pointedly argued that operating systems do not provide adequate flexibility for databases [12]. Since then, many projects have improved application flexibility through three primary mechanisms: restructuring the operating system to offer more flexibility, exposing small but powerful APIs to allow applications or portions of their runtime to contribute to the kernel's policies without changing its mechanisms, and lastly modifying the compiler to automatically pass hints to the kernel.

The mid-1990s saw a surge in research in extensible kernels, which explicitly gave applications much more control than general-purpose monolithic kernels. The primary examples of these extensible kernels are Exokernel [7], Spin [3], and VINO [11]. VINO, for example, let applications download code into the kernel to run in kernel mode on their behalf. While the Exokernel project continues commercially using applications specific to its codebase [13], the other two extensible kernels experience little use primarily because porting applications to a new kernel is tricky and time consuming.

Other projects have taken the approach of modifying both the kernel and applications to introduce a new interface for up- and down-calls to let the kernel collaborate with an application in making decisions on its behalf [10, 6, 1]. For example, Cao's approach lets applications choose which buffer of theirs is evicted when the kernel's buffer cache is out of room. U-Net [14], DAFS [9], and raw disk I/O are other approaches aimed primarily at single-application systems that seek to eliminate the kernel from the common case. While these approaches are elegant, they require modifying the kernel and the application and have therefore not enjoyed widespread adoption.

A third approach chooses to modify the compiler so as to insert hints into the instruction stream for a program. Brown and Mowry [4] describe a system in which a new runtime layer receives the memory prefetching hints the compiler inserted into an application and makes decisions about whether to pass them down into a modified kernel. While this work is similar to ours in its use of a runtime layer to make decisions on behalf of the application, we require modifications to neither the application nor the kernel.

As discussed in the introduction, the gray-box approach marks the most recent popularized point on this trend of giving more "information and control" to applications [2].

6 Conclusions

Transparently inserting an intelligent layer between applications and operating systems with binary rewriting offers a novel approach to giving added control to applications. Keeping this control close to the application is important since changes in kernel-implemented policies are often not appropriate for all applications; for example, increasing the dirty page threshold would aid compilation but hurt the responsiveness of other processes. However, inserting these policies and mechanisms into most applications would be time consuming, bug prone, and redundant. Offering versatile kernel-based mechanisms onto which applications could tailor individual policies was one of the strengths of Vino [11]. In contrast, we move the mechanisms to the application layer in the form of Agents like the FS Agent, but separate out policies into pluggable modules. This reduces kernel crossings, eases development of the application-level policy code, and still lets programmers avoid reinventing the buffer cache.

We showed that using the FS Agent is a viable alternative to using the OS's file cache and that it offers far more versatility at low cost. In the future, we plan to introduce other agents, including a Net Agent, that transparently uses specialized user-level network hardware as it becomes available.

References

[1] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, 1992.

[2] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Symposium on Operating Systems Principles*, pages 43–56, 2001.

[3] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *In Proceedings of Fifteenth Symposium on Operating System Principles, Dec. 1995.*, 1995.

[4] Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation*. USENIX Association, 2000.

[5] Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX-02). Monterey, CA*, June 2002.

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

[7] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[8] Brian Forney, , Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage. In *The First USENIX Conference on File and Storage Technologies (FAST '02). Monterey, CA*, January 2002.

[9] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA*, pages 1–14, June 2002.

[10] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the ACM Symposium on Operating Systems Principals*, 1995.

[11] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, October 1996.

[12] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[13] Vividon: a streaming video appliance. `http://www.vividon.com`.

[14] Thorsten von Eicken, Anindya Basu, Vineet Bush, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, pages 40–53, 1995.