Performance Analysis of the CannyFS Shim Filesystem

Per Alonso
Abstract

Performance Analysis of the CannyFS Shim Filesystem

Per Alonso

In High Performance Computing, there is a constant effort to improve performance. One important part of performance in these systems is storage performance. This thesis investigates the performance of a shim filesystem called CannyFS that exploits ignoring certain consistency requirements of typical filesystems for gaining performance benefits. This thesis uses profiling tools to find a potential bottleneck in CannyFS. This potential bottleneck then has four candidate optimizations applied to it, and each is tested separately to determine whether it is a good optimization or not. Finally, the candidate optimizations are compared against each other in a limited set of benchmarks to determine which ones result in the best performance.

This thesis shows that the function get_filedata_inner in CannyFS is a bottleneck that is best optimized by either switching out an std::set it uses to an std::unordered_set or by doing so as well as removing redundant work. These optimizations are shown to reduce end-to-end program runtime by more than 8% in certain benchmark scenarios. This work also shows that seemingly obvious optimizations may not be good optimizations, and that optimizations on sequential code may still have a significant impact in concurrent scenarios.
1 Introduction

This report attempts to analyze the performance of the CannyFS shim filesystem and find bottlenecks in CannyFS itself. This has been done by making use of certain profiling tools, as well as investigating the usefulness of the data they produced. This report takes a Linux-focused approach to analyzing CannyFS and its various surrounding components.

CannyFS is a filesystem in userspace using Linux’s FUSE that exploits the transaction-like behaviour of batch processing for better performance. The transactional nature means that if something fails, the whole program is re-tried or aborted as recovering from errors may be meaningless. This means that if a program e.g. writes to a file it does not have permission to write to - a regular filesystem (FS) would have to notice that and report it upon returning to the caller so the caller can recover from the error. CannyFS however notes that a batch program (in an HPC environment) likely won’t cause any errors, and that any errors will likely cause the whole program to have to be restarted anyway. CannyFS uses this to optimistically return to the caller before performing the write as if the write had completed successfully. 

Studying the results presented in the paper describing CannyFS shows that the performance benefits of CannyFS are less on directory tree removal than on zip-archive extraction [11]. This may indicate certain scenarios causing performance issues, and is therefore worth investigating.

As performance issues are created by bottlenecks in the program, it is necessary to understand what causes bottlenecks for one who wishes to analyze performance. J. Weidendorfer claims that there are three main causes for bottlenecks in sequential code [20].

- Redundant calls, such as multiple initialization, or always calling a function when this is only needed for certain inputs.
- An unfitting algorithm with worse than necessary performance.
- Poor usage of the platform, e.g. incurring additional cache misses, branch mispredictions, page table walks etc.

This report will explore how these three points can affect performance, even in concurrent scenarios due to e.g. sequential parts of the program and locked critical sections enforcing serialization.

This report contributes four candidate optimizations for CannyFS. The first optimization presented explores the effects of changing out a data structure (and therefore also the algorithms used on it) and is presented in section 6. Following this, two optimizations that explore removing redundant function calls as well as poor platform usage as a result of an attempted optimization are explored in section 7. Finally, a combination of two of the previous optimizations is presented in section 8. In section 9, these optimizations are compared against each other to find the best optimization of the ones presented. Table 1 presents the names of the optimizations and the sections they are presented in.

<table>
<thead>
<tr>
<th>Section</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Unordered</td>
</tr>
<tr>
<td>7</td>
<td>No-redundant</td>
</tr>
<tr>
<td>7</td>
<td>One-redundant</td>
</tr>
<tr>
<td>8</td>
<td>Unordered-1-redundant</td>
</tr>
</tbody>
</table>

Table 1: Table relating candidate optimization names and the sections they are presented in.

1.1 Related and prior work

CannyFS is a filesystem based on FUSE (Filesystem in USErspace) designed to be layered on top of any underlying filesystem provided by the administrators of the environment being used. FUSE is a software framework that allows for the implementation of filesystems in userspace rather than in the kernel [18]. This allows a user of an HPC system to get the benefits of CannyFS, such as optimistic return from certain requests or faked results from e.g. stat without requiring administrator-intervention for changing filesystems. CannyFS exploits the fact that many HPC jobs behave like transactions in a relational database management system, that is if one operation fails, the entire job ought to be rolled back and retried. This makes e.g. reordering of operations,
caching of operations etc. possible as long as the final contents of all files is the same as if a regular filesystem had been used. Another assumption must be made to exploit this fact, and that is that any IO operation can complete successfully as CannyFS otherwise would have to check for e.g. exceeding disk quotas, file permissions etc. for which the full operations may have to be completed to even notice any errors and thus removing benefits of deferring IO. Using FUSE allows this to be flexibly applied only when/where necessary and fitting.

Using all these assumptions, one can (and CannyFS does) buffer IO operations and defer them. Operations that return some easy to predict data - such as stat - or none at all can use this a priori knowledge to return to the requester of IO without having completed the request. An example of this would be a write. CannyFS can buffer the write for deferred completion and return an indication of success allowing the writer to continue as if the write had completed. Do however note that certain operations, such as reads, have to complete synchronously and even force any previously buffered writes to such a file to be synced to the underlying filesystem [11]. So while one of the main functionalities of CannyFS is caching writes, it also predicts and fakes return values. It may also e.g. defer closing files.

This is specifically implemented by CannyFS via having one work queue for each file, thus serializing the requests on each file. Each work queue is then handled asynchronously by a thread per file [11].

2 Background

FUSE is based on an idea of implementing the filesystem in user-space primarily borrowed from the concept of microkernels (in contrast to monolithic kernels like Linux). The microkernel model suggests that the OS kernel is to be based on several smaller components, some of which reside in user-space while the kernel provides a limited set of services. This was created with the idea of making development easier. Early microkernels had the issue of introducing too much overhead, and likewise early user-space filesystems were limited by overhead [16]. However, more modern FUSE-based filesystems can do anything between degrading performance by up to 83% and not being noticable depending on the workload (these numbers are based on passing through the filesystem operations directly from user-space to the kernel) [18]. Thus it may still be worth optimizing a filesystem based on FUSE as it will not necessarily be the case that the overhead from FUSE dominates.

In Linux, FUSE consists of two parts. One user-space daemon, and one kernel module. The kernel module creates a block device (represented in Linux as a file) which is used to exchange messages between the user-space daemon and the FUSE driver [18]. The FUSE kernel module is also responsible for registering the FUSE driver with Linux’s Virtual File System layer (VFS), which is the kernel’s standard way of handling any filesystem implementation (whether in the kernel or in user-space). In the case of a FUSE filesystem, the VFS takes the operations requested by applications using storage and forwards these to the FUSE driver, which in turn communicates with the user-space daemon, that then responds to the request [16].

FUSE provides a number of benefits over implementing a filesystem in the kernel. Development gets significantly easier in user-space since a kernel developer has to recompile the whole kernel and may suffer complete system lock-ups if they make a mistake. While there are indeed build-systems that make recompiling the kernel faster by only recompiling what is needed, recompiling still likely takes longer for a kernel-based filesystem than for one in userspace. Also consider that the developer can be spared many reboots by only working in user-space. This also allows fast and easy investigation of new algorithms and datastructures in filesystems allowing the general performance trend of filesystems to increase faster. Especially if one takes the findings from investigations on FUSE-based filesystems and applies it to kernel filesystems. Also, since the filesystem is not a part of the kernel - an issue causing a crash does not crash the entire system, rather a single program which is more easily recovered from [16].
3 Method

3.1 Preparation for experiments

Before running any experiments, it was necessary to prepare CannyFS as compiling it on GCC 11.1.0 presented some problems. At the time, documentation suggested to compile it with `-std=c++14`. However, the program used `string_view` which is absent from the C++14 standard [14]. This appeared to be a documentation mistake, however it was decided to bring the code to C++ 20 standard compliance. Doing so required making the filemap’s (the set data discussed later in the report) comparison object’s comparison function `const`.

Another problem encountered was crashes when compiling CannyFS with compiler optimizations. Using GCC’s `-fsanitize=undefined` option when compiling, and running CannyFS, it was shown that the crashes were due to a missing return statement in `get_filedata_inner` causing undefined behaviour that by chance worked without certain optimizations.

Undefined behaviour is a common cause of issues in software, and compiler optimizations can in many cases cause the effects of the undefined behaviour to be magnified by e.g. the assumption that programmers would not invoke undefined behaviour. Newer compiler versions may treat a certain type of undefined behaviour differently than previous compiler versions due to e.g. more aggressive optimizations [19]. Thus one must be cautious when using newer compilers than previously tested on for the purpose of better performance.

3.2 The experiments

To evaluate different versions of the programs against each other, six realistic experiment types have been used. The first three are unzipping the Linux 5.16 source tree with local mechanical disk mounted as ext4 as the underlying storage system, as well as storage over the network mounted as nfs, and also a shared memory virtual filesystem mounted as tmpfs. Although a tmpfs filesystem technically does not constitute a ramdisk as it does not use dedicated parts of physical memory, rather makes use of the OS virtual memory functionality [13] - the tmpfs filesystem will sometimes be referred to as a ramdisk for the sake of readability. The three remaining experiment types are removing the resulting directory tree from the unzipping operation on all three types of storage. The experiments will be named `unzip-ramdisk unzip-local`, `unzip-nfs`, `rm-ramdisk`, `rm-local`, and `rm-nfs`.

Consider each of `-nfs`, `-local`, and `ramdisk` as representing a class of overhead from the underlying storage. Although, one should note that this is not completely representative as filesystems can have significantly different characteristics, even spanning just mount options. The nfs filesystem was used as it provides a scenario with a network mounted filesystem. A Lustre filesystem was also available in the environment, but was not used due to intermittent performance issues with it at the time of running the experiments. The nfs filesystem shows the performance in one type of realistic HPC scenario as it provides the latency and possibly other characteristics of large, high throughput parallel filesystems such as those seen in e.g. the V4HIO IO-500 SC21 list [7]. The local ext4-mounted hard disk storage provides insight into the IO usage pattern where - for the duration of the job, any IO is done on local disk, and after job completion being moved to other (external) storage. Note that since this report only explores effects during the runtime of the job, any data movement off the local disk is not considered. The tmpfs storage is not representative of any real workload. Rather it shows the most pure overhead of CannyFS as overhead from underlying storage is minimized by using main memory. Note that swapping may occur (which causes overhead) as the OS virtual memory facilities are used, however given large enough main memory (as seen in section 3.4, this is 128 GB in this case) this ought to cause no issues.

These experiments are referred to as the real-world benchmarks or experiments. Each experiment was run 10 times to reduce the impact of anomalous results. This was done by running each program type (CannyFS optimized in different ways) in sequence, and repeating this sequence 10 times. All of this was repeated one time for each underlying filesystem being tested on (nfs, ext4, tmpfs). Additionally, a set of experiments meant to compare the different opti-
mizations presented was run on the tmpfs filesystem. This used a sample size of 80 runs per optimization type. The sample sizes were chosen based on targeting a 5 hour runtime of a complete set of experiments (all of the small sample size experiments or all of the large sample size experiments). The 5 hour target was chosen for practicality reasons. The sample size of 10 provides a base set of data that can be analyzed although conclusions need to be based on both data analysis and understanding of any performance effects. The sample size of 80 provides opportunity for drawing conclusions with less understanding of performance effects as the data has a lower probability of deviating significantly from the real data distribution.

Running in the round robin fashion described minimized the possibility for temporarily poor performance due to e.g. network congestion to introduce systematic bias in one particular optimization type. The system reference manual page for “Sync (GNU coreutils) 8.22” showed that this command writes cached writes to the underlying storage. Note that for each individual run of some benchmark, the command sync was used to force full completion of all writes on underlying storage before any experiment was considered done. The aim of using sync was to limit experiment timing inaccuracies and variability coming from writes not being performed as expected. Note that this makes no mention of entry and inode caches (caches in Linux that cache the internal objects representing files and directories). This could cause certain performance differences [8, ch.13]. This however could not be considered due to limitations in permissions for using the /proc/sys/vm/drop_caches interface. Note that CannyFS itself syncs to the underlying filesystem upon termination, however a sync does not necessarily happen for the underlying filesystem.

Setting up an experiment consisted of any required files being copied to the determined storage type (the nfs, ext4, or tmpfs filesystem) and the mountpoint being created in a designated location. CannyFS would then be mounted with the subdir option being set to the location in which any necessary files for the benchmark were located. Note that to avoid other users affecting the experiments as much as possible, a whole physical machine was allocated for the experiments. Thus, no other users could use the same machine.

For each of the experiment types, CannyFS was layered directly over the underlying file system. The Linux 5.16 source tree zip archive had a size of 243 MiB, and an uncompressed size of 1.3 GB spanning over a total of 74265 regular files and 4874 directories.

The benchmarks chosen for evaluating the performance were chosen to be as similar to the benchmarks from the original CannyFS paper as possible [11]. For these experiments, the unzip and rm programs provided by the OS (CentOS 7.9) were used. For all benchmarks and profiling, the following invocation of CannyFS - modeled after cannywrapper.sh provided with CannyFS - was used.

```
./cannyfs -f -o kernel_cache -o big_writes -o max_write=65536 -o attr_timeout=86400 -o negative_timeout=86400 -o entry_timeout=86400 -omodules=subdir,subdir=$SUBDIR $MOUNTPOINT
```

The -f option causes CannyFS to stay in the foreground (not daemonizing). The kernel_cache option disables flushing the file content cache on open(). Note that this requires the assumption that no modifications happen to the file outside of the CannyFS filesystem. As this is not expected to happen, this can be done to not cause additional slowdowns. The big_writes option allows multi-page write requests, and the max_write=65536 option sets the maximum write request size to 64 KiB. These options are used to not cause issues with any benchmark that for any reason might perform large writes. The options attr_timeout, negative_timeout, and entry_timeout set the timeouts for caching file and directory attributes, lookups of nonexistent files, and lookups of paths in general to 24 hours so the caches will not be invalidated during an experiment which could cause e.g. amplification and masking of slowdowns if a cache entry were to be accessed just before invalidation for one run, and just after invalidation for another run. The -omodules=subdir,subdir=$$SUBDIR $MOUNTPOINT option loads the subdir module into the FUSE setup and sets it up appropriately for the experiment. The
subdir module prepends the path given here as $SUBDIR$ to every path used through CannyFS. This essentially places the mount inside the path given by $SUBDIR$. As the last argument, the path to the mountpoint is specified.

Also note that for measuring the execution time of CannyFS and the benchmarks being run on it, the method used to time was the following. First, the command sync was used to ensure a cache state that is as similar as possible for all optimization types. After this, CannyFS would be immediately started along with the timing. Once the mountpoint would be ready, the benchmark would be started. Once it finished a kill-signal would be sent to CannyFS, and wait would be used to await CannyFS syncing to the underlying filesystem and terminating. After that, sync would be called once again. Once sync would return, the timing would stop. Note that the “real” timing value from time was used.

Using data from 200 runs of the real-world benchmarks, it was found that the median of the time taken to start CannyFS, mount the FS, unmount the FS, and terminate CannyFS was 0.135 seconds, the 75th percentile was 0.158 seconds, and the 90th percentile was 0.184 seconds. Even the 90th percentile is not a significant enough fraction of any experiment’s runtime to make the results difficult to interpret. Furthermore, ensuring low overhead is not as important as ensuring consistent overhead (although sometimes low overhead means consistent overhead, and of course the overhead cannot be completely dominant) as consistent overhead still allows seeing the same absolute performance difference as no overhead would, although the percentual difference is slightly smaller.

3.3 Synthetic benchmarks

To illustrate optimizations and gain further understanding regarding the differences between the various program versions, two synthetic benchmarks were created, these will be called unique-open and seq-multithread.

unique-open uses 16 threads (as that is the amount of hardware threads available per socket on the machines used to run the benchmarks) to open a set of files using random file names consisting of numbers between 1 and 400 000 for a total amount of calls to open a file at 400 000 times. The reason for using random file names rather than a predefined sequence is so that no artificial advantage can be gained by an optimized program version that for example uses a data structure that favours insertion in a particular order. Certain files may be opened multiple times, however this is not an issue as it is the general behaviour of using many different files that is interesting rather than the exact behaviour when all calls to open are unique (so the benchmark name may be a bit misleading). Note that the timing measurements using unique-open also factor in the time to remove all files created, and also creating a directory in which to place the files. unique-open aims to demonstrate the overhead of CannyFS when handling a large amount of files. Thus it attempts to avoid as much time as possible e.g. doing large file operations, rather attempting to maximize overhead from CannyFS in relation to the useful work in file operations. This can be thought of as an exaggerated workload where many files are used.

seq-multithread works by creating a file and filling it with 4 MiB of garbage data, and after that starting up a team of 16 threads (as that is the amount of hardware threads available per socket on the machines used to run the benchmarks). 8 of those threads are readers and the remaining 8 are writers. The reader threads read the entire file sequentially 16 times into 4 KiB thread-private buffers. Concurrently, the writer threads write 4 MiB to the file from the beginning of the file. They do this 16 times. This totals to 128 full file reads and 129 full file writes (one write to initialize the file) or a total of 512 MiB read and 516 MiB written. Note that the timing measurements using seq-multithread also include the time taken to remove the file created by seq-multithread. seq-multithread aims to show what overhead is introduced by CannyFS when the same file is being read from and written to concurrently. By only using one file, overhead from handling multiple files is avoided. Consider this to be an exaggeration of any workload where very few files are used. It is notable that the mixed concurrent reads and writes may cause additional overhead from Can-
nyFS since it forces CannyFS to flush the IO queue when reading a file that has queued writes. However, the inherent serialization of this is not an issue as this benchmark aims to force as serial a scenario as possible from CannyFS. This overhead is further discussed in section 10.5.

None of these synthetic benchmarks were run on the nfs-mounted filesystem as the usefulness of doing so is questionable since high overhead does not aid in (rather, it hinders) providing the insights into CPU-time spent in CannyFS. This is as the high overhead may mask any advantages or disadvantages of a program, however note that it is still useful in the real-world experiments as it provides a realistic scenario. The synthetic benchmarks were run on the local ext4-mounted disk and the shared memory tmpfs filesystem. For all synthetic benchmarks, before timing an instance of the benchmark - sync was run. sync was also run after the benchmark completed. The second invocation of sync was included in the timing. Timing measurement from the synthetic benchmarks are presented in seconds unless otherwise mentioned. The reason for using these specific benchmarks is that they can aid in predicting performance for workloads that either use very few files or use a large amount of files by only taking the file operations part of any such hypothetical workloads and making the characteristics of the file operations be more clear in a result (by taking more time, creating more lock contention etc).

The synthetic benchmark timing runs were run with the different programs (different optimization types) in a round robin fashion. This was repeated 10 times, and such groups of 10 runs were run on each type of underlying storage in order. For the benchmarks comparing the different optimizations, the process was the same although the amount of runs was 80 rather than 10.

### 3.4 Hardware & Platform

The experiments were (unless otherwise mentioned) run on HP ProLiant SL230 Gen8 machines with dual Xeon E5-2660 CPUs, 128 GB of memory, gigabit Ethernet, and InfiniBand. Each machine had local disks (HDD) mounted using Ext4. This was used for the -local versions of the experiments. There was also a networked volume mounted via NFS. This was used to run the -nfs experiments. This NFS storage was provided by a NetApp-based solution using SAS-disks. Each machine was running CentOS version 7.9 with Linux 3.10.0. These machines were a part of the Snowy cluster at UPPMAX (Uppsala Multidisciplinary Center for Advanced Computational Science) and represent a typical HPC environment. When running any experiments, one whole physical machine as described in this section was allocated, so the only job running on the machine would be the experiment.

Note that some profiling work was done on a laptop with an Intel Core i5-7200U with 8 GB of RAM, running Arch Linux with the 5.16.0 Linux kernel and GCC 11.1.0. This was done due to difficulties with profiling on the same machines as those used for running benchmarks. This should not constitute a large problem as profiling was never used to determine the performance level of a candidate optimization. Rather it was only used to inform decisions on what to optimize or finding reasons as to why performance may not have been as expected (or as expected for that matter).

### 3.5 Choice of tools

Several tools were used to attempt to find performance bottlenecks, however the data from each tool was not necessarily useful in finding an optimization. One such example is the causal profiling tool coz. What causal profiling attempts to do is to give data such as: “speeding line $x$ up by $y\%$ would result in a total $z\%$ performance improvement”. It does so by running various performance experiments where a piece of code is virtually sped up by - at runtime - slowing down other pieces running code by way of inserting pauses. It makes use of so called progress points placed by the programmer to measure overall performance as the reality of software is that end-to-end runtime may not always be the most important metric [2].

Coz was not used as it did not provide useful data on the size of experiments used, also not being a mature enough tool or technique caused it to have lacking documentation. The data that Coz
did present after running a prohibitively long experiment was that speeding up the comparison of strings (\texttt{basic\_string}) by any amount would cause the program to have a slowdown of 100%.

Gperftools was also tried. It aims to give data on (among other things not used in this project) which parts of the code cause the most heap allocations. In this particular project, the heap profiler of gperftools (through tcmalloc) was used. Pprof was used to generate a graph representing the collected data [6]. While the data appeared easy to read on the graph and clear in building an intuition for the program’s heap behaviour, exploring the data to find candidates for optimization was difficult. The documentation was satisfactory, but there seemed to be little community use making examples of how the gperftools heap profiler had been used previously difficult to find. An example of the output from pprof run on results from the gperftools heap profiler can be seen in Figure 1.

One flexible tool that was used for primarily exploring platform inefficiencies such as poor cache usage or poor usage of OS facilities as well generating as timing profiles with callstacks was Linux’s perf (or perf\_events). This was used for several different types of analysis, among others - cache misses, branch misprediction, and stack sampling. Tracking heap allocations was also tried using perf, however this was abandoned due to difficulties with correctly being able to track malloc calls for system setup reasons. Tracking cache misses and branch mispredictions gave the data necessary to properly analyze the program, however due to time constraints this was not further acted upon. Stack sampling was the most used feature of perf, as Brendan Gregg had created a data visualization tool that generates so called flamegraphs that (in this case) were used to display CPU-time usage per function and the reason for that function running (its call stack) [5].

These flamegraphs in combination with perf’s stack sampling provided clearly readable data that allowed building an intuition for what causes overhead in the program. The visualization provided good facilities for exploring the data by clicking various functions to zoom in on them, and also showing the results in absolute terms in a status message.

Figure 1: Partial example of a graph generated by pprof for a heap profile generated by gperftools. What can be seen here is that \texttt{get\_filedata} is responsible for 0.0 MB of allocations, however itself and its callees are responsible for 6.6 MB of allocations. This can be seen in the \texttt{get\_filedata} node’s last two lines. Of these allocations, 4.8 MB are from \texttt{get\_filedata}’s callee \texttt{\_M\_construct}, while 1.6 MB are from another callee not seen in this part of the graph. \texttt{\_M\_construct} allocates 4.9 MB in total.
Overall, the flamegraphs provided a method of qualitative analysis of the program’s performance and reasons why certain overheads were incurred. However, this was not used as a final method for discovering optimization opportunities; rather, it was to gain an intuitive understanding for the program’s various overheads as it was also able to show kernel operations. This aided in knowing what to analyze using other tools. An example of such a flamegraph can be seen in Figure 2.

The Intel VTune profiler was used for understanding the low-level details of the code (e.g. TLB usage, good use of speculative execution etc.) as well as hotspots in the code. It provided an easy to navigate interface with easily understandable data, often colour-coded to clearly indicate what the tool determined to be potential issues. Each datapoint also showed built in explanations of what the data represents along with an explanation for what is desirable in certain cases. One feature of VTune that was heavily used in this project was the source code and assembly browser. It allowed selecting a line of code and seeing the assembly instructions corresponding to that line highlighted among the assembly for the entire function/method. This provided both useful information on how the compiler influenced the code after various changes as well as some ability to more easily understand why some line may be slower than expected etc [1].

VTune mainly contributed with insights into cache and TLB usage, as well as pipeline usage per line as a forensic measure (understanding why a piece of code performed in a certain manner). However, as mentioned by A. Marowka, VTune is limited in terms of analyzing high-level concepts of the code [9].

Valgrind is a framework that provides a core and upon which skins are built. It provides the basis for heavy dynamic binary instrumentation allowing fine-grained instrumentation of programs regardless of source code availability. The core is the base of the framework. It provides the instrumentation functionality to the skin developers. The skin is the actual tool that provides data. It essentially provides the core with what to do in certain runtime scenarios such as on calls to malloc. One limitation of Valgrind is that it cannot instrument kernel code, but it can wrap syscalls and use a “description” of the syscall to infer information about what is done. Valgrind adds significant overhead, but allows for easy development of skins and analysis of applications without requiring any recompilation. However, the overhead of Valgrind differs for different scenarios. For example, Valgrind implements its own pthreads library in userspace and thus may behave differently in single threaded scenarios in contrast to multi threaded scenarios [10]. Valgrind was was the tool used most to understand the performance of CannyFS. More specifically, DHAT, Massif, Cachegrind, and Callgrind were used.

Cachegrind was used along with cg annotate to annotate the source code with the data generated by Cachegrind, although exploring this data was difficult as it required a complete picture of the code’s behaviour and layout. As it required an idea of what to look for before attempting to gain an understanding of its data, it was not further used. Massif along with massif-visualizer was used. This provided data that was easy to explore and gain an intuition for the program’s heap allocation behaviour through a
line graph with time on the X-axis and MB allocated memory on the Y-axis. For each function performing many allocations there was a line, the same for each allocator, and also a line for total allocation. DHAT also provided data on the program’s heap allocation behaviour, although this was more detailed and allowed for filtering allocations on e.g. short-lived allocations and allocations that were used few times. DHAT appeared better suited for gaining in-depth understanding of various behaviours of the program (and whether some specific allocation pattern can be optimized) rather than discovering an optimization opportunity in general. DHAT provided data in the form of a tree with a node representing some allocations’ stack trace. Each child node represents a subset of the allocations of its parent.

Callgrind (formerly known as CallTree [4]) allows for a close approximation of various hardware event counters by way of a simulated processor with a single cache, and lacking out-of-order execution and speculation. This allows for a cost approximation along with so called context call trees (CCTs) that can be thought of as callgraphs with some slight modifications. Callgrind was used in conjunction with KCachegrind which provides a GUI to visualize data and callgraphs with varying levels of granularity [21]. Callgrind and KCachegrind were useful both for gaining an intuitive, coarse-grained understanding of time-intensive parts of the code along with the reasons for these being called, as well as finding concrete numbers showing more fine-grained details. Callgrind and KCachegrind were the primarily used tools to inform which optimizations have high potential to yield performance improvements. Some results from Callgrind and KCachegrind can be seen in Section 4.1.

4 Analysis of the baseline CannyFS

This section analyzes the baseline version of CannyFS by attempting to first of all find parts of the program that could provide significant enough benefits to warrant optimization efforts and then attempting to find what about them could potentially be optimized.

4.1 Experiments & Results

In tables 2 and 3, the details of running the Linux utility unzip version 6.00 on the Linux kernel source tree for version 5.16 and then removing the resulting directory tree using rm from the GNU Coreutils version 9 can be seen. The data was gathered by Callgrind from Valgrind 3.18.1 and summarized by KCachegrind 21.12.1. Two main types of “costs” are explored in this section: self-cost and inclusive cost. A cost is an analogue for time. That is, a high cost for a function means a lot of time was spent in that function. A function’s self-cost describes the time spent in that function excluding the costs of its callees whereas a function’s inclusive cost describes the time spent in that function including the costs of its callees. All self-costs add up to 100%, whereas inclusive costs may add up to more than 100%. These costs can prove useful in many ways. For example - if a high self-cost function does not do much useful work, rather mainly calls other functions doing work and combining their results, it has introduced high overhead. If a high inclusive cost function calls many high self-cost functions, it may be worth minimizing the calls to those functions. These are only examples, but they illustrate the idea that optimizing a high self-cost function requires improving the algorithms used in the function or reducing overhead, whereas optimizing a high inclusive cost function may also require optimizing its callees or calling them less.

To understand the optimizations that will be presented, it may be helpful to understand what the functions in tables 2 and 3 represent. get_filedata_inner is a function that essentially fetches file metadata for CannyFS’s internal handling of files. fuse_session_process_buf is internal to FUSE and processes requests to the FUSE filesystem. It calls several of the functions in CannyFS. start_thread and all functions below it in Table 3 represent asynchronous IO work done by CannyFS. These functions are all a part of the same call stack. boost::...::lexically_normal() converts a filesystem path to one that e.g. does not contain redundant directory separators or dots (such as ./ in /a/./b/c). memcmp_avx2_movbe() is a part of std::Rb_tree<...>::find() which is called by
Table 2: The top 6 highest self-cost functions with costs given as a percentage of total program cost. The data for this table was gathered on a laptop with an Intel Core i5-7200U, and 8 GB of RAM running Arch Linux with the 5.16.0 Linux kernel. This data was collected using Callgrind in Valgrind 3.18.1.

<table>
<thead>
<tr>
<th>Self cost (%)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.62</td>
<td>int_free()</td>
</tr>
<tr>
<td>7.67</td>
<td>malloc()</td>
</tr>
<tr>
<td>6.12</td>
<td>_memcmp_avx2_movbe()</td>
</tr>
<tr>
<td>5.76</td>
<td>std::Rbtree&lt;...&gt;::find()</td>
</tr>
<tr>
<td>5.74</td>
<td>free()</td>
</tr>
<tr>
<td>5.40</td>
<td>boost::...::lexically_normal()</td>
</tr>
</tbody>
</table>

get_filedata_inner. The find function actually fetches the metadata that get_filedata_inner returns.

Note that the get_filedata_inner function has an inclusive cost of 53.53% of total execution cost which is the highest inclusive cost of any singular function. According to the callgraph generated by KCachegrind, one of the functions called by get_filedata_inner is find on std::Rbtree which in turn calls _memcmp_avx2_movbe(), two high self-cost functions making up 5.16% and 6.12% of the execution time respectively. Note that std::Rbtree::find() is only ever called by get_filedata_inner as seen by the total number of calls to it and the number of calls to it from get_filedata_inner in the callgraph as well as from reading the CannyFS source code. It is worth noting that get_filedata_inner is the second most called function in CannyFS itself (i.e. in the CannyFS source file). It is also notable that the asynchronous IO queue handling only takes up 29.80% of the time, as can be seen in Table 3 by the start_thread entry and below.

Among the baseline measurements for the real-world benchmarks, there was one extreme outlier. It is a 33.338 second timing value in unzip-ramdisk. Which is 6.203 seconds higher than the second highest timing value, or 27.4 IQRs higher than the upper quartile. This is far higher than what Tukey considered enough to be a far outlier [17, p. 44]. This outlier can be seen in comparison with the rest of the distribution in Figure 3. This outlier will be considered highly anomalous due to its large deviation from the rest of the population and may be excluded in data visualization for visual clarity in the results as it likely was disproportionately affected by some unknown system overhead.

4.1.1 Discussion & potential optimizations

It is notable that the time spent performing asynchronous IO in the scenario presented in tables 2 and 3 is fairly small (≈ 30% as seen from the entry start_thread and down in Table 3), indicating a fairly sequential scenario. This may indicate that optimizations on sequential code can have especially noticeable effects in this scenario.

The data in Table 2 opens up for some potential optimizations. Looking at the two highest self-cost functions, malloc and free (which is called directly from free), one can propose that lowering the number of allocations happening on the heap could potentially lower the execution time of the entire application by calling these functions fewer times. Potentially, other implementations of these functions -
such as tcmalloc - could also be used to improve performance.

Likewise, the find function seen in Table 2 has a high self-cost. This is (as will be seen in sections 6 and 7) a significant part of the use of a C++ STL set called data within get_filedata_inner (as seen in Table 3). Since get_filedata_inner is also a function being called often in the workloads being investigated, and it calling high cost functions, optimizing it may yield significant results. It should be noted that it is likely not justified for get_filedata_inner to have such a high inclusive cost as it is a helper function not doing significant “useful work”, rather just fetching metadata for files.

One could notice lexically_normal() in Table 2. It has a high self-cost, and an inclusive cost of 21.28%. Whilst this is a high cost it may be necessary as reading the code yields that there likely are no redundant calls to this. Solutions such as memoization (caching function results) may be possible. However, due to the perceived complexity of optimizing this it is not further explored in this report.

5 Data presentation and analysis

For presenting the data from the six real-world experiments - unzip-nfs, unzip-local, unzip-ramdisk, rm-nfs, rm-local, and rm-ramdisk - box plots have been chosen as they can give both useful point estimates such as the median as well give a graphical intuition about the data distribution. To determine outliers (outliers are plotted as points in the box plots), Tukey’s fence method has been used (any value above the upper fence, or below the lower fence is considered an outlier). It should however be noted that the statistical significance of using this method is low for the small sample size experiments (10 data points), but slightly better for the large sample size experiments (80 data points) [3]. This means that no conclusions should be drawn from the box plots’ outliers alone (without other supporting evidence). The (inner) fences are placed at lower quartile $-$ 1.5$IQR$ and upper quartile + 1.5$IQR$ (Inter Quartile Range). The type of box plots used is as described by John W. Tukey in Exploratory data analysis [17, p. 39-56]. Although this report uses some slightly different terminology than Tukey’s descriptions, the meaning is the same.

For each results section for an individual candidate optimization, the results are presented with six box plots, and three tables. Each of the six box plots represent the distribution of the timing results in seconds from one of the real-world benchmark configurations. These box plots aim to give an intuitive understanding of the data distribution. One of the tables has colour-coded cells and is presented in Appendix A, this more precisely describes the results seen in the box plots in order to provide a more detailed understanding of the data distribution along with a colour-based overview showing “at a glance” how likely it is that a candidate optimization is indeed an optimization. Each of these cells has a percentage value describing the difference between the corresponding baseline value and experimental program value as a percentage of the baseline value. If the experimental value is lower than the baseline value (negative percentage), the cell is coloured green. If the experimen-
tal value is higher than the baseline value (positive percentage), the cell is coloured red. If the difference is zero, the cell is coloured yellow. This means that a large fraction of green cells indicates that it is likely that the particular candidate optimization is faster than the baseline, while a large fraction of red cells means the opposite.

The two remaining tables show the results of the seq-multithread and unique-open synthetic benchmarks. These tables contain values in seconds for the baseline measurements on the tmpfs filesystem (Base-ram), and on the ext4 filesystem (Base-ext4). Additionally, the table contains values in seconds for the experimental program on the tmpfs filesystem (Exp-ram), and on the ext4 filesystem (Exp-ext4). The values presented are abbreviated AVG for average, MED for median, and MIN for minimum.

One important note to make about the results is that there ought to be different levels of importance placed on different statistics originating from measurements. For example a maximum value may not be useful in analysis as some underlying filesystem or other part of the computer system may add unpredictable overheads. Consider for example one of the -nfs experiments. It is entirely possible that sudden network congestion or high usage of the storage may cause one single experiment or a sequence of experiments to display abnormally high timings. On a local disk these effects may happen due to e.g. sudden higher level of disk vibration [12]. Due to this type of effect, maximum values are largely ignored, and relatively low importance is placed on upper quartiles. Instead, the focus is placed on the lower timing portions of the data as well as statistics such as averages and medians where any such effects have a somewhat limited impact. The reason that these statistics need to be analyzed rather than just inspecting minimum values is that the program itself may have some stochastic behaviour, either due to e.g. the file name randomization in unique-open or due to the OS performing some tasks concurrently with the performance experiment. Thus a trade-off is necessitated as there needs to be enough analysis emphasis put upon the upper parts of the data distribution to include the program’s stochastic behaviour, while not accidentally including too much of the non-CannyFS overhead. Note that the variability of non-program overhead has been minimized by calling sync just before running the experiments, and including a call to it after the experiment in the timing. However, as mentioned there may still be other types of non-CannyFS overheads (with both constant and variable magnitudes).

6 Using unordered_set for data

By reading the implementation of the C++ STL used on the machine that was used to run the experiment detailed by tables 2, and 3 (from GCC-11.1.0), it was found that the implementation uses red-black trees for the set data (which is a set containing data for the files CannyFS is tracking at any given moment that may be referred to as “the filemap”). This confirms that std::Rb_tree<...>::find() represents the find calls on the set data (this was not called from anywhere else than get_filedata_inner). This along with the costs seen in tables 2, and 3 allows for the conclusion that get_filedata_inner has a high cost and that more specifically finding entries in the set data has a high cost.

Also note that the time complexity of find for an associative container, such as std::set is logarithmic in a standard-compliant implementation of C++ [15, p.825]. However, unordered associative containers such as unordered_set are defined to run find in constant time on average and linear time (in the size of the collection) for the worst case [15, p.836]. Given large enough size of data and small enough overhead from some appropriate collection type with constant time lookup, changing the type of data would result in higher performance. It would likely do this by removing the costs of _memcmp_avx2_movbe() (used in comparison of elements) and other parts of std::Rb_tree<...>::find(), replacing them by far lower self-cost functions. Also note that the find operation happens under a global lock on the filemap which is needed for both reading and writing before returning to the caller. Therefore less contention on this lock by way of faster operation under its critical sections may allow reads to complete quicker, and deferred writes to return to the caller quicker.
The optimization being explored in this section (called unordered) consists of switching the data structure used for `data` to `std::unordered_set`. Since `get_filedata_inner` is called for both reading and writing, this allows building the hypothesis that when many different files are accessed - this optimization should speed the program up. This covers the bottleneck type described by Weidendorfer that is an unfitting choice of algorithms [20].

### 6.1 Results

Results from the real-world experiments are detailed graphically in Figure 4 and in table format in Table 6. What can be noticed is that `rm-local`, `unzip-ramdisk`, `unzip-nfs`, and `rm-ramdisk` show performance improvements on all parts of the box plots in Figure 4. This in contrast to `unzip-local` which shows performance degradation on all parts of the box plots. 4 out of 6 real-world experiment types display a speedup on all parts of the box plots of Figure 4, whereas 1 experiment type shows mixed results, and 1 experiment type shows a slowdown on all values.

From profiling the experimental program and baseline program using Callgrind from Valgrind 3.18.1 on unzipping the Linux 5.16 source tree and subsequently deleting it on ext4 atop an SSD on a laptop with an Intel Core i5-7200U, and 8 GB of RAM running Arch Linux with the 5.16.0 Linux kernel - the following was found. The calls to `find` on `data` (that in the experimental program happen on a hash table) had an inclusive cost of 10.56% of total program CPU-time for the experimental program, and 11.54% inclusive cost for the baseline. However, `_memcmp_avx2_movbe` in the `find` operation reduced from 5.78% of the `find` operation’s runtime in the baseline to 0.33% in the experimental program. `_memcmp_avx2_movbe` was called in majority (as decided by CPU-time) by functions handling filesystem paths as strings. This is in line with what was presented as expected in the introduction to this section (the inclusive cost of `find` has been reduced indicating that some parts of it have lower cost than the baseline, and the time spent in `_memcmp_avx2_movbe` has been reduced).

It is notable that reading the implementation of the particular hash function used for the unordered set (on the systems used to test) yielded that it hashes the string representation of each separate part of the path separately. It then combines these partial hashes for a complete hash. In simplified terms, each “directory level” (essentially parts separated by `/` or other separator) in the path is hashed separately, and finally combined.

The results on `seq-multithread` are shown in Table 4. What is noticeable is that the results are mixed overall with more values suggesting a performance improvement rather than degradation. For the tmpfs runs, there appears to be an improvement on the order of 1.5% (by the median). However, for the ext4 runs, the results are more mixed with only the median suggesting an improvement of 0.5% whereas the other values suggest a performance degradation.

The results of `unique-open` can be seen in Table 5. These results consistently show a performance improvement for this candidate optimization. No value suggests a performance degradation. The medians suggest that the performance improvement is 4.6% on ext4, and 6.3% on tmpfs.

![Table 4: Results for `seq-multithread` on the program where `data` uses `std::unordered_set` (called unordered).](image)

<table>
<thead>
<tr>
<th></th>
<th>Base-ram</th>
<th>Base-ext4</th>
<th>Exp-ram</th>
<th>Exp-ext4</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>32.50</td>
<td>32.35</td>
<td>32.35</td>
<td>33.19</td>
</tr>
<tr>
<td>MED</td>
<td>32.89</td>
<td>33.33</td>
<td>32.40</td>
<td>33.16</td>
</tr>
<tr>
<td>MIN</td>
<td>29.61</td>
<td>30.48</td>
<td>28.36</td>
<td>31.38</td>
</tr>
</tbody>
</table>

![Table 5: Results for `unique-open` on the program where `data` uses `std::unordered_set` (called unordered).](image)

<table>
<thead>
<tr>
<th></th>
<th>Base-ram</th>
<th>Base-ext4</th>
<th>Exp-ram</th>
<th>Exp-ext4</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>36.77</td>
<td>209.13</td>
<td>34.35</td>
<td>203.74</td>
</tr>
<tr>
<td>MED</td>
<td>36.77</td>
<td>210.80</td>
<td>34.44</td>
<td>201.10</td>
</tr>
<tr>
<td>MIN</td>
<td>35.31</td>
<td>194.25</td>
<td>32.87</td>
<td>191.20</td>
</tr>
</tbody>
</table>
7 Removing redundant find calls on data

Consider the following function.

```cpp
cannyfs_filedata* get_filedata_inner(const bf::path& path, bool always) {
    cannyfs_filedata* result = nullptr;
    bf::path normal_path = path.lexically_normal();
    {
        shared_lock<shared_timed_mutex> maplock(this->lock);
        auto i = data.find(normal_path);
        if (i != data.end()) {
            result = const_cast<cannyfs_filedata*>(&*i);
            maplock.unlock();
        }
    }
    if (always && !result) {
        unique_lock<shared_timed_mutex> maplock(this->lock);
        auto i = data.find(normal_path);
        if (i != data.end()) {
            result = const_cast<cannyfs_filedata*>(&*i);
        } else {
            result = const_cast<cannyfs_filedata*>(
                &(*data.emplace(normal_path).first));
        }
        maplock.unlock();
    }
    return result;
}
```

This function is the original, unoptimized version of the function receiving attempts to be optimized (`get_filedata_inner`). Note the “then-branch” of the if-statement if (`always && !result`). It uses `find` to determine whether there already is an equivalent element in `data` before attempting to emplace (construct an element and insert it) a value in the non-multivalued set while under a unique lock. However, this is redundant as `emplace` does not insert a value if an equivalent value already exists in the set. In that case, it returns (in part) the same output as `find` would have [15, p.821]. Thus, one could remove...
that call to `find` and the related logic, whilst keep-
ing only the call to `emplace`. This particular call
to `find` may be there due to the programmer not
noticing that `emplace` has a special case that does
not insert an element when an equivalent element is
already present in the set or for readability reasons.
Additionally, one could remove the other call to `find`
seen in the above code as it is functionally redundant.
This call - in contrast to the previously mentioned
one - serves the purpose of allowing a potential early
return while using only a shared lock (rather than an
exclusive lock) if what is being looked for is already
present in the set.

Removing only the call to `find` that happens un-
der a unique lock is one candidate optimization being
tested. It is referred to as one-redundant, whereas
removing both calls to `find` is another candidate op-
timization. It is referred to as no-redundant.

This covers the reason for sequential bottle-
necks presented by Weidendorfer that is redundant
calls [20]. The first call to `find` can be removed
as it is functionally redundant, however keeping it
may allow better concurrent/parallel performance as
that `find`-operation happens under a shared lock (al-
lowing multiple readers to hold the lock at the same
time) whilst allowing earlier return from the function
without needing exclusive access to the lock.

Due to the fact that `get_filedata_inner` is called
for both reads and writes, the hypothesis being tested
for one-redundant is that it performs better than or
equal to the baseline on any type of workload as it
removes a redundant call whilst keeping the reader-
writer lock mechanism. For no-redundant, the hy-
pothesis being tested is the same. The reason for this
hypothesis on no-redundant is that any advantage of
a reader-writer lock being lost may be outweighed by
the removal of a functionally redundant call.

7.1 Results

Figure 5 and Table 17 present the results from the no-
redundant program, likewise Figure 6 and Table 18
present the results from the one-redundant program.
For both of these, the results are presented in com-
parison to the baseline.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{AVG} & \text{MED} & \text{MIN} \\
\hline
\text{Base-ram} & \text{266.77} & \text{209.13} & \text{39.39} & \text{206.62} \\
\text{Exp-ram} & \text{205.34} \\
\hline
\end{array}
\]

Table 6: Results for `unique-open` on the no-
redundant program.

\[
\begin{array}{|c|c|c|c|}
\hline
\text{AVG} & \text{MED} & \text{MIN} \\
\hline
\text{Base-ram} & \text{32.50} & \text{32.89} & \text{29.61} \\
\text{Exp-ram} & \text{32.93} & \text{33.33} & \text{30.48} \\
\hline
\end{array}
\]

Table 7: Results for `seq-multithread` on the no-
redundant program.

The real-world results of no-redundant show a con-
sistent performance degradation in terms of four out
of five box plot parts (whereas the rest are mixed and
leaning heavily toward performance degradation) for
the experiments on local ext4-mounted disk and the
tmpfs filesystem. For the nfs experiments, the re-
sults are more mixed, however these still show perfor-
mance degradation on metrics such as lower whisker
and lower quartile.

The real-world results of one-redundant show a
clear performance degradation for `unzip-local`. The
nfs experiments show mixed results, although `rm-nfs`
has more box plot parts indicating a speedup than
`unzip-nfs`. `rm-local` shows a consistent performance
improvement across all box plot parts. The tmpfs
experiments show a slightly mixed performance im-
provement with a majority of box plot parts indicat-
ing a speedup in both cases.

For `unzip-local` using the no-redundant program re-
sulted in the timings presented in Table 6. These
results show a clear performance degradation on the
order of 7% for the tmpfs filesystem, however a per-
formance improvement on the order of 3% for the
ext4 filesystem.

The timing values of `seq-multithread` (with no-
redundant) are presented in Table 7. These re-
sults are mixed for both the ext4 and tmpfs filesys-
tems with the baseline being slower than the no-
redundant program on all statistics on tmpfs except
the minimum, whereas all statistics but the median
on ext4 show a performance degradation for the no-
The benchmark unique-open with the one-redundant program resulted in the values presented in Table 8. These results show a consistent performance improvement on the order of 10% across all statistics on the tmpfs filesystem, and a performance improvement on the order of 2% on the ext4 filesystem (also across all statistics).

Running seq-multithread under one-redundant resulted in the values presented in Table 9. The results for the ext4 filesystem are mixed. However, on the tmpfs filesystem a consistent performance degradation on the order of 13% for the one-redundant program is seen on all statistics.

Additionally, unique-open modified to open one million files was used to profile one-redundant and the baseline program with Intel VTune 2022.1.0. VTune’s hotspot and microarchitecture analysis features were used. The machine that was used to profile was a laptop with an Intel Core i5-7200U, and 8 GB of RAM running Arch Linux with the 5.16.0 Linux kernel. Consider the following code snippets delimited by comments. The first snippet comes from the baseline program, and the second snippet comes from the one-redundant program.

```cpp
//Baseline
shared_lock<shared_timed_mutex> maplock(this->lock);
auto i = data.find(normal_path);
if (i != data.end())
{
    result = const_cast<cannyfs_filedata*>(&*i);
    maplock.unlock();
}
//End of snippet 1

//One-redundant
shared_lock<shared_timed_mutex> maplock(this->lock);
auto i = data.find(normal_path);
result = i != data.end() ?
        const_cast<cannyfs_filedata*>(&*i) : nullptr,
    maplock.unlock();
//End of snippet 2
```

The lines on which calls to find are placed have results from the profile using a 0.01 ms sampling rate presented in Table 10. The number of basic blocks is the amount of basic blocks the individual line was compiled to, and the CPU-time is the time spent by the CPU doing useful work. The pipeline usage is the percentage of a pipeline slot that is doing useful work (fraction of the maximum amount of retiring micro-ops per pipeline slot actually being retired), while the bad speculation metric shows what fraction of a pipeline slot is wasted due to incorrect speculation (such as incorrect branch prediction). The back-end dependence metric describes what fraction of a pipeline slot is being wasted due to the processor’s back-end not being able to accept more micro-ops.

What can be noticed is that the amount of basic blocks for the line with find has seemingly increased by 49% with the optimization. The amount of wasted time and resources due to poor use of the processor’s speculative features also appears to have
increased with the one-redundant program. On that line, the results showed that the baseline program retired 40 200 000 instructions at 0.525 CPI (cycles per instruction), while the one-redundant program retired 470 300 000 instructions at 1.945 CPI, an increase in number of retired instructions of 11.70 times.

However, note that when using GCC’s option `-fdump-ipa-inline` and inspecting the inlining decisions, the baseline program did not have the `find` function inlined, but the one-redundant program did (thus indicating that the previous comparison does not include the call to `find` on the baseline profile). This shows that the previous comparison (increase in number of basic blocks, retired instructions etc.) compares the line excluding the call to `find` (for the baseline) with the the same line including the call to `find` (for the one-redundant program). Inspecting the profile for the baseline program further yielded that its calls to `find` had 14.7% pipeline waste due to bad speculation and 67.1% due to the back-end being unable to accept new micro-ops. It also ran at 1.329 CPI. Note that according to the hotspot analysis the calls to `find` on the baseline program took 2.933 seconds (inclusive), and on the one-redundant version 2.524 seconds (inclusive).

8 Combining optimizations from Sections 6 and 7

It is possible that removing a redundant call to `find` performs significantly different for a program using an unordered set for `data` in relation to one that uses an ordered set. Thus, this section explores combining the optimization from section 6 and the one-redundant optimization from section 7. In other words, this optimization consists of the set `data` using an `std::unordered_set` whilst also having the removal of the redundant call to `find` in `get_filedata_inner` that happens under a unique lock. The hypothesis being tested in this section is that this combined optimization makes workloads using many files, and particularly many unique files perform better. The reason for this hypothesis is...
In Figure 7 and Table 19 the real-world results from the main experiments for the combined optimization can be seen. For rm-local, unzip-ramdisk, and rm-ramdisk, there is a consistent performance improvement seen in the lower parts of the box plots. The improvement is smaller for the unzip-ramdisk experiment than for the rm experiments in terms of the medians. unzip-local and unzip-nfs show no majority of box plot parts indicating neither improvement or performance degradation. rm-nfs shows a clear and consistent performance improvement on all box plot parts.

unique-open resulted in the timing summaries presented in Table 11. In these results, an overall consistent speedup can be seen. On the tmpfs filesystem, that speedup is on the order of 9% whereas the speedup on the ext4 filesystem is on the order of 3%.

seq-multithread gave the experiment results presented in Table 12. These results show a consistent performance degradation with only the median on the ext4 filesystem being lower for the experimental program than the baseline program. The perfor-
mance degradation is on the order of 2% on the tmpfs filesystem.

9 Comparative Results

In this section, a comparison of all optimizations on the -ramdisk experiment variants is presented. The -ramdisk variety was chosen due to the lower overhead originating from the underlying storage. This shows any differences in CPU-time the clearest in the end-to-end timing measurements. Another reason for choosing to compare on this data is that on the baseline real-world benchmarks - for both the unzip and rm benchmarks - the smallest IQRs belong to the -ramdisk variants.

In Figure 8 the results for all program varieties explored in this report are compared based on their results on the unzip-ramdisk experiment type. The lowest lower quartile, median, and minimum value (lower portions of the data distribution) all belong to the unordered-1-redundant program, followed by the unordered program in all these values except the minimum value.

In Figure 9 one can see the results for all program varieties explored being compared based on their results on the rm-ramdisk benchmark. The lowest minimum value and median belongs to the unordered program, but the lowest upper and lower quartiles belong to the unordered-1-redundant program.

Table 13 presents the results of all explored optimizations in comparison to each other on the seq-multithread synthetic benchmark using a large sample size of 80 runs per optimization type. The lowest median belongs to the unordered program, and so does the second lowest minimum value. The lowest minimum value belongs to the baseline program, and the second lowest median value belongs to unordered-1-redundant.

Table 14 also presents results for all explored optimizations in comparison to each other (also using the 80 run sample size), however for the unique-open synthetic benchmark. Here, all optimized programs except no-redundant performed better than the baseline on average, median, and minimum values. No-redundant performed worse than the baseline on all

Figure 7: Details for the set of experiments where all redundant calls to find on data but one have been removed in get_filedata_inner as well as having the data structure for data changed from std::set to std::unordered_set (unordered-1-redundant). The results shown as “base” are from the baseline experiments. The results shown as “exp” are experimental results from the version of CannyFS using this change. All measurements are in seconds and as reported by the time-command from starting CannyFS to it finishing and sync completing.
Figure 8: Combined results for the unzip-ramdisk series of experiments over the various optimizations with a sample size of 80 runs per optimization. All measurements are in seconds and as reported by the `time`-command from starting CannyFS to it finishing and `sync` completing. Base represents the baseline program, no-redundant and one-redundant represent the no-redundant and one-redundant programs of section 7, whereas unordered represents the experimental program of section 6, and unordered-1-redundant represents the experimental program of section 8.

Figure 9: Combined results for the rm-ramdisk series of experiments over the various optimizations with a sample size of 80 runs per optimization. All measurements are in seconds and as reported by the `time`-command from starting CannyFS to it finishing and `sync` completing. Base represents the baseline program, no-redundant and one-redundant represent the no-redundant and one-redundant programs of section 7, whereas unordered represents the experimental program of section 6, and unordered-1-redundant represents the experimental program of section 8.
these values. The two best performing programs were unordered and unordered-1-redundant. The differences between their average, median, and minimum values were all less than one second, and the lowest of all three of these values belong to unordered-1-redundant.

10 Discussion

It is worth mentioning that what is considered a “good optimization” in this report is one that improves performance and/or provides clear benefits in some typical workload whilst not predominantly degrading performance or causing other significant disadvantages in other typical workloads. This allows for trade-offs if they provide more or better benefits than drawbacks.

It is notable that one main bottleneck has been found in CannyFS, and that is the get_filedata_inner function. Changing it has been shown to make a difference in end-to-end timing for the various benchmarks, and it appears heavily used in many different workloads. There are some interesting properties to note about get_filedata_inner that may have contributed to its status as a bottleneck. One such property is the fact that it implements functionality that is needed in most operations that contribute to the program’s goal. Another property is that it is a low level building block of the program, i.e. it is used to build up higher level functionality within the program. The final property is that it shows up as a high cost function in a flat profile of the program. These properties combined lead to it affecting the program’s performance in many different code paths that directly contribute to the goal of the program while a small fractional improvement of its performance may lead to a large improvement in absolute performance.

10.1 Using unordered_set for data

As the hypothesis being tested for this particular optimization is that it will have a performance benefit when handling a high number of files - the most relevant benchmarks are the real-world benchmarks as they end up handling a high number of files as well as the unique-open synthetic benchmark.

What can be noticed in the real-world benchmarks is that the unzip-ramdisk benchmark indicates that a significant part of the timing data is distributed such that the optimized program in a significant number of cases performs better. Solely using this data would seem to indicate that the optimized program indeed performs better, however inspecting unzip-local and unzip-nfs causes this conclusion to be less credible. This is as the unzip-local experiment clearly has an experimental data distribution that is above that of the baseline. unzip-nfs seems to show very similar performance between the two programs since the baseline distribution covers a similar range of values as the optimized version with a similar distribution, except that the main distribution is slightly lower on the optimized version. It is interesting to note that - in theory - as the underlying filesystem’s overhead decreases (represented by the different filesystems tested on), the improvement is expected to be more clear due to a higher fraction of the time being spent on CPU-time in CannyFS. However this is not upheld due to the -local version of the unzip experiments appearing to show performance degradation in contrast to the -nfs and -ramdisk experiments which show this expected behaviour.

Consider the following three different possibilities for this non-expected behaviour.

1. unzip-nfs and unzip-ramdisk incur atypical behaviour in CannyFS and show a false optimization.
2. unzip-local incurs atypical behaviour in CannyFS and therefore has unexpected behaviour.
3. No unzip benchmark incurs atypical behaviour in CannyFS and the unexpected behaviour is due to the stochastic nature of the measurements.

While option 1 certainly is possible, it is unlikely for the following reasons. For one, inspecting the rm-benchmarks shows the expected behaviour where rm-nfs shows little to no improvement, rm-local appears to show a small improvement, while rm-ramdisk shows the largest improvement. Similarly -
on unique-open it is clear that the speedup is greater on the tmpfs filesystem than on the ext4 filesystem thus also following the expected behaviour. These reasons show that all the most relevant experiment types (as determined in the beginning of this subsection) except unzip-local follow the expected pattern, making option 1 highly unlikely.

Therefore options 2 and 3 remain as the more likely explanations. However, both of these options allow for the conclusion that workloads that handle a lot of files see performance benefits from using this optimization. Since all experiments outlined as highly relevant to the hypothesis being tested except unzip-local show equal or better performance than the baseline, the data gathered exhibits a high likelihood of behaving as outlined by the hypothesis.

To show that this is an optimization, it is also - as previously discussed - necessary to show that other typical workloads are not overly negatively impacted by the optimization. This is supported by the seq-multithread synthetic benchmark as it shows a consistent slight improvement on the tmpfs filesystem, and varying speedups and slowdowns of similar magnitudes on the ext4 filesystem, likely indicating no significant change. This allows for concluding that replacing the std::set data with an std::unordered_set likely is a good optimization.

### 10.2 Removing redundant find calls on data

As the no-redundant program displayed a consistent slowdown on all real-world benchmarks except possibly the -nfs variants, it is highly likely that the no-redundant program is not an optimization. The -nfs variants’ seemingly better performance may be due to high overhead of NFS masking any slowdowns or creating some more efficient resource usage pattern, thus not providing any grounds to conclude that it is an optimization.

This is further supported by unique-open which shows a significant slowdown on the tmpfs filesystem.

<table>
<thead>
<tr>
<th></th>
<th>base</th>
<th>0-red</th>
<th>1-red</th>
<th>unordered</th>
<th>unordered-1-red</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>32.77</td>
<td>32.81</td>
<td>32.98</td>
<td>32.45</td>
<td>32.80</td>
</tr>
<tr>
<td>MED</td>
<td>32.82</td>
<td>32.66</td>
<td>33.15</td>
<td>32.43</td>
<td>33.03</td>
</tr>
<tr>
<td>MIN</td>
<td>28.66</td>
<td>30.16</td>
<td>29.97</td>
<td>28.97</td>
<td>30.32</td>
</tr>
</tbody>
</table>

Table 13: Combined results for the seq-multithread series of synthetic benchmarks over the various optimizations on ramdisk using a large sample size of 80 runs per configuration. All measurements are in seconds and as reported by the time-command from starting CannyFS to it finishing and sync completing. Base represents the baseline program, 0-red and 1-red represent the no-redundant and one-redundant programs of section 7, whereas unordered represents the experimental program of section 6, and unordered-1-red represents the experimental program of section 8.

<table>
<thead>
<tr>
<th></th>
<th>base</th>
<th>0-red</th>
<th>1-red</th>
<th>unordered</th>
<th>unordered-1-red</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>37.97</td>
<td>40.17</td>
<td>37.70</td>
<td>35.28</td>
<td>35.01</td>
</tr>
<tr>
<td>MED</td>
<td>37.95</td>
<td>40.14</td>
<td>37.59</td>
<td>35.30</td>
<td>35.21</td>
</tr>
<tr>
<td>MIN</td>
<td>35.93</td>
<td>37.96</td>
<td>35.23</td>
<td>33.31</td>
<td>33.00</td>
</tr>
</tbody>
</table>

Table 14: Combined results for the unique-open series of synthetic benchmarks over the various optimizations on ramdisk using a large sample size of 80 runs per configuration. All measurements are in seconds and as reported by the time-command from starting CannyFS to it finishing and sync completing. Base represents the baseline program, 0-red and 1-red represent the no-redundant and one-redundant programs of section 7, whereas unordered represents the experimental program of section 6, and unordered-1-red represents the experimental program of section 8.
However, since \texttt{unique-open} on the ext4 filesystem displayed mixed results there may be some special cases where no-redundant is faster than the baseline program. This may be due to e.g. latency from ext4 and the disk causing the lock on data to used in a more efficient manner, or some latency masking certain other issues. Another possibility is that this may be due to the stochastic behaviour of the program timings and that the actual data distribution suggests a slowdown.

The synthetic benchmark results that show some possibility of the no-redundant program improving performance do not provide enough grounds to justify calling it a good optimization since the performance degradation on the real-world benchmarks was significant. This also leads to rejecting the hypothesis for this optimization (that it would perform better on any workload).

What can be noticed for the one-redundant program’s results on the real-world benchmarks is that in terms of medians, \texttt{unzip-nfs} and \texttt{unzip-ramdisk} both display a near insignificant difference in relation to the baseline. However - similar to what is seen with the no-redundant program and unordered program - the \texttt{unzip-local} results are significantly worse than the baseline. This once again supports the idea that some interaction between ext4 and some part of certain optimizations (or CannyFS in general) cause a performance degradation. As for the \texttt{rm} experiments, they consistently show small speedups. Notice that on \texttt{rm-local} the differences in median and minimum values are so small as to be insignificant, and thus it is reasonable to conclude that the performance is at least equal in this scenario. For \texttt{rm-nfs}, the data points are far more concentrated toward the lower timing values for the one-redundant program than the baseline, so this suggests at least as good performance as the baseline, but likely better.

The results from \texttt{unique-open} on one-redundant show a clear performance benefit when handling many unique (unique to the CannyFS mountpoint’s lifetime) files. However, since the \texttt{seq-multithread} results show significant slowdowns on the tmpfs filesystem on the same order as the speedups seen on the tmpfs filesystem for \texttt{unique-open} there appears to be a tradeoff that is beneficial when handling many files, but highly detrimental when handling few files. Since it was also unclear whether a speedup was seen on \texttt{seq-multithread} for ext4 or not, the conclusion that one-redundant is a good optimization cannot be drawn. The hypothesis for this optimization (that it would perform better than the baseline for any workload) can clearly also be rejected.

This is unexpected as simply removing an often made call ought to reduce time spent and thus be an optimization. However, note the VTune profile results. The reason for the expansions in terms of basic blocks for the source code line investigated by the profile was that the \texttt{find} function was inlined in the one-redundant program, but not in the baseline program. This caused the profile to compare the line without the call to find to the line with the inlined call to find included in the statistics. Inspecting the calls to \texttt{find} instead showed that the baseline program had a clear advantage in terms of cycles per instruction for the \texttt{find} call. One reason for this might be the increased amount of bad speculative work done in the optimized program, or possibly slower front-end operation (as could possibly be seen by the decrease in back-end dependence). Note that since the benchmark used to gather the VTune profile was the \texttt{unique-open} benchmark, a speedup (as seen in the end-to-end timings) is expected. However, what is interesting here is the decrease in efficiency (in terms of CPI) as that can be the reason for the slowdown in the other scenarios. This indicated that microarchitectural features or compiler factors may provide lesser benefits on the optimized program than what is seen on the baseline program.

### 10.3 Combining optimizations from Sections 6 and 7

For this optimization, the \texttt{unzip} experiments show similar performance to the baseline for both the nfs and ext4 experiments. However, the data is more concentrated toward the lower parts of the timing distributions with \texttt{unzip-nfs} showing a possible small performance degradation. \texttt{unzip-ramdisk} however shows a clear performance benefit. These timings alone are neither significant enough nor monotone enough to conclude neither a performance improve-
ment nor performance degradation. Do however note that there appears to be slightly more evidence supporting a performance improvement as the lowest overhead underlying storage (tmpfs) shows an improvement. The rm experiments on the other hand show a clear and consistent performance benefit on all underlying filesystems. This clearly supports a performance benefit on the rm workloads and at least not a performance degradation on the unzip workloads. Therefore these data are in support of this being a good optimization.

The results of unique-open appear to support a performance improvement as experiments on both the ext4 and tmpfs filesystems show clear and consistent performance benefits. Since - for all averages, medians, and minimums - the optimized program displayed speedups of multiple seconds, this workload appears to heavily benefit from this particular optimization. However, since seq-multithread shows consistent slowdowns, it is highly likely that this optimization degrades the performance on workloads behaving like seq-multithread. However, also note that all these performance degradations were small. Given the significantly smaller magnitude of performance degradation seen in seq-multithread in comparison to the performance improvement seen in unique-open, and with the fact that seq-multithread was created to exaggerate the effects of using few files concurrently, this can still be concluded to be an optimization. Also note that the hypothesis for this optimization clearly holds, as when many files are accessed (such as in unique-open, rm, and unzip) there is a clear performance benefit at best and at least equal performance to the baseline (but likely slightly better) at worst.

### 10.4 The best optimization

Note that this subsection uses the data from Section 9, so e.g. rm-ramdisk in this section refers to Figure 9. As these optimizations are essentially mutually exclusive, it is necessary to find which one is the most advantageous. Since neither no-redundant nor one-redundant could be concluded to be good optimizations, they need not be considered here. As there is little difference between unordered and unordered-1-redundant on unzip-ramdisk while unordered-1-redundant may show a small improvement over unordered one cannot conclude that one is better than the other, but it gives merit to the idea that unordered-1-redundant is slightly better than unordered.

The results of rm-ramdisk allows one to argue that unordered-1-redundant could be the most advantageous optimization as it has the lowest median, and upper and lower quartile values of all optimizations on the rm-ramdisk benchmark. However, as the unordered program has the lowest minimum value by a significant margin, this can be disputed. The low minimum value may however simply be due to coincidence of the unordered program running and low system overheads being present. While the probability of something like that happening to only one program type is lowered both by the round-robin manner of running each benchmark and the relatively large sample size of 80 samples per optimization type, this cannot be ruled out. Especially as the differences in timing are relatively small. This allows for the idea that the likelihood of unordered being the fastest on rm-ramdisk is slightly higher than the likelihood of unordered-1-redundant being the fastest on rm-ramdisk. What can however clearly be concluded is that unordered and unordered-1-redundant are the two best optimizations on the rm-ramdisk benchmark as they consistently outperform the other program variants for a significant part of the timing distributions.

As the seq-multithread benchmark on the tmpfs filesystem shows consistently better performance for the unordered program in comparison to the unordered-1-redundant program by a significant amount, it is clear that the unordered program performs better on the seq-multithread. However, note that on unique-open the results show the opposite. It shows consistently better performance for unordered-1-redundant.

While this discussion has shown that the two best optimizations are clearly the unordered program and the unordered-1-redundant program, it is not clear which of these two is the best optimization. As both have been concluded to be good optimizations, none of them would be “a bad choice”.

27
In summary, \textit{unzip-ramdisk} suggests that the unordered-1-redundant program may be the best optimization by a small margin, while the \textit{rm-ramdisk} results suggests that the unordered program might be the best optimization by a small margin. The synthetic benchmarks also show an ambiguous picture as \textit{unique-open} shows better performance for the unordered-1-redundant program, and \textit{seq-multithread} shows better performance for the unordered program.

Intuitively, the unordered-1-redundant program should perform better than the unordered program as they are the same except the unordered program making a redundant call that unordered-1-redundant does not. If the reason for unordered-1-redundant not being clearly faster than unordered is assumed to be microarchitectural (similar to the effect seen in section 7), other processors may show different performance. However, if the data is taken without consideration for such factors, and performance degradation in any form is considered unacceptable - the unordered program is the best optimization as it is better than or insignificantly different to the baseline on \textit{seq-multithread} whereas unordered-1-redundant is clearly slower than the baseline on \textit{seq-multithread}. This is when using both data from Table 13 and the respective optimizations’ individual analyses. As this report did not run the experiments on other processor types, it is not possible to - from the data found - conclude that there is one best optimization. Rather, it can be concluded that the unordered program presented in section 6, and the unordered-1-redundant program presented in section 8 are the two best optimizations.

However, if one simply aims to answer the question “what is one bottleneck in CannyFS” - this report should make it clear that one significant bottleneck is the \texttt{get\_filedata\_inner} function as changing its performance characteristics clearly changes the performance characteristics of the whole program’s runtime. More specifically, the handling of the set \texttt{data} in \texttt{get\_filedata\_inner} is a bottleneck that is heavily investigated in this report.

10.5 Uncertainty of results & limitations of methods

There are some factors that may limit the reliability and value of the various results presented in this report. One such factor is that the benchmarks do not cover all workload types. There may be workload types that are disproportionately negatively or positively affected by the various optimizations presented in this report. Note that to attempt to cover as many relevant program behaviours as possible, it was noted that the part of CannyFS that was studied and optimized was how the filemap - handling keeping track of CannyFS’s files - was used. Thus one of the main factors affecting performance characteristics would logically be the pattern of file usage. While \textit{seq-multithread} and \textit{unique-open} capture using many files and few files, they don’t capture e.g. order of file usage, different intensities of usage (such as accessing files in bursts with different intervals etc.) or using different file operations.

A potential problem affecting \textit{seq-multithread} is the following. While it is possible for the overhead in \textit{seq-multithread} stemming from flushing IO queues (such as thread synchronization) to dominate, very clear and significant differences have been seen between the various optimizations. This indicates that overhead is at least small enough to see differences in \texttt{get\_filedata\_inner} and its related program behaviours. Although, some differences may still be masked and the overhead may be unrealistic, the value of investigating a scenario that is highly serialized by CannyFS is high enough to warrant using this benchmark. Additionally, what is most important is that the overhead from e.g thread synchronization is consistent across runs, which it likely is.

Since the data from all experiments contained large, unpredictable system overhead, several methods were used to reduce data noise from runs disproportionately affected by this overhead. One potentially problematic such method is the increased value placed on the lower parts of the data distributions. This may cause a poorly performing program to be labeled as performing well due to unintentional cherry-picking of data. However, with the particular optimizations being presented in this report - it is

28
unlikely that some optimization adds large random overheads with low enough probability to occur so that it is not included in the data being considered.

Another thing to consider is potentially misleading profiling results from Callgrind which was heavily used to inform which optimizations to attempt. Callgrind makes use of the architecture simulation technique to make sure its results are accurate. More specifically it uses cache simulation of a two-level inclusive cache [20]. While this should limit the measurements’ unreliability due to the instrumentation, a significantly different microarchitecture may cause these results to be unrepresentative. As cache simulation is used in this case and Callgrind has not been used for purposes other than aiding in finding and understanding candidate optimizations while the optimization results have been verified by other means, this poses no problem for this particular work. Neither should the fact that Callgrind was used on other machines than those used to test as the architecture simulation should cause this to make little difference.

It is worth discussing why the tmpfs scenarios were used to evaluate the differences between the candidate optimizations. One could argue that the tmpfs benchmarks are the least representative of a true scenario and therefore should not be used. However, the tmpfs benchmarks clearly showed the most consistent and the least noisy data. Also, since every filesystem has different overhead characteristics - optimizing for one underlying filesystem (or a selection of commonly used underlying filesystems) could be pointless as those optimizations may not carry over to other underlying filesystems. While this effect can be seen in tmpfs, it is seen far less than for other filesystems as tmpfs minimizes overhead overall by only using RAM until swapping to disk is necessary [13]. This minimized overhead causes the FS overhead variability to contribute with a far smaller difference than other filesystems. Also it is free of e.g. network effects, spinning hard drive effects, and SSD effects. This makes tmpfs show the largest amount of pure CannyFS performance with the least amount of particular effects from storage medium or underlying filesystem.

11 Future work

There are a number of potential bottlenecks this work did not cover. It could be worth further investigating these. One such potential bottleneck is the possibility of excessive heap allocations causing various performance issues. The reason for suspecting this is that the two highest self cost functions both relate to heap allocations (malloc and free). It is also the case that Valgrind Massif and DHAT showed clear spikes in the heap allocations that were quickly deallocated (large, short-lived allocations).

Also, causal profiling was tried. However, due to low tool maturity and lack of documentation, this did not provide useful data. Profiling CannyFS with a causal profiler (such as Coz) could lead to interesting results as the metrics that need improvement can be set as the performance metric to be measured. Also, a causal profile gives a directly testable hypothesis by giving information such as speeding up line $a$ by $b\%$ improves performance on $c$ by $d\%$. In other words, it can be decided that e.g. reads need to be optimized and a causal profiler could find a potential optimization opportunity automatically that is then easily tested after making an optimization.

It could also be worth exploring how compilation factors affect CannyFS performance. This could for example be link order, enabling/disabling optimizations, labeling certain code as hot code (although GCC appears to do this fairly well on its own), as well as influencing inlining decisions. It could even be worth comparing performance between GCC, clang, and e.g. the Intel C++ compiler.

Due to the limitations of the benchmarks used, it could also be worth exploring how these optimizations perform using different benchmarks capturing different behaviours. Another possibility would be to use some standardized benchmark suite that captures many different behaviours such as the IO500 benchmark [7].

12 Conclusions

This report has shown that the function `get_filedata_inner` is a bottleneck in CannyFS.
It also shows that this bottleneck can be optimized by two methods. One of these optimizations is to replace the filemap named data (an `std::set`) with an `std::unordered_set`. The second optimization is a combination of the first optimization and removing a redundant call to `find` on data.

Additionally, two candidate optimizations were not shown to be optimizations. The first one of these consists of removing a redundant call to `find` on data. The second one consists of removing the same call, and additionally another `find` call along with a shared (reader-writer) lock mechanism being supported by this call.

Finally, the candidate optimizations (including those that were not concluded to be optimizations) were compared against each other on the benchmark scenarios deemed to give the least noisy data while also having the smallest dependence on the underlying storage’s performance characteristics possible, whilst still giving a complete picture of the performance of CannyFS. It was found that there were two best optimizations from which it was not possible to distinguish which of the optimizations was the best (switching the data structure of data to `std::unordered_set`, and doing so while also removing a redundant call to `find` on data).

This report has also made some more general findings. One of these findings is that end-to-end runtime is an imprecise measurement that - while it provides an easily understandable performance approximation - does not show the complete picture of performance. This is especially valid in complex, complete systems such as CannyFS layered upon a networked filesystem. This is as there is no representation of e.g. overhead. Other, more focused measurements are far more complex to implement, use, and interpret.

Another general finding is that seemingly obvious optimizations may not cause clear performance improvements. An example of this is the one-redundant program. What was seen was that removing a purely redundant call did not clearly improve performance. In that particular case, the reason for not causing a performance improvement appears to have been microarchitectural. The reason in and of itself however is not important, but it describes the phenomenon that causes this, which is the removal of one performance problem causing the creation of other performance problems.

However, the possibly main general conclusion that can be drawn is that performance problem types seen in sequential code can make large differences in concurrent scenarios. This has been seen throughout the whole report as it has been relating performance problems from Weidendorfer’s list of three performance problem types presented in section 1 to the concurrent performance effects seen and explored throughout the whole report.

References


### A Tables describing box plots

This appendix contains tables describing the box plots seen throughout the report more quantitatively. Each cell describes the percentual difference between the baseline’s value, and the optimized program’s value.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>UW</th>
<th>LW</th>
<th>MIN</th>
<th>MAX</th>
<th>UQ</th>
<th>MED</th>
<th>LQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>unzip-nfs</td>
<td>-1.8%</td>
<td>-0.2%</td>
<td>-0.2%</td>
<td>-1.8%</td>
<td>-0.5%</td>
<td>-0.4%</td>
<td>-0.5%</td>
</tr>
<tr>
<td>rm-nfs</td>
<td>+1.9%</td>
<td>-3.3%</td>
<td>-3.3%</td>
<td>+1.9%</td>
<td>+16.4%</td>
<td>+2.6%</td>
<td>-5.9%</td>
</tr>
<tr>
<td>unzip-local</td>
<td>+4.3%</td>
<td>+4.1%</td>
<td>+3.7%</td>
<td>+4.3%</td>
<td>+3.1%</td>
<td>+4.2%</td>
<td>+1.6%</td>
</tr>
<tr>
<td>rm-local</td>
<td>-8.4%</td>
<td>-5.9%</td>
<td>-5.9%</td>
<td>-8.4%</td>
<td>-6.0%</td>
<td>-5.5%</td>
<td>-5.8%</td>
</tr>
<tr>
<td>unzip-ramdisk</td>
<td>-2.1%</td>
<td>-1.4%</td>
<td>-2.0%</td>
<td>-19.1%</td>
<td>-1.7%</td>
<td>-1.3%</td>
<td>-1.5%</td>
</tr>
<tr>
<td>rm-ramdisk</td>
<td>-7.6%</td>
<td>-7.6%</td>
<td>-7.6%</td>
<td>-7.6%</td>
<td>-6.2%</td>
<td>-6.8%</td>
<td>-6.2%</td>
</tr>
</tbody>
</table>

**Table 15:** Table describing the meanings of the abbreviations seen in the tables in this appendix.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>UW</th>
<th>LW</th>
<th>MIN</th>
<th>MAX</th>
<th>UQ</th>
<th>MED</th>
<th>LQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>unzip-nfs</td>
<td>-10.3%</td>
<td>+1.4%</td>
<td>+1.4%</td>
<td>-10.3%</td>
<td>-1.1%</td>
<td>-3.0%</td>
<td>+0.1%</td>
</tr>
<tr>
<td>rm-nfs</td>
<td>+5.3%</td>
<td>-0.1%</td>
<td>+30.1%</td>
<td>+5.3%</td>
<td>+4.7%</td>
<td>+6.5%</td>
<td>+1.4%</td>
</tr>
<tr>
<td>unzip-local</td>
<td>-0.9%</td>
<td>+4.9%</td>
<td>+4.9%</td>
<td>-0.9%</td>
<td>+2.9%</td>
<td>+4.3%</td>
<td>+4.3%</td>
</tr>
<tr>
<td>rm-local</td>
<td>+5.8%</td>
<td>+0.2%</td>
<td>+1.3%</td>
<td>-13.9%</td>
<td>+5.3%</td>
<td>+4.3%</td>
<td>+2.0%</td>
</tr>
<tr>
<td>unzip-ramdisk</td>
<td>+2.5%</td>
<td>+1.2%</td>
<td>+1.2%</td>
<td>+2.5%</td>
<td>+4.1%</td>
<td>+3.7%</td>
<td>+2.7%</td>
</tr>
</tbody>
</table>

**Table 16:** Result description of Figure 4 (the unordered program).

**Table 17:** Result description of Figure 5 (the no-redundant program).
<table>
<thead>
<tr>
<th>Experiment</th>
<th>UW</th>
<th>LW</th>
<th>MIN</th>
<th>MAX</th>
<th>UQ</th>
<th>MED</th>
<th>LQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>unzip-nfs</td>
<td>-3.3%</td>
<td>+0.5%</td>
<td>+0.5%</td>
<td>+2.6%</td>
<td>-0.9%</td>
<td>-0.2%</td>
<td>0.0%</td>
</tr>
<tr>
<td>rm-nfs</td>
<td>-15.1%</td>
<td>-1.8%</td>
<td>-1.8%</td>
<td>+9.2%</td>
<td>-4.8%</td>
<td>-12.0%</td>
<td>-4.1%</td>
</tr>
<tr>
<td>unzip-local</td>
<td>+1.4%</td>
<td>+0.3%</td>
<td>+30.6%</td>
<td>+1.4%</td>
<td>+3.3%</td>
<td>+4.5%</td>
<td>+2.4%</td>
</tr>
<tr>
<td>rm-local</td>
<td>-3.1%</td>
<td>-0.5%</td>
<td>-0.5%</td>
<td>-3.1%</td>
<td>-1.9%</td>
<td>-0.7%</td>
<td>-0.8%</td>
</tr>
<tr>
<td>unzip-ramdisk</td>
<td>+2.9%</td>
<td>-1.3%</td>
<td>-0.2%</td>
<td>-16.3%</td>
<td>+1.6%</td>
<td>-0.3%</td>
<td>-1.2%</td>
</tr>
<tr>
<td>rm-ramdisk</td>
<td>-3.9%</td>
<td>-2.3%</td>
<td>-7.3%</td>
<td>+0.2%</td>
<td>-3.0%</td>
<td>-3.0%</td>
<td>-3.1%</td>
</tr>
</tbody>
</table>

Table 18: Result description of Figure 6 (the one-redundant program).

<table>
<thead>
<tr>
<th>Experiment</th>
<th>UW</th>
<th>LW</th>
<th>MIN</th>
<th>MAX</th>
<th>UQ</th>
<th>MED</th>
<th>LQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>unzip-nfs</td>
<td>-1.5%</td>
<td>+1.1%</td>
<td>+1.1%</td>
<td>+2.3%</td>
<td>-0.8%</td>
<td>-0.4%</td>
<td>+0.1%</td>
</tr>
<tr>
<td>rm-nfs</td>
<td>-22.2%</td>
<td>-2.7%</td>
<td>-2.7%</td>
<td>+1.8%</td>
<td>-9.6%</td>
<td>-9.0%</td>
<td>-6.0%</td>
</tr>
<tr>
<td>unzip-local</td>
<td>-5.9%</td>
<td>+0.6%</td>
<td>+0.6%</td>
<td>+5.6%</td>
<td>-2.1%</td>
<td>+0.2%</td>
<td>+0.4%</td>
</tr>
<tr>
<td>rm-local</td>
<td>-8.5%</td>
<td>-6.5%</td>
<td>-6.5%</td>
<td>-8.5%</td>
<td>-6.3%</td>
<td>-5.7%</td>
<td>-6.1%</td>
</tr>
<tr>
<td>unzip-ramdisk</td>
<td>+1.2%</td>
<td>-3.1%</td>
<td>-2.0%</td>
<td>-12.0%</td>
<td>-0.2%</td>
<td>-0.3%</td>
<td>-1.6%</td>
</tr>
<tr>
<td>rm-ramdisk</td>
<td>-6.2%</td>
<td>-9.5%</td>
<td>-9.5%</td>
<td>-6.2%</td>
<td>-8.1%</td>
<td>-8.4%</td>
<td>-9.1%</td>
</tr>
</tbody>
</table>

Table 19: Result description of Figure 7 (the unordered-1-redundant program).