Preload — An Adaptive Prefetching Daemon

by

Behdad Esfahbod

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2006 by Behdad Esfahbod

Abstract

Preload — An Adaptive Prefetching Daemon

Behdad Esfahbod

Master of Science

Graduate Department of Computer Science

University of Toronto

2006

In this thesis we develop *preload*, a daemon that prefetches binaries and shared libraries from the hard disk to main memory on desktop computer systems, to achieve faster application start-up times. Preload is adaptive: it monitors applications that the user runs, and by analyzing this data, predicts what applications she might run in the near future, and fetches those binaries and their dependencies into memory.

We build a Markov-based probabilistic model capturing the correlation between every two applications on the system. The model is then used to infer the probability that each application may be started in the near future. These probabilities are used to choose files to prefetch into the main memory. Special care is taken to not degrade system performance and only prefetch when enough resources are available.

Preload is implemented as a user-space application running on Linux 2.6 systems.

Acknowledgements

First, I would like to thank my supervisors, Allan Borodin and Angela Demke Brown, for helping me get past playing around the code and get to write my ideas in the form of this thesis.

I wish to thank Søren Sandmann and Lorenzo Colitti for useful discussion, and sharing their measurements and tools with me.

I also want to thank Bert Hubert for readily sharing his work with me prior to publication, and Rik van Riel for answering all my technical questions promptly.

I would like to thank all my colleagues in the Systems Software Reading Group for introducing me to recent research in the systems software area, for thought provoking discussions, and for providing useful feedback during the early days of my work.

During the course of this project, I solicited advice from many, many colleagues here at the University of Toronto, all of whom have helped me very generously. In particular I would like to thank Shiva Nejati, Mehrdad Sabetzadeh, Reza Azimi, and Faye Baron.

Finally, I would like to thank Google and the Fedora Project, for partially supporting my work through the Summer of Code program.

Contents

List of Algorithms

List of Tables

Chapter 1

"And every single block
Looked like every single block
Looked like every single block
Looked like every single block
But she kept driving
'Cause everyone else kept driving
And cause gridlock is evil
And not knowing your way is evil"

—Dan Bern, *Wasteland*

Introduction

It's been a long time since computer processors have been the bottleneck for most uses of computer systems. The data flow in today's computation goes through a hierarchy of memories and mediums whose speeds vary by orders of magnitude. To browse a web page, for example, the data is first requested by the client from the web server. The server will send a request to the database server to fetch the data. The database server finds and loads data from physical storage and serves it back to the web server, which will send it back to the client. When in the client side, the web page will go through the process of text layout and rendering, requiring fonts to be loaded from the local storage, and finally the resulting image is pushed to the display. Caches have been used in a

variety of layers to impedance-match these various levels. In the simplest case, a typical two-level memory architecture consists of a small but fast cache and a relatively large but slower memory. Examples include: the cache in the CPU versus the main memory; the page cache (the main memory as the cache, versus the hard disk as the external memory); the web browser cache versus the web resources; the web server's cache versus all the documents it serves from local files or network resources like databases.

For caches to be effective they should have a high hit ratio. The cache performance is mostly determined by the cache size, the cache evacuation algorithm, and the workload. In many caching scenarios, the external memory is unused for long periods of time. To further improve cache performance, many computer systems try to predict which chunk of data will be needed next and fetch it into the cache (if it is not already in cache) before it is requested. This method is called *prefetching*.

1.1 Desktop Computers

This thesis focuses on desktop computers. Desktop computers are commonly used in homes and small offices. They are mainly used for web browsing, email exchange, instant messaging, reading and writing letters, listening to music, and watching movies. For the purposes of our analysis, desktop computers typically have one or two regular users—unlike servers. While many desktop computers are often not turned off for long periods of time (days and weeks), statistically speaking they spend most of their time idling. This is mostly because of their single-user-at-a-time nature and having few users in total, which means, even at times that the computer is being used by a user, the processor is not busy processing for most of the time. In fact, assuming the system is not short in main memory, the majority of times that the system limits are pushed are when new applications are started. Application start-up consumes a lot of processing time and generates lots of I/O traffic from the external storage that is the local hard disk.

Hard disks are the main means of storage on desktop computers (as opposed to network storage). Because of their mechanical nature, hard disks are a few orders of magnitude slower than the main memory. The ones used in desktop systems have a transfer rate of 20 to 50 megabytes per second, have disks rotating at 5400 rounds per minute, and a *seek-time* of 10 to 20 milliseconds [3]. Seek time is the time it takes for the head to move to the target track. The overhead associated with a reading an arbitrary location on the disk involves the seek time and the rotational latency for the head to be positioned on top of the target sector on the track. We call this time *disk access time*. It is this large disk access time that disk I/O schedulers try to minimize by sorting and merging disk access requests in batches, such that the disk head moves in an *elevator-like* path (going to one side and then the other in a loop) instead of jumping around randomly. In the next section we identify disk access time as a bottle-neck of application start-up time, and in the rest of the thesis we will try to come up with a scheme to avoid delays caused by disk access time during application start-up.

1.2 Startup-Time Problem

When hard disks in their current shape and scale found their way into micro-computers in the eighties they were not particularly a bottleneck of the system's operation. However, during the past decades various measures have had growing rates of varying scales. In particular, in the personal computer market, processor power, size of main memory, hard disk capacities have been growing at least 40% a year on average: doubling every two years [21]. However, disk throughput and access time have been improving at most at a 10% a year rate, being limited by the mechanically moving parts of the drive. This has caused hard disks to have a more important impact on overall system performance.

With the computer's processing power and memory sizes growing fast, software applications have also grown in processor and memory size requirements at a high rate.

Where an application typically was a few tens of files that could fit on a 1.4 megabyte floppy disk in the 1980's, there are applications and games available in 2006 that fill a CD or DVD, broken into thousands of files.

The bigger the software applications become, the more they are broken into modular and partially standalone pieces, for sharing and for manageability reasons. Shared libraries are one way that this is achieved. Shared libraries (or shared objects) were introduced as a way to decrease memory and disk requirements by sharing the code for common functionality between applications and keeping only one copy of such code in main memory and on the external storage. A side effect of using shared libraries is that not every application uses all the code in all the shared libraries that it uses (by linking to them). Another elegant idea in the history of operating systems has been load-on-demand: instead of reading all the needed files in memory before running the application, they are mapped into the address space and the program starts running. The kernel then encounters a page fault when data is missing, leading to disk read operations to fill the memory with the required data [3].

To understand what we call *the startup-time problem* now, we just need to summarize the logical consequences of the above changes and compare their rates:

- Faster processors and bigger hard disk and main memory sizes result in more and larger applications,
- More/larger applications result in more files in the application distribution, and consequently, more files to load at application start-up time,
- Use of shared objects results in more files to be loaded at application start-up time,
- More files loaded cause more disk access times to wait as the hard disk head has to move around the hard disk to find various files to read,
- Use of load-on-demand causes more disk access times to be experienced. As elegant as the load-on-demand idea is, it can lead to unpredictable seek behaviors, like

occasional hits going backwards on disk, and disks perform very poorly in those situations [3].

Given that disk access time has been the slowest of all the factors to improve, it is not surprising that software start-up time (and system boot time similarly) has been slowing down over the years no matter how fast the hardware running it has been getting faster [18]. The end result of this all is that in 2006, on a decent desktop computer it takes:

- from 30 seconds to 2 minutes to boot, depending on configurations and services to start up,
- 30 seconds to log into the desktop environment before the system gets back to a steady state,
- 15 seconds to start a word processor application
- 11 seconds to start a web browser

For hardware and software configurations, see Chapter 6.

1.3 Prefetching

Prefetching has been studied in depth across all areas of the systems literature as a way to improve performance. Hardware prefetching is performed in all modern processors to load instructions ahead of time into the cache. Database and virtual memory are two other systems in which prefetching is widely studied [6]. With the wide-spread use of networks and the Internet in recent years, Web prefetching and remote file system prefetching have gained in interest too.

When reading data from the hard disk, most modern kernels detect sequential read access pattern and prefetch data ahead, saving time that could have been wasted in disk access time when the read operation for the next chunk comes in. In fact, designing the

readahead algorithm has become one of the crucial aspects of file-system performance [20]. There has been a lot of work to go beyond sequential readahead. The goal is to reduce hitting the lengthy disk access time. Prefetching and caching strategies are either heuristic or hinted [17]. There has been work on both areas, and even work that falls in between, heuristically adding hints to applications. Hinted approaches only affect programs that are modified to generate prefetch requests and so are mostly viable in server-side applications that are optimized for the highest possible performance. In the rest of this thesis we mainly deal with heuristic approaches to prefetching.

In the absence of hints and sophisticated heuristics, several operating systems targeting desktop computers have taken naïve approaches to prefetching and reducing boot and start-up times. We review some of these systems in the rest of this section.

1.3.1 Windows XP

Windows XP claims to be a self-tuning operating system. It essentially records file accesses during boot and application start-up times and uses this information to prefetch those files in subsequent events and to lay those files out near each other on the disk, and towards the more dense outer edge of the disk [14].

It has been part of the design goals for Windows XP to boot to a usable state in a total of 30 seconds, measured from the time the power switch is pressed to being able to start a program from a desktop shortcut. To achieve this, Windows XP also *cheats* by delaying various initializations and service start-ups to after the login [15]. This gives the impression of a faster boot process but may also make the first few seconds after login almost unusable due to heavy I/O traffic.

1.3.2 Fedora Readahead

The Fedora Core[1] 5 system contains a package called `readahead` that contains a static list of about 2500 files that are read ahead during the boot process, in two stages. This is supposed to make for a smoother login experience as many files needed during the login on a system with default settings will be in memory already. We analyze the performance of this system more in Chapter 7.

1.3.3 SuSE Preload

Similar to Fedora's `readahead` package, SuSE[2] systems have a package called `preload`[3] that reads files and directory entries into memory, from static lists pre-generated by tracing system calls during the system boot in the standard configuration.

1.3.4 GNOME Display Manager

A display manager is the piece of software that manages the graphical login screen. The GNOME Display Manager (`gdm`) can be configured to call a *prefetching program* when the initial login screen is displayed. This period of time is particularly good for prefetching as the system is idle otherwise. Solaris systems use this functionality of `gdm` to prefetch a list of about thirty shared library files. It is documented as "preloading these libraries improves first-time login performance for the GNOME desktop." [13]

1.4 Contributions

We introduce a new approach to prefetching from hard disk into main memory on desktop systems: to predict what applications a user may run soon based on her currently-running

[1] A GNU/Linux-based operating system
[2] Another GNU/Linux-based operating system
[3] Incidentally, we did not know of this other system called *preload* before naming ours.

applications. This is the first work to perform prefetching at this level as far as we know. We further develop a probabilistic model of the problem based on continuous-time Markov chains, and deduce prediction algorithms that are used to implement a user-space prefetcher.

We implemented this scheme as a user-space daemon monitoring applications and predicting and prefetching online on Linux 2.6 systems, and we measured its performance on various tasks.

While our experimental results look promising, we question the merits of prefetching on desktop systems, and recommend alternatives to prefetching for desktop systems seeking to improve boot and application start-up times.

1.5 Organization of Thesis

This thesis is organized as follows. Chapter 2 reviews related work. Chapter 3 discusses the terminology we use and our system model. Chapter 4 provides the algorithms for implementing the proposed model. Chapter 5 presents the implementation of the system. Chapter 6 presents our experimental results. Chapter 7 discusses the solution and the experimental results achieved, and Chapter 8 concludes the thesis.

Chapter 2

Related Work

Prefetching and caching strategies are either heuristic or hinted. O'Rourke [17] compares issues and results of various file-system prefetching schemes based on both approaches. It concludes that while hinted prefetching can remove almost all cache misses and modifying applications to generate hint streams is straightforward, it is impractical to rewrite more than a few applications on a standard UNIX system.

Chang and Gibson [4] described an intriguing approach to modify application binaries to produce prefetching hints for future I/O requests during I/O stalls. This works by running a *shadow* copy of the application in a low-priority thread that does not stall on I/O and proceeds running, gathering information about future I/O requests. For most applications, they report, the performance comes within a few percent of that achieved by hand-written hints. However, in some cases automatic prefetching can perform worse than no prefetching at all, and the prefetching code added increases binary size by 130 to 600 percent.

In the remainder of this chapter we review previous work on heuristic-based prefetching.

Papathanasiou and Scott [21] discuss that with the drastic growth of processor power and main memory sizes in the past decade, the time may have come to employ aggres-

sive prefetching to offset the rather slow improvement rate of external storage. They discuss why aggressive prefetching makes sense now, research challenges in it, and finally identify several traditional prefetching problems that may require improved solutions as prefetching becomes more aggressive.

Curewitz et al [6] analyze the practical aspects of using data compression techniques for prefetching. The idea is that compression algorithms work by predicting future elements in a stream and assigning shorter codes to them. A compressor is most efficient if it best predicts upcoming input. The algorithm then can be adapted to act as a predictor for a prefetching system.

2.1 Prefetching Integrated with Caching

Prefetching is an old idea. There has been a lot of heuristic prefetching work conducted during the past decade that focus on file-system, page-cache, and web prefetching. A common property to many of these works is that they integrate prefetching with caching. This is feasible when prefetching is implemented at the same level as caching, for example, in the kernel or in the web browser. This has raised the question of to what extent access history can already be used to improve the cache evacuation algorithm instead of using LRU with prefetching. Vellanki and Chervenak [26] demonstrate that well over half of all accesses in a file-system are cacheable based on history, significantly more than LRU and prefetching in most cases.

Griffioen and Appleton [7] devise a *probability graph* connecting files in the file system together as nodes. They do that by connecting file open requests within a certain *look-ahead period*. They achieve up to 280 percent improvement over LRU, or alternatively, reduce cache size by up to 50 percent.

Amer et al [1] introduce a new file access predictor, *Recent Popularity*, that works by predicting a successor for each file access that is the *best j of k* successor of the specific

file in tracked history. Their approach allows for improving accuracy through reducing offered predictions, by adjusting the parameters j and k. They report a less than two percent error rate while offering predictions for 60 percent of accesses.

2.2 Markov-based Prefetching

Bartels et al [2] review potentials and limitations of fault-based Markov prefetching for virtual memory pages. They found that fault-based virtual memory prediction can achieve reasonably high levels of accuracy for some scientific applications, mainly those accessing large matrices stored externally. They also conclude that high precision of one-fault prefetching hardly results in significant speedup, as applications that can benefit from such an accurate short-sighted prediction already have a sufficiently low fault rate and latency that the I/O overhead is not prohibitive in the first place. One of their most surprising results is that the Markov predictor of order 1 often outperformed the Markov predictor of order 2, because the second-order model was too conservative.

The limitation of Markov-based approaches in that orders higher than one or two are impractical is elevated by using the Partial Prefix Matching (PPM) technique. PPMs work by constructing a tree of match prefixes of all lengths up to a limit, and so can be thought of as a multi-order Markov scheme. It essentially trains Markovs of order 1, 2, 3, up to the limit all at the same time, and picks the best predictions out of them all. It is easy to observe that unlike Markov-based approaches, increasing the order limit in PPMs cannot result in inferior prediction results in practical situations.

Kroeger and Long [10] develop a PPM based prefetching scheme with intriguing performance result: that their four megabyte predictive cache has a higher cache hit rate than a 90 megabyte LRU cache for their simulations. However, it is likely that such a result is limited to the particular use cases that their data-set represented.

Hidden Markov Model (HMM) is another model based on Markovs that has been

used in prefetching. Madhyastha and Reed [12] describe an HMM approach to block access classification that can then be used to adjust cache replacement or prefetching. Their model works on blocks of a file at a time and so is limited in scope to large files like databases. Their approach offers significant performance improvement over similar artificial neural network based approaches.

2.3 Block-based, File-based, and Web Prefetching

Prefetching in the file system level can either be done at the block level, or the file level. Block level prefetchers are much harder to develop as the hard disk and memory sizes increase, and offer significantly less improvement as processors grow faster and more blocks should be prefetched to have any measurable effect. Bartels et al [2] confirm this.

Choi et al [5] devise an application/file-level characterization of block references that can be used to employ different cache replacement policies per application/file to maximize performance.

File-based approaches are a lot more promising as they can trigger prefetching of several files by a specific request. In all previous work that we reviewed, file-based prefetching has focused on network file-systems (NFS). Many of them we already mentioned in previous sections. For comparison, our work goes a level higher and tries to prefetch groups of files needed by working on the application level. This is the first work in this level as far as we know.

Lei and Duchamp [11] develop an application/file based approach that records trees of process creations (forks) and file accesses of those processes, in a sequential order. It then uses such trees to make prediction for the next time when those processes are run. They report reducing application latency by up to 40 percent for wireless remote file-system access.

Similar work has been done in web prefetching, many of them simply adapting file-

system prefetching algorithms to the web, but others exploiting features unique to the way the web is navigated. Web prefetching is related to our work in that, like our work, and unlike block-based and file-based file-system prefetching, it tries to come up with predictions of what action the computer user will take next.

Jiang and Kleinrock [9] develop a prefetching scheme for network use. Their scheme has a very simple predictor that, upon seeing a request, predicts all resources linked from the resource being requested based on the history and the number of times each of those links were navigated. However, they back this simple prediction algorithm up by deriving a formula for the prefetching threshold based on system load, capacity, and costs such that a lower average cost can always be achieved. The scheme can be implemented on the client or the server.

Nanopoulos et al [16] propose a PPM based prefetching algorithm for web browsing. They train their PPM using sequences of access in each user *session*. Yang and Zhang [29] propose a very similar scheme, though they do not use the term PPM, and they believe that their work is the first to integrate prefetching and caching in web browsers. They use a fixed part of the cache for prefetching and always fill that with no threshold.

Padmanabhan and Mogul [19] use the prefetching algorithm of Griffioen and Appleton [7] for web prefetching in a cooperative mode where the server suggests some pages to prefetch to the client based on the page requested, and client decides which pages to prefetch based on the suggestions and other criteria including access history.

Sow et al [24] take a compression-based approach to web prefetching, coming up with an algorithm based on the original Lempel-Ziv algorithm.

2.4 Summary

Prefetching file-systems has been widely studied before. However, all previous approaches work on a block-based or file-based manner. In the rest of this thesis, we design a file-

system prefetcher that works on application-level. Like many other prefetchers, we use Markov predictors in a mixture model.

Chapter 3

Fundamentals

This chapter shapes the core of the thesis, by formalizing the problem and the solution we propose. This includes defining the terminology and notation, the main idea, design decisions that were made, and finally the model.

This work lies in the intersection of operating systems and machine learning areas of computer science. For this reason, we provide a detailed definition of terminology and mathematical concepts we use throughout the thesis.

3.1 Terminology

There are two fundamental objects that we deal with in our model: *applications* and *maps*. The following four definitions make it clear what we exactly mean by these two terms.

We provide specific details and examples drawn the Linux kernel and GNU C library. Other Unix variants provide similar features.

3.1.1 Processes

We use the term *process* as it is always used in the systems literature: a program in execution. It is completely characterized by a single current execution point and address space. The list of current processes in the system as well as information about each process can be found by scanning the `/proc` pseudo-filesystem.

Each process has exactly one file on the file-system which is the program that this process is executing. The file `/proc/<pid>/exe` is a symbolic link to this file. When the program file is unlinked from the file-system, the string " `(deleted)`" will be appended to this symlink, so that can be detected easily.

3.1.2 Applications

An *application* is a program that provides the user with tools to accomplish a task. On systems that we focus on in this thesis, an application is almost always a program with a graphical user interface. An application may be started by the user choosing a launcher from an application menu, or by other means. Some examples of applications are the Firefox web browser, the OpenOffice.org Writer word processor, or the Totem movie player.

An application is said to be running at a certain point in time, if and only if there is at least one process which is executing this application program. Applications typically have larger binaries in size compared to other kinds of programs available on a modern Unix system, and they have larger working sets when running, and longer running times too. These are consequences of the inherent complexity of applications as programs interacting with users in a graphical user interface and performing complex tasks, compared to the "do one thing and do it well" Unix mantra. While there are hundreds, even thousands of programs installed on a typical Unix system, there are hardly more than a hundred applications installed on such systems. Even then, a user rarely uses more than ten or

twenty different applications in her day-to-day interaction with the computer.

Since our goal is to achieve faster application start-up time, there are mechanisms in preload to roughly distinguish application programs from other programs. Preload ignores any processes that are very short-lived, or their address space is smaller than a certain size. This has the extra benefit of keeping the model in a manageable size. The details of this filtering is described in Section 4.2.1.

Since applications are the focus of this thesis, we use this term regularly, but most of the time what we really mean is a "program that has passed preload's tests for being an application program".

3.1.3 Shared Objects

By *shared object*, we simply mean a file that a program uses when running, and uses by mapping the file into its address space using the `mmap(2)` system call. When several processes use a shared object, only one copy of its contents is kept in the main memory, and mapped into the address space of each process using it.

The most common type of shared objects are shared libraries, but the program binary itself, fonts, locale definitions, static system-wide caches (like the icon cache) and other static data files are some other types of shared objects used by programs.

By their nature, most of the shared objects used by a process are mapped when the process is starting. By prefetching these shared objects, as we will see, we can improve the time it takes for the application to start up.

A shared object is uniquely identified by a file-name on the file-system.

3.1.4 Maps

A map is a contiguous part of a shared object that a process maps into its address space. This is identified by an offset and a length; in practice, both of them are multiples of the page-size of the system, 4kb on 32-bit and 8kb on 64-bit processors.

A process may use multiple maps of the same shared object. The list of the maps of a process can be accessed through the file `/proc/<pid>/maps`. This contains a list of address ranges, access permissions, offsets, and file-names of all maps of the process. When the shared object file of a map is unlinked from the file-system, the string "`(deleted)`" will appear after the file-name of the map in the maps file, so this can be detected easily.

3.2 Mathematical Background

In this section we review the mathematical concepts and notation from probability theory that are used in the following sections. A reader familiar with the definition of the subsection titles in this section may skip to the next section.

3.2.1 Exponential Distributions

The *exponential distributions* are a class of continuous probability distributions. They are often used to model the time between events that happen at a constant average rate. [28]

The probability density function (pdf) of an exponential distribution has the form

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & , \ x \geq 0, \\ 0 & , \ x < 0. \end{cases} \tag{3.1}$$

where $\lambda > 0$ is a parameter of the distribution, often called the rate parameter. The distribution is supported on the interval $[0, \infty)$.

The exponential distribution is used to model Poisson processes, which are situations in which an object initially in state A can change to state B with constant probability per unit time λ. The time at which the state actually changes is described by an exponential random variable with parameter λ. Therefore, the integral from 0 to T over f is the probability that the object has transitioned into state B by the time T.

In real world scenarios, the assumption of a constant rate (or probability per unit time) is rarely satisfied. For example, the rate of incoming phone calls differs according to the time of day. But if we focus on a time interval during which the rate is roughly constant, such as from 2 to 4 PM during work days, the exponential distribution can be used as a good approximate model for the time until the next phone call arrives. Likewise, the rate of an application being started differs according to the time of day. But when the computer system is on and being used, the exponential distribution can be used as a good approximate model for the time until the next start-up of the application.

3.2.2 Continuous-time Markov Chains

A *continuous-time Markov chain* is a stochastic process $\{X(t) : t \geq 0\}$ that enjoys the Markov property and takes values from amongst the elements of a discrete set called the state space. The Markov property states that at any times $s > t > 0$, the conditional probability distribution of the process at time s given the whole history of the process up to and including time t, depends only on the state of the process at time t. In effect, the state of the process at time s is conditionally independent of the history of the process before time t, given the state of the process at time t. [27]

Intuitively, one can define a Markov chain as follows. Let $X(t)$ be the random variable describing the state of the process at time t. Now prescribe that in some small increment of time from t to $t + h$, the probability that the process makes a transition to some state j, given that it started in some state $i \neq j$ at time t, is given by

$$\Pr(X(t + h) = j | X(t) = i) = q_{ij}h + o(h), \tag{3.2}$$

where $o(h)$ represents a quantity that goes to zero as h goes to zero. Hence, over a sufficiently small interval of time, the probability of a particular transition is roughly proportional to the duration of that interval.

Continuous-time Markov chains are most easily defined by specifying the transition

rates q_{ij}, and these are typically given as the ij-th elements of the transition rate matrix, Q. The continuous-time Markov chains that we use have Q-matrices that are:

- conservative—the i-th diagonal element q_{ii} of Q is given by

$$q_{ii} = -q_i = -\sum_{j \neq i} q_{ij}, \tag{3.3}$$

  Note that this is only a convention, to make rows of Q sum to zero, hence the name conservative. The diagonal is never used as a rate parameter, and we never use this property of the matrix, but include it to have a well-defined diagonal.

- stable—for any given state i, all elements q_{ij} (and q_{ii}) are finite.

When the Q-matrix is stable, the probability that no transition happens in some time r is

$$\Pr(X(t+r) = i | X(s) = i, \forall r \in [t, t+r)) = e^{-q_i r}. \tag{3.4}$$

Therefore, the probability distribution of the waiting time until the first transition is an exponential distribution with rate parameter q_i ($= q_{ii}$), and continuous-time Markov chains are thus memoryless processes.

Given that a process that started in state i has experienced a transition out of state i, the conditional probability that the transition is into state j is

$$\frac{q_{ij}}{\sum_{k \neq i} q_{ik}} = \frac{q_{ij}}{q_i}. \tag{3.5}$$

3.2.3 Exponentially-fading Mean

The *exponentially-fading mean* of a function $f(t) : t \mapsto \mathcal{R}$ at time T is defined as:

$$\mu(f, T; \lambda) = \frac{\int_{-\infty}^{T} f(t) e^{\lambda t} dt}{\int_{-\infty}^{T} e^{\lambda t} dt} = \frac{\int_{-\infty}^{T} f(t) e^{\lambda t} dt}{\frac{1}{\lambda} e^{\lambda T}} = \lambda \int_{-\infty}^{T} f(t) e^{-\lambda(T-t)} dt \tag{3.6}$$

where λ is called the decay factor, and identifies how fast the older values are faded.

From this definition, it follows that for any $T_0 < T$

$$\mu(f, T; \lambda) = e^{-\lambda(T-T_0)}\mu(f, T_0; \lambda) + \lambda \int_{T_0}^{T} f(t)e^{-\lambda(T-t)}dt \tag{3.7}$$

Moreover, if $f(t)$ is constant between T_0 and T,

$$\mu(f, T; \lambda) = e^{-\lambda(T-T_0)}\mu(f, T_0; \lambda) + (1 - e^{-\lambda(T-T_0)})f(T) \tag{3.8}$$

Now if we sample the continuous signal $f(T)$ at regular intervals of length τ, the resulting discrete signal F_n will have

$$\mu(F, n; \lambda, \tau) = e^{-\lambda\tau}\mu(F, n-1; \lambda) + (1 - e^{-\lambda\tau})F_n \tag{3.9}$$

where $e^{-\lambda\tau}$ may be called the mixing-factor. This last equation is useful even if F_n is any arbitrary sequence of numbers. It allows to maintain an on-line average of the sequence without keeping the number of elements seen so far, and for the average to be fading.

An extension to the above case is when we want to find the fading mean of a sequence of measurements at arbitrary points in time, as opposed to a sampling with a fixed frequency of samples. If F_n is the sequence of measurements, and T_n is the sequence of times the measurements were performed, and assuming that T_n is ascending, the exponentially-fading mean of the measurements at time T_n is:

$$\mu(F, n; \lambda, T) = e^{-\lambda(T_n - T_{n-1})}\mu(F, n-1; \lambda, T) + (1 - e^{-\lambda(T_n - T_{n-1})})F_n \tag{3.10}$$

3.3 Main Idea

There are two fairly isolated components in preload: the data gathering and model training component, and the predictor. These two are connected together using a shared probabilistic model. The former component trains the model online based on data gathered by on-going monitoring of user actions, while the latter uses the model to make predictions and perform prefetching.

The data gathering component will gather information about running applications periodically, once each *cycle* where a cycle is a tunable parameter that defaults to twenty seconds. The list of running applications is produced by filtering the list of the processes running on the system, and for each application, the list of its file-backed memory maps is fetched, and used to update the model parameters.

The predictor component also takes action once every cycle, and uses the trained model and the list of currently running applications. For every application that is not running, the predictor derives a probability that this application is going to be started during the next cycle. The predictor then uses these per-application probabilities to assign probabilities to their maps, and sorts the maps based on their probabilities, and proceeds with prefetching the top ones into main memory. Memory statistics and system load are used to decide how much prefetching is performed in each cycle, to minimize the effect of preload on the system load.

The problem can be seen as a stochastic Markov chain whose states are members of the power-set of all applications that the user may run, and given the current state, we are interested to know the transition probabilities to every other state during the next cycle. If we could build and train this Markov chain then we would be done, but this is not feasible, given the total number of states and the little training data we have. To overcome this shortcoming, we make independence assumptions and model every pair of applications separately in a four-state continuous-time Markov chain whose states correspond to the four combinations of each of the applications being running or not. Each of these Markov chains models the correlation between the state of the two applications involved.

Every state in each of these Markov chains has a waiting time parameter that is the average time before leaving this state. When a transition happens, every outgoing edge from the current state has a probability assigned to it that this edge is taken. All these parameters are trained as an exponentially fading mean of their values over time, such

that when a user's habits change, the model adapts to it in a constant time with high probability.

3.4 Design Decisions

In this section we discuss the main decisions behind the design above, as well as some remaining details that should be resolved in order to make a complete implementation of preload.

3.4.1 Bayesian Probabilistic Model

We use a Bayesian probabilistic model, which means, we assign a probability (real number between zero and one including) to our (the system's) belief of an event. This holds equivalently true for events in the past and current time as well as for events in the future.

As an example, for a map M, we *may* assign a probability number incore_M to M that is our belief of the event that map M is in core at this time. This probability may take any value between zero and one as long as we do not query (using the `mincore(2)` system call for example) the map for being in core, although we may be able to do that.

3.4.2 Memoryless Model

Another important decision made in the design process that needs justification is the use of a memoryless model. Memoryless in this context means that given the trained model, at any point in time, all decisions are made only based on the current list of applications running, no matter when they were started or what the previous states of the system have been. This is not necessarily what happens in reality. For example a user may play his favorite video game in his break times that are exactly thirty minutes long. So as time approaches the half hour, the probability that the game is closed increases, but a

memoryless model cannot capture that. Not all user actions are like that though: a user may leave the web browser open for very long periods of time, making a memoryless model as good as one can get.

We have chosen a memoryless Markov-based model mainly because of its simplicity and ease of implementation. However, this is not much of a limitation as we train the system on-line. We believe this is a good compromise that keeps the model simple like a memoryless system, while still keeping it powerful by using the entire history of the system as training data.

3.4.3 Mixture of First-Order Markov Predictors

Markov predictors are an implementation of prediction by partial matching (PPM). PPM estimates probabilities based on prior observation. A PPM of order n encodes sequences of length n and predicts the next element of a sequence given the $n-1$ immediately preceding elements. The predictor chooses the most frequently occurring transition from the static beginning with the $n-1$ observed elements and predicts the final element of the sequence [2]. For this scheme to work well, the data stream should expose patterns of order n. The larger that n is, the more conservative the predictor will be, and will fail to make predictions when no known sequences match the initial $n-1$ elements. With lower values of n, the predictor will be more aggressive, and more likely to mispredict. The complexity of a PPM of order n with a domain of elements of size d is $\Theta(d^n)$, as one has to keep the frequency of all d^n sequences of elements in the model (many of them may be zero), to perform predictions. This obviously falls short if the number of the elements in the training data stream is *not* significantly larger than d^n. For this reason, in predictors used for prefetching, a PPM model of order greater than two or three is unlikely to be effective.

When talking about Markov chains and Markov predictors out of the scope of PPMs, the order definition is different. A first-order Markov is one that chooses next state

based on the current state and a second-order Markov is one that chooses next state based on two previous states, while a zero-order Markov chooses next state independent of current or any previous states. This means a first-order Markov precitor is equivalent to a second-order PPM. This should be kept in mind to avoid confusion.

Previous work on fault-based Markov prefetching for virtual memory pages shows that the Markov predictor of order one outperforms the Markov predictor of order two [2]. Another system uses a Hidden Markov Model (HMM) for file-system input/output access pattern classification [12]. While an HMM has the potential to model a higher-order Markov model (by encoding sequences of elements in its hidden state), in this particular application, they construct a composite HMM from the HMMs trained for each unique access pattern. Each of their basic HMMs then have a total number of states equal to the size of the domain of elements, and hence can be best thought of as a first-order model.

For web prefetching, a PPM predictor of order two is more common [16, 9]. That is due to the fact that the number of the elements in the working set of a web prefetcher is typically much smaller than the number of elements in page/block-based or file-based prefetchers that have to deal with tens of thousands or even millions of elements.

In the domain of our problem, elements are the applications, and the data stream is the sequence of application start-ups. Like in the case of web prefetching, we have the luxury of having a fairly limited domain set (of size a few tens of items). However, our training data stream is fairly low-frequency too, in the rate of zero or a few application start-ups per minute, if not per hour. As a result, it is not obvious whether a first-order predictor performs better or a second-order one, but the approach we take in fact resolves this problem by using a mixture of very small Markov chains (two elements and four states each), that can be easily trained and used as first-order Markov predictors.

Mixture models are a common tool in machine learning to compose complex models out of simpler building blocks. The beauty of this approach is that training each of the

basic models used is easy and feasible, the composition rule is well-understood, and the final model is powerful.

For the above reasons, we have chosen the mixture of first-order Markov predictors as our learning model.

3.4.4 Using the Correlation Coefficient

Another question we had to answer was whether the probabilistic correlation coefficient of the random variables of two applications being run over the time should be used to weight the effect of them on each other. We answer no to this question, for the reason that follows.

Consider the following scenario: There are two programs, one of them is a `cron` job running periodically every thirty minutes, taking a few seconds on each run, the other program is the user's internet browser, running at their will for long periods of time, with no recognizable pattern. It is easy to see that the correlation coefficient of the random variable of these two programs running is very near to zero, because the odds of both of them running is almost the same as the odds that the `cron` job is running independent of the browser. Now lets see if these two programs can give us any information about each other: The browser tells us that no matter if it is running or not, the `cron` job is likely to be started very soon (in fifteen minutes on average). This is useful information we did not have otherwise. In other words, if the correlation coefficient is insignificant, the predictions become independent of the application predicting, and this can be thought as a zero-order prediction.

The example sketched above may not be a real-world case for preload (for example, because we ignore very short-running processes), but the argument holds for the general case.

Another reason that forced us to make this decision was that we were unable to fit it in our probabilistic inference with firm theoretical reasoning.

3.4.5 User-space versus Kernel-space

A common problem with the prefetching literature is that most of the systems are implemented (by design) in the lower levels of the operating system, mostly in the kernel space. This design has several benefits, like tight integration with file-system caching. This, on the other hand, increases the complexity of the implementation drastically, and makes deploying the system on a normal system much harder. Mostly because the kernel needs to be patched, and since prefetching has not shown enough benefits to justify the complexity of the implementation, no sophisticated prefetching system (like those developed in academic circles) has been widely deployed by major operating system distributors. For this reason, we have decided to implement preload completely as a user-space program running in the background (also known as a daemon).

Preload gathers information about running processes and their shared objects by scanning the `/proc` pseudo-file system and performs prefetching using a few system calls, mostly `readahead(2)`, `posix_fadvise(2)`, `posix_madvise(2)`, `mmap(2)`, `madvise(2)`, `fadvise(2)`, and `mincore(2)`.

The main problem with this approach is that preload cannot track file accesses by any means other than `mmap(2)`ing the file. This includes `open(2)` and `stat(2)` accesses.

3.4.6 The User Running the Application

Another issue we had to deal with was whether the pair of (user, application) should be used in the inference phase instead of individual applications. That generally makes sense, since different users have different habits and sets of commonly-used applications, but we decided to not do this, basically because this complicates the implementation and also adds another orthogonal axis on the object space of the model. This is not much of a problem, since preload is targeted for desktop systems that usually have very few number of users (one or two most of the time).

If the per-user behavior is desired, different preload daemons can be run for different users. This is a compromise though, since these daemons will race for using resources (main memory mostly), and no cross-user inference is performed. Cross-user inference is very powerful at login time, since the login manager is normally run as the super-user, while after login, the desktop is run as the user who just logged in.

3.5 The Model

In this section we define the objects used in the model representation of the problem. These objects will then be used in the algorithms in the next chapter. For each object, the member properties are divided into two parts, the persistent properties and the runtime properties. The persistent properties are updated by the data gathering and training component, and will be saved and restored across runs of preload. The runtime properties are used by the predictor to keep track of its state at the current time, and so are not kept across runs. For each object, some of the persistent properties form a key, in that the object is uniquely identified by values for those set of properties.

In the following subsections, the key properties are prefixed by an arrow.

3.5.1 Map Object

A *Map object* corresponds to a single map that may be used by one or more applications. A Map is identified by the path of its file, a start offset, and a length. The *size* of a Map is its length.

Struct Map {

PERSISTENT PROPERTIES

→ **char** ∗path; // *full name of the file being mapped*

→ **size_t** offset; // *start offset in bytes*

→ **size_t** length; // *length in bytes*

}

3.5.2 Exe Object

An *Exe object* corresponds to an application. An Exe is identified by the path of its executable binary, and as its persistent data it contains the set of maps it uses and the set of Markov chains it builds with every other application.

The runtime property of the Exe is its running state which is a boolean variable represented as an integer with value one if the application is running, and zero otherwise. The `running` member is initialized upon construction of the object, based on information from `/proc`.

The *size* of an Exe is the sum of the size of its Map objects.

Struct Exe {

PERSISTENT PROPERTIES

→ **char** ∗path; // *full name of the executable binary*

Set of Map maps; // *the maps this application uses*

Set of Markov markovs; // *the Markov chains with other applications*

RUNTIME PROPERTIES

int running; // *one if running, zero otherwise*

}

3.5.3 Markov Object

A *Markov object* corresponds to the four-state continuous-time Markov chain constructed for two applications A and B. The states are numbered 0 to 3 and respectively mean: none of A or B is running, only A is running, only B is running, and both are running. A Markov object is identified by its links to the Exes A and B, and has as its persistent data the (exponentially-fading mean of) transition time for each state, timestamp of when the last transition from that state happened, and probability that each outgoing transition edge is taken when a transition happens.

The runtime property of a Markov is its current state and the timestamp of when it entered the current state. Upon construction, the current state is computed based on the `running` member of the two Exe objects referenced, and transition time is set to the current timestamp.

Struct Markov {

PERSISTENT PROPERTIES

→ **Exe** a, b; *// the two applications involved*
// in this Markov chain

double tt[4] *// mean transition time from each state*

double tp[4][4] *// probability that transition from*
// one state goes to another

int timestamps[4] *// timestamp of last time leaving each state*

The

RUNTIME PROPERTIES

int state; *// current state, 0, 1, 2, or, 3*

int time; *// timestamp of the last transition*

}

trasition times `tt` are the inverse of transitions rates defined in Section 3.2.2.

The `state` of a Markov object can be computed as follows:

$$M.state = M.a.running + 2 \times M.b.running \tag{3.11}$$

and is always updated to maintain this as an invariant.

3.5.4 MemStat Object

The *MemStat object* holds various statistics about total and available memory in the system as well as disk activities. All values are in kilobytes.

```
Struct MemStat {
    int  total    // total memory
    int  free     // free memory
    int  cached   // page-cache memory
    int  pagein   // total data paged in (read from disk)
    int  pageout  // total data paged out (written to disk)
}
```

3.5.5 State Object

The *State object* holds all the information about the model except for configuration parameters. It contains the set of all applications and maps known, and also a runtime list of running applications and memory statistics which are populated from `/proc` when a State object is constructed.

There is a singleton instance of this object at runtime that is trained by the data gathering component, and used by the predictor. It has methods to read its persistent state from a file and to dump them into a file. This will load/save all referenced Markov, Exe, and Map objects recursively.

Struct State {

Persistent properties

HashTable of Exe	exes	*// all the applications known*
HashTable of Map	maps	*// all the maps known*

Runtime properties

Set of Exe	running_exes	*// all applications running*
MemStat	memstat	*// memory statistics*

}

3.5.6 Parameters

There are two important parameters that are used with the model. These are left as configuration variables and are set by the user with other less important configuration parameters that are described in detail in Section 5.2. The two are:

τ: the length of each cycle in seconds,

λ: the decay factor used for exponentially-fading means.

We will use these parameters in Chapter 4. τ is used in the inference algorithms when we predict what happens during the next cycle. λ is used in the training algorithm as the decay factor of the exponentially-fading means we maintain.

3.6 Summary

In this chapter we provided background material used in our work, and described the main idea of this thesis. After discussing design decisions, the model was presented. In the next chapter we will present algorithms for training the model; making predictions based on it; and prefetching those predictions.

Chapter 4

Algorithms

In this section we present algorithms for training the model introduced in Section 3.5, inference and prefetching based on the trained model, and the main body of the preload daemon. The inference algorithms are presented as their probability equations to maintain readability.

In the algorithms, we access properties of the objects as defined in the structures in the model (Section 3.5). For example, for a Markov object m, its transition probability from state 1 to state 3 may be written as $m.tp_{1,3}$ or $m.tp[1][3]$ depending on the context. We use the former in equations, and the latter in algorithms.

4.1 Main Algorithm

When the preload daemon is started, it will periodically run the training and prediction algorithms. This is shown in Section 4.1.

4.2 Data Gathering

Data gathering happens by scanning the list of currently running processes from `/proc`, parsing their maps, and reading memory and I/O statistics from `/proc/meminfo`.

```
1: Load configuration
2: Load state
3: while not terminated do
4:     timestamp ← current time
5:     GatherData
6:     Prefetch
7:     Train
8:     Sleep (timestamp + τ− current time) seconds
9: end while
10: Store state
```

Algorithm 4.1: Main algorithm of the preload daemon

The daemon loads configuration and state. Next, it gathers data, predict and prefetch, and train in a loop until terminated. Finally it saves data and exit. The GatherData procedure is described in Section 4.2, and the Prefetch and Train algorithms are presented in Section 4.3.2, and Section 4.4 respectively.

The data gathering algorithm populates a list of the file-name of currently-running applications, named `running_applications`, and a MemStat object named `current_memstat`. This information is used during prefetching.

4.2.1 Exe and Map Filtering

We do a very simple filtering on the scanned processes and maps. For both we allow the user to black-list certain files by matching patterns on the file names. The details of this black-listing can be found in Section 5.2.3. The only other filtering we do is to require a minimum size for the sum of the size of the maps for a process to consider it as an Exe object. This parameter is explained in Section 5.2.1.

4.3 Predictor

The predictor is responsible for inferring probabilities that each map may be needed during the next cycle, and to choose and prefetch the high ranking ones.

4.3.1 Inference

In the following sections, all the probability estimates of the form $\Pr(X)$ are implicitly conditioned on a given state and parameter τ (the length of one cycle). For example when we write $\Pr(X)$, we really mean is $\Pr(X|state, \tau)$.

Inferring Exe Probabilities

For every Exe object E, we are interested in finding $\Pr(E \text{ starts})$ which is the probability that E is not currently running but will be started during the next cycle (τ seconds). For a running application, this is obviously zero. We can encode this observation as:

$$\Pr(E \text{ starts}) = (1 - E.running) \Pr(E \text{ is needed}) \tag{4.1}$$

where $\Pr(E \text{ is needed})$ is the probability that the application E will be running at the next cycle. Similarly for $\Pr(E \text{ is not needed})$:

$$\Pr(E \text{ starts}) = (1 - E.running)(1 - \Pr(E \text{ is not needed})) \tag{4.2}$$

Now to find $\Pr(E \text{ is not needed}|state, \tau)$, we observe that $\Pr(E \text{ is not needed})$ is independent of all the variables in state except for the Markov chains that it forms with other applications. In other words:

$$\begin{aligned}
\Pr(E \text{ is not needed}) &= \Pr(E \text{ is not needed}|state, \tau) && (4.3)\\
&= \Pr(E \text{ is not needed}|E.markovs, \tau) && (4.4)\\
&= \prod_{m \in E.markovs} \Pr(E \text{ is not needed}|m, \tau) && (4.5)\\
&= \prod_{m \in E.markovs} (1 - \Pr(E \text{ is needed}|m, \tau)) && (4.6)
\end{aligned}$$

Now consider the case of a single Markov m that has E as one of its Exe links. Assume without loss of generality that $m.a = E$. The case for $m.b = E$ is similar. We are only interested in the case that E is not currently running, so $P(E \text{ is needed}|m, \tau)$ is equal to the probability that m makes a transition in time τ, and that the state it goes into has E running (states 1 and 3):

$$\begin{aligned} \Pr(E \text{ is needed}|m,\tau) &= \Pr(m \text{ makes transition in time } \leq \tau) && (4.7)\\ &\times \Pr(m \text{ goes into states 1 or 3} \,|\, m \text{ changes state}) && (4.8)\\ &= (1-\exp(\frac{-\tau}{m.tt_{m.state}}))(m.tp_{m.state,1}+m.tp_{m.state,3}) && (4.9) \end{aligned}$$

and we are done. The probability that application E is not currently running but will be started during the next cycle is:

$$\begin{aligned} \Pr(E \text{ starts}) &= (1-E.running)\Pr(E \text{ is needed}) && (4.10)\\ \Pr(E \text{ is needed}) &= 1-\prod_{m\in E.markovs}(1-\Pr(E \text{ is needed}|m,\tau)) && (4.11)\\ \Pr(E \text{ is needed}|m,\tau) &= (1-\exp(\frac{-\tau}{m.tt_{m.state}}))(m.tp_{m.state,1}+m.tp_{m.state,3}) && (4.12) \end{aligned}$$

Inferring Map Probabilities

For every map object M, we are interested in finding $\Pr(M \text{ is needed})$ which is the probability that M will be used by a process during the next cycle (τ seconds). This happens when an already running application accesses the map, or if a new application that uses the map is run. The former case cannot be tracked easily, so we only handle the latter.

$\Pr(M \text{ is needed})$ is the probability that at least one application using map M that is not already running will be started by the next cycle. Similarly for $\Pr(M \text{ is not needed})$.

With this definition, $\Pr(M \text{ is not needed})$ can be computed using $\Pr(E \text{ starts})$:

$$\begin{aligned} \Pr(M \text{ is not needed}) &= \Pr(M \text{ is not needed}|state, \tau) && (4.13) \\ &= \Pr(M \text{ is not needed}|\{E : M \in E.maps\}, \tau) && (4.14) \\ &= \prod_{E:M \in E.maps} (1 - \Pr(E \text{ starts})) && (4.15) \end{aligned}$$

where $\Pr(E \text{ starts})$ is inferred in previous section. Hence:

$$\Pr(M \text{ is needed}) = (1 - \prod_{E:M \in E.maps} (1 - \Pr(E \text{ starts}))) \quad (4.16)$$

4.3.2 Prefetching

With the list of currently running applications and memory statistics available and the inference equations, the prefetching algorithm simply sorts all maps based on the probability that they will be needed during the next cycle, cuts at a threshold depending on the memory conditions, and fetches. The prefetching algorithm is listed in Section 4.3.2.

4.4 Training

The training algorithm is straightforward: It checks for any new applications that are not known to preload currently and registers them. Then, it updates the running status of all Exe objects, and finally updates all Markov objects. The training algorithm is listed in Section 4.4 and uses the UpdateMarkov algorithm.

The UpdateMarkov algorithm is listed in Algorithm 4.4. It computes the new state of the Markov, and if it is different from the old state, a transition has been occurred. In that case, it updates all different timestamps of the object, as well as the transition time and probabilities.

The transition time of the previous state is updated to reflect the transition happening. It follows Equation 3.10. The transition time maintained in the Markov object is an

```
1: maps ← state.maps
2: Sort maps descending using Pr(M is needed) as key
3: Compute available_mem for prefetching based on configuration parameters and
   current_memstat
4: selected_maps ← ∅
5: for all M in maps in the sorted order do
6:    E.running ← 1 if E ∈ running_exes, 0 otherwise
7:    if M.length > available_mem then
8:       Break out of loop
9:    end if
10:   selected_maps ← selected_maps ∪ M
11: end for
12: Fetch maps in selected_maps into memory
13: state.memstat ← current_memstat
```

Algorithm 4.2: Prefetch

The prefetching algorithm first sorts maps based on their probability of being needed during the next cycle, derived in 4.3.1. Then, it computes available memory for prefetching according to Equation 5.1. And finally it prefetches maps from the most probable to the least, until it exhausts the available memory.

exponentially-fading mean of the sequence of times to leave the state, over all transition events that ever happened from the previous state.

All transition probabilities are also updated in a similar fashion. The transition probability of the current event is 1 for the arc connecting previous state to the new state, and 0 for all other arcs. This value is combined with the exponentially-fading mean value, using Equation 3.10 again.

```
1: known_applications ← {E.path|E ∈ state.exes}
2: for all path in (running_applications - known_applications) do
3:    Create a new Exe object E for path
4:    Populate E.maps by scanning /proc; create new Map objects and add them to
      state.maps if necessary
5:    for all E′ in state.exes do
6:       Create a new Markov object M for E and E′
7:       Add M to E.markovs and E′.markovs
8:    end for
9:    Add E to state.exes
10: end for
11: running_exes ← {E|E ∈ state.exes and E.path ∈ running_applications}
12: for all E in state.exes do
13:    E.running ← 1 if E ∈ running_exes, 0 otherwise
14: end for
15: state.running_exes ← running_exes
16: for all E in state.exes do
17:    for all M in E.exes do
18:       UpdateMarkov M
19:    end for
20: end for
```

Algorithm 4.3: Train

To train the model after gathering data in each cycle, we first register all applications never know to preload before. This is achieved by creating an Exe object for it, populating it with Map objects and Markov objects, and adding it to the list of all applications. After that, we update the running status of all Exe objects. Finally we update all Markov objects using the UpdateMarkov algorithm presented in Algorithm 4.4.

Input: M

1: `new_state` $\leftarrow M.a.running + 2 \times M.b.running$

2: **if** `new_state` $\neq M.state$ **then**

3: `time` $\leftarrow$ current time

4: $t \leftarrow$ `time` $- M.timestamp[M.state]$

5: $M.tt[M.state] \leftarrow e^{-\lambda t} M.tt[M.state] + (1 - e^{-\lambda t})(\texttt{time} - M.time)$

6: $t \leftarrow$ `time` $- M.time$

7: **for** $i = 0$ to 3 **do**

8: **for** $j = 0$ to 3 such that $i \neq j$ **do**

9: $p \leftarrow 1$ iff $i = M.state$ and $j =$ `new_state`, 0 otherwise

10: $M.tp[i][j] \leftarrow e^{-\lambda t} M.tp[i][j] + (1 - e^{-\lambda t})p$

11: **end for**

12: **end for**

13: $M.timestamp[M.state] \leftarrow$ `time`

14: $M.time \leftarrow$ `time`

15: $M.state \leftarrow$ `new_state`

16: **end if**

Algorithm 4.4: UpdateMarkov

To update a Markov object we determine the state it is making a transition to. If no transition is happening there is not much to do. Otherwise, we update the transition time going out of the previous state to reflect the transition happening. This is done on line 5 using Equation 3.10. Then we update all transition probabilities in a similar way. This is done on line 10 using Equation 3.10. Note that line 10 is mixing probability values linearly with coefficients adding to 1, so, the result is still a probability value. Finally we record the transition by updating timestamps and current state.

Chapter 5

Implementation

This chapter contains some of the details of the implementation of preload. Preload is implemented in C and is less than 3000 lines of code, and becomes a small 35kb binary when compiled. However, it uses the GLib library[1] for basic data structures and the application main loop. GLib is a convenience library for C programming that is widely used in the GNOME project[2].

5.1 Dependencies

Preload uses the `readahead(2)` system call that is specific to the Linux kernel and supported by the GNU libc implementation. Most other Unix kernels provide system calls with similar functionality, so it is rather straightforward to port preload to other UNIX systems, like the BSDs or Solaris. Depending on the implementation, the `madvise(2)` system call may be used as a substitute. See Section 5.4 for details.

The GLib library that preload depends on is ported to various systems and can be compiled (and is usually available) on all modern and legacy desktop systems.

[1] Available from `http://www.gtk.org/`
[2] `http://www.gnome.org/`

5.2 Configuration Parameters

Preload reads configuration parameters from an INI-style text file that is normally located at `$prefix/etc/preload.conf`. This can be changed at compile time, or using the `--conffile` command line argument when invoking preload. The recognized configuration parameters follow.

5.2.1 Model parameters

The model parameters control various aspects of the model and algorithms as described in Section 3.5 and Chapter 4. The following model parameters are recognized:

- `model.cycle`:

  Type: integer

  Unit: seconds

  Default value: 20

  This is the quantum of time for preload. Preload performs data gathering and predictions every cycle.

  Note: Setting this parameter too low may reduce system performance and stability.

- `model.halflife`:

  Type: integer

  Unit: hours

  Default value: 168 (one week)

  The time it takes for statistic information to lose importance to a level half of the current importance. This parameter controls how soon preload forgets about the past. Setting it to a higher value makes it take longer for preload to change its mind about its beliefs, while setting it lower makes preload more sensitive and

responsive to the current happenings. This is used to compute the decay factor for all exponentially-fading mean computations.

Setting this parameter too high will essentially prevent preload from learning new things.

- `model.minsize`:

  Type: integer

  Unit: bytes

  Default value: 2 000 000

  This is the minimum sum of the length of maps of the process for preload to consider tracking an application.

  Note: Setting this parameter too high will make preload less effective, while setting it too low will make it consume quadratically more resources, as it tracks more processes.

5.2.2 Memory Usage Parameters

The total memory preload uses for prefetching is computed using the following formula:

$$\begin{aligned} \max\ (0,\ & \texttt{memstat.total} \times \texttt{model.memtotal} \\ +\ & \texttt{memstat.free} \times \texttt{model.memfree}) \\ +\ & \texttt{memstat.cached} \times \texttt{model.memcached} \end{aligned} \tag{5.1}$$

where `memstat` is the MemStat object filled with memory statistics of the system at runtime.

The formula above is written such that it can use configuration parameters to express all (interesting) linear combinations of system memory variables. The more memory preload receives for prefetching, the more effective it is, at the cost of more memory used.

The following parameters control how much memory preload is allowed to use for prefetching in a cycle. All values are percentages and are clamped to the range -100 to 100.

- `model.memtotal`:

  Type: integer

  Unit: percentage

  Default value: -10

  Multiplier for total memory variable part of the memory usage formula.

- `model.memfree`:

  Type: integer

  Unit: percentage

  Default value: 100

  Multiplier for free memory variable part of the memory usage formula. Reducing this value makes preload less agressive during the boot time, where most of the memory is free.

- `model.memcached`:

  Type: integer

  Unit: percentage

  Default value: 30

  Multiplier for cached memory variable part of the memory usage formula. Increasing this value makes preload more agressive during steady state after the boot. That is, for the most of the operating time of the system. During this time the kernel typically does not let memory to stay free for long and uses it for page-cache. When preload prefetches files, it is essentially causing other pages to be evicted from the cache.

The default values for the memory usage formula result in:

$$\begin{aligned} \max \quad (0, \quad & -10\% \times \texttt{model.memtotal} \\ + \quad & 100\% \times \texttt{model.memfree}) \\ + \quad & 30\% \times \texttt{model.memcached} \end{aligned} \tag{5.2}$$

which essentially means: use all free memory except for ten percent of total memory, and thirty percent of memory already used for caches.

When a system is in steady state, there is little free memory available since the kernel utilizes most of the free memory for caching. During boot time on the other hand, there is little cached memory and a lot of free memory. Given this, the `model.memfree` and `model.memcached` controls enable tuning preload's aggressiveness during boot process and steady state fairly separately.

5.2.3 System Parameters

System parameters enable customizing aspects of the preload daemon that are specific to the implementation and unlike model parameters, do not directly affect the internal working of the scanning and prefetching subsystems.

- `system.doscan`:

  Type: boolean

  Default value: true

  Specifies whether preload should monitor running processes and update its model state. This is the monitoring subsystem of preload and should normally run, but you may want to temporarily turn it off for various reasons like testing predictions.

  Note that if scanning is off, predictions are made based on whatever processes have been running when preload started again and again, and the list of running processes is not updated at all.

- `system.dopredict`:

  Type: boolean

  Default value: true

  Specifies whether preload should make prediction and prefetch anything off the disk. Similar to `system.doscan`, you normally want this to be enabled: this is the other subsystem in preload. But you may want to temporarily turn it off, for example to only train the model without prefetching anything.

  These two parameters allow turning scan/predict subsystems on/off on the fly, by modifying the configuration file and signaling the daemon.

- `system.autosave`:

  Type: integer

  Unit: seconds

  Default value: 3600

  Preload will automatically save its state to disk periodically, and this parameter determines how often. This is only relevant if `system.doscan` is enabled.

  The state is saved when the daemon exits normally (using termination signals). So auto-saving is not strictly required.

- `system.mapprefix`:

  Type: string

  Default value: *empty*, accept all

  A list of path prefixes that controls which mapped files should be scanned by preload. The list items are separated by semicolons. Matching stops as soon as an item matches. For each item, if it matches the beginning of the full file name, a match occurs, and the file is accepted. If on the other hand, the item starts with an exclamation mark as its first character, the rest of the item is considered for matching, and if a match happens, the file is rejected.

As an example a value of `!/lib/modules;/` means that every file other than those in `/lib/modules` should be accepted. In this case, the trailing item can be removed, since if no match occurs, the file is accepted. It is advised to make sure that `/dev` is rejected, since preload does not differentiate device files internally.

- `system.exeprefix`**:**

  Type: string

  Default value: *empty*, accept all

  A list of path prefixes that controls which binary executables should be scanned by preload. The syntax is exactly the same as `system.mapprefix`.

5.3 Persistent State Storage

The persistent state of the model is stored in a simple text file that is normally located at `$prefix/var/lib/preload/preload.state`. This can be changed at compile time, or using the `--statefile` command line argument when invoking preload.

The file is read and used to populate the model when preload starts, or if it does not exist, an empty model is constructed. It will be saved periodically as set with the `autosave` configuration parameter, when preload is shutting down, or upon receiving the `SIGUSR2` signal.

5.4 Prefetching

Preload uses the `readahead(2)` system call that is specific to the Linux kernel and supported by the GNU libc implementation. Most other Unix kernels provide system calls with similar functionality, so it is rather straightforward to port preload to other UNIX systems, like the BSDs or Solaris. There are an entire family of *advise* system calls that can be used as a substitute, depending on the implementation: `madvise(2)`,

`fadvise(2)`, `posix_madvise(2)`, and `posix_fadvise(2)`. If all fails, one can even use the `read(2)` system call to fetch data into main memory, this is a bit wasteful however, as it makes the kernel copy the data into user-space unnecessarily.

One way to improve prefetching performance is to make sure all files to be prefetched are queued instead of sequentially read, such that the kernel has more opportunities to avoid seeking around the hard disk. However, the `readahead(2)` system call is implemented as a command and will block the caller until the request is fulfilled. The `posix_madvise(2)` implementation specification is more promising with regard to this. However, the current implementation in Linux is synchronous, like `readahead(2)` [25].

The Fedora readahead package described in Section 1.3.2 uses a filesystem-specific API to determine the place of the first block of each file on the disk, and sorts files to be prefetched on the position of their first block and reads them in that order, supposedly reducing disk access time.

Since preload tries to be fairly conservative and not affect system balance in a noticeable way, reading files one at a time is probably acceptable.

Note that preload generates prefetching system calls for all the selected maps even if it prefetched some of them at the previous cycle. This means, if no applications are started or shut down, preload generates the exact same system calls as the previous cycle and if those maps are still in memory, the system call essentially becomes a no-operation.

We also call `sched_yield(2)` after reading every few files to give away processor volunteerly instead of exhausting our time slice. This is useful to keep system responsive as being I/O bound during the prefetching phase, preload will get elevated priority over other processes.

5.5 Resource Consumption

Preload has a modest memory footprint. Its main memory consumption is the model that is kept in memory, and with an uncommonly large setting of 1000 maps and 100 applications, it operates in less than 3MB of memory, the main contender being the Markov objects that are quadratic in the number of applications, and take less than 200 bytes.

The process is in the sleep state most of the time, waiting for the next cycle or blocking on I/O, so the load on the processor is minimal. At each cycle it involves scanning `/proc` to gather data and the mathematical computation to update the model and make predictions. When the system is in an stable state (not swapping or under tight memory conditions) and the set of running applications does not change, preload enters a steady state after a few cycles and stops making new I/O requests. This means that it does not interfere with power-saving activities on laptop systems, like turning the hard disk drive off.

5.6 Running Preload

When invoked, preload starts running as a background daemon on the system. It accepts the following command line arguments:

- `--help`: writes out information about invoking preload and exits.
- `--version`: writes out the version number of the preload binary and exits.
- `--conffile` *`file`*: sets the file containing configuration parameters (defaults to `$prefix/etc/preload.conf`). The configuration file is used for tuning model parameters as well as other behaviors of preload. Configuration parameters are enumerated in detail in Section 5.2.

- `--statefile` *`file`*: sets the file for storing persistent state data (defaults to `$prefix/var/lib/preload/preload.state`).
- `--logfile` *`file`*: sets the file used to write out informative messages to (defaults to `$prefix/var/log/preload.log`). An empty *`file`* argument redirects messages to the standard output.
- `--foreground`: instructs preload to run in the foreground, not as a daemon.
- `--nice` *`adjustment`*: adjusts the nice level of the daemon (defaults to +15).
- `--verbose` *`level`*: adjusts the verbosity level of the daemon. Levels 0 to 10 are recognized with 10 being the most verbose (defaults to 4).
- `--debug`: enable debugging mode. Equivalent to `--logfile '' --foreground --verbose 9`.

When running, preload responds to a variety of signals:

- `SIGHUP`: Reloads configuration file and reopens the log file. Reopening log file allows for rotating the log file (backing up current content and starting a new one) without restarting the daemon.
- `SIGUSR1`: Dumps messages containing the current state and configuration parameters. Useful for debugging purposes.
- `SIGUSR2`: Saves the current state to the state file.
- `SIGINT`, `SIGQUIT`, `SIGTERM`: Saves the state and quits.

5.7 Source Code

The source code of preload is publicly available at `http://preload.sf.net/`. It is also distributed for Debian based systems in the Debian unstable package repository.

The source code is fairly straightforward and closely maps to the algorithms presented in Chapter 4.

5.8 Other Issues

Preload currently does not handle package updates or removals. However, doing that is as simple as removing objects that have a probability of less than a certain threshold, and removed files will be eventually removed from the model.

5.8.1 Floating-Point Precision

A minor issue when implementing the inference algorithms of Section 4.3.1 is how to maintain the products in the probability equations within an acceptable precision. To solve this, we use *log-probabilities*, ie. compute the sum of the natural logarithm of the elements instead of computing their product, and convert back to a probability value when necessary. This drastically reduces the error in the floating point computations and is common practice in machine learning.

5.8.2 Prelink

Prelink is a daemon available on various Linux systems, including Fedora, that runs on the system periodically (once every 24 hour or less frequently), analyzing all installed shared library and applications, and assigning a unique virtual address slot for each shared library, and *relocating* them to that address. The idea is that when running applications, if the shared library can be loaded at its allocated address, no relocation is necessary anymore. This improves application startup-time [8].

Prelink's operation conflicts with preload, in that prelink modifies shared library files, creating a new one and renaming it to the original one. This causes running processes to see deleted files as their maps. However, this is not a major problem because:

- When preload prefetches, it will prefetch the current version of the file, and when a new application starts, it is linked against the current version. Except for the rare occasions that the current version between prefetching and application start-up changes, prefetch always loads the version that is going to be used by new applications started,

- While prelink changes shared objects, the only way it does that is by modifying some relocations structures in the file. The modifications are in-place and do not change the offsets of interesting regions of the file that preload uses in its data structures. So, prelinking does not nullify preload's work.

Preload handles the conflict by behaving as if no maps of currently running applications were deleted.

Chapter 6

Experimental Evaluation

In this chapter we present our experimental evaluation results. Evaluating preload's performance through experiments is inherently hard compared to block/file-based or web prefetching because of the much higher variance of user actions it depends on. For that reason, we break our experiments into two parts: to measure how much prefetching improves application start-up time, and how precise preload's predictions are.

6.1 Experimental Setup

Many prefetching experiments use trace-based simulation to measure performance. This has the benefit of comparing before/after numbers of the exactly same run, reducing noise caused by external factors, and being comparable to results from other systems when performed on standard trace sets. This approach however has its own limitations. Namely, that only hit rate is measured this way, not actual performance improvements on wall-clock operation time [17]. Moreover, most of the trace-based simulations assume an infinite I/O bandwidth.

There exist commonly-used traces for measuring file-system performance (e.g. Auspex, Sprite, Andrew [17]), however, most of them are more than a decade old and so not quite useful today (in fact many of them are not available on the Internet anymore), and

more importantly, they do not have enough information traced to be useful for measuring preload's performance. They do not have any requesting-process information tagged with file accesses. Moreover, trace-based simulation of preload is harder than similar systems because simulating the behavior of the main memory that preload uses for caching is not easy, given all other processes that are using it at the same time (all memory allocations come from the same pool that is the main memory).

6.1.1 Methodology

We perform two sets of experiments: start-up time measurement and hit ratio measurement.

For wall-clock time experiments, we run certain applications multiple times (five trials in average) with and without preload running, and measure their start-up time, ie. the time from launching the application until the application window is fully exposed. The start-up time is measured by modifying the source code of the program to print out the time of day once as the first operation in `main()`, and another time in an idle callback called from the main loop of the application. We then use the average time of the multiple runs as the value reported. For cold-cache experiments, we drop all pages cached from the page-cache. For warm-cache, we start the application, close it, and start again. For testing under preload, we run preload, clear the cache and wait until preload prefetches the application in question, and run the application.

For hit ratio computation we run a modified version of preload over the period of two weeks, with the user(s) using the system normally. Whenever an application is started, the modified preload checks which of the maps that the application requires were prefetched during the previous cycle and counts those as hits; the rest are counted as misses. These numbers are used to compute a total hit ratio for the particular scenario. This is very conservative as a map may already be in memory but not prefetched by preload. In that case it will be counted as a miss.

We test two scenarios: a single-user scenario with a single user using the system for her day-to-day computer uses (email, web, document processing, instant messaging, and games), and a two-user scenario with two users using the system, one of them using the GNOME desktop, and the other the KDE desktop environment. They both use the Firefox browser but use mostly different applications for the rest of their needs.

Moreover, our modified version of preload also implements a naïve prediction algorithm that assigns to each application not-running a starting probability relative to the total number of times it has been started. We compute hit ratio for the naïve algorithm too.

6.1.2 Operating Environment

We have performed all of the experiments on a system with an Intel Pentium M 1.7 GHz processor with 2 MB of CPU cache, 512 MB of main memory, and a 4500RPM 60GB hard-disk drive. The operating system used is a stock Fedora Core 5 distribution, with Linux kernel version 2.6.17-1.2139_FC5, and the default I/O scheduler (AS).

6.1.3 Limitations of Experiment

Our experiments fail to measure effect of preload on the I/O subsystem. In particular, we do not measure how preload slows down I/O-intensive running processes. Moreover, we do not measure preload's hit ratio compared to that of the bare kernel page cache. Instead, we compare it with our naïve prefetching algorithm and show significant improvements. This comparison shows a lower bound on the improvements of preload over not prefetching at all. The experiment also assumes that prefetched files are not evicted from the cache in the next cycle (that is 20 seconds by default). Given the total memory that preload uses for prefetching, and given the LRU cache replacement algorithm, this assumption is fairly realistic.

As we discuss in Chapter 7, measuring the true effects of preload on the overall

Application	cold	warm	preload	gain	# Maps	size
OpenOffice.org Writer	15s	2s	7s	53%	323	90 MB
Firefox Web Browser	11s	2s	5s	55%	288	38 MB
Evolution Mailer	9s	1s	4s	55%	308	85 MB
Gedit Text Editor	6s	0.1s	4s	33%	216	52 MB
Gnome Terminal	4s	0.4s	3s	25%	184	27 MB

Table 6.1: Application start-up time with cold and warm caches, and with preload

behavior of the system is a very hard task. We also discuss in the same chapter other ways to minimize the negative effects of preload that are not implemented in the version we test.

6.2 Measurements

The following two subsections present the results of our measurements.

6.2.1 Startup-Time

Table 6.1 shows start-up time for several applications from a cold cache, warm cache, and with preload prefetching them. Cold cache times are achieved by running the application after clearing the page cache by the command "`echo 1 > /proc/sys/vm/drop_caches`" which is supported in Linux 2.6.16 and newer kernels, designed specifically for simulating a cold cache. Warm cache are achieved by running the application, exiting, and running again. Measurements under preload are performed by running preload, dropping caches, waiting for preload to pass the next prefetching cycle, and run the application.

Table 6.2 shows system boot time and login time with and without preload running. Boot time is from the moment the computer is turned on until the login page is shown. Login time is from the moment login information is entered up to when the entire desktop

	without preload	with preload
Boot time	95s	103s
Login time	30s	23s
Total time	125s	126s

Table 6.2: Boot and login times with and without preload

is rendered completely. As can be seen prefetching during the boot process slows it down, but improves login-time. Since a system once booted is used to login at least once, and possibly many more times, reducing the login time while keeping the total boot+login time constant is a net gain. We discuss the effect of prefetching on boot-time in Section 7.3.

6.2.2 Hit Rate

We modified preload such that whenever an application is started, it checks which of the maps the application requires are prefetched during the previous cycle and counts those as hits, and the rest as misses. It also implements a naïve prediction algorithm that assigns to each application not-running a starting probability relative to the total number of times it has been started. This modified prediction algorithm is not used for prefetching and is only used to compute hit ratio for the naïve algorithm.

We tested two scenarios: a single-user scenario with a single user using the system for her day-to-day computer uses (email, web, document processing, instant messaging, and games), and a two-user scenario with two users using the system, one of them using the GNOME desktop, and the other the KDE desktop environment. They both use the Firefox browser but use mostly different applications for the rest of their needs.

The computed hit ratios are presented in Table 6.3.

Scenario	naïve	preload	improvement
Single-user	93%	93%	0%
Two-user	63%	91%	44%

Table 6.3: Hit rate for preload and the naïve algorithm for two scenarios

6.3 Performance Analysis

While an exact analysis of preload's performance is not possible, we have evaluated two measures of performance: wall-clock application start-up time and hit rate improvement for predictions preload makes over the naïve algorithm. The following subsections shortly analyze the numbers we obtained. The rest of the discussion is presented in Chapter 7.

6.3.1 Startup-Time

Preload improved application start-up time by 50% for larger applications, compared to a cold-cache start. However, for the very same applications, starting from a warm cache is at least twice as fast as preload can achieve. The differences comes from the fact that preload only tracks and prefetches those files that applications use by mapping into their process address space, and access to them does not involved any system calls. As a result, files that the application reads using the `read(2)` system call are not prefetched. The more the application uses `mmap(2)` instead of `read(2)`, the better preload performs on it.

To verify this, we look into Gnome Text Editor and Gnome Terminal applications in more detail. We traced all system calls they make during the start-up for cold cache, warm cache, and under preload. The trace also contains the time spent in each system call invocation. This was achieved using the "`strace -T`" command. Table 6.4 shows the total time spent in system calls for these three cases.

For both Gnome Text Editor and Gnome Terminal, time spent in system calls is

Application	cold	warm	preload	preload minus warm
Gnome Text Editor	3.3s	0.82s	3.2s	2.4s
Gnome Terminal	1.7s	0.07s	1.4s	1.3s

Table 6.4: Time spent in system calls with cold and warm caches, and under preload
The last column shows the difference between the third and fourth columns; that is, the extra time spent in system calls under preload that is not with a warm cache.

almost the same for cold cache and under preload (Table 6.4). That is expected as preload does not prefetch anything that directly affects time spent in system calls[1]. The warm cache case however spends significantly less time in system calls. The difference between system call time with a warm cache and under preload (Table 6.4 last column) accounts for more than half of the start-up time difference under a warm cache and under preload (Table 6.1).

We can confirm that the time difference is indeed accumulated disk access time by checking out which system calls are taking long, and how long. For this purpose we filter all system calls taking longer than one millisecond when Gnome Terminal is starting under a warm cache, and under preload: Under a warm cache there are only five system calls (out of about 4900) lasting more than a millisecond, for a total of 10 milliseconds. Under preload, however, there are 110, half of them taking more than 10 milliseconds each. These are very obviously stalled system calls hit by the disk access time. Of the system calls taking more than 10 milliseconds, 80% are `read(2)`, and the rest are `getdents(2)`. The `read(2)` ones are obviously those reading from a file, while `getdents(2)` ones are for reading directories.

[1]The exception is when a system call reads the same part of a file that has also been mapped and prefetched by preload.

6.3.2 Hit Rate

Our hit rate measurement is flawed because we do not measure hit rate with no prefetching. However, we show that by letting preload use 30% of the page cache it can guarantee a greater than 90% hit rate for all the files that applications map (which from Section 6.3.1, we know are responsible for about half of the I/O stall on larger applications). While preload produces similar results as the naïve algorithm for a single user, we have shown significant improvement for the case of two users using different sets of applications. The reason that the single-user scenario yields the exact same results as the naïve algorithm is that all the maps of all the interesting applications actually fit into preload's share of the cache and so they are all prefetched. However, in the two-user scenario not all the applications that the users use fit in that range, and so preload's application-tracking nature outperforms the naïve approach of prefetching regardless of what applications are currently running.

Chapter 7

Discussion

In the course of designing and implementing preload as a file prefetching system that works on a higher level than previous ones, we faced several issues and problems. While we did not solve every one of them, we grasped an intimate knowledge of how other prefetching systems work. In the following sections we discuss limitations and possible improvements of our approach, and will come up with recommendations for systems seeking to improve application start-up time through prefetching.

7.1 Limitations

Preload's major limitation in reducing I/O stall during application start-up is that it only tracks mapped files. While mapped files are known to be a superior way to access read-only data for various reasons[1], not all applications make use of it. In fact, most of the hundreds of mapped files for the applications we measured are shared libraries and font-related files, handled by the linker and libraries down into the application stack. If applications put more effort to make best use of `mmap(2)`, preload can be more successful in reducing their start-up time. Another source of I/O stall that preload does not help

[1] Using a shared copy for all processes, and avoiding copy to user-space.

with is reading directories. And finally there is one more system call that can cause an I/O stall, `stat(2)`.

It also happens to be the case that while all blocking I/O operations take about the same time to complete, which is the disk access time (10ms in our experiments), the `stat(2)` and `getdents(2)` calls take significantly less memory to cache. So, in a computer with various memory-hungry applications eating all the lunch off the page-cache's plate, it seems most beneficial to go after caching the restuls of these two system calls[2] instead of caching files. This is in fact one of the advantages of the SuSE Preload approach to boot speed-up that we covered in Section 1.3.3. In defense of prefetching files, applications should really avoid performing more than a few `stat(2)` and `getdents(2)` calls. And unlike reading files, changing them to not do this is typically very easy.

If one wants to target all the I/O stalls, they need to be able to instrument all I/O accesses made by applications. This is not feasible to be performed in user-space[3], and so automatically out of scope of preload. When that data is available, it seems logical to prefetch them all, *and* to reorder blocks on the disk to make sure the files required for starting popular applications are put in the same area on the disk to reduce disk access time when reading them. As we covered in Section 1.3.1, this is roughly what Windows XP does. Windows XP however prefetches the files upon application launch. We discuss that approach in Section 7.2.

7.2 Aggressive Prefetching

Papathanasiou and Scott [21] argue that with the drastic growth of processor power and main memory sizes in the past decade, the time may have come to employ aggressive prefetching. However, that is only possible if prefetching is integrated with caching, and

[2]Or more practically, caching the I/O blocks that these system calls read.

[3]It is possible though, by preloading a shared library to sniff system calls, rewriting the binary to generate hints, or by using the debugging API in the kernel, similar to `strace`. However, all have measurable effects on the application.

probably only relevant in other levels of prefetching that can achieve a high prediction accuracy.

For preload, prediction accuracy is hardly a performance measure when you think about what preload does under a steady state (no application starting or shutting down): it prefetches the same set of maps again and again, every cycle. This is a property of the memoryless model. Although when a file is already in memory, the next prefetch request for it is a light no-op, preload still must repeat this every cycle to make sure that the predicted maps will be in memory when the user starts the next application. There are two basic reasons for this: (i) preload does not have a separate cache, nor does it have any control on the cache replacement algorithm, and (ii) the time that the next application starts has a drastically high variance. None of these issues exist in most other prefetching frameworks and implementations. For example, in most file-based prefetching systems, the prefetching engine is implemented in the kernel and has direct control over the cache, but even more important is that patterns in file accesses are mandated by a limited set of commonly-used applications that always access the same set of files in the same order with the same almost constant delay in between.

For the reasons stated above, preload's operation can be best thought of as *keeping the cache warm* for popular applications, based on the set of currently running applications. A major problem with keeping the cache warm without proper integration with the cache is that lots of extra work needs to be done. For example, playing a DVD movie on a computer trashes the entire page cache, because the DVD content is 4.7 GB worth of data read into the main memory over a two hour period and accessed only once, but the kernel has no idea whatsoever that caching DVD contents is hopeless. When we put this DVD playing scenario in contrast to what preload is doing, we get back to the question of whether we really need prefetching to improve performance, and, how much can a more sophisticated, history-based, cache replacement algorithm improve performance. Vellanki and Chervenak [26] raised the same question and demonstrated that well over

half of all accesses in a file-system are cacheable based on history, significantly more than LRU and prefetching in most cases.

Another aspect of the way preload works is whether we need to prefetch prior to application launch to be successful in improving start-up time. The answer is implied to be yes in the design of preload, but that is not necessarily the case. In particular, since we failed to remove all the stall time from application start-up, it may not be unrealistic to start prefetching all files the application needs upon its launch. Again, that needs to be handled in the kernel[4], but if one has the ability to do that, it also means that they have information for full I/O requests (not only maps), then they can get rid of all the prediction logic and polluting page cache and start prefetching upon application launch. If correctly implemented, that can improve start-up time, and improve as much as preload could achieve (about 50% for larger applications). Windows XP does this, as described in Section 1.3.1, and claims to highly improve application start-up time. However, we failed to find any academic evidence of Windows XP's prefetching performance. We found instead a technical how-to on the web [22] that suggests removing the prefetching database files in Windows XP[5] as a way to *speed up Windows XP boot up and shutdown*. Windows XP uses the same mechanism to prefetch files during the boot process. So the technical article may be sacrificing the application start-up time to get a faster boot. This in fact lines up with preload's results of slowing the boot process down, and our measurements of Fedora's Readahead system covered in Section 1.3.2 revealed the same behavior. In our measurements the Readahead service in Fedora slowed the boot process down, and sped up login-time.

[4]Or using a preloaded library or tracing

[5]Located in the `Prefetch` directory in side the Windows folder.

7.3 Improvements

There are various improvements that can be applied on preload, as well as other prefetching systems that are widely in use today and were covered in Chapter 1.

As we noted in Section 7.2, prefetching during the boot process can very well negatively affect the boot time. This is in part due to the fact that the boot process is mostly I/O intensive already. The I/O bus is not fully utilized during the entire boot process, but weaving prefetching requests into the holes of the normal I/O load is a hard problem. The way we implemented prefetching, the I/O load caused by the prefetcher *is* going to delay I/O requested by other processes no matter how distributed it is in the boot process. This is a direct result of the scheduling guarantees the kernel makes about not blocking any process for too long. The rest of the poor behavior can be associated to poor kernel I/O scheduling performance, and in fact Seelam et al suggest that the Anticipatory Scheduler (AS) that is the default I/O scheduler in Linux 2.6 starves processes [23]. The Anticipatory Scheduler works by delaying moving the disk head for a few milliseconds, hoping that the process that caused the head to be moved to its current position may be rescheduled and request I/O blocks around the same position on the disk. This policy has negative impacts on a prefetcher reading hundreds of files spanned all across the hard disk.

Starting at the 2.6.13 version, the Linux kernel supports I/O scheduling priorities, including an *idle* class that is ideal for boot-time prefetching, but unfortunately I/O scheduling priorities are only implemented for the Completely Fair Queue (CFQ) I/O scheduler.

An improvement would be to postpone prefetching until the boot process is done and the log-in screen is shown. This can be performed using the GNOME Display Manager as described in Section 1.3.4.

The newer Linux kernels implement the `MADV_REMOVE` advice to the `madvise(2)` sys-

tem call, but only for tmpfs/shmfs[6] file-systems, and so is not useful for cache eviction hinting by applications.

7.4 Summary of Recommendations

We recommend that systems seeking to use prefetching to improve boot time should limit prefetching to blocks required by `stat(2)` and `getdents(2)` system calls, and do that very mildly, and call `sched_yield(2)` regularly. For further boot time speed up, parallelizing boot tasks should be explored.

To improve the log-in time, it is best to start prefetching when the log-in display manager becomes idle. This is a good time to prefetch: the system is idle, and it can be predicted with high probability what to prefetch. This feature is implemented in GNOME Display Manager for example. When possible, the idle I/O scheduling class should be used for prefetching in this stage.

Application start-up time can be improved by modifying applications to reduce the number of `stat(2)` and `getdents(2)`[7] system calls. Moreover, using `mmap(2)` instead of `read(2)` improves performance on its own, and allows for more prefetching opportunity, like what preload does. Finally, applications can take advantage of the `madvise(2)` system call to let the kernel know that they will need a section of a mapped file, and let the kernel prefetch decide to prefetch it.

File-based prefetching integrated with the cache subsystem may be used to further improve application start-up performance, and reorganizing file layout on the hard-disk can be used if all other routes have been taken.

[6] Two in-memory file-systems.
[7] Usually caused by the `readdir(2)` POSIX function.

Chapter 8

Conclusions

We designed and implemented *preload*, a Markov-based adaptive prefetching scheme that works on application-level predictions. Moreover, preload is implemented in the user-space and does not change the application run-time environment in any sense. This is the first work experimenting with file-system prefetching at this level as far as we know.

Our experimental results show promising improvements on application start-up time compared to cold caches, and a decent hit rate compared to a naïve prediction algorithm. However, being in user-space introduces major obstacles into making preload a competitive solution to the startup-time problem. In particular, not having full information about applications' I/O requests, and lack of strong communication channels with the page-cache subsystem degrades preload's effectiveness drastically, especially under tight memory conditions.

Another inherent problem with the preload design is high variance and low prediction confidence caused by the relatively loose correlation of application start-ups. While we successfully build a model to track application correlations, the fact that application launches are very rare events compared to the timescale that computers work on, an application-level prefetching scheme is condemned to consume huge prefetching memory over practically infinite periods of time. This memory can be used to improve short-term

cache behavior.

Finally, we come up with a set of recommendations for system developers on how to improve boot-time, login-time, and application startup-time without falling back to a prefetcher integrated with the cache subsystem in the kernel. Of course, a file-based prefetcher in the kernel can improve on top of that.

Bibliography

[1] Ahmed Amer, Darell D. E. Long, Jehan-Françios Paris, and Randal C. Burns. File access prediction with adjustable accuracy. In *Proceedings of the International Performance Conference on Computers and Communication*, 2002.

[2] Gretta Bartels, Anna R. Karlin, Darrell C. Anderson, Jeffrey S. Chase, Henry M. Levy, and Geoffrey M. Voelker. Potentials and limitations of fault-based markov prefetching for virtual memory pages. In *Measurement and Modeling of Computer Systems*, pages 206–207, 1999.

[3] bert hubert. On faster application startup times: Cache stuffing, seek profiling, adaptive preloading. In *Proceedings of the Linux Symposium*, 2005.

[4] Fay Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of OSDI '99*, 1999.

[5] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Measurement and Modeling of Computer Systems*, pages 286–295, 2000.

[6] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 257–266, New York, NY, USA, 1993. ACM Press.

[7] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 197–207, 1994.

[8] Jakub Jelínek. Prelink. Technical report, Red Hat, Inc., 2004. [Available Online].

[9] Zhimei Jiang and Leonard Kleinrock. An adaptive network prefetch scheme. *IEEE Journal on Selected Areas in Communications*, 16(3):358–368, 1998.

[10] Thomas M. Kroeger and Darrell D. E. Long. Predicting file-system actions from prior events. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 319–328, 1996.

[11] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference*, Anaheim, California, USA, 1997.

[12] Tara M. Madhyastha and Daniel A. Reed. Input/output access pattern classification using hidden Markov models. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 57–67, San Jose, CA, 1997. ACM Press.

[13] GNOME Display Manager. gdmprefetchlist.in, 2006. [Online; accessed 12-July-2006].

[14] Microsoft. Benchmarking on Windows XP, 2001. [Online; accessed 12-July-2006].

[15] Microsoft. Fast System Startup for PCs Running Windows XP, 2004. [Online; accessed 12-July-2006].

[16] Alexandros Nanopoulos, Dimitrios Katsaros, and Yannis Manolopoulos. Effective prediction of web user accesses: A data mining approach. In *Proceedings of the WebKDD Workshop*, 2001.

[17] Sean O'Rourke. Improving I/O parallelism through hints and history: Future reads and future-reading. [Available Online], 2001.

[18] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, 2003.

[19] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve World-Wide Web latency. In *Proceedings of the ACM SIGCOMM '96 Conference*, Stanford University, CA, 1996.

[20] Ram Pai, Badari Pulavarty, and Mingming Cao. Linux 2.6 performance improvement through readahead optimization. In *Proceedings of the Linux Symposium*, 2004.

[21] Athanasios E. Papathanasiou and Michael L. Scott. Aggressive prefetching: An idea whose time has come. In *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

[22] Dennis Roche. Speeding Up Windows XP Boot Up and Shutdown. [Online; accessed 12-July-2006].

[23] Seetharami R. Seelam, Rodrigo Romero, Patricia J. Teller, and William Buros. Enhancements to linux i/o scheduling. In *Proceedings of the Ottawa Linux Symposium 2005*, 2005.

[24] Daby M. Sow, David P. Olshefski, Mandis Beigi, and Guruduth Banavar. Prefetching based on web usage mining. In *Proceedings of ACM/IFIP/USENIX International Middleware*, 2003.

[25] Rik van Riel. kernelnewbies mailing list archives; Re: readahead'ing questions. [Online; accessed 12-July-2006].

[26] Vivekanand Vellanki and Ann L. Chervenak. A cost-benefit scheme for high performance predictive prefetching. In *Proceedings of SC99: High Performance Network-*

ing and Computing, page 50, Portland, OR, 1999. ACM Press and IEEE Computer Society Press.

[27] Wikipedia. Continuous-time Markov chain — Wikipedia, The Free Encyclopedia, 2005. [Online; accessed 20-December-2005].

[28] Wikipedia. Exponential distribution — Wikipedia, The Free Encyclopedia, 2005. [Online; accessed 20-December-2005].

[29] Qiang Yang and Henry Hanning Zhang. Integrating web prefetching and caching using prediction models. *World Wide Web Journal*, 4(4):299–321, 2001.