Improving System Performance using Application-Level Hints

Björn Döbel
Technische Universität Dresden,
Department of Computer Science,
Operating Systems Group

8th June 2005

Supervising professor:	Prof. Dr. Hermann Härtig
Supervisor:	Dipl.-Inf. Martin Pohlack

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst zu haben. Ich habe keine anderen als die im Quellenverzeichnis angegebenen Literaturhilfsmittel verwendet.

Dresden, 08.06.2005

Björn Döbel

Contents

1 Introduction 5
- 1.1 State of the art . . . 6
  - 1.1.1 Monitoring . . . 7
  - 1.1.2 Resource management . . . 8
  - 1.1.3 Linux kernel modules . . . 9
  - 1.1.4 Caches in Linux . . . 10
  - 1.1.5 Using prefetching to increase application performance . . . 11
- 1.2 Related work . . . 14
  - 1.2.1 Transparent Informed Prefetch . . . 14
  - 1.2.2 Speculative Hinting . . . 15
  - 1.2.3 Integrating prefetching and caching . . . 16

2 Design 19
- 2.1 Disk accesses and application startup . . . 19
- 2.2 Design goals . . . 20
- 2.3 Design decisions . . . 21
  - 2.3.1 Collecting data . . . 21
  - 2.3.2 Modifying the Linux kernel . . . 23
  - 2.3.3 HintMod - the kernel module . . . 24
  - 2.3.4 HintGen - the hint generator . . . 25
  - 2.3.5 Application of hints . . . 25
  - 2.3.6 Result of my design . . . 26

3 Implementation 28
- 3.1 Evolution of the solution . . . 28
- 3.2 Acquiring sensor data . . . 29
- 3.3 Determining the current working directory . . . 30
- 3.4 HintMod internals . . . 31
- 3.5 Minimizing the amount of data . . . 32
- 3.6 HintRec - the hint recorder . . . 34
- 3.7 A look at HintGen . . . 35
- 3.8 ReadPref - the prefetch daemon . . . 36

4 Evaluation 38
- 4.1 The measuring environment . . . 38
- 4.2 Measuring Results . . . 39
  - 4.2.1 Booting . . . 39
  - 4.2.2 Application startup . . . 40
  - 4.2.3 Parallel prefetching . . . 41
- 4.3 Cooperation with other tools . . . 43

5 Conclusions and Outlook 47
- 5.1 Conclusion . . . 47
- 5.2 Future work . . . 47

References 52

List of Figures

1 Observing shared library calls . . . 21
2 Application-Level Hints architecture . . . 26
3 Layout of the read and accept context fields . . . 33
4 Startup performance compared . . . 43
5 Bootchart for my test computer . . . 49

1 Introduction

Modern operating systems try to perform well in common cases, so that they are applicable and usable by many different users on many different computers with many different applications. This is the best an operating system developer can do from his point of view, because general purpose operating systems tend to be more applicable than special purpose solutions in most situations.

From a user's or application's point of view this generality is not always desirable. Custom tailored solutions for resource management often outperform general algorithms implemented in an operating system's kernel. This is because the overall system will perform better as more and more specific knowledge about a user's application domain is available, so that it can adapt to the user's needs.

Knowledge about users' and applications' behavior isn't easy to obtain in most cases. To increase system performance by exploiting useful information, one will have to invest plenty of computing time and memory. This does not speak in favor of implementing information retrieval and data processing within the operating system kernel.

In my thesis I develop an architecture capable of automatically observing a system to collect data on resource usage within this system. This data is transformed into knowledge in form of hints that may be applied to improve the system's performance in similar situations.

I show that most of the architecture may be implemented outside the operating system kernel without losing control or performance. Furthermore, I describe an enhancement to the Linux kernel making data for automatic hint generation available to user space applications. This information will then be read, processed, and applied in user space.

Possible scenarios for my architecture include:

- Monitoring application file access patterns to prefetch files into the cache before they are accessed.
- Recognizing periods of high and low resource utilization and improving overall system performance by performing resource accesses during high load in earlier low utilization periods.

Layout of this thesis

Sections 1.1 and 1.2 introduce concepts that are useful to understand my design and implementation. I introduce *Transparent Informed Prefetch* and *Speculative Hinting*, work related to prefetching application data; afterward work on the relationship between caching and prefetching is explained.

Section 2 introduces an abstract design for my architecture consisting of four separate components:

1. A small *kernel patch* inserting sensors and a basic monitoring architecture to the Linux kernel,
2. A kernel module performing *system monitoring*:
   - triggering monitor runs, and
   - storing and filtering retrieved data.
3. User space *hint generators* processing the collected data and generating application-level hints for later use, and
4. *Hinting applications* making use of the generated hints to increase system performance.

Section 3 presents an implementation of my architecture. This implementation solves a concrete problem: large applications need to read a lot of data from hard disk at startup. If the system knows about this data ahead of time, system idle times may be exploited to read these files from disk and thus bring them into cache. I will show how to solve this problem using my hinting architecture.

I evaluate my implementation in Section 4, showing that application startup times may be decreased up to 50% by prefetching without noteworthy influence on the overall system performance. Finally, I conclude my thesis and present thoughts on future work in Section 5.2.

1.1 State of the art

In this chapter I introduce basic principles and concepts necessary for the understanding of my work. These are:

- *Monitoring* as a way of evaluating the performance of informational systems on-line,
- *Linux Kernel Modules* forming the main part of my monitoring system described later on,
- The *Linux Page Cache*, which is helpful in improving application start up performance, and
- *Prefetching* methods, performing time-consuming read operations before they are explicitly requested by an application.

After introducing these concepts I give an overview of related work, namely the *Transparent Informed Prefetch* (TIP) infrastructure developed at Carnegie-Mellon-University and *Speculative Hinting*, a prefetch system based upon TIP.

1.1.1 Monitoring

In the field of system diagnostics there exist several approaches for testing and evaluating systems, the most widespread approach being hardware and software debuggers. These tools enable developers to control their applications by setting breakpoints at arbitrary positions within their code. Reaching one of these breakpoints, the application is stopped. The developer may then inspect and change values of variables, so that he is able to find errors within the code or influence the application's further control flow.

Unfortunately, the debugging approach has several disadvantages rendering it useless in the following situations:

- The system is highly influenced by being stopped. For many cases, for instance for real-time and safety critical systems, this is not acceptable because it is necessary for these applications to be running all the time.
- A debugger adds code to an application. This may lead to situations, where the application runs fine with the debugging code enabled and produces errors when the debugging code is removed. Adding code may for instance remove occurrence of race conditions or add some new ones.
- Often developers and analysts are not able to predict, which parts of an application are error prone or critical with respect to performance or when these parts of the application will be reached. Working with breakpoints then will require plenty of time and work. It is often not possible to put a human being in front of a long running system for all the time. Critical moments may thus be missed.

System monitors, as described by Kabitzsch in [11], provide another approach to diagnostics and performance analysis and remedy the disadvantages of debugging systems. Monitors automate the task of collecting data over a long period of time. The following are the main elements of a monitor:

1. *Sensors* are inserted into applications at arbitrary positions and generate data, which may then be read from outside the application. This is somewhat similar to the breakpoints provided by debugging systems but sensors do not stop the whole application. It is up to the developer to make sure that the time needed to send out data is as small as possible so that it does not influence overall application performance.

   However, a main difference between sensors and breakpoints is that sensors are hardcoded into an application. The user therefore needs to know the data he is interested in before running the system monitor. With breakpoints it is possible to step through an application one-by-one and check for unexpected behavior.
2. *Efficient data storage* is needed so that storing monitoring data adds as little overhead as possible. Furthermore sensor data must be available in a timely order for later evaluation. A time stamp is added to sensor data by the monitor for that reason.
3. *Triggers and Filters* are used to restrict the amount of sensor data that needs to be stored already at run time. Triggers are used to start and stop measuring at specific points in

time or under specific circumstances, so that only interesting data is collected. Filters are applied to the data collected, to enable selection and classification of interesting data.

An example: A system consists of a processor serving requests and a wait queue of length 10 to store incoming requests, which may not be served at the moment. The system administrator wants to check if the queue's length is large enough and therefore inserts a sensor into the system displaying the current number of elements within the queue. If this output is sent every second, the administrator receives 86.400 numbers a day needing to be checked.

If the administrator knows that the system is under heavy load mainly from 11 a.m. to 3 p.m., he could insert a trigger into his monitor starting the collection of data at 10.50 a.m. and stopping at 15.10 p.m. He then would receive only 15.600 numbers, which is still a lot.

As the administrator wants to know if the queue is long enough, he is not interested in all the data telling him that the queue is long enough, he only needs to know whether there are times when the queue is full. To only obtain these numbers, he can use a high-pass filter storing data only when the number of elements within the queue is equal to the maximum queue length. This will once again decrease the amount of data collected.

4. Monitors may produce an immense quantity of data. Checking this for abnormal patterns requires investment of time and is error prone when performed manually. Therefore there exist *diagnostic tools*, which support the user basically by visualizing data so that irregularities may be discovered more easily.

Also system monitors have their disadvantages. Inserting sensors into the system is not always easy. If the system's internals (e.g., the source code or its hardware layout) are not accessible to the analyst, this may even be impossible.

My application-level hint architecture will make use of system monitoring techniques to retrieve data on applications' behavior from the operating system.

1.1.2 Resource management

Resources in the field of operating systems summarize all hardware and software utilities needed by a process to perform its tasks. Resources may be peripheral devices (printers etc.), hardware (e.g., memory), and also software constructs like for instance semaphores.

Most resources are limited in quantity and may be restricted to be used exclusively by one process at a time. This means, not all processes may gain access to a resource whenever they want. It is the operating system's task to manage the assignment of resources to processes. A basic principle of resource management in general purpose operating systems is *fairness*. To be widely applicable, the resource management has to assure that a process may acquire a requested resource within a finite amount of time. In most cases the operating system does not make assumptions on future resource usage of processes when deciding, which resource to assign to which process, mainly because this is either impossible or trying to predict the future would require too much processing time and thus slow down the system.

However, fairness is not always the most profitable way of resource management with respect to the user. Most users on desktop computers have a certain set of applications they use for their daily work, a kind of *working set*. A software developer's working set may for instance include a development environment, a compiler and a debugger, whereas a secretary's working set might consist of a word processing application and a calendar application.

With the working set being constant over a longer period, an operating system could be regarded as quick and responsive when only performing well for the applications belonging to the user's working set. Unfortunately there are millions of different users out there, each with his own working set, so there is no way to predict type and number of applications within a specific set. Within this work I have a look at how to create an architecture on top of the operating system, spending system idle time to improve performance with respect to the applications of a user's application working set.

1.1.3 Linux kernel modules

Monolithic operating systems include all subsystems necessary for the system within the OS kernel. These are for instance memory management, scheduling of processes and resources, and device drivers. In contrast *microkernel* operating systems only provide basic functionality within the kernel while the rest of the services (file systems, device drivers, ...) is implemented in form of user space applications.

Adding new functions to a monolithic operating system is uncomfortable as it requires recompiling and rebooting the operating system. However, adding something new can be a frequent task. It is for instance needed when the user purchases new hardware and this hardware's device driver needs to be added to the system. A solution for that would be to provide drivers for all possible devices within the kernel. This fails for two reasons:

1. There are many different types and versions of hardware, so that providing drivers for every device will require a lot of time and make the operating system kernel very large. As most users only own a small amount of hardware, drivers for all the rest will be of no interest for them.

2. New hardware is announced every week. An operating system with all drivers inside will fail to collaborate with newer hardware. It thus lacks extendability.

Kernel modules are Linux' approach to this problem. Kernel modules are programs that may be inserted into the Linux kernel at runtime using the `insmod` utility. Once inserted, the kernel module runs as part of the kernel until it is removed. Modules are mostly used to provide new device drivers within the Linux kernel. However, it is also possible to add different functions to the kernel (e.g., an operating system monitor).

Kernel modules have access to all public Linux kernel data structures. This may lead to design flaws, because there would always be developers accessing a hard disk driver directly from their memory management module — simply because it is easier than sticking to the correct interfaces. As a consequence, this way of development leads to obfuscation and invisible dependencies between certain kernel subsystems. To prevent these errors, kernel and module developers explicitly declare their methods as public using the `EXPORT_SYMBOL`-macro. These symbols define a module's interface and are the only ones that may be accessed all over the modular kernel.

Bovet and Cesati [2] state that kernel modules were not only introduced because of their software engineering aspects, but also because they provide some of microkernels' advantages over monolithic operating systems (modularity, extendability) while not suffering microkernels' performance losses.

1.1.4 Caches in Linux

Caches are used to make often needed data quickly available. This means that data is copied from a slowly accessible device — like for instance a hard disk — to a device that provides faster accesses for instance from the computer's main memory. We can distinguish hardware and software caches. Hardware caches are controlled directly by hardware. For instance modern CPUs include Level-1 and Level-2 hardware caches that are managed by an internal cache controller.

Software caches — as the name says — are managed by software. Applications control their own software caches internally. Web browsers for instance store a cache of recently visited internet pages. Other software caches are controlled and used by the operating system. The cache that is most often used by the Linux kernel, is the page cache. This cache uses physical memory that is currently not allocated by other applications to store data read from files for later accesses.

Whenever an application accesses a file, Linux first of all tries to serve the request from the page cache. If this is not possible, data is read from the underlying device (e.g., a disk) and is stored within the page cache. As memory accesses to the page cache are much faster than hard disk accesses, cached read operations perform better than those from disk.

A *page* is the basic unit in Linux' memory management. The main software cache is therefore managed on a per-page basis. A page in virtual memory may contain blocks of files residing in physically non adjacent disk sectors, so that it is impossible to exactly identify a page only by the disk's major and minor number and a single sector number. An exact mapping between pages, file blocks and hard disk sectors is however necessary for read and write operations.

To manage pages containing file blocks, every page within the page cache is mapped to an `address_space` object. This structure then establishes a logical connection between an inode (identifying a file) and the corresponding part of the page cache.

Main elements of an address space object are:

- `host` pointing to the inode owning this part of the cache,
- `clean_pages`, `dirty_pages`, `locked_pages` - lists of all pages of the file residing within the page cache,
- `nr_pages` - the number of pages contained by the address space, and
- `a_ops` - a set of function pointers used to access and modify address space objects.

On modern computers the page cache may occupy hundreds of Megabytes of data, caching data from thousands of different files. Searching all those pages therefore becomes a time consuming operation. To improve performance in searching the cache, the Linux kernel maintains a page cache hash table, which is indexed by a hash function incorporating a file's inode and an offset into this file.

1.1.5 Using prefetching to increase application performance

Most of the file accesses performed by applications are carried out synchronously. This means the application is blocked by the operating system until the data to be read has been fetched from hard disk or cache. As hard disk accesses are much slower than memory accesses, read operations take a long time when the system's caches are still empty (so called "cold cache").

If no other applications are running in parallel to a read operation being carried out, the CPU is idle until the data arrives. Decreasing system idle time (and thus increasing resource utilization) is a main design goal for operating systems as long idle and input–output times make the system appear unresponsive to user interaction.

Asynchronous input–output prevents applications from blocked waiting for the completion of read and write accesses. This kind of input–output returns control to the application immediately, signaling whether data is currently available. The application may then perform other computations and will receive a signal when the input–output operation has been finished. This increases CPU utilization.

On the other hand, more context switches are necessary, signal delivery and signal handling also consumes time. Asynchronous input–output therefore does not always increase system performance. Practically, when performing many small asynchronous read operations the overhead will be perceivable.

There are two more drawbacks to the asynchronous approach:

1. Programmers have to adapt their application to the needs of asynchronous input–output. This will require a large investment of time and work and will complicate programs.
2. Asynchronous input–output is a solution if it is possible for the application to determine the files to be read at a later point in time. The application may then start the

read operations, perform other tasks while the data is being fetched and will afterward be able to use the data read. Unfortunately an application's further program flow often depends on the data read from a file. In such cases the application must wait for the input–output operations to be finished to continue operation. If this is the case, asynchronous input–output is not an option.

An example where asynchronous input–output fails is starting up a large application. Libraries are loaded, configuration files and other data needed for the application's work are being read. Waiting for the data may not be avoided in such cases.

Another solution for the previously exposed problem is *prefetching*. Prefetching means to read data from hard disk before it is actually requested by an application. Knowledge about future data accesses is needed to apply prefetching.

Prefetching is an old idea to increase application performance and hence has been examined by many researchers. It is mostly performed by software, although it can also be carried out by special hardware.

One can distinguish three main approaches to prefetching that have been examined:

1. Heuristic prefetching,
2. Statistical prefetching, and
3. Informed prefetching.

Heuristic prefetching derives its name from the ancient Greek $\epsilon\upsilon\rho i\sigma\kappa\epsilon\iota\nu$ meaning "to find." Heuristics name information on a process or system that is commonly known. Heuristic prefetching uses these facts to predict read operations. For instance, files are often read sequentially from the beginning to the end. Hence it is reasonable to prefetch a whole file's data when an application starts to read this file. Furthermore the probability of disk blocks being accessed sequentially is also high enough, so that disk controllers mostly read more blocks from a hard disk than are requested by the operating system. The rest of the data is stored within the hard disk's cache as it is highly probable to be read soon.

Statistical prefetching uses data acquired on an application's behavior by means of statistics and probabilistic calculations. Many applications access the same files in the same order every time. These file access patterns may be observed and then be used to prefetch these files so that their data is within the cache when the application needs it. This observation holds for instance for startup of large applications.

Probabilistic methods consider differences in the obtained access patterns and calculate the file with the highest access probability. For instance, if file A is followed by file B in ten observed patterns, while file A is followed by file C in only one pattern, the probability that file B is needed after file A is much higher than for file C.

The drawback of probabilistic and pattern-based solutions is their low flexibility when it comes to changes. A software developer might for instance heavily use a working set of files over a long period of time. But then, after the project is finished, these files will not be touched ever again. A probabilistic approach that simply uses the number of times a file was accessed, will still prefetch these files. CPU cycles are then wasted to prefetch files that are not needed. Further these files will then consume memory space that is possibly not available anymore for the files with which the developer really needs to.

This drawback may be overcome in two ways:

1. *Windowing* — only consider files that have been accessed within a certain time window to calculate the probabilities. This will lead to a faster solution to the problem mentioned before. However, it is difficult to determine the window's size. If it is too large, unneeded files are prefetched. If it is too small, the probabilistic calculations may be incorrect.

2. *Weighting* — more recent file access patterns will receive a higher weight when computing the access probability. This will help newer files' access probabilities to raise fast, while older files' probabilities will sink.

Unfortunately statistical prefetching does not work for applications without perceivable access patterns, like `grep` or `make`. For these applications necessary files are known earliest at the moment these applications are started, so that prefetching is not possible before that.

Informed prefetching uses hints dynamically generated by an application to prefetch data as soon as it is needed. Hence, informed methods receive their knowledge directly from the application, thereby easily adapting to its special needs.

The main drawbacks of informed methods are:

- That the operating system kernel needs to support informed prefetching,
- Applications need to be altered to produce dynamic hints at runtime, and
- Applications adapted to generate correct hints may be favored by the operating system above others, thus contradicting fairness.

In addition to the three prefetching approaches mentioned previously, another automated approach exists. Mowry [16] and Luk [14] describe possibilities to automatically add prefetching operations at compile time and thereby increase application runtime performance. However, these approaches are out of my thesis' scope, because my goal is to improve performance for existing applications without modifying or recompiling them.

Within my thesis I design an architecture to retrieve resource usage patterns from the operating system. I then implement a tool to improve startup performance of large applications by using statistical prefetching.

1.2 Related work

In this section I give an overview of prefetching implementations. At first I introduce *Transparent Informed Prefetch* (TIP), developed at Carnegie-Mellon-University and described by Patterson and Chang [20, 6, 5]. TIP is a change to the buffer management within a Unix 4.3 BSD kernel.

Speculative hinting described next, uses TIP to dynamically generate hints on accessed files. I conclude this section with information on work related to the connection of prefetching and caching.

A main aspect of my thesis will be automatic generation of *hints* providing information on applications' resource usage. Opposite to existing approaches described here, I try to generate and use these hints mainly in user space, because I hope that this will extend the possibilities to apply my architecture.

1.2.1 Transparent Informed Prefetch

Developers of TIP came up with the idea to enable applications to send hints on future read operations to the operating system. TIP is a patch to the cache management of a BSD Unix providing `ioctl`'s to transmit the following kind of hints to the operating system.

- File name,
- Areas of the file that will be accessed, and
- Number of accesses planned.

Further TIP changes the page replacement algorithm used by the cache, to fit its needs. The basic LRU page replacement used by most Unix systems, is not suitable for TIP as it first replaces pages that have not been accessed for a long time. As TIP brings pages into the cache that have not yet been accessed, it has to ensure that these pages are not replaced before they are referenced for the first time. Furthermore knowledge about the number of accesses to a file may be used to determine when a page is not needed anymore.

Adapting applications to use TIP has several disadvantages. The system's performance increase mainly depends on the correctness of the hints generated by an application. With good hints and a good program structure an applications execution times can be reduced by 50%. Bad hints however will lead to unnecessary read operations, thus significantly slowing down the system.

Application developers may not always be in the lucky position to predict their application's file access patterns. In such cases Heuristic hinting may not be applied, however dynamic generation of hints at runtime might be successful.

A first attempt of dynamic hint generation was the following experiment:

The text searching application `grep` is provided with a list of files to search through at startup. This list may not be predicted earlier but without it prefetching is not possible. Heuristic hinting therefore fails in this situation. TIP developers started `grep` from a shell script. The script started a prefetch application in user space, providing it with the same list that with which `grep` is started. Then `grep` itself is launched.

This type of prefetching in user space was not successful, because:

- Starting the prefetch process at the latest time possible requires resources and puts heavy load on the operating system.
- As the prefetch process runs in parallel to `grep` within user space, file accesses of both applications could not be synchronized. This may lead to the following situations:
  - The prefetch process falls back behind `grep` thus only reading data already accessed by `grep`. Execution time will not be any smaller than without prefetching, probably it will even be higher because an additional application is putting load on the system.
  - The prefetch process reads data too fast so that the cache becomes filled and the page replacement algorithm deletes pages from the cache that have not yet been read by `grep`, thus also having no positive prefetching effect.
  - Both processes compete against each other with respect to using the hard disk at the same time. This leads to several seek operations that would have been avoided without the prefetching process and consume a reasonable amount of time.

1.2.2 Speculative Hinting

From their results TIP developers concluded that effective informed prefetching could only be achieved, if it was possible to manipulate binary files directly for hint generation. An approach pursuing this way is called *Speculative Hinting*, developed in the late 1990ies and described in by Faye Chang in [6, 5].

Speculative hinting requires the following preconditions to be met:

1. The underlying operating system has to provide an appropriate hint and cache management. Speculative Hinting is therefore developed based on TIP.
2. The operating system needs to support a thread model and needs to deal with thread priorities.

If these conditions are satisfied, it is possible to add a low-priority thread to each application. This thread will then only run, whenever no other threads of this application are runnable — for instance because they are blocked waiting for input–output.

In this case the additional thread begins to *speculatively* execute the application's code. All future code is executed and thus possible further read operations may be found. If the thread steps over such a read operation, hints are issued to the underlying TIP system and data is prefetched.

Speculative execution has several side-effects, like for instance variables being changed during this process. To circumvent these problems, the speculative thread is not allowed to execute any system calls apart from accesses to the hinting system and read operations. Variable values are backed up by a software copy-on-write strategy. Before each write access to a variable it is copied by the speculative thread so that the original value is not modified. This practice is called *shadow copying*. For shadow copying to work without having an effect on the application's main threads (which would cause serious performance losses due to the copying), Speculative Hinting creates a full copy of the application's text segment for the speculative thread and only inserts shadow copy operations there.

Adding threads and a shadow copy implementation makes changes to the original application unavoidable. To automate this, Speculative Hinting comes with a tool called SpecHint, which is able to modify executables to use Speculative Hinting. This results in considerable increases of program size while being able to apply Speculative Hinting to every application.

The previously described procedure has problems with synchronization. It is possible for the speculative thread to fall back behind the original. This case results in no useful hints being generated anymore because speculative execution accesses data already read by the main application. A solution for this problem is to establish cooperation between the speculative and the native parts of the application. The speculative thread stores all hints sent to the operating system within a *hint log*. The original thread counts all read accesses and checks whether this number is smaller than the number of entries in the hint log. As long as this assertion holds, the speculative thread is ahead of the original application and generates correct hints. If this is not the case, the speculative thread has fallen back and needs to be restarted from the main application's current position.

Speculative Hinting produces in favorable results. Application's execution times can be decreased by 70% while the same applications in the worst case do not need more execution time than without Speculative Hinting. However, disadvantages of Speculative Hinting are applications growing much larger by adding the hinting components — in certain cases binaries became up to six times larger than the original executable.

Within this thesis I design and implement a application-level hinting architecture. In contrast to Speculative Hinting I automatically generate hints by monitoring applications' behaviors. I show that reasonable performance increases of up to 50% may be achieved without modifications to the applications and with very few modifications to the operating system kernel.

1.2.3 Integrating prefetching and caching

Prefetching and caching are two ways to improve application performance. Caches improve access times by storing data in fast accessible memory instead of slowly accessible external devices. Prefetching makes sure that data resides within the cache *before* it is accessed by an application.

However, prefetching forces cache replacement decisions earlier than when they would have been needed without prefetching. Therefore it is possible that the caching policy replaces a different cache block than it would replace at a later point in time, thereby making suboptimal decisions. The contradiction between prefetching and caching may be summed up like this: Prefetching needs to be started as early as possible, so that a lot of data may be brought into the cache. However, cache replacement should be performed as late as possible because information on usage of cache blocks becomes better the longer an application runs.

Because of their intersection it seems clear, that the relationship prefetching and caching needs to be considered. Cao and associates were the first to come up with a theoretical background and an implementation for a combined prefetching and caching policy for the Ultrix operating system. Their efforts have been documented in [3, 4].

Theoretically, the authors examined a stream of data references data $(r_1, r_2, \dots)$ and a cache able to store at most K blocks. Fetching a block from disk takes F time units. They formed four rules that need to be fulfilled by an integrated caching and prefetching policy with the goal to minimize the waiting time for all fetches to complete:

1. *Optimal prefetching:* every prefetch operation should bring into cache the next block in the reference stream that is not already in the cache.
2. *Optimal replacement:* every prefetch should discard the block whose next reference is furthest in the future.
3. *Do no harm:* Never discard block A to prefetch block B, when A will be referenced before B.
4. *First opportunity:* Never perform a prefetch-and-replace operation, when the same operation could have been performed previously.

Rules 1 and 2 tell the system, *what to do* when the decision to prefetch has been made. Rules 3 and 4 guide in *when* the decision has to be made and when prefetching should be delayed. As applications work, there will be opportunities to start prefetching. At every opportunity the prefetching and caching policy will have to decide whether to start prefetching based upon rules 1 – 4.

In [3], Cao and associates introduced the *controlled-aggressive* prefetching algorithm. This algorithm is aggressive, because it always starts prefetching when it is allowed by the rules. The authors proved that by overlapping fetch operations with cache references, the total application runtime using their prefetching algorithm is at least $1 + F/K$ times the optimal run time. The algorithm is therefore near optimal, when the quotient of F and K is near zero. This can be achieved in two ways:

1. decrease F by providing fast accessible hard disks
2. increase K by raising cache sizes

With memory being very cheap today, the second option is easy to be used. In [4], Cao and colleagues state that in practice F/K is below 0.02, so that the controlled-aggressive algorithm is very close to the optimal solution.

The controlled-aggressive prefetching algorithm was implemented within an Ultrix kernel using the following three components:

1. A *buffer manager* mapping cache blocks to applications. This was needed, because the operating system performs global cache management, so that applications can only manage a smaller part of the cache themselves.
2. An *application control module* managing each application's cache.
3. A *prefetch control module* receiving prefetch commands from the application, integrating it into the cache management via the application control module and scheduling prefetch disk jobs by interfacing with the disk driver. Prefetching commands are issued from an application itself as it is done in Speculative Hinting.

Measurements in [4] show that application runtimes can be decreased by up to 46% using the integration of caching and prefetching described above.

The architecture and implementation presented in this thesis achieve performance increases comparable to the ones of Cao while not implementing an own cache management policy.

2 Design

This chapter is going to document important design decisions that I made during my work on automatic hint generation. To highlight practical relevance, I will give a short introduction to a specific problem that can be solved by automatic hinting: long startup times of large applications. After this introduction I will describe the design of an architecture for automatic hint generation and design goals that I keep in mind during design and implementation.

2.1 Disk accesses and application startup

As mentioned in Section 1.1.2, most PC users have a set of applications they work with most of the time. For them it is especially annoying to see the following happen to applications from their working set: An application starts with CPU-intensive operations and then blocks for a read operation. After this is done it continues until it has to stop for the next disk access and so on. While waiting for disk jobs to complete, the system is often idle. This leads to low CPU utilization and long waiting times for the user. On the other hand of course peripheral devices are idle during times of high CPU utilization, but in most cases this is not accompanied by unpleasing effects for the user.

Name	cold cache startup	warm cache startup	gain in %
GDM (login manager)	16.01 s	9.17 s	42.72%
KDE (desktop environment)	28.07 s	10.88 s	61.24%
Mozilla (web browser)	10.42 s	2.52 s	75.82%
Kate (editor)	4.97 s	2.41 s	51.51%
The Gimp (imaging software)	9.86 s	6.8 s	31.03%

Table 1: Differing startup times with cold and warm cache

Table 1 shows varying startup times for some widely used Linux applications. One will observe that these startup times are longest for the first time an application is started after system bootup. There are two main reasons for that:

- *Cold caches:* On a freshly booted system all data has to be read directly from disk, no data is located within the cache. This empty cache state is called *cold cache*. As disk accesses take much longer than memory accesses, cold caches lead to a large penalty for disk operations. If the application is shut down and restarted at a later point in time, it is likely that the data still resides within the cache. Startup is then much faster than for the first time.

  Table 1 shows how startup times for applications with cold cache differ from those with warm cache. [1]

  If the operating system knows about future file accesses, it can use disk idle time to warm up the cache.

[1]For exact details on how these times were measured, please refer to Section 4.1.

- *Distribution of disk jobs:* Applications usually do not read a whole file at once. The file is read using many separate read operations distributed over a long period of time instead. Additionally, read operations may depend on preceding read operations to be completed because the application needs to compute them from previously read data. In this case and if the timely distance between operations exceeds the anticipatory I/O-scheduler's deadline — developed by Sitaram Iyer and Peter Druschel [8, 9] — there is no possibility to merge these jobs into a larger one. Thus the operations have to be carried out separately, which may result in many unnecessary moves of the disk's read head. As the movements are controlled by a mechanical actuator, they are slow and one of the reasons for long disk access times. In addition to that, overhead for processing requests increases.

  However, if the operating system knows about future disk accesses, it is possible to merge everything into a small number of relatively large read operations and start these within a short period of time. The system's I/O scheduler may then do its best in optimizating disk jobs as it gets to know all future disk requests at once.

For both reasons wait time may be reduced by acting provident. To do so, it is necessary to collect knowledge about applications' resource usage. A main part of this thesis will therefore deal with retrieving and processing this knowledge.

2.2 Design goals

Throughout this thesis I develop an architecture, enabling users to gather information on specific applications' resource usage. This information will then be used to increase system performance with respect to these applications. As this information is gathered from and processed by applications and is used to give hints to the operating system, I will call this information *application-level hints*.

This architecture will be *as general as possible*, hoping that it will thus not only be applicable for the problems I am aware of at the moment, but also for problems emerging in the future. On the other hand I introduce an implementation of my architecture solving the afore mentioned problem of application startup times.

Increasing performance for specific applications is good, but not applicable if there are noticeable decreases in other areas such as slowing down the rest of the system while obtaining application-level hints. I therefore try to design and implement my solution as performant and *as less intrusive* as possible.

The system slows down while collecting and processing application-level hints. Furthermore, when modifying the operating system kernel, unwanted side effects may easily be produced, destabilizing and slowing down the system. To avoid this, I make as few changes to the Linux kernel as possible.

A final goal of my architecture is *ease of use*. Only solutions providing the user with an easy and automated way of application are considered useful in daily practice. There is not enough time to come up with a completely user-friendly solution from research work. However, I try to ensure that there will be no major barriers preventing later improvements of usability.

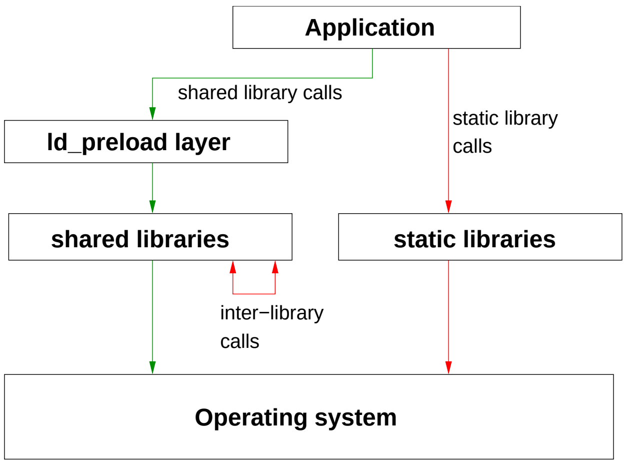

Figure 1: Shared library calls are issued through the `ld_preload` layer while inter–library calls and calls to static libraries may not be examined using this approach.

2.3 Design decisions

2.3.1 Collecting data

Collecting data is the first task to be accomplished for generating application level hints. The problem is retrieving data on an application's actions from the operating system. In Linux this can be done in several ways, as there are already tools available supporting such tasks:

1. The *`strace`* tool enables users to protocol all system calls performed by an application and all signals sent to this application. It therefore lists the system call, its parameters and its return value.

   However, `strace` has got its drawbacks. It is simply unusable when it comes to handling multiple threads of an application as its output gets extremely complex then. Another drawback is the lack of absolute path names on file accesses. These are essential to generate correct application-level hints. One could however regenerate absolute paths from the relative ones by monitoring calls to the `cwd` system call.

   Measuring execution times of `strace`'ed applications, I found out that execution with `strace` took at least 20% more time than without it. This is another handicap that prevented me from using this tool.

2. System calls in Linux are wrapped by function calls to the standard C library. `ld_preload` is a linking mechanism that gives programmers the opportunity to catch calls to specific library functions and provide their own handling. The programmer therefore provides his own implementation of these functions within a shared library and specifies this library within the `LD_PRELOAD` environment variable before executing the application he wants to observe. The dynamic linker ensures that `LD_PRELOAD` libraries are searched for function definitions before all other shared libraries.

`ld_preload` thus establishes a layer between an application and the shared libraries used by this application without access to the application's source code. For purposes of collecting hints one could generate logging output within this layer and then call the original library function manually. This solution is elegant and can easily be implemented.

Performance decreases caused by wrapping functions with `ld_preload` are acceptable — Fetzer [7] points out, that execution times for wrapped functions are increased by up to 10%.

`ld_preload` is also not the tool of choice, because it only wraps calls to library functions issued by the application. The shared libraries defined by the `LD_PRELOAD` variable are only searched, when an application issues calls to a shared library. Whenever the application calls an internal function or a function provided by a statically linked library, the function call is not caught.

For the special case of acquiring knowledge about file access patterns `ld_preload` can be a solution. One can build wrappers for the `open()` and `read()` library functions and therein obtain all necessary data. However, I did not chose to use this possibility, because it puts to many constraints on other future solutions with working only for shared library calls, but for instance not for function calls within an application or within the kernel.

Figure 1 shows functioning and problems of `ld_preload`.

3. It is possible to change the system call table of a running Linux system from a kernel module. This is described in [18]. Therewith the idea of `ld_preload` may be extended to observing system calls from a module. This restricts a solution to observing system calls only, although this might be sufficient for instance for solving the problem described in 2.1.

4. In Linux it is possible to implement an own *operating system monitor*. This is of course more complex than the previous alternatives but may be adapted to the architecture's needs. Developers gain the highest level of control over the system this way. Calls to and within the kernel may be observed at whatever granularity is necessary. Major drawbacks of an own implementation are complexity and a higher probability of software errors as the other solutions have already been tested for years.

After looking at the preceding possibilities, I decided to work on an own operating system monitor as this was the only solution fulfilling my requirements:

- Arbitrary data needs to be retrieved from kernel subsystems, and
- My architecture's users need to have complete control over the type and amount of obtained data, and
- A system monitor enables me implementing it being as less intrusive to the kernel as possible.

2.3.2 Modifying the Linux kernel

Modifications to the operating system kernel are critical with respect to performance losses, because the Linux kernel is complex and one can easily establish unwanted side effects. However the kernel needs to be adapted to create an operating system monitor: sensors must be inserted to points of interest and the sensor data must be provided to user space applications through some kind of interface.

With one of my design goals being to make as few modifications to the kernel as possible, I decided to use the concept of kernel modules as far as possible. Within the kernel I need sensors collecting data and means to send data from these sensors to a data storage. A stub function will be introduced to the kernel. Kernel modules may register to receive data via this stub function. If no modules register themselves, the function will simply discard the sensor data.

Storing data within a kernel module is necessary because of another design goal: speed. A user space application reading out data directly from the kernel sensors will require lots of system calls, thus slowing down the system because of many switches between user and kernel mode. With a kernel module this data may be stored internally and then be handed over in larger chunks so that less communication between kernel and user space is needed. Martin Pohlack's kernel patch described in [21] built up the basis of my enhancements to the Linux kernel.

Sensor data within the Linux kernel will consist of integer values as this is the most common way of handing over data within the kernel. The kernel-internal stub function therefore will be able to hand over a certain amount of integer information at a time. This data will then be added a *time stamp* and an *event type*, so that user space applications processing this data will be able to distinguish data from different sensors. A time stamp is needed to obtain information about events' order. The complete values will then be sent to the kernel module for intermediate storage.

In certain cases, integer data will not fit the needs and textual data will be needed (e.g., to determine the file name during the `open()` system call). Therefore I will also add a possibility to hand over strings from the kernel to the kernel module. This will be basically done by sending a pointer to the string and its length to the kernel module. The module then has to find out that this data is a string by looking at the event type and will thus be able to handle strings separately, if this is required for monitoring.

Apart from having an event type with them, sensor data does not have any semantics associated within the kernel. Most semantics will be added by user space applications processing data read from the kernel. The solution is therefore applicable also for future uses: programmers simply add sensors to the kernel where they need them and then read this data out via a kernel module.

Special care has to be taken for handing out strings from the kernel: As handing out pointers to a string within the kernel to user space applications will not suffice, because these applications must not access the kernel address space, we need to separate string data from integer data already within the kernel module.

2.3.3 HintMod - the kernel module

As already mentioned, the kernel module is used to store monitoring data before handing it out to user space applications. HintMod will have to separate textual data from integer data and provide different interfaces for reading out these:

1. `/dev/hint` is the device read by user space applications to obtain integer data
2. `/dev/hintname` is used by these applications to retrieve textual data

So how is data managed within HintMod? A memory saving solution is holding linked lists for both types of sensor data and dynamically adding new data members whenever it is necessary. However dynamic memory management is rather slow. Therefore I decided to statically allocate buffers for sensor data. This will provide a faster data storage and is an easy way to prevent the kernel module from filling up too much memory.

For every event type, HintMod manages a separate ring buffer. Separating these buffers is needed to perform filtering. The user may choose to read data for specific event types while ignoring others, which may then be retrieved later. Filtering is already done within the kernel module, because the amount of data handed over to user space applications may thus be minimized. Reading out only the necessary data to user space will lead to faster read accesses to the hint devices.

Accessing the ring buffers may lead to race conditions. Processes may try to add data to a ring buffer in parallel. This would cause unpredictable data losses. To prevent such situations, spin locks are used to guard the sections for adding data to the buffers.

Reading out sensor data in user space needs to be concerned, too. With static ring buffers, HintMod's memory is limited. Therefore it will not be possible for long monitoring runs to store all the data within the module and retrieve them afterward when time is not a critical component anymore. User space applications have to be able to read out data during monitoring instead, therefore it is necessary to read out data fast and with a small memory footprint. Even then, reading data during monitor runs will have an influence on the system — the only thing we can do is to keep this influence as small as possible.

To compress data, the `/dev/hint` device provides data in a binary format. The reading user space application will have to know this format and create human or machine readable representations of it, depending on the needs of user space hint generators.

2.3.4 HintGen - the hint generator

Data processing will be completely carried out by user space applications. HintGen is responsible for interpreting sensor data and generating useful hints for later use. Three main steps are necessary to generate hints:

1. Data is read out from the HintMod kernel module. This reading needs to be performed fast, to keep influence on the system low.
2. Data is processed. Semantics are brought in by distinguishing between the different event types, data is sorted, depending on the hint generator's needs.
3. Eventually, application-level hints are generated. Therefore data is processed further by:
    - Merging data if possible to generate compact hints,
    - Removing redundant data,
    - Establishing connections between the different types of events generated by the sensors, and
    - Storing the hint data to be used later on.

The final two parts — data processing and hint generation — are not time critical as they may be performed off-line after monitoring. However reading sensor data needs to be fast. Therefore I will further divide the task of hint generation into:

1. HintRec — a small tool dedicated to fast read out sensor data, and
2. HintGen — a larger tool for hint generation.

2.3.5 Application of hints

The final link of my architecture will be applications using the automatically generated application-level hints to improve system performance. A daemon process will be used for that. Even though it is running in background, this daemon may influence the system. To keep this influence low, Linux FIFO mechanisms may be used: At startup, the daemon opens a FIFO for reading. It will block within this operation until another application also opens this FIFO for writing and emits a command through it. The daemon will only be woken up, when it is necessary, sleeping for the rest of the time.

In my implementation of an application-level hint architecture for speeding up application startup, hints will be deployed by ReadPref — the read prefetch daemon. The basic idea of prefetching is to exploit times of low CPU and disk utilization. If applied to the daemon process, this will also support low influence on the system, because it will only work when no other work is being performed. The easiest way to achieve that, is to let the daemon run with low priority. In later experiments we will see, whether this low priority is enough or whether more sophisticated methods need to be applied.

2.3.6 Result of my design

So far I have introduced all components that will make up my architecture. Figure 2 gives an overview of these components and their relationships.

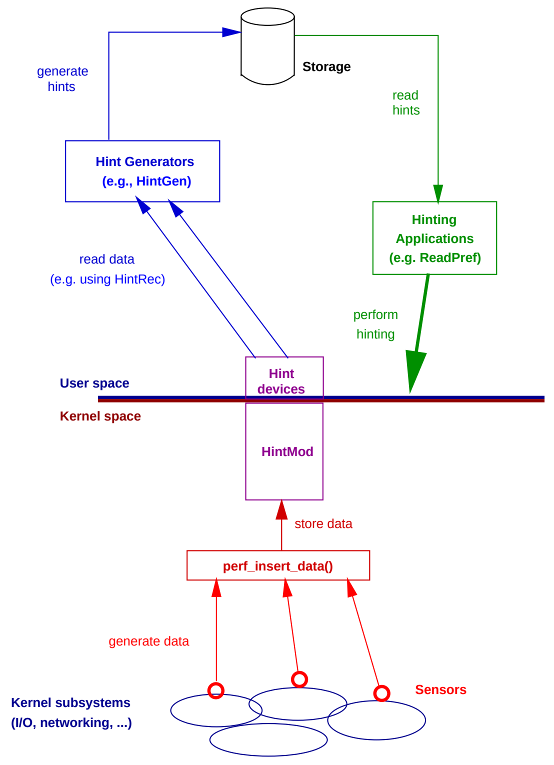
Storage
generate hints
read hints
Hint Generators (e.g., HintGen)
Hinting Applications (e.g. ReadPref)
read data (e.g. using HintRec)
perform hinting
User space
Hint devices
Kernel space
HintMod
store data
perf_insert_data()
generate data
Sensors
Kernel subsystems (I/O, networking, ...)

Figure 2: Application-Level Hints architecture

Besides this functional representation, the components may also be distinguished with respect to their level of abstraction:

1. *HintMod* works completely independent from the problem that a specific solution solves. It only provides means to store data within ring buffers within the kernel.
2. *Reading out data from HintMod* is therefore also not depending on a specific problem.
3. *The kernel patch* consists of two parts:
    - Handing out data to HintMod is independent from the problem.
    - Sensors within the kernel are dedicated to the problem. Programmers will have to investigate, which data is needed from the kernel to find patterns related to their problem.
4. *Hint generation and application* largely depends on the problem to be solved. This is caused by specialized kinds of sensor data being processed for every problem.

Semantics are needed with the data at the moment of data generation and during processing. These components will be specific for each scenario. For data transfer and storage the solution will be more general and may be reused for scenarios different from the scenario discussed here.

3 Implementation

While my architecture's design was developed to be general, making it applicable for a broad range of possible scenarios, the implementation of my architecture will concentrate on the specific problem of application startup times as described in Section 2.1.

In this section I describe how my implementation evolved from the first ideas. Then I will give details for the single components of my implementation which I think will be useful for future work.

3.1 Evolution of the solution

The first idea that came up when thinking about executing disk accesses ahead of time was to exploit information from Linux' disk driver and thus determine the blocks being read from hard disk. These blocks might then be prefetched and brought into the cache.

Unfortunately this did not work as expected, because of the cache management within Linux. As described in Section 1.1.4, the Linux page cache is managed on a per-file basis. This is reasonable for a file system as applications never directly access hard disks. For my solution this is a handicap, because when prefetching disk blocks via the disk file, say `/dev/hda`, these blocks are stored within the disk file's page cache. When accessing the file belonging to a disk block, cache lookups are however performed for the file itself and not for the belonging disk. That's why no data will be found in the cache and read operations will be performed from disk again.

Two solutions for this problem came to my mind:

1. *Implementing a block cache* within the operating system's hard disk driver would make it possible to exploit information on accesses on hard disk blocks for application-level hinting. This is however a large intrusion to to kernel (comparable e.g., to the intrusions made by TIP as described in Section 1.2.1) and does not fit the design goal of being as less intrusive as possible.

2. *Adapting the idea to reality*: Given knowledge about Linux' cache management, we can also base a solution to the startup problem on top of it. Instead of tracing disk block accesses, we need to get information on:

   - File names,
   - Absolute paths, and
   - Which sections of files were accessed.

   Name and path are inevitable because they provide the only possibility for locating the files later on. The accessed sections are interesting because we may thus relieve the cache from unneeded data by prefetching only these sections.

   Further on, files — especially libraries — are not always accessed in sequential order, but in a rather random fashion instead. Prefetching sections from a file in this random order will possibly require unnecessary moves of disk actuators. If we know all accessed sections before, we may prefetch these sections in a sequential order, thus minimizing disk latencies.

I finally preferred to implement the second solution because of its alignment to my design goals. Furthermore it allows not only to prefetch data read from a hard disk but also data mounted via non disk file systems, like for instance a network-mounted NFS, because the Linux page cache is managed in the same manner for every file system due to the comprehensive architecture of the Virtual File System.

3.2 Acquiring sensor data

Sensors within my monitoring approach are active, which means they insert data into the monitoring system. This is basically done by calling the function `perf_insert_data()`.

This function is added to the Linux kernel by my patch and serves as a wrapper method for calling the HintMod kernel module. It first of all tests whether a kernel module has registered a function for receiving sensor data. If this is not the case, `perf_insert_data()` returns immediately and all data is dropped. This is necessary because of two reasons:

1. The kernel needs to be able to run even without a monitoring module inserted. Therefore a wrapper is necessary.
2. The wrapper must return immediately, if no monitoring is performed at the moment, so that system performance is not decreased by executing unnecessary code.

A monitoring module may register itself at the kernel by calling

```
void put_entry_pointer(int (*funcPtr)([...])
```

where `funcPtr` is a pointer to an arbitrary function accepting sensor data within the kernel module. This function has 13 integer arguments and returns an integer value.
`put_entry_pointer(NULL)` may be used to unregister a once registered callback function. Programmers of monitoring modules have to make sure, that this is done no later than at module unloading. Otherwise, the kernel will run into critical errors because of dereferencing a pointer to a non existing address.

If a callback function is registered, `perf_insert_data()` adds a time stamp to the sensor data and then hands data over to the kernel module for further processing.

Creation of sensor data is done by calling `perf_insert_data()` from certain points within the kernel, where the required information is available and may be sent out. The `perf_context` parameter of `perf_insert_data()` determines an entry's event type. All further parameters may be used for arbitrary sensor data. Currently the types of events as shown in Table 2 are defined within my kernel patch, others may be added as needed.

Name	Description
PERF_CONTEXT_DISKIO	IDE disk driver data
PERF_CONTEXT_SCSI	SCSI disk driver data
PERF_CONTEXT_NETIO	Network I/O data
PERF_CONTEXT_SCHED	Scheduling data
PERF_CONTEXT_GENERIC_READ	data from `read()` system call
PERF_CONTEXT_OPEN_FILENAME	data from `open()` system call

Table 2: Currently defined event types

To retrieve data about file accesses that is needed for prefetching files accessed during large application startup, I added the following sensors to the Linux 2.6.9 kernel:

1. The function `sys_open()` in `fs/open.c` handles the open system call. At this point of operation the opened file's name is available as a string. Therefore sensor data of the type OPEN_FILENAME is produced, sending a pointer to the string and its length to HintMod.
2. The function `add_to_page_cache()` in `mm/filemap.c` is adding pages to the page cache. This is the point where we may collect information on accessed sections of a file. Therefore sensor data of the event type GENERIC_READ is generated, sending the following data:
   - the file's inode,
   - offset of the currently added page within the file,
   - major and minor number of the device the file is located on, and
   - current number of this file's pages within the page cache.

For evaluation purposes I additionally wanted to show disk accesses in contrast to CPU load. The Linux kernel provides statistics on CPU load via the `/proc/loadavg` file by summing up runnable and blocked processes. For measuring purposes I added a kernel thread to HintMod, which is generating data of the type PERF_CONTEXT_SCHED by sending this sum of processes to HintMod every 20 ms. This scheduling data is not necessary for prefetching files, it is only in the module for testing reasons. Therefore this sensor is disabled by default.

3.3 Determining the current working directory

File names collected by the sensor within the `open()` system call often are relative. Relative paths need to be transformed into absolute paths, because prefetching must not rely on the prefetching application to be executed from a specific path. There are three possible solutions to determine the working directory:

1. HintMod may check at runtime, whether data of the type OPEN_FILENAME is an absolute or relative path by checking whether the name starts with a slash (for absolute paths) or not. If the file name is a relative one, it then may add the current working directory to the name before storing the string.

2. Another opportunity is instrumentation of the `cwd()` system call, which is handling changes within the working directory. One can simply log all changes and off-line apply them to received data later by examining their time stamps.

3. The hint generator may filter relative file names and try to determine their absolute locations using the `find` tool. This would however require parsing the whole directory tree of a disk for every relative file name, which is time consuming and would not even solve the problem in case of duplicate file names in different directories. Thus it is not an acceptable option.

For my implementation I decided to implement the first option, because it was the easiest one to implement. At first it seemed necessary to implement this solution within the kernel patch itself, thus breaking with the design goal of not putting semantics to sensor data within the kernel. Luckily I later on found that the function `d_path()` from `fs/dcache.c` is able to determine the working directory for a file and may be used from kernel modules.

3.4 HintMod internals

HintMod is my implementation of a kernel module to store sensor data before handing it out to user space applications in a compact format. HintMod registers the function `hintmod_get_entry()` as a callback for sensor data within the kernel (See Section 3.2). For storing data, HintMod handles all sensor entries as data of the struct type `buffer_entry_t`. There are several ring buffers for data of this type — one buffer for each known event type, each buffer being able to store 10.000 entries. 10.000 entries per buffer proved to provide an acceptable trade-off between using too much memory for the kernel module and needing to read out data too often. For every ring buffer HintMod additionally maintains:

- An indexing pointer to the next element to read,
- An indexing pointer to the next element to write, and
- A spin lock to prevent processes from writing to the buffers in parallel.

As ring buffers are limited in size, it needs to be decided what happens when the ring buffer is full. In a classical producer-consumer problem the producer will wait until the consumer has freed one or more elements of the buffer. This is not possible for monitoring, because

- We may not be sure whether data will be read soon and
- Monitoring must not have a large influence on the system. Blocking a process is a fairly large influence.

Because of that, I decided to continue writing to the ring buffer by overwriting the oldest entries. This is acceptable because this data is probably of less importance than the newer entries. The consumer needs to make sure that he reads out data before it gets overwritten.

In addition to the ring buffers, HintMod manages a buffer for text data, like for instance file names retrieved from the `open()` system call. The buffer for this data is 250 kByte large. As data during my monitoring runs never exceeded 180 kByte, this size is large enough and might even be decreased if necessary.

To calculate how often data needs to be read from the ring buffers, let us have a look at the following scenario: As already mentioned a high load on the monitoring system is put by the kernel thread generating CPU load events. This thread adds 50 elements to the ring buffer each second. As the ring buffers are separated, it has no influence on other entries being generated at the same time. As each ring buffer may store 10.000 entries, there is enough space to store CPU load events for 200 seconds.

From that I conclude that the buffer needs to be read out at about every three minutes to not lose any data. The sensors necessary for prefetching files will generate events at a much lower rate. Typical application startups do several hundreds of open system calls and probably thousands of read operations.

Note that there are of course possible scenarios generating more sensor data than in the preceding example. One can for instance decide to insert a sensor into the process scheduler — this will result in lots of events being generated. In this case programmers will need to read out HintMod's devices more often.

Reading data in larger periods will lead to more data being read at a time. Reading more data however takes up more time, thus influencing the system. To avoid that, there are two possible solutions:

1. For rather short monitoring runs where the amount of data does not exceed 10.000 entries for each event type, it is recommended to not read out the data until the monitoring run has finished. This avoids any influence on the system caused by reading monitoring data.

2. For longer runs the amount of monitoring data is hard to predict and may quite well exceed 10.000 entries per event type. It is not recommended to delay reading out data until the ring buffers are completely full, as data might get lost then. Reading out small amounts of data in smaller steps will result in many small disturbances caused by monitoring. These are less perceivable to the user than fewer, but longer, disruptions. It is up to the monitor's user to decide which way of distributing reads from HintMod fits his needs best.

   My implementation of a hint recorder, called HintRec, reads out data periodically every 10 seconds. This period was chosen arbitrarily during my measurements and did not lead to noticeable performance decreases.

3.5 Minimizing the amount of data

As introduced in Section 1.1.1, monitoring systems provide means for triggering data extraction and filtering data, so that only data, which is of concern for the user, is stored. This is reasonable also with respect to the cost of monitoring as the monitor's influence on the system is lower if less data is being collected.

My implementation of the HintMod kernel module supports filtering and triggering. Both are implemented using `ioctl` commands that may be issued to the module from user space.

Triggering is implemented by the ioctls for starting and stopping recording. IOCTL_-STARTREC may be used to set a valid callback function for sensors. IOCTL_STOPREC resets this callback function to NULL, so that no further sensor data is sent to the module.

`ioctl` command	Description
HINTMOD_IOCTL_STARTREC	recording = ON
HINTMOD_IOCTL_STOPREC	recording = OFF
HINTMOD_IOCTL_GET_READCONTEXT	get content of read_context
HINTMOD_IOCTL_GET_ACCEPTCONTEXT	get content of accept_context
HINTMOD_IOCTL_SET_READCONTEXT	set read_context
HINTMOD_IOCTL_SET_ACCEPTCONTEXT	set accept_context

Table 3: Ioctl commands for HintMod

Filtering is implemented by means of an *accept context* and a *read context*. Both contain flags determining, which event types will be accepted by the module (`accept_context`) and those, which are read by reading the `/dev/hint` device (`read_context`). HintMod's sensor callback function `hintmod_get_entry()` checks all entries and only stores them within a ring buffer, if the event's flag within `accept_context` is set. Otherwise the sensor data is discarded. When reading the `/dev/hint` device only those ring buffers are written out to the reading application for which the corresponding flag in `read_context` is set.

`read_context` and `accept_context` have an identical layout which is shown in Figure 3. Table 4 explains the bits used within these fields.

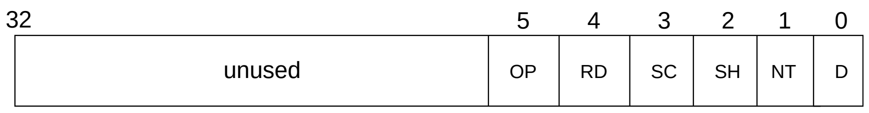

Figure 3: Layout of the read and accept context fields

D	disk I/O data
NT	network I/O data
SH	scheduling data
SC	SCSI disk I/O data
RD	data from the `read()` system ccall
OP	file name data from the `open()` system call

Table 4: Bits used within the read and accept context fields

3.6 HintRec - the hint recorder

The HintRec hint recorder is a small application, written in C, to read out data from HintMod and send `ioctl` commands to it. HintRec accepts the following options:

- `-start, -s` - start recording by issuing the STARTREC `ioctl`
- `-stop, -t` - stop recording by issuing the STOPREC `ioctl`
- `-read, -r` - read data from the `hint` device and print it to the console
- `-readcont, -c` - get the read_context using the GET_READCONTEXT `ioctl`
- `-acceptcont, -a` - get the accept_context using the GET_ACCEPTCONTEXT `ioctl`
- `-setRead <num>, -R <num>` - set the read_context to `num`
- `-setAccept <num>, -A <num>` - set the accept_context to `num`
- `-querycont, -q` - query both of accept_context and read_context

A sample monitoring session using HintMod and HintRec will look like the following and may easily be automated with a shell script:

1. Load HintMod: `insmod hintmod.ko`
2. Empty HintMod's ring buffers as there may be data left from earlier sessions:

```
hintrec -stop
hintrec -read >/dev/null
cat /dev/hintname >/dev/null
```

3. Start monitoring

```
hintrec -start
```

4. Execute whatever actions or applications you want to be monitored. During work periodically read out data:

```
while /bin/true; do
    sleep 10
    hintrec -read >>hintrec_data
    cat /dev/hintname >>hintrec_names
done;
```

5. Finally stop recording and flush HintMod's memory if necessary:

```
hintrec -stop
hintrec -read >>hintrec_data
cat /dev/hintname >>hintrec_names
```

After these steps you will have two files within your directory. `hintrec_data` contains all data read from `/dev/hint`, `hintrec_names` includes all file names read from the `/dev/hintname` device. These files are ready to be processed by HintGen, the hint generator.

3.7 A look at HintGen

HintGen is my hint generator. For the purposes of speeding up application startup times HintGen's input is:

- The file names of all accessed files from `hintrec_names` and
- Data on all the files' sections that have been read from `hintrec_data`.

HintGen parses all the data and generates a file containing application level hints to be used by a prefetch daemon. A large amount of data has to be processed by HintGen, whereas speed is not that important anymore, because HintGen's work is done off-line after monitoring. Therefore I chose the Python programming language to implement HintGen as it enables easier ways of processing data than C does.

HintGen will first parse the input files and build lists containing the file names and object representations of the buffer entries read from HintMod. After this, HintGen will build a HintStat object from the monitoring run, containing separate lists of all events.

After data has been ordered, the following steps are performed:

1. First run through the list of entries describing changes to the page cache. Merge separate read operations affecting continuous sections of a file.
2. Then reorder the disk jobs sequentially by
   - The device they belong to, and
   - The offset into the file.

   As modern file systems mostly store regular files within adjacent disk blocks, this sorting is used to make sure that disk head movements are minimized.
3. Now HintGen establishes a mapping to distinguish, which inodes on which devices have been accessed. This mapping is used to indicate which numerical data from `hintrec_data` belongs to which of the file names contained in `hintrec_names`.

4. With this knowledge we can generate a hint file looking like this:

   ```
   [...]
   /usr/lib/libz.so.1
   /lib/tls/libnsl.so.1
   /lib/tls/libcrypt.so.1
   /usr/bin/startkde
   0       3
   /usr/lib/libDCOP.so.4
   0       51
   /usr/lib/libqt-mt.so.3
   0       495
   508     330
   861     143
   1007    32
   [...]
   ```

   This file contains all files accessed by an application. Next to a file name we find sequences of tuples `(start,length)` where:

   - `start` is the offset within the file from which a section starts, and
   - `length` is the number of memory pages to be read starting from the offset.

   If no such sequences are given next to a file, the application tried to open a file, but was not successful. This happens quite often, for instance when the system's PATH is searched for an executable and when the LD_LIBRARY_PATH is searched for a library. If we produce these failing open operations ahead of time, the kernel's dentry cache — responsible for directory entries — is filled with information that this file does not exist. The application will later not need to look up this information from disk again. Therefore knowledge about failing `open()` operations will help to speed up application startup.

3.8 ReadPref - the prefetch daemon

So far application-level hints have been successfully generated. Now it is time to apply these hints, so that data resides within the page cache before an application needs this data. As described in Section 2.3.5, I therefore implemented a prefetch daemon. This daemon is called ReadPref and will run in the background from the moment it is launched.

ReadPref at first opens a FIFO special file for receiving commands. With Linux' FIFO architecture, ReadPref will be blocked within this open operation until another application opens this FIFO for writing. ReadPref will thus not influence the system until there is work to be done.

Another application may issue commands to ReadPref by writing a string to the FIFO device. ReadPref will then start to open a hint file matching this string and prefetch all the files described within this hint file. After ReadPref has finished its work, data will reside within the page cache and will thus be read much faster by applications than if it was still only available from hard disk.

To be available as soon as possible, I added a bootup script starting ReadPref as a background process at system startup. The bootup script then runs a simple Python script in background that sends bootup commands to ReadPref. Therefore prefetching starts while the bootup scripts are still running, which is the earliest time for user space applications to be in need of data. As we will see in Section 4.2, it turned out that bootup time was heavily increased by starting ReadPref from a bootup script. This was caused by ReadPref adding disk jobs while the bootup processes themselves largely performed disk input–output. I therefore implemented another small kernel module — *HintDrive* — that makes the current number of disk jobs waiting available to ReadPref via an `ioctl` operation. ReadPref now checks HintDrive before prefetching a file and only starts reading data when there are no other disk jobs waiting. If the disk's wait queue is filled, ReadPref sleeps for some time and then checks again. During this time the system gets the possibility to perform the queued operations without being disturbed by ReadPref. This decreases ReadPref's influence on the system, but introduces busy waiting.

Another approach of reading data without disturbing other system operations is to use the `readahead` system call. This system call may be directly used to populate the page cache from a file. Its main advantage is that data is not copied to user space. This makes the readahead system call faster than a default read operation. However, `readahead` does not solve the problem of performing disk input–output in parallel to bootup applications. This will only be done, if the disk driver distinguishes between `read` and `readahead` operations and handles `readahead` calls with a lower priority.

4 Evaluation

Within this chapter I evaluate the effects my implementation had on the problem described in Section 2.1. I first introduce my measuring environment and describe how I measured times. Afterward, I present my measurement results.

4.1 The measuring environment

At first I will introduce the environment in which my measurements were carried out. The computer was a standard workstation computer from the OS department with the technical details given next:

- AMD Duron processor at 1200 MHz
- 256 MB RAM
- CPU cache size: 64 kB
- 10/100 Mbit connection to the local network
- `/home` directory mounted from external server via NFS
- Maxtor 6E030L0 hard disk drive
  - 60 GB disk space
  - one primary and two logical partitions with EXT3 file system
  - one swap partition
  - read transfer rate of 48 MB / second [2]

With the main goal of my architecture's implementation being to speed up application startup times, I chose three applications for reference that are very common within people's working sets. These applications are shown in Table 5.

Name	Function
Mozilla	a web browser
The Gimp	a graphics editing application
Kate	a text editor

Table 5: Applications within my working set

Another widely used application on Linux is the X-server for graphical display. It is not used as a standalone application but in connection with a login and a window manager. Therefore I decided to extend my measurements to the GDM login manager and the KDE window manager to see whether my architecture could also be able to perform prefetching for these applications.

[2]Measured using `hdparm -t`.

As GDM and KDE are started immediately after bootup, I added a prefetch script to my bootup scripts. This script starts the ReadPref prefetch daemon as a background process and then runs a simple Python script sending prefetch commands to this daemon. These commands include all applications needing to be prefetched. Prefetching is carried out for GDM, KDE, Mozilla, Gimp and Kate in that order.

To see how much adding the daemon affected my system I also decided to measure the bootup time for the system. Bootup times were measured by reading out `/proc/uptime` immediately after the bootup scripts had exited.

GDM and KDE startup times were measured by hand with a timer clock, because this was the only way to perform such measurements. I am aware that this incorporates incorrectness caused by my reaction times. The same goes for the applications: I measured their startup times by starting them with the Linux `time` tool and shutting them down as soon as possible [3]. This also leads to incorrectness caused by personal reaction.

Measured times are therefore not completely correct, but are probably a little higher than the real values. Furthermore the measurement incorrectness is not constant allover my measurements, but varies remarkably. To compensate for this, I repeated every measure ten times and then calculated the mean value and the standard deviation.

4.2 Measuring Results

I will now describe and explain the results of my measurements, starting with the boot process as it comes first in my prefetch process.

4.2.1 Booting

As mentioned in Section 4.1, starting the prefetch daemon and prefetching was added to the system's bootup sequence. I therefore expected bootup time to increase. This occurred and bootup times increased by more than 10%. The reason for this was, that ReadPref started read operations all over the boot process. This process is already producing high disk loads, so that the new jobs sometimes heavily disturbed the default operations on bootup.

Fortunately, I could decrease the loss of bootup performance in a simple way. I added the HintDrive module as described in Section 3.8. This module determines the current length of the hard disks' work queues. ReadPref is therefore able to draw conclusions on the current disk load and only issue disk requests if the system is idle otherwise. It then may sleep for some time and check the load again.

With this enhancement, measures as shown in Table 6 were obtained. As can be seen from the values, introducing the HintDrive module lead to bootup times being nearly the same as without my prefetching solution. This was not as I expected, because time needs to be invested for prefetching. The reason for the performance loss being nearly zero is simply that prefetching is not yet finished, when the bootup process is done. Only GDM and parts of KDE are already within cache. My expectations thus had to be realtered — I now expected to lose time during GDM startup, when the rest of the prefetching work had to be done.

[3]with the Ctrl-Q keyboard shortcut

	avg. bootup time	maximum time	minimum time	std. deviation
default bootup	44.68 s	48.16 s	46.60 s	1.48 s
modified bootup	44.85 s	48.16 s	42.43 s	1.60 s
performance loss	0.17 s / 0.40%			

Table 6: Bootup performance

4.2.2 Application startup

So far little or no performance loss was obtained during bootup time. Prefetching is not finished at the end of bootup so that I expected losses for GDM startup. After the login manager is up, prefetching was expected to be finished and I hoped to see startup increases for the KDE window manager and the applications. The maximum possible gain for application startup is to achieve the same startup time as with a warm cache startup, so this is what I expected to see in the best case.

Tables 7-11 show the obtained results for the individual applications as introduced within Section 4.1. Figure 4 visualizes the average startup times of each application for cold cache startup, warm cache startup, and prefetched startup.

The performance gains for prefetching are considerable, ranging from 5% for the X server and GDM startup (Table 7) up to 44% for the Mozilla web browser (Table 9). Observing the absolute time gains, we see that already the 7 seconds saved by prefetching KDE are considerable and will be recognized by the user. Seeing it this way, a prefetch solution using my application-level hints architecture may be considered successful.

However, the tables show that the performance of a warm cache startup is never reached by the prefetch solution. One imaginable reason for this is that my hint generation ignores files where prefetching data is useless because it is generated dynamically, like for instance files in Linux' `/dev` or `/proc` directories. As these files are not touched on prefetching, their inode information is not stored within the dentry cache and needs to be retrieved from the file system at run time. A warm cache startup already finds these information within the dentry cache and is thus faster. This explains the small differences between warm cache and prefetched startup for the tested applications.

Furthermore, applications create temporary files during the first startup after booting and use these files until the system reboots (e.g., files in the `/tmp` directory). These files may not be created and populated at runtime without knowledge of application internals, therefore ReadPref cannot do that. However they exist after the first application startup and may then be cached so that a startup from warm cache results in better startup times. This is the case for the KDE window manager (which creates some files and directories in `/tmp`) and leads to additional performance loss in comparison to warm cache startup time.

Special care has to be put onto the obtained values for GDM startup. As mentioned in Section 4.2.1, I expected its startup time to increase because prefetching has to be performed in parallel to GDM running. This is however not directly visible from the measurements. Instead, GDM's startup time is a little lower than with a cold cache but 4 seconds away from warm cache startup time.

The reason for this is complex: GDM's files are prefetched during bootup. Therefore the real GDM startup time is not much above warm cache startup time. But as prefetching is performed in parallel to GDM, the time we gain by prefetching is already spent for performing other tasks again. Two reasons lead me to this conclusion:

- GDM startup times and bootup times are varying around the average value. However there is a connection between them in my measurements. Whenever bootup takes a longer time, GDM startup is faster. This is, because more prefetching is then performed during bootup and not that much work is being delayed until GDM startup. The other direction is also true: Whenever bootup is fast, prefetching is mainly done during GDM startup which then results in longer GDM startup times.

- I repeated my measurements with DMA turned off for my hard disk. The measurements are not presented in detail, because they are of no use for evaluation purposes as they consider old-style hardware. However, the distribution of disk times is more visible with these measures:

  - Time is lost during bootup. As bootup with a slow non DMA disk takes longer, nearly all of the prefetching work may be done during bootup. This leads to startup time increases of about one second for bootup.
  - With most prefetching being finished after bootup, we immediately see performance increases for GDM. The one second lost during bootup is amortized by a three second performance gain for GDM startup.
  - My measurements with DMA turned off also lead to the conclusion that prefetching is more effective for slow disks

	avg. startup time	maximum time	minimum time	std. deviation
startup with cold cache	12.78 s	15.28 s	10.60 s	1.53 s
startup with warm cache	8.18 s	8.70 s	7.80 s	0.25 s
startup after prefetch	12.15 s	14.01 s	9.10 s	1.58 s
performance gain warm vs. cold cache	4.60 s 36.0%			
performance gain prefetch vs. cold cache	0.63 s 5.0%			

Table 7: Startup performance for X-Server plus GDM

4.2.3 Parallel prefetching

So far we have seen that prefetching using application-level hints is a considerable option of increasing application startup time *if there is enough time for prefetching*. Up to now the scenario demanded that enough time was left between the start of the prefetch operation and the start of the application so that ReadPref could perform prefetching in background.

	avg. startup time	maximum time	minimum time	std. deviation
startup with cold cache	18.90 s	29.40 s	17.30 s	3.51 s
startup with warm cache	9.65 s	9.90 s	9.20 s	0.24 s
startup after prefetch	11.81 s	12.60 s	10.60 s	0.57 s
performance gain warm vs. cold cache	9.25 s 48.90%			
performance gain prefetch vs. cold cache	7.09 s 37.50%			

Table 8: Startup performance for the KDE window manager

	avg. startup time	maximum time	minimum time	std. deviation
startup with cold cache	5.14 s	5.50 s	4.93 s	0.20 s
startup with warm cache	2.35 s	2.67 s	2.16 s	0.13 s
startup after prefetch	2.89 s	3.50 s	2.61 s	0.26 s
performance gain warm vs. cold cache	2.80 s 54.30%			
performance gain prefetch vs. cold cache	2.25 s 43.80%			

Table 9: Startup performance for the Mozilla browser

	avg. startup time	maximum time	minimum time	std. deviation
startup with cold cache	4.10 s	5.17 s	3.78 s	0.50 s
startup with warm cache	2.54 s	2.63 s	2.50 s	0.04 s
startup after prefetch	2.91 s	3.32 s	2.80 s	0.14 s
performance gain warm vs. cold cache	1.66 s 38.00%			
performance gain prefetch vs. cold cache	1.19 s 29.00%			

Table 10: Startup performance for the Kate editor

	avg. startup time	maximum time	minimum time	std. deviation
startup with cold cache	7.86 s	11.59 s	7.22 s	1.26 s
startup with warm cache	6.27 s	6.43 s	5.99 s	0.13 s
startup after prefetch	6.83 s	7.68 s	6.41 s	0.39 s
performance gain warm vs. cold cache	1.59 s 20.02%			
performance gain prefetch vs. cold cache	1.03 s 13.10%			

Table 11: Startup performance for TheGIMP

With this proved, the next step was to check how my solution performs when prefetching is started the same moment the application is started. If this works, prefetching becomes

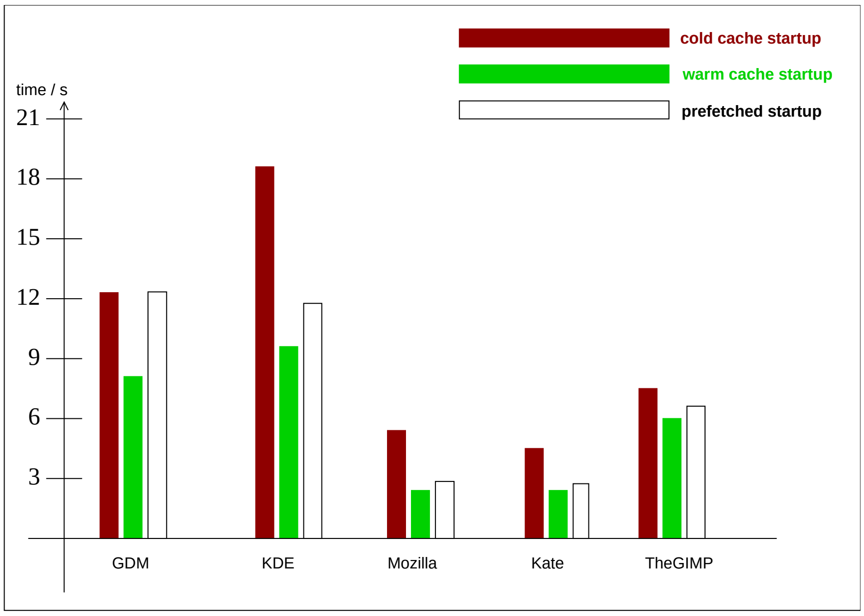

Figure 4: Startup performance compared

applicable for a wider range of applications. As mentioned in Section 1.1.5, there are many applications to which statistical prefetching is no solution because the files needed for them are unknown until application startup. These applications — including `make` and `grep` — may only benefit from prefetching in parallel to startup.

Table 12 shows the obtained measures for my experiment. The results show that to achieve a reasonable performance gain, there needs to be enough time for prefetching before applications startup. Otherwise my prefetching solution is currently not working. Parallel prefetching leads to a speedup only worth a split second. Users will not even realize this speedup.

The reason for the failure of parallel prefetching is the same that the developers of Speculative Hinting (Section 1.2.2) were facing: The prefetching application will fall back behind the original application and then will only prefetch data that has already been read and brought into the cache by the application itself. Speculative Hinting solved this problem by logging the prefetched files and restarting the prefetch thread when it fell back behind the application. I will give ideas about how to cope with this problem within my architecture in Section 5.2.

This experiment lead me to the following conclusion: Even when falling back behind the original application, ReadPref does not heavily slow down the system.

4.3 Cooperation with other tools

Reading data ahead of time to speed up application startup is in no way a brand new idea. Several other tools and projects exist to perform this task. In this section I am going to dis-

	avg. startup time	maximum time	minimum time	std. deviation
startup with cold cache	5.14 s	5.50 s	4.93 s	0.20 s
startup with parallel prefetch	4.89 s	5.01 s	4.77 s	0.09 s
performance gain prefetch vs. cold cache	0.25 s 4.87%			

Table 12: Startup performance for prefetching in parallel to starting the Mozilla browser

cuss, whether my prefetching solution using application-level hints and the ReadPref daemon is able to collaborate with such solutions.

Prelink, developed by Jakup Jelinek [10], is based upon the assumption that most large applications are built with shared libraries to make use of common Unix libraries and to scale down application binaries. This means, that application developers rely on specific libraries being installed on every Unix system and use calls into these libraries from their application.

For the binary this means that there are several function calls and other symbols which are not resolved when building and linking the application. Resolution of these symbols is performed when the program makes use of these symbols. Libraries are mapped into the application's address space at startup — a symbol's address may thus be calculated with knowledge of the region to which the library is mapped. The more symbols an application uses, the more time needs to be spent to calculate this lookup table. This process is called *dynamic relocation*.

Prelink's way of speeding up application startup is to assign a unique memory slot to each library. Thus each library will be mapped into the same memory slot for each application. With this ensured, calculation of symbol addresses may be accomplished at once by running the prelink tool on every shared library and every application. Dynamic relocation is then just a lookup in a table belonging to a library - no more calculations need to be performed.

Jelinek [10] improved application startup by up to 32% using Prelink. These improvements could unfortunately not be reproduced by my own measurements. I think the reason for this is that improvements have been made to the dynamic linker, so that it optimizes symbol lookup itself and there is not that much space for performance gains by prelinking than it was two years ago.

As Prelink's work is orthogonal to the way my prefetching tool works, it seemed interesting to test how these two tools performed in combination. Table 13 shows startup times for Mozilla on my test computer after prelinking all libraries and the Mozilla binary.

As can be seen from the measurements, prelinking alone results in nearly no performance gain for Mozilla (1.5%). However prefetching data for the prelinked Mozilla decreases startup time by 51.8% - this is 7% more than without prelinking. To come to a clear assumption on the collaborative performance of Prelink and ReadPref, I carried out measurements for KDE as given in Table 14. KDE's startup time decreases by 42.2%. This is 4.7% increase compared to prefetching without Prelink. With these measures I come to the conclusion that ReadPref may be combined with Prelink, resulting in about another 5% decrease in application startup time.

	avg. startup time	maximum time	minimum time	std. deviation
un-prelinked, cold cache	5.14 s	5.50 s	4.93 s	0.20 s
prelinked, cold cache	5.06 s	5.26 s	4.90 s	0.14 s
prelinked and prefetched	2.56 s	2.60 s	2.50 s	0.05 s
performance gain by prelinking	0.08 s 1.5%			
performance gain by prelinking and prefetching	2.58 s 51.8%			

Table 13: Startup performance for Mozilla using Prelink and ReadPref

	avg. startup time	maximum time	minimum time	std. deviation
un-prelinked, cold cache	18.90 s	29.40 s	17.30 s	3.51 s
prelinked and prefetched	10.93 s	12.80 s	10.10 s	0.72 s
performance gain by prelinking and prefetching	7.97 s 42.20%			

Table 14: Startup performance for KDE using Prelink and ReadPref

Linux features the `readahead` system call since kernel version 2.4.19. This system call populates the page cache by reading data from a storage medium. In contrast to the standard `read` system call this data is however not copied to userland. Prefetched data is not required by the userland prefetching application — therefore prefetching with `readahead` is faster than a simple `read` while not losing quality.

A tool with the name `readahead` is available to perform operations similar to my prefetching daemon ReadPref: the readahead tool is provided a list of files to read and populates the page cache from these files at bootup. In comparison to my solution, where prefetched files are determined by evaluation of statistics on applications, the list of files for the readahead tool is simply generated by its developers' experience and may be adapted by users.

My evaluation of `readahaed` is based on information found on the internet ([23]). The authors of this site adjusted a Knoppix Linux to use readahead during bootup. Their measurements show that readahead may speed up computer bootup by about 30%. They further found out that readahead works better, the more memory is provided to the system. Systems with 256 MB RAM or less did not even perform better with readahead but went worse instead.

The authors of [23] give no clues why this performance losses occur. My interpretation is that the operating system and applications they use need an amount of the 256 MB of memory for their own purposes. Reading ahead data which is not needed at the moment fills up the page cache. This leads to cache pollution — necessary data is thrown out of the cache and needs to be read again from disk. Startup times therefore increase.

The experiences of ReadaheadKnoppix lead me to the following conclusions:

1. As the readahead tool does nearly the same as ReadPref, I do not expect considerable decreases for application startup times when both tools are combined. ReadPref is more fine grained than the readahead tool because it takes care of specific knowledge about users' application working sets.

2. The `readahead` system call may be interesting for further improvements of ReadPref. Prefetching with this system call will probably decrease the times needed for prefetching data, especially CPU time. When prefetching becomes reasonably faster than performing real read operations, the chance that ReadPref is outrun by an application during parallel prefetch becomes smaller. Therefore it may be a good idea to use this system call when trying to make ReadPref perform better in those situations.

5 Conclusions and Outlook

5.1 Conclusion

Aim of my thesis was to design an architecture for automatic generation and application of application-level hints. As possible scenarios spread over a wide area, this architecture needed to be as general as possible.

During my thesis I developed the following architecture and its components:

- Sensors are introduced into the Linux kernel with a small kernel patch.
- A runtime monitoring system has been implemented. Small changes to the kernel were needed to implement this. However, most of the work was put into a loadable kernel module. The monitor's influence on the system may therefore be kept small by the user, because the module may only be loaded into the kernel if it is needed.
- An automatic hint generator is used to evaluate data obtained by the runtime monitor and to generate application-level hints.
- A hinting application uses the automatically generated application-level hints to improve overall system performance with respect to the specific scenario involved.

As a proof of concept I implemented my architecture for the specific scenario of speeding up startup of large applications in Linux. Measurements lead to the conclusion that prefetching using automatically generated hints leads to reasonable performance increases, given there is enough time to perform prefetching in background before application startup. With low overhead for prefetching, application startup times could be increased by up to 44%. Furthermore my prefetching architecture may be combined with other tools like for instance Prelink [10]. Such combinations increase the benefit of prefetching and lead to decreased startup times of up to 51%.

5.2 Future work

During my work several ideas to improve my architecture and my implementation of hinting have come to me by myself and by hints of others. I would like to outline these ideas now.

1. *Prefetching kernel startup:* From the very beginning of my work on prefetching there has always been the idea of applying the solution to speedup Linux kernel bootup. This is however not possible with the current solution as my prefetch daemon is launched quite late during the boot process. An idea to improve this situation is to strip down the kernel to a bare minimum and compile all the necessary components as kernel modules. One could then try to boot the kernel into an initial ram disk and then start to prefetch the modules from hard disk while in parallel starting to load them.

   Figure 5 shows CPU and disk utilization during bootup on my test computer. This figure was created with the BootChart tool, available from [1]. As can be seen in the second row, there is a large gap in disk throughput before there is much work to be done in the end. If the data read in the end is known from the beginning, it may be read ahead of time and booting will become faster.

2. *Prefetching parallel to application startup:* My current prefetch implementation does not perform well when started in parallel to the prefetched application. Means to retrieve data from the kernel at runtime have been established by my kernel patch and module. One can therefore try to implement synchronization between the prefetch daemon and the rest of the system based upon the HintMod interface, thus improving performance when prefetching in parallel.

   Two possible ideas come into my mind regarding this problem:

   a) First of all it is worth a try to do some reordering of the data within the prefetch file. As applications start to read data from the beginning, it should be evaluated if there are performance gains for parallel prefetch, when the ReadPref daemon takes these files into account in reverse or random order.

   b) Another idea is to implement a low-traffic, fast readable sensor within the operating system kernel to perform synchronization between ReadPref and the prefetched application. This could happen in a way similar to the logging approach used by Speculative Hinting and described in Section 1.2.2.

3. *Automation of hint generation:* Application-level hints are generated and applied automatically. However, my software's users will have to manually start the process of hint generation. I think it is possible to automate this process by determining a user's application working set and then automatically performing the necessary steps of hint generation in background. The main problem is to handle program and library updates, which potentially render old application-level hints useless. One will therefore have to monitor the working set and determine when such conditions occur, so that hint generation may be restarted. A precondition for this is a useful implementation of synchronization between the prefetched applications and the prefetch daemon, because this will enable us to perform runtime monitoring of a user's working set.

4. *Investigation of applicable domains:* In my thesis I implemented the application-level hints architecture for the specific scenario of speeding up application startup. As the architecture's goal was to be as general and reusable as possible, it will be the task of future work to determine other domains where this architecture may be applied. I currently think there are applications within the networking area. I can also imagine that it could help to apply hints for scheduling so that a system might prioritize applications from a user's working set over others.

5. *Manual hinting:* Main target of my thesis have been application-level hints generated by monitoring applications automatically. I expect that application-level hints issued to the system directly by applications give programmers the opportunity to issue information on future resource usage to the underlying system. If the system does have enough resources, it may then decide to make use of these hints, so that applications perform better. Otherwise the system may decide to drop hints and thus perform as if no hints were given.

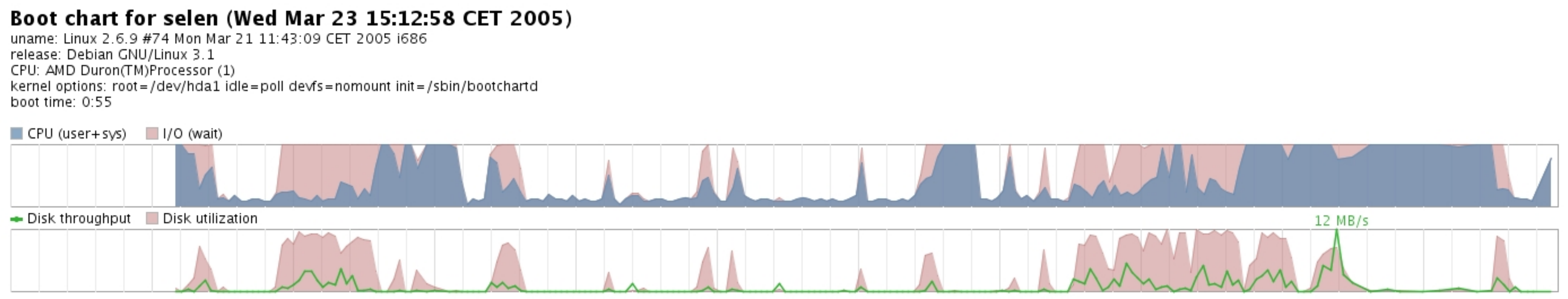
Boot chart for selen (Wed Mar 23 15:12:58 CET 2005)
uname: Linux 2.6.9 #74 Mon Mar 21 11:43:09 CET 2005 i686
release: Debian GNU/Linux 3.1
CPU: AMD Duron(TM)Processor (1)
kernel options: root=/dev/hda1 idle=poll devfs=nomount init=/sbin/bootchartd
boot time: 0:55

Figure 5: Bootchart for my test computer

Further investigation into the area of application generated hints needs to find out types of resources and resource accesses and to define which kinds of prefetching or pre-caching can be performed for these types. However, it should be avoided to implement solutions to problems that already may be solved using application generated hints like for instance asynchronous input–output.

6. *Considering cache replacement:* Cao and associates showed in [3, 4] that there is a strong relation between prefetching and the cache replacement policy (see Section 1.2.3). During my measurements I did not run into problems caused by suboptimal cache replacement. The reason for this is the type of my scenario: prefetching is performed during bootup, when the computer's physical memory is nearly empty and the page cache may therefore occupy large parts of it. Because modern computers tend to have lots of memory, the page cache will normally not become filled during bootup, even if prefetching operations are added.

   The caching policy may however not be ignored as there will be scenarios where prefetching will require cache replacement decisions to be made. Possible scenarios are:

   - Embedded systems, which typically do not possess as much physical memory as desktop computers do,
   - Desktop systems on which several memory-consuming applications are already running, so that only a small part of memory is still available for the page cache, and
   - Server systems running applications for many users where each user's application working set will potentially differ from the other users' working sets.

   Future work will therefore have to test, how the application-level hint architecture described in this thesis behaves for prefetching that is performed with a limited page cache. It needs to be tested, if the caching policy might be adapted so that it recognizes prefetch requests issued using the `readahead()` system call. The cache manager may then decide to ignore such requests under high load and simply return an error message to the prefetch tool, signaling that a requests could currently not be served. The prefetch application — similar to the HintDrive approach for hard disks — then needs to decide whether it wants to issue the request again at a later moment or simply drop it.

   Another way to handle problems with caching would be to monitor the amount of memory currently available to the page cache. As the prefetch application knows the amount of data to be prefetched ahead of issuing the requests, it may compute how many requests will fit into the cache before it becomes filled up. This approach appears to be less intrusive than adapting the cache manager.

7. *Focus on per-user working sets:* My current prefetching implementation collects and applies application-level hints globally. Future enhancements to the prefetching tool chain should enable to determine file access patterns on a per-user basis.

   Therefore one will have to collect hint data for one application from several users. The hint generator may then be extended to merge all these information, creating two types of hint files:

   a) A system-wide hint file containing data on the files that are accessed for every user. It will include for instance application binaries and libraries.
   b) For every user the hint generator will create a user hint file. This will include data specific to the user, for instance user-local configuration files.

   With these two types of hint files, hinting may be better distributed over time. During bootup we will only need to prefetch data described in the system-wide hint file. Prefetching user-local data may be delayed until the specific user logs in.

   It is possible that this approach might lead to the prefetch operations having less effect on the bootup process, because less data needs to be prefetched during system startup. However, this will only happen if separating user-local data from global data results in a considerably smaller amount of data to be prefetched during bootup.

References

[1] The bootchart project. http://www.bootchart.org.

[2] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. Second edition, 2002.

[3] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Measurement and Modeling of Computer Systems*, pages 188–197, 1995.

[4] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[5] F. Chang. Using speculative execution to automatically hide i/o latency, 2001.

[6] Fay W. Chang and Garth A. Gibson. Automatic i/o hint generation through speculative execution. In *Operating Systems Design and Implementation*, pages 1–14, 1999.

[7] Christof Fetzer. Software Fault Tolerance - Vorlesungsskript, TU Dresden, 2005.

[8] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[9] Sitaram Iyer and Peter Druschel. The effect of deceptive idleness on disk schedulers. Technical Report CSTR01-379, Rice University, June 2001.

[10] Jakub Jelinek. Prelink. ftp://people.redhat.com/jakub/prelink/prelink.pdf, December 2003.

[11] Klaus Kabitzsch. Test und Diagnose von Anwendungssystemen - Vorlesungsskript, TU Dresden, 2000.

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

[13] Linux cross reference. http://lxr.linux.no.

[14] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems*, pages 222–233, 1996.

[15] Wolfgang Maurer. *Linux Kernelarchitektur*. Hanser Fachbuchverlag, 2003.

[16] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 62–73, New York, NY, 1992. ACM Press.

[17] Sean O'Rourke. Improving I/O parallelism through hints and history - a class project. www.cse.ucsd.edu/classes/fa01/cse221/projects/group8.ps, November 2001.

[18] Patching a running Linux kernel. http://uranus.it.swin.edu.au/~jn/linux/kernel.htm, 2003.

[19] R. Hugo Patterson, Garth. A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles, Copper Mountain, CO, December 1995*, December 1995.

[20] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, 27(2):21–34, 1993.

[21] Martin Pohlack. Komplexpraktikumsbericht. October 2001.

[22] Porting linux device drivers to kernel 2.6. http://lwn.net/Articles/driver-porting/.

[23] Readahead performance on knoppix with linux 2.6. http://unit.aist.go.jp/itri/knoppix/readahead/index-en.html.

[24] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. Second edition, 2001.