COMP9242: Advanced Operating Systems
Final Examination

Session 2, 2002

Contents

1 Information and Control in Gray-Box Systems

1.1 Introduction

In this paper, the authors make the observation that mainstream operating systems are large and complex bodies of code, which for many reasons are difficult to change. They note that research into extensible systems seeks to make changing the OS easier, but requires modifications (albeit minimal ones in some cases) to the OS code base. The authors contend that even the slightest change to an OS is unacceptable in some situations, and go on to develop a methodology for treating the OS as a *gray box* in which general information regarding the algorithms used by the OS is assumed, and combined with data gathered at run time to provide information regarding the internal state of the system as well as a certain degree of control over this internal state.

After providing an overview of gray-box techniques, the authors go on to develop and use a number of these techniques in the implementation of three *information and control layers* (ICLs):

- a *file-cache content detector* for determining which files are likely to be in the system's cache;
- a *file layout detector and controller* for determining and optimising the layout of files on disk;
- a *memory-based admission controller* for controlling the allocation of physical memory to processes.

Finally, the authors present their work on a common toolbox for implementing gray-box ICLs, cover some further related work, and conclude that gray-box ICLs are a useful tool when building systems involving components over which one has no control.

1.2 Analysis of case studies

File-cache contents

The primary ICL presented by this paper is a file-cache content detector (FCCD), which makes use of the fact that most such caches are implemented using a *least-recently used* (LRU) eviction algorithm. The LRU algorithm and variants thereof are widespread, but all have a well-known pathological case: when the reference pattern of data is cyclic and the data does not entirely fit in the cache, data is always discarded before being reused—in this case the cache almost always misses, and data must be re-fetched from disk.

Much research has gone into improved cache eviction algorithms to avoid the problems of LRU behaviour, but for the purposes of this example the authors look at what can be done from purely user-level code running on an OS with an LRU-based file cache. An ICL is implemented which uses a series of *probes* (file accesses used to determine if a file is in the cache) to guess the content of the cache, and then reorders access to files in order to provide optimal throughput.

Essentially what this ICL is doing is avoiding the pathological case of LRU caches by reordering the references to be non-cyclic. This reordering of references requires changes and support within the various applications, a problem which will be discussed later.

Benchmarks To prove the effectiveness of the FCCD, the authors benchmark two applications exhibiting the pathological-case: repeated runs of *grep* over a large set of files, and sorting data from a large file. By selecting benchmarks which exhibit this worst case behaviour of LRU, the authors avoid showing the overhead imposed by their methods on non-pathological applications, and instead show only the performance gains possible by reordering file input.

The second major drawback of the benchmarks is the selection of parameters. Having stated that they avoid benchmarking the in-cache access time of file data, but instead determine it heuristically at run time to ensure portability, the authors select fixed access and prediction units of 20MB and 5MB, which are most likely tuned to the specific data set sizes used in the benchmarks. Furthermore, the size of benchmark data used is selected to be slightly larger than the file cache in all cases (1GB for Linux and Solaris, 65MB for NetBSD), which maximises the performance gain possible through reordering.

Finally, in the multi-platform sort benchmark, the authors again provide unrealistic input data and maximise the performance benefit, by placing the data file with the matching string last on the command line. This means that a standard application is guaranteed to find the data last, whereas any reordering of file accesses will deliver advantages—again, the benchmarks are selected explicitly to show an advantage for the system under test, and make any conclusions about the general applicability of the system impossible.

File layout

The second ICL introduced by this paper, a file layout detector and controller (FLDC), makes use of an optimisation common to file systems based on the Berkeley *fast file system* (FFS), which places data for files located within the same directory close together on disk.

To control the layout of files on disk, the FLDC makes use of a *directory refresh* technique, which recreates a directory and all files within it. This has the effect (on most FFS-based systems) of moving the data close together on disk, and allocating increasing *inode numbers* (used to identify each file) for the new data. When accessing files, the FLDC system makes use of the refreshed directories, and accesses files in order of their inodes, which results in faster access times since the data is located close together.

This is very similar to the problem of filesystem fragmentation, which has been well understood for a long time—when files are modified and deleted they can no longer be allocated close together, and fragments of files and empty space form all over the disk. To address this problem, some systems make use of a filesystem defragmenter, which reorders files on disk to place them close together once again, improving localised access times. The directory refresh technique used in FLDC is nothing but a crude version of a defragmenter, freeing up fragmented space and allowing the filesystem to reallocate data in a contiguous region. Of course, since it is done purely at user level, it suffers several disadvantages not normally seen in defragmentation—the inode numbers of files change, and hence the change is not transparent to user applications, limiting the FLDC's applicability (as the authors note).

Benchmarks Again, the benchmarks selected for the FLDC specifically show the performance of the system in its best case (a single directory where small files are being created and deleted). The authors show that file accesses is improved if ordered by inode number on three different platforms, although they fail to specify which filesystem is actually used on each platform (Linux, for example, supports a huge number of filesystems) reducing the reproducibility of the results.

Finally, there is no data provided regarding the overheads of the ICL, in particular the directory refresh operation.

Memory-based admission control

The final ICL developed is a memory-based admission controller (MAC), which uses a series of probes to guess at the total amount of physical memory available to the process, and provides an extended version of the *malloc* routine to allocate virtual memory as long as it will fit in the available physical memory.

The whole approach of this ICL is questionable, since it makes several strange assumptions about the behaviour of modern VM subsystems. For example:

- Only memory allocation through *malloc* is controlled, but memory is allocated in many other ways (for example, starting processes, using *mmap*, cached files, shared memory, and other mechanisms), so this is a severe limitation on the applicability of the approach.
- The implicit assumption in controlling *malloc*, which allocates virtual memory, is that all of this allocated memory will be used and will require physical memory to back it. However, modern systems allocate physical memory only when it is used, not when the virtual memory is allocated (through a mechanism such as *malloc*), so a process which allocates large buffers but commonly only uses a small part of them will unnecessarily starve others of memory.
- To determine the available memory size, the ICL uses probes which involve allocating and accessing progressively larger sets of data until paging activity occurs. This means that another application could be forced out of memory by the probe—the authors attempt to account for this (using the troublesome assumption that any other processes have their full working set resident), but ignore other users of physical memory such as the system's internal caches.

The fundamental problem here is that the MAC ICL works against the operating system's memory allocation algorithms in determining the amount of available RAM, when such data is normally available through other means. This leads to strange application behaviour at startup, which may cause the operating system's normal memory allocation routines to behave suboptimally.

To limit memory use, the MAC system denies allocations which would exceed its view of physical RAM. The authors note that this can easily lead to deadlock, and suggest that applications may allocate all required memory at startup, or release memory when an allocation fails—this is not generally applicable, is extremely restrictive, and pushes resource management out of the operating system into the applications, which do not have the full information required in order to make such decisions. The applications are also not notified when memory pressure changes.

The MAC system is only implemented on Linux, and the authors note that the system is particularly tied to the underlying page-replacement process of the OS. This is in stark contrast to their former claim that gray-box techniques were easily portable—in particular, the Linux VM system is a complex and constantly changing one, even maintaining support within Linux will require much ongoing work.

Benchmarks The single benchmark used shows that a modified sort program selects a reasonable memory size, however it also shows something that the authors have successfully hidden from all other benchmarks: the overhead required for an ICL. In this case, a huge 54% overhead is required over the static version of the sort.

This overhead is, in my opinion, prohibitive. A normal application cannot be expected to sustain a 50% overhead on execution time, just to avoid pathological cases arising from memory competition. Furthermore, for most common applications, adjusting memory usage to current memory pressure is not a feasible option—once again, we see that gray-box techniques offer performance benefits only for certain specialised cases, and come with a severe performance disadvantage in the general case.

1.3 General criticisms

Motivations

This paper maintains the key motivation that changing the OS is impossible. With the growth in open source and extensible systems, this is becoming a significantly less realistic limitation. Furthermore, to avoid modifying the OS, the solution adopted requires significant modifications to the applications—often modifying applications is just as troublesome as modifying the OS, for the same reasons listed by the authors.

Reference to related work

The authors' reference to previous work is haphazard and often non-existent. They do discuss briefly the trend in extensible systems, and provide a discussion of previous gray-box approaches, but when it comes to the specific ICLs implemented, previous (and usually more general) solutions to the same problems are totally ignored. For example, there is no reference to non-LRU algorithms for cache eviction, to the concept of defragmentation, or to more complete resource control mechanisms for physical memory allocation.

The authors make a number of apparent factual errors in their discussion of related work. They state that in scheduler activations the user-level library knows how many processors it runs on, but this is untrue—the user-level library knows how many *virtual processors* it runs on, but these may not map directly to physical processors. Furthermore, the paper claims that "running low-importance processes only during idle time is a feature missing from many modern operating systems", even though almost all regular systems allow an application to be run at low priority without influencing normal workloads.

Finally, there is a section briefly discussing "other related work"—four ideas are introduced here, which should have been considered much earlier in the paper and tied into the development of gray-box techniques.

Shortcomings of the approach

The shortcomings of the general gray-box are largely obvious—by implementing everything outside the system, the ICLs are stuck working around the shortcomings of the OS rather than actually fixing the problem. In many cases, they must degrade the overall performance of the system (due to the "Heisenberg effect") to make improvements only for certain specific cases.

ICLs rely on changes to applications, which are often as hard, if not harder, than changes to the OS. Many systems strive to maintain backwards compatibility with a huge pool of existing applications—surely it is harder and less productive to change hundreds of applications to suit an imperfect OS than to fix the problem in the OS.

The authors claim that modifying applications to make use of gray-box techniques is easy, but they only demonstrate this in the case of the file cache detector for simple modifications to *grep*. Surely changing an application to perform all its memory allocation differently, as is the case with the MAC ICL, is significantly harder—the paper is conspicuously silent on the level of modification required in the MAC case.

Finally, in the cases of the file cache detector and disk location controller, the ICLs rely on modifying the external behaviour of the applications (through file ordering)—often such a change is unacceptable for an application.

1.4 Summary

The key failure of this paper comes from its flawed approach—the authors attempt to change the behaviour of applications to cope with a broken OS, rather than just fixing the OS. The problems caused by this approach are many, and are evidenced time and time again in the three case studies, the most obvious of these are:

- modifications to applications are usually non-trivial;

- ICLs only help in specific pathological cases, and most likely impose a significant performance overhead in normal cases;
- use of an ICL works against the natural behaviour of the OS.

Furthermore, the benchmarks used are non-representative of the overall performance of the modified system, merely showing specific improvements in the worst cases over the unmodified system, and hiding the cost of the ICLs.

In conclusion, this is a case of a paper starting from a poor premise, trying to do far too much to prove this a general result, but instead doing it poorly (the authors often cite “space constraints” as reasoning for lack of detail), and failing to prove anything conclusively. One suspects they would have been better served by concentrating on a specific example of the use of ICLs, rather than three case studies.

2 High-Performance Memory-Based Web Servers: Kernel and User-Space Performance

2.1 Introduction

The second paper is an analysis of various high performance web servers, and in particular a comparison of kernel- and user-level web serving techniques. The authors introduce the various issues in high-performance web serving, and cover various approaches to them. They then introduce a software architecture based on a state-machine for high-performance web serving, known as the Adaptive Fast Path Architecture (AFPA), and describe its implementation on Linux and Windows 2000. The paper then analyses the performance of eight different production web servers, as well as an experimental user-level server, and draws some general conclusions.

2.2 General problems

Unlike the first paper considered, the authors of this paper make no glaring overall errors. The biggest problem I experienced stemmed from the somewhat unstructured nature of the writing. It seemed to me that the authors of this paper lacked a clear defining goal—this paper seemed to attempt three related but different projects:

- a comparison of user-level versus kernel-level web serving;
- an investigation of the performance of different high-performance web servers;
- a presentation of the design and implementation of the AFPA system, and its improved performance over other kernel-level web servers.

As a result of the lack of clear direction, the paper is structured strangely (since it attempts to discuss all three projects), lacks a clear conclusion from the results obtained, and tends to repeat itself in different places.

2.3 Specific shortcomings

Experimental results

On the whole, the experimental methodology used by the authors seems sound. This section points out some possible flaws in their results, some of which have already been noted by the authors.

The use of different versions of the Linux kernel, 2.3.51 and 2.4.0, is problematic—both of these versions were under heavy development, which could lead to unusual or suboptimal results depending on the specific version used. The paper claims that these versions were necessitated due to the features required, but one would expect that it was possible to at least use the same version for all benchmarks.

The authors pick the SPECWeb96 benchmark over SPECWeb99, partly because the results are comparable to other systems, but then immediately go on to note that the results presented do not follow the correct reporting guidelines (although they claim that the results are unaffected). Furthermore, the authors blindly used the tuning parameters for the various webservers as supplied with the SPECWeb96 benchmark to achieve maximum performance, but did not test other parameters to verify that they were in fact getting the best result for their particular configuration.

Perhaps the most significant limitation is the extremely small number of client machines—only ten were used. It is highly likely that this would affect the results, for example by reducing the overhead incurred in the TCP/IP stack—a small number of clients would result in unusually small routing and ARP (address conversion) tables, and hence faster TCP/IP stack performance. The authors note a number of these limitations, including the small number of clients, but do not discuss their possible outcomes.

The analysis of the best-case user-level server performance using Howl is cursory, and should have been included with the normal benchmarking of the other servers rather than added to the end of the section. In particular, the implementation of Howl uses the optimised *TransmitFile* system call under Windows 2000, but not the corresponding operation (*sendfile*) under Linux. This is an unreasonable comparison.

Miscellaneous

The general writing, typesetting and presentation standards of this paper leave something to be desired. The use of mono spaced font for emphasis (in the AFPA section) is unusual, the graphs in Figure 3 confusingly use different symbols for IIS and Apache, and the references are poorly prepared. Furthermore, the paper uses a number of acronyms without explanation (for example, the reference to *mbufs* and *skbuffs* in Section 3.3), and confusingly introduces some concepts multiple times (*TransmitFile* is discussed twice on Page 3).

Finally, the conclusion includes an unfinished sentence, possibly indicating that the paper was written in a hurried manner:

> The results showed that software interrupt based kernel-mode servers perform 10% to 20% better than Overall, the best performing Web servers share three attributes. . .

Of course these are only minor issues—although the standard of writing is poor in some places, the content presented by the paper is generally sound.

2.4 Unconsidered issues

Possibly as a result of the unclear goals of the paper, several important issues are not addressed. Most significantly, the paper lacks any discussion, apart from the obvious speed aspect, of the tradeoffs involved in kernel- versus user-level web serving. Running a web server in the kernel of the OS carries with it huge reliability and security risks, although in some cases these risks are justified—the authors should have discussed this, if only briefly. They can be forgiven somewhat for neglecting this discussion, since they do not attempt to conclude that either user- or kernel-level web serving is necessarily better.

The other part most lacking for a paper comparing kernel- and user-level web serving, is a discussion of the reasons why running the server in the kernel is cheaper. The most obvious difference is the cost of context switches between user mode and kernel mode, which is particularly high on the Intel x86 platform used for all the tests. A comparison of the costs across different architectures would have been instructive.

2.5 Summary

Despite the problems outlined here, I was more impressed by this paper than the gray-box paper. It provides a fairly thorough performance analysis of web servers at kernel- and user-level on both Linux 2.3/2.4 and Windows 2000. The biggest shortcoming of the paper is its lack of clear direction, which is evidenced in the overall conclusions.

In my opinion, this paper could be significantly improved by reworking the introduction and conclusion and reorganising its contents, targeting it toward a comparison of the performance issues involved in kernel- and user-level web serving. The title of the paper reflects this, but its contents do not—in particular, the AFPA system would best be served by a separate publication, not a somewhat misplaced section within the paper.

To summarise: a good paper, with generally well analysed results, suffering from a poorly motivated and organised write up.