Application Buffer-Cache Management for Performance: Running the World's Largest MRTG

David Plonka, Archit Gupta, and Dale Carder – University of Wisconsin-Madison

ABSTRACT

An operating system's readahead and buffer-cache behaviors can significantly impact application performance; most often these better performance, but occasionally they worsen it. To avoid unintended I/O latencies, many database systems sidestep these OS features by minimizing or eliminating application file I/O. However, network traffic measurement applications are commonly built instead atop a high-performance file-based database: the Round Robin Database (RRD) Tool. While RRD is successful, experience has led the network operations community to believe that its scalability is limited to tens of thousands of, or perhaps one hundred thousand, RRD files on a single system, keeping it from being used to measure the largest managed networks today. We identify the bottleneck responsible for that experience and present two approaches to overcome it.

In this paper, we provide a method and tools to expose the readahead and buffer-cache behaviors that are otherwise hidden from the user. We apply our method to a very large network traffic measurement system that experiences scalability problems and determine the performance bottleneck to be unnecessary disk reads, and page faults, due to the default readahead behavior. We develop both a simulation and an analytical model of the performance-limiting page fault rate for RRD file updates. We develop and evaluate two approaches that alleviate this problem: *application advice* to disable readahead and *application-level caching*. We demonstrate their effectiveness by configuring and operating the world's largest[1] Multi-Router Traffic Grapher (MRTG), with approximately 320,000 RRD files, and over half a million data points measured every five minutes. Conservatively, our techniques approximately triple the capacity of very large MRTG and other RRD-based measurement systems.

Introduction

Sometimes common case optimizations by the operating system can adversely affect an application's performance instead of improving it. For instance, in most OS, readahead intends to optimize sequential file access by reading file content into buffer-cache with the expectation that it will soon be referenced. In this paper, we identify and remedy a situation in which the performance of a popular time series database, the Round Robin Database (RRD) Tool, is adversely affected by the default OS readahead and caching behaviors.

We present an investigative method to discover the RRD system's performance bottleneck and an analysis of the bottleneck identified: the OS default file readahead and caching behavior. We describe two approaches to optimize system resource usage for maximum performance: (*i*) *application advice* to the OS to disable readahead and (*ii*) *application-level caching*. We validate our results by configuring and operating a Multi-Router Traffic Grapher (MRTG) system that performs over a half a million measurements every five minutes and records them into a set of 320,000 RRD files in near real-time. We identify the additional factors that limit further scalability of the RRD system after these improvements. We also discuss OS improvements to the readahead behavior that could generally avoid the application performance problem we observed.

We investigate the scalability issues in a real world scenario. During the deployment of new network equipment (routers and switches) over the past few years at our university, the number of manged devices grew significantly, nearly doubling each year. This required our network measurement system's capacity to scale similarly. Today, for approximately 60,000 measured network interfaces, about 160,000 RRD files need to be updated every five minutes. These record interface byte, packet, and error rates. As the number of measurement points grew with the network size, we found that an increasing number of measurements did not get recorded into the database within the required five minute interval (20 to 80 percent failures).

We are motivated to study the scalability issues of RRD for two main reasons: (*i*) we were confounded

[1]Based on the authors' experience in the MRTG and RRDTool user community

by our system's poor performance given that it is generously-sized with respect to processor, memory, and disk, and (*ii*) any performance gains achieved would benefit many, given the popularity of RRDTool. To the first point, our prior understanding of RRD file structure and access patterns led us to believe the amount of work should not overwhelm our system. RRD files are organized in such a way that a small number of blocks are accessed per update cycle. The set of blocks in a series of updates has a low entropy: that is, most updates touch the same set of blocks. Thus, we were of the opinion that the "working set" of blocks for the RRD files in our system could reside completely in the OS file buffer-cache. Unexpectedly, our system's CPU spent the majority of its time in an "I/O wait" state due to disk reads.

To study the state of the buffer-cache, we wrote a tool called fincore that exposes the cache "footprint" of a given set of files; it takes a snapshot of the set of file blocks or pages in the buffer cache. This helps us determine at any given time what pages were brought in memory by the OS and helps us discover the readahead effects. We are also able to study the average number of pages per file brought into the memory by an unmodified MRTG. This helps us determine the maximum number of RRD files the system could handle with fixed hardware resources. We wrote another tool called fadvise that can advise the operating system about the file access pattern using the posix_fadvise system call. This tool enables the user to forcibly evict any file's pages from the buffer-cache, providing a key function in controlled experiments.

Our work makes the following contributions:

- We provide two tools and a methodology to study buffering behavior. These enable a system administrator or analyst to study the buffer-cache of *any system* (provided it implements the requisite APIs) and draw conclusions about readahead behavior, cache eviction policies, and system capacity.
- We develop an analytical model and simulation that determine the number of RRD files that can be managed given fixed memory resources or to determine the memory required for managing a given number of RRD files.
- We present two optimizations to RRDTool and evaluate their performance and scalability. The first employs *application-level buffering or caching* to coalesce file updates. The second offers *application advice* to the operating system that RRD files are accessed randomly rather than sequentially, thus causing readahead to be disabled.

The remainder of this paper is organized thusly: We first provide background on MRTG and RRD, and introduce our network measurement system. Next, our investigation technique is described in the "Method and Tools" section. Then two complementary performance optimizations to RRDTool are presented in the and "Application-Offered Advice" sections. The subsequent "Analysis" section contains our analytical model and simulation details. Ultimately, in the "Scalability" section, we report the scalability of the optimization techniques by running what we suggest is the world's largest MRTG on a single server. Therein we also discuss the factors limiting the further scalability of RRDTool after these improvements. The "Related Work" and "Discussion and Future Work" sections follow and we close with our conclusions.

Overview of MRTG and RRD

The Multi-Router Traffic Grapher (MRTG) is a perl script that collects network measurements and stores them in RRD files. Figure 1 shows a simplified MRTG in pseudo-code form. MRTG performance is satisfactory as long as it can consistently complete one loop iteration, consisting of one "poll targets" and one "write targets" phase, in less than the update interval, typically five minutes (300 seconds.)

```
# read configuration file
# to learn targets
readConfiguration();
do {
   # POLL TARGETS:
   #  collect values via SNMP:
   readTargets();
   # WRITE TARGETS:
   #  update values in RRD files:
   foreach my $target (@targets) {
      RRDs::update(...);
   }
   sleep (...); # sleep balance
                # of 300 seconds
} while (1); # forever
```

Figure 1: The MRTG daemon in pseudo-code.

MRTG refers to the configurable metrics it collects as *Targets*. Each target consists of two objects collected via SNMP, typically one inbound and one outbound measurement for a given network interface, i.e., a router or switch port. Thus, the number of targets per network device is typically a function of its number of interfaces.

The paired objects are each referred to as a *Data Source* (DS) in a *Round Robin Database* (RRD). The RRD file name itself and the file's Data Sources define the database "columns." *Round Robin Archives* (RRAs), or tables of values observed at points in time, are the database "rows." Figure 2 shows a typical RRD file managed by MRTG.

RRD performance is influenced by the RRAs defined within a file. Each RRA has an associated consolidation function, such as AVERAGE or MAX, that operates on a set of one or more *Primary Data Points* (PDPs), i.e., data points collected at the measurement interval. Additional RRAs typically require additional work to be done periodically, such as on every half hour, two hours and one day. These aggregation times are defined as offsets from zero hours UTC. Thus all like-configured MRTG RRD files require aggregations to be done every half hour, more every two hours, and then the most aggregations at midnight.

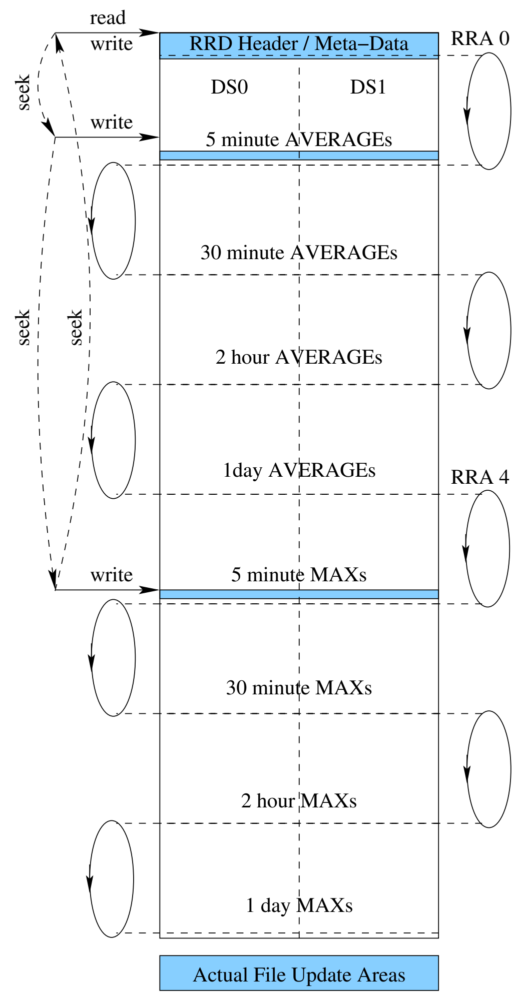

Figure 2: A typical MRTG RRD file and update operation. This RRD file stores two Data Sources (DSes) in eight Round Robin Archives (RRAs): four AVERAGE and four MAX RRAs. The 5 minute AVERAGE and 5 minute MAX RRAs are being updated.

Our MRTG System

Our MRTG system use RRDTool and currently measures approximately 3,000 network devices. Primarily, the devices are switches and routers in our campus network including those in the core and distribution layers and most of the network equipment at the access layer, serving users in approximately 200 campus buildings.

In this work, we refer to this production MRTG network measurement system as the System Under Test (SUT.) The SUT's characteristics including its software are summarized in Table 1.[2] The system's page size is 4KB and our file-systems are configured with a 4KB block size. Thus, we will conveniently use the terms "block" and "page" interchangeably when referring to segments of a file whether they are on disk or in memory.

Component	Characteristics
Processors	8 × Intel Xeon @ 2.7 GHz
Processor Cache	2 MB
Memory	16 GB
Disk	SAN: RAID-10, 16 × 2 disks
Operating System	Linux 2.6.9
File System	ext3 and ext2, 4KB blocksize
I/O Scheduler	Deadline
Software	**Version**
MRTG	mrtg-2.10.5
RRDTool	rrdtool-1.0.49

Table 1: Characteristics of the System Under Test and its software.

As our centrally-manged network has grown, our MRTG system has grown in terms of computing power and storage. One significant technique we employ to improve MRTG's scalability is to divide the targets amongst a configurable number of MRTG daemons that we increase as our number of targets increases; we process about 10,000 targets per daemon. So, our one MRTG "instance" is actually a collection of MRTG daemons running on one server. Another dimension in which our MRTG system is larger than most is that we resize RRA 0 (the five minute averages) to store up to one year or five years of data. This increases an MRTG RRD file's size from the typical 103 KB to 1.7 MB or 8.2 MB, respectively, of course requiring much more disk space. (We see that this does not adversely affect performance in the "Analysis" section.)

Prior to this work, our network growth exceeded the scalability of the SUT. The Appendix lists system and MRTG configuration recommendations that we've tested and used in our system to meet our performance goals.

Method and Tools

Examining System Activity

We started by examining the SUT's activity to determine the nature and extent of the performance problem. We present three measurements that led us to the root cause of the problem and that allow us to evaluate potential solutions.

First, we measured the time to completion of each measurement interval by each MRTG daemon on

[2]MRTG 2.10.5 patched as follows: Modified fork code to use select as in mrtg-2.10.6. Added a `--debug=time` option to report poll targets and write targets times. Removed test for legacy ".log" files (log2rrd), and threshcheck.

our system, with approximately 160,000 targets in total. As shown in Figure 1, this consists of two phases, first polling the network statistics via SNMP and then updating the pertinent RRD files. Figure 3 is a scatter plot with the measurement's time of day on

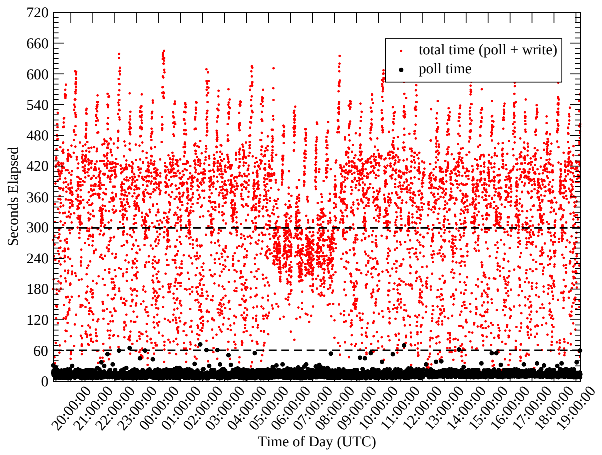

Figure 3: Original MRTG performance on the SUT with 160,000 targets processed by 28 MRTG daemons. The total time for poll and write targets phases often exceeds the five minute performance goal.

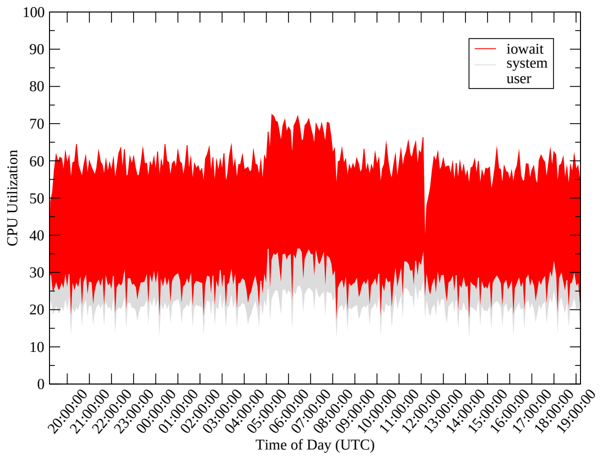

Figure 4: Original CPU utilization on the SUT with 160,000 targets processed by 28 MRTG daemons. CPU I/O wait state is excessive due to page faults for RRD file content.

the horizontal axis and the seconds elapsed on the vertical axis. The "poll time" dots show the time taken (in seconds) to poll the network and the "total time" dots signify the time taken by the poll and the subsequent write phase in one loop iteration by an MRTG daemon. Note that all the network polling finishes well below 60 seconds (marked by a horizontal line.) The writing phases very often do not finish within the period of 300 seconds (our five minute performance goal, also marked with a horizontal line) for the daemons; some even take 10 minutes to complete. This is clearly unacceptable performance because it delays the measurements for the subsequent poll phase in the single-threaded MRTG daemon, resulting in missing measurements.

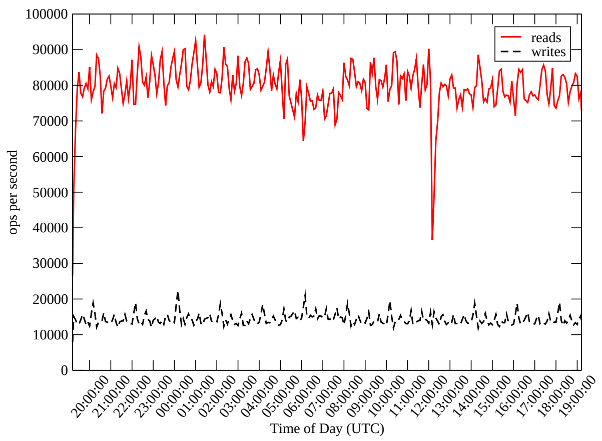

Figure 5: Original Disk utilization on the SUT with 160,000 targets processed by 28 MRTG daemons. Block read operations unexpectedly exceed write operations on RRD file updates.

Further examination reveals that the CPU was in I/O wait state for a majority of the time. Figure 4 shows the CPU utilization. The user and system CPU utilization levels are ~20% and ~10%, respectively, and are not the bottleneck. However, the CPU is spending more than half its time in the I/O wait state. Thus, the CPU wastes most of the time waiting for I/O to complete.

To understand the I/O wait, we studied the actual number of reads and writes involving the disk. Unexpectedly, the system was doing close to 90,000 reads per second (see Figure 5.) In contrast, the number of writes was stable at ~12,000 writes per second. The high number of reads suggests that files are not being cached effectively. This led us to examine the contents of the buffer-cache to determine why.

Examining Buffer-Cache Content

The system's buffer-cache content gives a good indication of which files' accesses are benefiting from caching in core memory. Unfortunately, a system's buffer-cache content is generally hidden from users and user processes. Prior work has resorted to timing file block accesses to surmise whether or not a given page already resides in the buffer-cache memory [4]. While suitable in some situations, this technique is indirect and has the unwanted side-effect of modifying the cache because it references the pages about which it inquires, causing them to be brought into cache and likely evicting other pages. Thus, a user tool to passively investigate buffer-cache content is desired.

We introduce a new user command called fincore that is used to determine which segments of a *file* are *in core* memory, presumably because they reside in the buffer-cache. The fincore command takes file names as arguments and displays information about file blocks or pages in memory. The fincore command uses two common system calls to accomplish this: mmap and mincore. That is, it first maps a file into its process' address space, then asks which pages of that segment of the process' address space are in core at that time.[3] Using fincore we were able to uncover the readahead effects on the buffer-cache. As an optimization, Linux reads pages ahead from the disk into the buffer, anticipating locality of subsequent reads. This improves performance for most applications by decreasing subsequent read latencies. With the current implementation of RRDTool, the readahead can have a highly adverse impact on performance and scalability. A brief discussion of the readahead algorithm within the context of the RRD file shown in Figure 2 can make this clearer.

The readahead algorithm tries to guess whether the file being accessed is going to be read sequentially (when readahead is actually useful) or randomly. On an RRD file update, the first read is for the meta-data at file offset zero, i.e., the beginning of the file. The maximum readahead window size is 32 blocks for ext2 and ext3, and, on an initial read, the readahead window starts at half that maximum in anticipation of sequential access. So, 16 pages are read into buffer-cache when the application read just one. The file offset of the second block needed to update the AVERAGE RRA depends on the current update position within the RRA.

For the typical MRTG RRD file, this will lie within the first 16 pages. The file offset of the third block needed for the MAX RRA update sometimes lies beyond the first 16 pages which can lead to further 8 pages being read in to the memory. (Eight pages are read as the readahead algorithm reduces the readahead window at the random seek into the MAX RRA.) These file block accesses are depicted in Figure 6.

A typical RRD file update consists of two RRA updates (AVERAGE and MAX). With the default readahead, most blocks that are read in are unnecessary. In the event of data kept over a longer period of time, as in our case with five minute averages for one year or five years, the write for the AVERAGE RRA often lays well beyond the first 16 pages. The readahead window is reduced to 8 pages for the next random read for the AVERAGE RRA update and then to 4 for the subsequent random read for the MAX RRA update. The readahead algorithm starts to adapt to the random reads by reducing the readahead window. A typical RRD file update requires just three block updates, yet we end up bringing 28 (16+8+4) blocks into the file cache. The file is then is closed which causes the adapted readahead value to be lost, reverting to 16 the next time the file is opened. For the typical RRD file with 800 recorded values, we end up bringing almost the full file into cache. If we could bring just the required "hot" blocks into the file cache by suppressing readahead from the beginning, we would get better performance and scalability.

[3]fincore is not entirely passive; it likely affects the cache slightly because it opens the file and thus causes an access to its inode block.

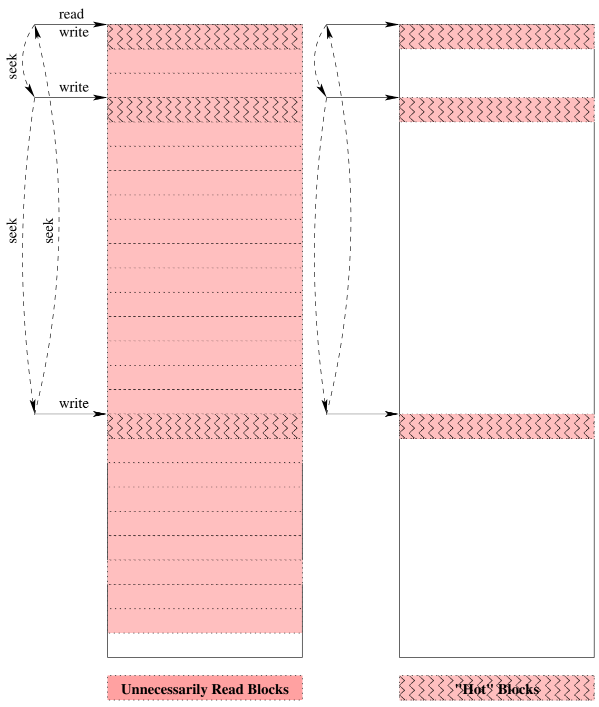

Figure 6: A sample RRD file update operation and the blocks involved. Readahead causes many blocks to be read unnecessarily, rather than just the "hot" blocks.

We see that the number of blocks most often required per MRTG RRD file per update is three. Based on our fincore observations (in the "Analysis" section), we note that there is also a low "churn" rate of these blocks. That is, once fetched into memory, these blocks were useful for a long period of time. If we were caching only the required blocks, there would be no waiting on reads, eliminating our performance problem.

Due to the default readahead behavior, we must wait for reads from the disk since we find almost nothing useful in cache. This is because, for instance for the file cache to accommodate 300,000 RRD files, 300,000 × 24 × 4 KB (close to 30 GB of memory for the cache) are required. Since the file cache isn't that large, the page replacement policy evicts the pages that will be needed later. However, with suppressed readahead, we would cache just three blocks per file: 300,000 × 3 × 4 KB (3.6 GB) and an order of magnitude less memory is required to fit everything desired in file cache. Actual buffer-cache behavior for RRD files is not quite as simple as this example; see the "Analysis" section for details.

Evicting Buffer-Cache Content

For repeatable experiments involving the buffer-cache, we require fine-grained control over the buffer-cache content. For instance, one run of an experiment such as an RRDTool update, brings pages of the RRD file from the disk into the buffer-cache. If we wish to see the effects of a subsequent update (reading the pages again from the disk), we need to evict the pages that were brought in earlier. Generally, the only methods available to forcibly evict pages from the buffer-cache were to either (*i*) unmount the file-system containing the cached files or (*ii*) populate the cache with hotter pages by accessing other content more frequently or more recently, thus invoking the systems page replacement algorithm to evict the unwanted pages. To perform controlled experiments we wanted a more convenient method for a user to forcibly evict specific files' blocks from the buffer-cache. To do so, we introduce a new user command called fadvise that is used to provide file advisory information to the operating system. The fadvise command takes file names as arguments.

Our typical use of fadvise is to advise the system that we ''don't need'' a file's blocks and that we'd like them to be evicted from the buffer-cache.[4] In this case, the file is first synchronized so that its dirty pages are transferred to the storage device, e.g., the disk, and then the advice is issued. The fadvise command uses the fsync then fadvise system calls to accomplish this.

Application-Level Buffering

We've described our system and showed that it does not meet our performance goal. A number of RRDTool users have proposed significant modifications to RRD measurement systems to improve performance by modifying I/O behavior [22, 12, 9]. These proposals generally involve intercepting RRD file updates and recording them to be written later. The updates are thus deferred, then later coalesced and written. The result is improved performance by the introduction of an independent thread to perform application writes and by the better locality characteristics of these periodic, coalesced writes.

[4]The Linux 2.6.9 source code and experimentation show that fadvise DONTNEED immediately evicts non-dirty pages from the buffer-cache. Other implementations might not evict the pages immediately.

Since this essentially implements a buffer-cache within the application, we call this technique *application-level buffering.*

Technique

We now describe our application-level buffering implementation called RRDCache, shown in Figure 7. RRDCache has three main components:

1. The RRDCache Library: a perl module that handles an application's calls to RRDTool's library. Specifically, it is used in place of RRDs perl module and provides the same functions, i.e., update, graph, etc.
2. The RRDCache Journal Buffer: a tmpfs file-system [24] to which updates are temporarily stored sequentially. (This is reminiscent of a journal in a journaling file-system.) We selected a memory-based file-system because it reserves a portion of memory exclusively for RRD and completely eliminates disk I/O during the RRD update operation.[5]
3. The RRDCachewriter: a script scheduled hourly using cron that periodically organizes updates and applies them to the RRD files on disk. In

[5]The RRDCache journal buffer need not be a memory-based file-system; it could be a disk-based file-system and still yield improved performance due to the better locality characteristics of appended writes to files.

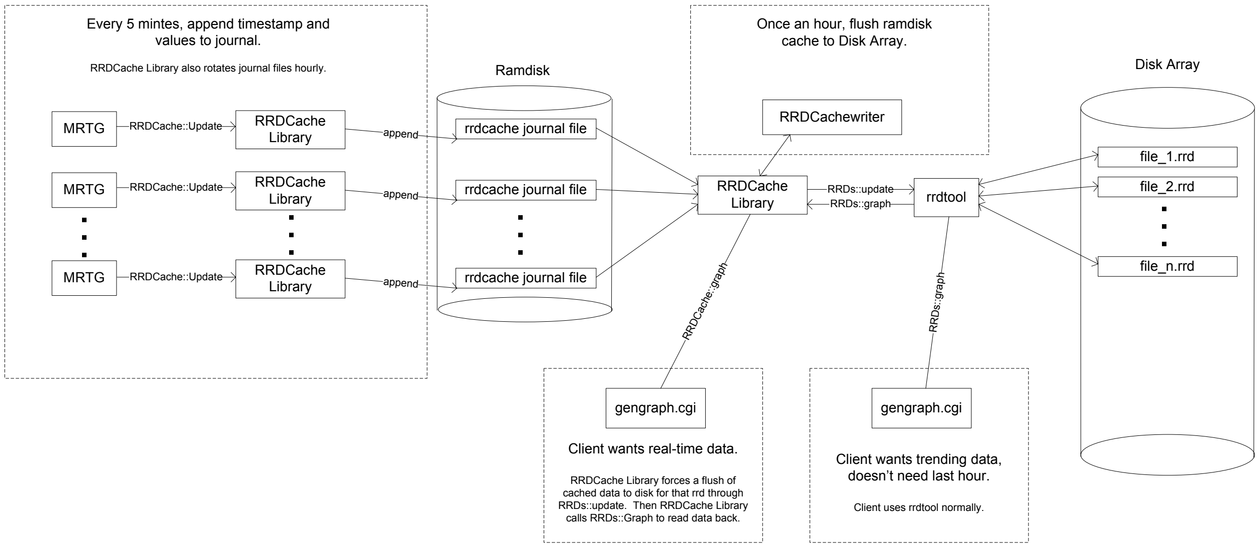
Every 5 mintes, append timestamp and values to journal.
RRDCache Library also rotates journal files hourly.
MRTG
RRDCache::Update
RRDCache Library
append
Ramdisk
rrdcache journal file
Once an hour, flush ramdisk cache to Disk Array.
RRDCachewriter
RRDCache Library
RRDs::update
RRDs::graph
rrdtool
Disk Array
file_1.rrd
file_2.rrd
file_n.rrd
RRDCache::graph
RRDs::graph
gengraph.cgi
Client wants real-time data.
RRDCache Library forces a flush of cached data to disk for that rrd through RRDs::update. Then RRDCache Library calls RRDs::Graph to read data back.
gengraph.cgi
Client wants trending data, doesn't need last hour.
Client uses rrdtool normally.

Figure 7: An Overview of RRDCache.

this way, RRDCachewriter performs both asynchronous writes and a sort of I/O scheduling on behalf of RRDCache and the application using it. This side-stepping of the operating system's default behavior, such as flushing dirty buffers to disk every five seconds, results in performance gains.

Normally an MRTG daemon (or any application that uses RRD files) accesses RRD files on disk directly by using the RRDTool API. With RRDCache, MRTG and other applications instead call functions in the RRDCache library. RRDCache presents the same API as the RRDTool. So, when an MRTG daemon calls the RRDCache:update function, the arguments are appended to a RRDCache journal file associated with calling process, i.e., the MRTG daemon.

The RRDCache journal file is located on the tmpfs file-system, eliminating disk I/O for the update. Periodically (once every hour), the RRDCachewriter runs to process any new data that has been written to the RRDCache journal files. The RRDCachewriter handles the updates from the journal files by committing the update to the appropriate RRD files on disk. In the process, it coalesces all the updates meant for a particular RRD file. If the requisite file pages are not already present in the buffer-cache, this has the benefit of bringing them into memory much less frequently. The RRDCachewriter can be run more often if the tmpfs runs out of space between runs. (We have been using 1 GB of our main memory for the tmpfs file-system and that has proven to be sufficient for the 160,000 targets polled at five minute intervals.)

Performance Impact

The performance of the measurement system using RRDCache is much improved. For our MRTG system, Figure 8 shows the results. We see that the MRTG daemons finish in well under 60 seconds which includes *both* the polling and writing. Contrast this with Figure 3, in which most of the updates were not achieving even the five minute performance goal.

Since the RRD file updates were performed by RRDcachewriter once an hour, and ordered by RRD file, there is a limited amount of I/O wait by the CPU at the start of every hour (Figure 8). This I/O wait is much less than the original system's I/O wait shown in Figure 4. The CPU utilization by the user and system processes remains the same as before.

Also evident in Figure 8 are spikes in disk read and write activity once per hour as the updates are being transfered from the journal buffer to the RRD files on disk. These disk I/O rates are much lower than the original system's rates shown in Figure 5.

One complication of RRDCache's technique is that the application-level journal buffer is not readable by RRD applications other than the RRDCachewriter; currently it is just a buffer for writing, not a cache for reading. While updates reside in the RRDCache journal buffer, they can't be directly accessed by applications that may wish to graph recent measurements, for instance. Thus RRDCache slightly changes near real-time access semantics for RRD files. To work around this, RRDCache provides a graph function that immediately flushes pending updates from the journal buffer into RRD files upon attempts to read them and then returns the result of the RRDs::graph function. Applications then have the option of accessing RRD files directly through the RRDs interface, thus reading perhaps only older data suitable for trend analysis. However, performance would degrade if applications were to read every RRD file once per update interval (e.g., five minutes) to retrieve the most recent measurements, reverting to approximately the poor performance originally observed. If ever this becomes a problem, RRDCache could be improved so that its journal buffer is a true buffer-cache, consistent amongst both reading and writing processes.[6]

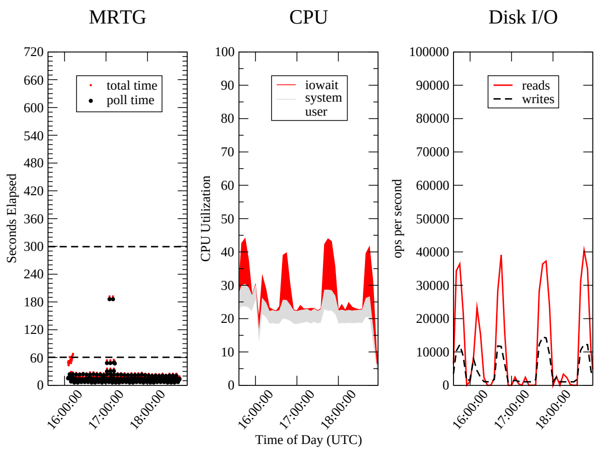

Figure 8: **RRDCache:** Performance on the SUT with 161,000 targets processed by 27 MRTG daemons. The five minute performance goal is easily met although CPU I/O wait and excessive block reads vs. writes are evident.

Application-Offered Advice

We find sufficient motivation to avoid the operating system default readahead and buffer-cache behaviors because of the latencies observed while updating RRD files. In the previous section, we've shown that modifications that drastically reduce RRDTool's file I/O can achieve better performance by *working around* the operating system's default behavior.

In this section, we instead improve performance by *directly influencing* the operating system behavior. Specifically, we identify a mechanism to cause just the desired RRD file blocks to be read and cached.

Technique

One technique to suppress readahead is to use the posix_fadvise system call. This allows applications

[6]Maintaining the journal buffer as a collection of very small RRD files would enable it to be read conveniently.

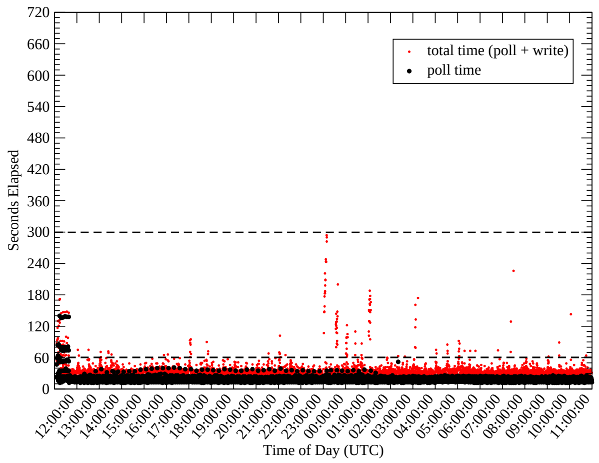

Figure 9: **RRD with fadvise:** MRTG performance on the SUT with 162,000 targets processed by 19 MRTG daemons. The five minute performance goal is clearly met.

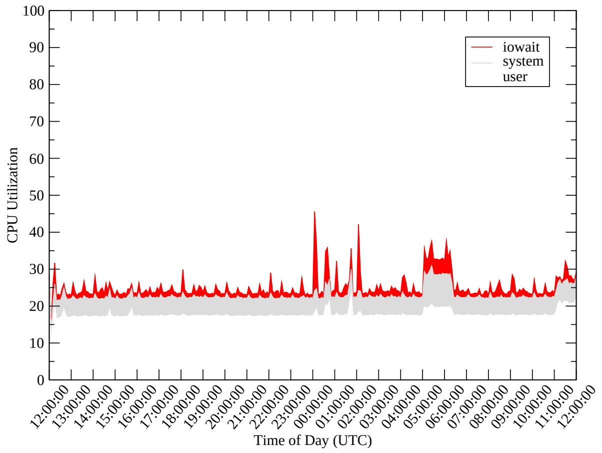

Figure 10: **RRD with fadvise:** CPU utilization on the SUT with 162,000 targets processed by 19 MRTG daemons. CPU I/O waits are minimal.

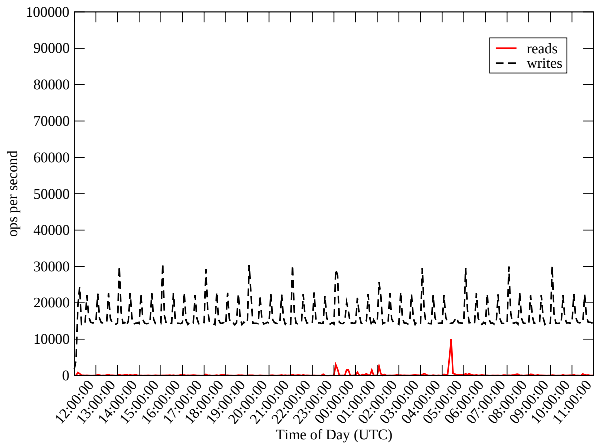

Figure 11: **RRD with fadvise:** Disk utilization on the SUT with 162,000 targets processed by 19 MRTG daemons. I/O Read operations are reduced to nearly zero due to efficient use of the buffer-cache.

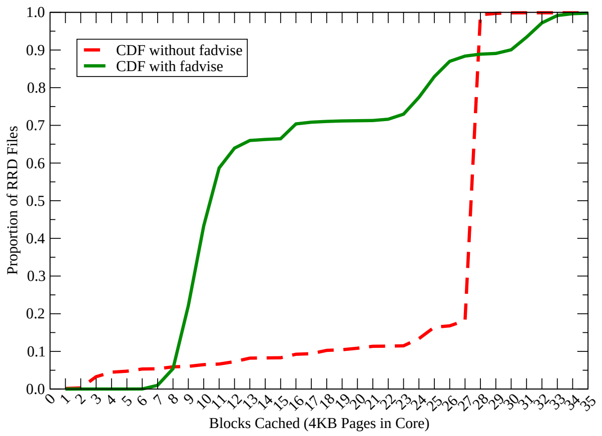

Figure 12: **fincore**: Comparison of proportion of RRD files by number of blocks cached without file access advice (before) and with advice (after). With fadvise RANDOM, unnecessary RRD file blocks no longer occupy the buffer-cache.

to advise the operating system of their future access patterns of files. The application identifies regions of an open file (by offset and length) and offers hints as to whether it will access them sequentially (the default) or randomly. Additionally, the application can inform the OS whether or not it expects to access those file regions again in the near future. Thus, for RRD files, we are able to advise the OS that the file accesses will be “random.” This turns off readahead. The result is that only the “hot” blocks shown in Figure 6 are read and cached. For Linux 2.6.9 with fadvise RANDOM, on a typical five minute update only three blocks are cached.

Performance Impact

The benefit of disabling readahead is realized immediately. Figure 9 shows the time elapsed per loop iteration of each MRTG daemon. With a system of 162,000 RRD files serviced by 19 MRTG daemons, we see that they finish within 60 seconds including both the polling and writing phases. This is contrasted with Figure 3 where we see that most of the updates do not meet our five minute performance goal. This also suggests that we can monitor even more targets within a five minute interval. Note that the increase in the time to update at zero hours UTC is due to aggregation (of the AVERAGE and MAX values) that is synchronized across all RRD files, and a potential performance limitation that we also discuss. The performance potential and limitations are further explored in the “Scalability” section.

As expected, the CPU is largely freed from waiting on I/O to complete (Figure 10). Again comparing with Figure 4, we note that the CPU utilization by the user level processes and the system remains the same as before. The significant difference is the low amount of waiting on I/O by the system when the readahead is suppressed. This implies a lowered number of reads with the buffer-cache becoming much more effective in caching the needed blocks. This is validated by our measurement of the reads issued to the disk per second by the system (Figure 11). The performance gain is significant, reducing approximately 90,000 reads per second to about 100 reads per second.

Figure 12 shows the plot of the proportion of RRD files vs. the number of blocks cached for both the original system without fadvise and the modified one with fadvise. One can see the sharp decrease in the number of blocks cached by the modified system with fadvise compared to the original system. For the original system, a sharp inflection point occurs at 27 pages. This indicates for a majority of the RRD files 27 pages were required. Also, the original system required more pages per file but couldn't fit them in the buffer-cache. For the modified system with fadvise, the inflection point occurs at 8, where the system requires 8 pages for most of the RRD files. Since the buffer-cache has space available for more pages, some of the RRD files get to keep more than 8 pages in the buffer-cache.

Analysis

To better understand the buffer-cache behavior when updating a very large number of RRD files in near real-time, we developed both an analytical model and a simulation. Analytical modeling improves our understanding of RRDTool's file update behavior so we have a solid foundation on which to propose general solutions. The resulting model also provides a convenient way to calculate expected page fault rates without experimentation and measurement. The simulation allows us to gather a broader set of results than either the model or real-world experiments that would prohibitively require repeated reconfiguration of a real system's RRD files or physical memory.

Analytical Model

We present an analytical model to predict the page fault rate given an RRD file configuration and an estimated number of pages available for the each RRD file in the system's buffer-cache memory. The modeling is done for a system with RRDTool patched to do fadvise RANDOM.

For this analytical model, we first need a list of the unique Primary Data Point (PDP) counts in increasing order, over which the consolidation is performed for every RRA (for AVERAGE, MAX, and so on). Recall that each RRA consolidates some number of PDPs that were gathered at the measurement interval, so an RRA's PDP count determines how often it is updated. For instance, for the RRD file shown in Figure 2, the ordered list of values of PDPs over all RRAs is: {1, 6, 24, 288}. These represent the periods of consolidation which in the case of Figure 2 refers to 5 minutes (1), 30 minutes (30/5 = 6), 2 hours (120/5 = 24) and 1 day (288). We also need a corresponding list of number of RRAs that are configured to consolidate each of those numbers of PDPs. That is for 2, since both MAX and AVERAGE RRAs are kept for each of the PDP value, the associated count list for {1, 6, 24, 288} is {2, 2, 2, 2}.

We denote the ordered PDP list as $\{x_1, x_2, \ldots, x_n\}$ and the associated RRA count list as $\{c_1, c_2, \ldots, c_n\}$. The cardinality of the ordered list is n. Hence in the example, $\{x_1, x_2, x_3, x_4\} \equiv \{1, 6, 24, 288\}$ and $\{c_1, c_2, c_3, c_4\} \equiv \{2, 2, 2, 2\}$. Here, $n = 4$.

Now, for one update of an RRA, let B be the number of bytes written into the RRD file. In our case $B = 16$ bytes, for the two 8-byte floating-point values. For simplicity, we assume that each RRA is block aligned. This does not sacrifice generality since the average number of page faults due to crossing page boundaries is still predicted accurately as long as an RRA is at least one block in size, which is typically the case. The block size is S bytes. $S = 4096$ bytes in our case.

The number of updates that fit in a block is $u = S/B$. That is, after every S/B updates a page fault will occur. $u = 256$ in our case.

Let T be the time after which the primary data point is updated. $T = 5$ minutes in our case.

Now we estimate, for a single RRD file, the rate at which page faults occur given maximum p pages are available in the buffer-cache for use for this file (excluding the inode and indirect blocks.) When more than p pages are needed, LRU is used to evict pages to make place for newer ones.

The time estimated to a page fault, t, is the following:

$$t = minimum(uT, x_sT) \tag{1}$$

where $\exists\, s$ such that

$$p - 1 \geq \sum_{i=1}^{s} c_i \tag{2}$$

$$p - 1 < \sum_{i=1}^{s+1} c_i \tag{3}$$

minimum() returns the lower of the two values.

The rationale behind x_sT: For each RRA, a page is needed in memory. So we calculate the count of RRAs which can be fit in $p - 1$ pages. (Only $p - 1$ pages are available to the RRAs because one page is required for the first block of the RRD file that is always read as it contains the RRD metadata.) The subscript s is used to index into the ordered list to determine the interval after which the fault will occur. Each index into the ordered list is a discrete point in time when a fault will occur. For instance, if $s = 2$, then x_2 implies fault would occur every half hour. (2) and (3) give the index s based on the count of RRAs that can fit in $p - 1$ pages.

The rationale behind uT: A page fault will surely occur after u updates.

The number of pages that will see a fault, m, after time t:

$$m = \delta \sum_{i=1}^{n} \frac{1}{x_i} \tag{4}$$

where:

$$\delta = p - \sum_{i=1}^{s} c_i \tag{5}$$

Rationale behind δ: The number of pages that need to be brought into memory is the count of the RRAs at a particular index s, which cannot be held by p pages.

Rationale behind $\sum_{i=1}^{n} \frac{1}{x_i}$: In our case, the update rate for x_1 is six times the update rate of x_2. Therefore, the number of faults for $x_1 = \frac{1}{6} x_2$. We sum for all the fault rates through x_n relative to x_1 and hence the $\frac{1}{x_i}$ factor.

Rate of page fault, r, is given by m/t:

$$r = \frac{m}{t} \tag{6}$$

$$r = \frac{\delta \sum_{i=1}^{n} \frac{1}{x_i}}{minimum(uT, x_s T)} \tag{7}$$

This model essentially predicts the average values shown with no readahead in Figure 13, as verified by simulation. Thus, the model provides a quick way to calculate either the expected page fault rate given a buffer-cache memory constraint, or vice-versa. In addition to this practical result, the analytical model led us to the following insights:

- The total RRD file size is practically irrelevant. This is ideal since it frees us to extend our data retention arbitrarily, bounded only by disk space. For instance, recall that in our system we regularly resize our RRD files to store five minute averages for up to five years. (MRTG typically stores them for less than three days.) The resulting 17× increase in file size only nominally affects the page fault rates.
- The total number of RRAs with a given aggregation level is important. For instance, removing the five minute MAX RRA,[7] which duplicates the values in the five minute AVERAGE RRA, results in a significantly lower page fault rate when buffer-cache memory is scarce.

Simulation

In addition to deriving the analytical model, we developed a page fault simulation. This simulation provides a means by which to validate the average page fault rate predicted by the analytical model. It also exposes the distribution, variance, and peak page fault rates by time of day.

[7]We suppose that the MRTG five minute MAX RRA exists for historical and convenience reasons. While it allows one to graph just the MAX values and see the entire time range, users typically put both the AVERAGE and MAX values on the same graph.

First we simulated an entire lifetime of an RRD file's updates using RRDTool itself, but with synthetic input data. Before each update, we used our `fadvise` command's "don't need" technique to evict the file's cached (hot) pages. After each update we used our `fincore` command's technique to determine the hot pages and recorded the page numbers to a log.

Secondly, we wrote a buffer-cache simulator with a Least-Recently-Used (LRU) page replacement policy and replayed the page operation log recorded earlier to determine the page faults with varying numbers of buffer-cache pages being available per RRD file. While our SUT's buffer-cache is actually managed using the Linux 2.6 page replacement algorithm [8], similar to 2Q [10], we make the simplifying assumption that LRU is suitably similar for the purpose of simulation. In addition to the RRD file data blocks, we also simulated access to the file's inode and indirect blocks. On ext2 and ext3 file systems, a typical MRTG RRD file incurs an indirect lookup (and therefore an indirect block must occupy space in the buffer-cache) for each data block above the twelfth since only the first twelve blocks are directly referenced in the inode.

From the resulting simulated behavior, we can determine the expected page faults for a single RRD file over time. We then extrapolate by multiplying by the target number of RRD files to determine what amount of buffer-cache (as limited by physical memory) reduces page fault disk reads to an acceptable level.

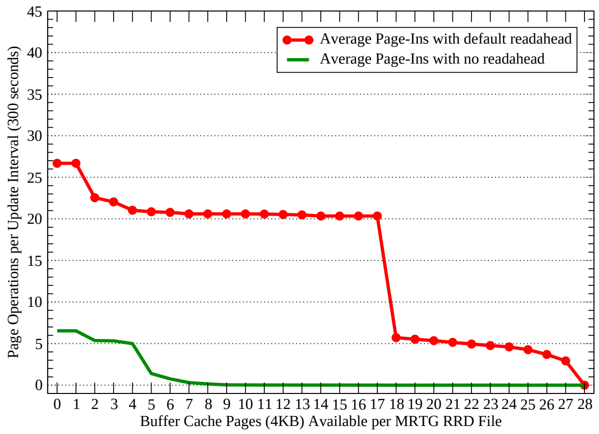

Figure 13: **Simulation with and without `fadvise`:** Page-ins as a function of the number of buffer-cache pages available per MRTG RRD file. Significantly reduced paging rates result following the dramatic drops.

We run the simulation for both the original RRDTool (default readahead). and the one patched with `fadvise RANDOM` (no readahead). Figure 13 shows the average number of page-in operations for both versions of RRDTool as a function of the number of buffer-cache pages available per MRTG RRD file. For the original RRDTool, when the number of pages available in the buffer-cache is less than 18, the number of page faults is very high. It falls at 18 because the 16 pages required for initial readahead are available in the buffer-cache at this point. (It is 16+2 because two extra blocks are required for the file inode and indirect blocks.)

Also, sometimes, the AVERAGE and MAX RRAs get written within these 16 pages. For the patched version with fadvise RANDOM, the average number of page-ins is close to zero if more than 7 pages are available in the buffer-cache. Page faults still occur when 8 pages are available in the buffer-cache but the average page-in rate is extremely low as shown in Figure 14. Note that more pages are written out at the aggregation intervals of 30 minutes, 2 hours and one day. The time of day 00:00 UTC shows the peak paging activity, when aggregation happens for daily RRAs.

Our simulation results are validated by the earlier observations of the real system's performance with fadvise RANDOM. Specifically, the read and write pattern of the simulation in Figure 14 agrees with the observations of the real system with fadvise RANDOM in Figure 11, i.e., near zero read (page-in fault) rate. Earlier, using fincore in the real system, we also observed that most RRD files had only 8 pages in cache (Figure 12); this is the value that simulation shows (Figure 13) is the minimum required to achieve an average page fault rate of nearly zero.

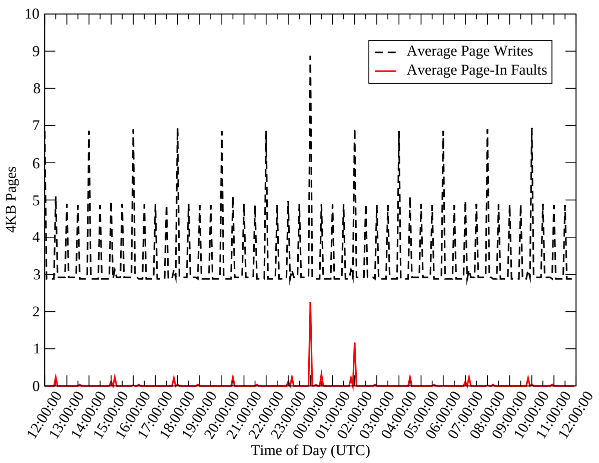

Figure 14: **Simulation with fadvise:** Average page-out and page-in operations by time of day given eight buffer-cache pages available per MRTG RRD file. As few as eight pages per file reduces page-ins to near zero.

Scalability

We have shown that using either *application-level buffering* or *application-offered advice* dramatically improves the performance of RRD systems. In this section we show the performance and capacity scalability characteristics of a very large RRD and MRTG-based network measurement system by testing it first with just *application advice*, and secondly with *advice plus application-level buffering*. Thus, we explore the performance of the simpler of the two techniques and also the two techniques combined to determine the upper-bound to the scalability of such a system.

Our production MRTG system today monitors approximately 3,000 network devices with approximately 160,000 MRTG targets. Recall that each target is typically a pair of measurements such as byte, packet and error rates, inbound and outbound. Thus, the production system measures and records approximately 320,000 data points every five minutes. Having already found that either improvement technique results in satisfactory performance and thus does not push our system to its limit, we now construct an even larger system to study scalability.

We create an MRTG system three times the size by replicating our existing production measurement system so that there are three like-configured MRTG instances all running on one server. Our experimental procedure is to begin with approximately 160,000 (i.e., the production MRTG system) and then progressively add 20,000 targets every twenty minutes (10,000 from each of the replicated instances), until all three systems are running in parallel for a total exceeding 480,000 targets.

MRTG with fadvise RANDOM

We first tested the scalability of MRTG with RRDTool patched to do fadvise RANDOM. The performance results are shown in Figures 15 and 16. This is the system we claim as the world's largest MRTG, operating with acceptable performance at around 320,000 RRD files. While there are some outlier points in the upper left of Figure 15, they occur at twenty minute intervals and there are exactly two per interval. Thus, these outliers are an artifact of the experimental procedure showing latency during just the very first loop iteration of each of the two new MRTG daemons as their the set of hot pages for their RRD files are read into buffer-cache. Beyond about 320,000 targets in the scalability test, performance is unacceptable because page faults increased and CPU utilization continually exceeded 65%, leaving little room for other tasks.

In Figure 16 note the spikes in CPU I/O wait state just following 1600 and 1800 hours. These are due to aggregations that occur every two hours in typical MRTG RRD files. Furthermore, note that as the number of targets increases, similar spikes are seen at half hour intervals following 1800 hours. These spikes indicate that the number of hot pages exceeds the capacity of the buffer-cache on the SUT, resulting in an excessive page fault rate. We estimate our buffer-cache requirement to be 8 pages per file and 480,000 × 8 × 4 KB = 14.6 GB. Although the SUT has 16 GB of memory in total, often only 10 GB is available for buffer-cache. Ultimately, the high CPU utilization interfered with SNMP polling (as evidenced by a drop in network traffic) so the test was stopped.

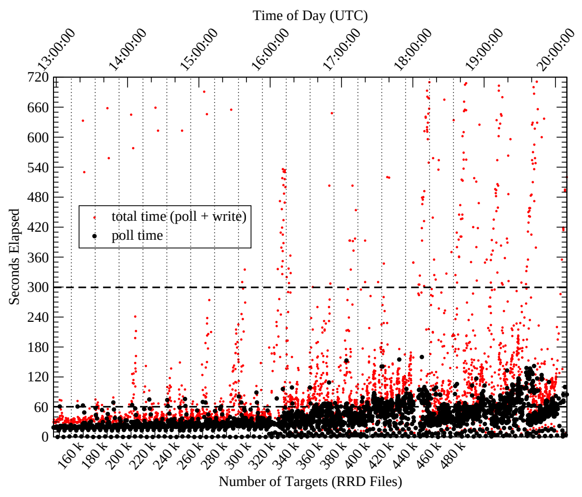

Figure 15: **Scaling with fadvise**: MRTG performance on the SUT progressively increasing to 483,000 targets and 53 MRTG daemons.

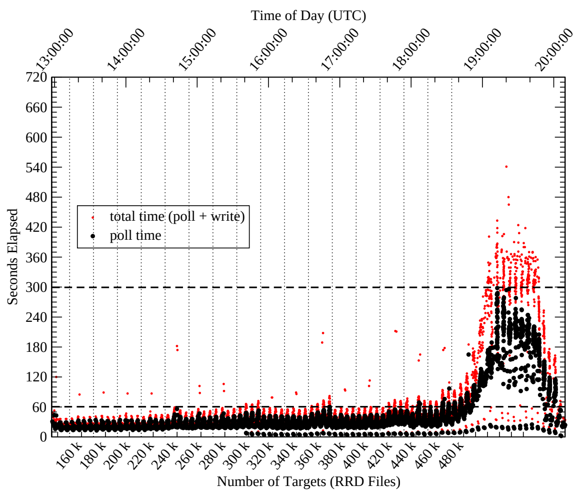

Figure 17: **Scaling RRDCache with fadvise**: MRTG performance on the SUT progressively increasing to 486,000 targets and 52 MRTG daemons.

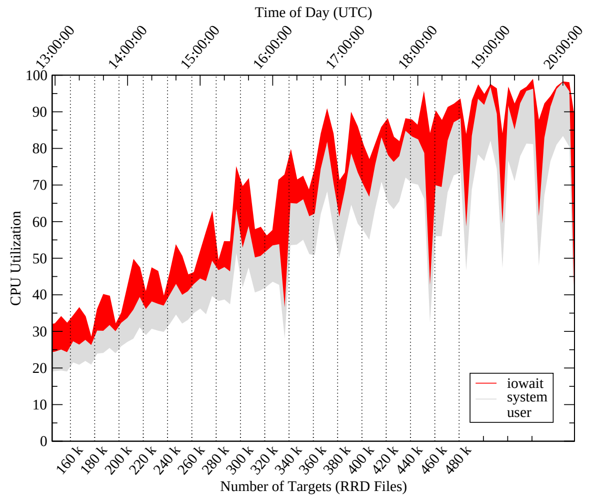

Figure 16: **Scaling with fadvise**: CPU utilization on the SUT while progressively increasing to 483,000 targets and 53 MRTG daemons.

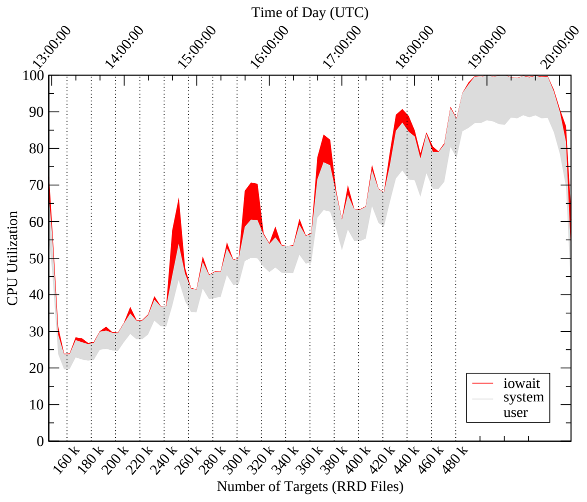

Figure 18: **Scaling RRDCache with fadvise**: CPU utilization on the SUT while progressively increasing to 486,000 targets and 52 MRTG daemons.

MRTG with fadvise RANDOM and RRDCache

Subsequently, we tested the scalability of MRTG using RRDCache combined with RRDTool patched to fadvise RANDOM. The performance results are shown in Figures 17 and 18. These combined techniques yielded the highest capacity, exceeding 400,000, but CPU utilization reached 100% and the RRDCache-writer could not complete its hourly updates within an hour, so an increasing backlog developed from which it didn't recover and the test was stopped.

Limitations

These scalability tests on our SUT show at least two capacity or performance limitations of large RRD and MRTG systems:

1. When buffer-cache is scarce, page fault rates peak at RRD aggregation times that are predictable offsets from zero hours UTC. Synchronized aggregations, and thus consolidated data points with matching timestamps, are a convenience to RRDTool users when fetching or graphing the data. However, when updating RRD files in near real-time, there is clear performance consequence to synchronized aggregation because work is not distributed evenly across time.
2. CPU utilization approached or reached 100% when updating around 480,000 RRD files. This consists primarily of user mode CPU that we attribute to the MRTG and RRDCachewriter perl scripts. Thus, the next performance bottleneck limiting the scalability of MRTG systems is likely CPU.

We've shown that MRTG and RRD systems can scale to hundreds of thousands of targets or RRD files.

Without changing the RRD file read semantics, our fadvise RANDOM method allows us to scale to 320,000 target and files with acceptable performance on our system with 16 GB of memory. With slightly changed read semantics (because of the deferred RRD file updates), the RRDCache method scales higher. The factors that limit further scalability are (*i*) the CPU required for target processing (in perl) and (*ii*) RRDTool's aggregations at synchronized times across all RRD files. Further gains can be achieved by profiling and optimizing the system software (e.g., MRTG) and by appropriately sizing the systems physical memory so that an even larger buffer-cache is available.

Related Work

Our work is informed by prior operating system and application performance improvement techniques.

Within the operating system, better buffer-cache management techniques can help reduce the number of disk reads and writes. There are a number of policies described in the literature (e.g., FIFO, LRU, LFU, Clock, Random, Segmented FIFO, 2Q [10], and LRU-K). The readahead by the OS can limit the amount of useful data that can be cached. In some circumstances, improvements to the adaptive readahead algorithm can significantly improve performance [14].

The ability to accept hints or advice from applications with the aim of more efficiently managing resources and improving performance was implemented in the Pilot operating system [20]. An interface by which operating systems can accept such advice specifically to provide buffer management has long-since been suggested [25]. Later work finds that application "hints" to inform the operating system of file access patterns improves performance [15]. Today, some operating systems have support for application advice via the fadvise and madvise APIs [19].

A related approach is to the change the kernel to include functionality which enables application guided buffer-cache control [2]. Another possibility is to simulate the cache replacement algorithm to build a reasonably accurate model of the contents of the cache for reordering reads and writes [4].

Within our version of the operating system (Linux 2.6), there are four I/O scheduling algorithms available: Completely Fair Queuing (CFQ) scheduling, deadline elevator, NOOP scheduler and Anticipatory elevator scheduling [21]. The scheduling algorithm can prioritize individual I/O requests, such as reads over writes, and therefore affects application performance when page faults occur.

A number of software systems inspired by the original MRTG have improved its performance in some ways. The current MRTG, Cricket [1], Cacti [5], and Torrus [23] applications use RRDTool [13] to achieve improved performance. Cricket [1] allows configuration of more parallel measurements per file, but this offers only a modest performance improvement since tens to hundreds of thousands of RRD files would still be required. RTG [3] made significant changes in the polling (but that is not our bottleneck) and replaced the file I/O with relational database I/O. We have no reason to believe this would offer better I/O performance, and it significantly changes the user interface to the data. JRobin [11] completely reimplements RRDTool in java, improving performance in some areas but decreasing it in others and modifying the RRD file format in the process.

Recently, RRD users have proposed design changes or made customizations to introduce an application-level cache maintained by a daemon that intercepts updates [22, 12, 9].

Discussion and Future Work

Our investigation and experimentation thus far suggests at least the following potential items of future work.

- **File Types:** UNIX-based operating systems lack file types; a file is simply a stream of bytes and this is often cited as an advantage or, at least, a successful simplification. However, this is one reason that the RRD file update access pattern is not handled well by the adaptive readahead algorithm. Perhaps the readahead and other behaviors, such as caching, could be influenced or determined at file open time based on a file's type, as defined by file name extension (e.g., ".rrd") or by magic, the file command's magic number file.
- **File Read Performance:** Although we have disabled readahead to achieve better update performance, we have not thoroughly investigated its effect on RRD fetch or graphing (read) performance. We surmise that advising for random I/O helps read performance too, but have not carefully measured it.

  Also, we selected the Linux deadline I/O scheduler because it prioritizes reads over writes, but evaluating this decision is left for future work. Linux' Completely Fair Queuing (CFQ) scheduler may perform acceptably as well; we have not compared them.
- **RRD Update Interval:** Some network operators desire more frequent measurements, such as a one minute interval rather than five. Future work might explore if RRD scales similarly in this situation. We believe our page fault rate model is valid for all update intervals somewhat greater than that of the system's page replacement algorithm. (Note that dirty pages are flushed at five second intervals by pdflush in Linux 2.6.)

  Our performance results suggest that the CPU load would be a limiting factor as the update interval decreases, i.e., if updates are more frequent. Perhaps simply choosing a one minute interval would constrain the capacity to about one fifth that of when using a five minute interval.

Judicious partitioning of the set of targets would help, e.g., using a one minute interval for measuring the core and/or distribution links and a five minute interval for more numerous access ports.

- **MRTG CPU Utilization:** As is to be expected in system performance work, we found that eliminating the I/O bottleneck exposed the next bottleneck, CPU, that limits scalability. It seems this high CPU utilization is primarily due to the MRTG perl script, thus profiling and optimizing it could improve performance.
- **"Gaming" the Readahead Algorithm:** While our platform provides the BSD and POSIX fadvise APIs, others do not or do not yet completely implement them. Can we instead "game" their readahead algorithms, for instance by performing otherwise unnecessary no-op seek operations, to likewise disable readahead? What is the performance cost of doing so? If adaptive readahead algorithms are similar, this might also have portability benefits. (We've observed that some operating systems use a readahead of zero initially,[8] so they exhibit the desired behavior for RRD files without needing advice or adaptation.)

  Alternatively, the adaptive readahead algorithm could be improved to respect a file's access *history*, i.e., by initially setting readahead based on a series of previous file access (open/close) "sessions" by the current process or prior processes.
- **RRD File Design:** We've seen how the RRD file organization influences its update performance. Is there a better organization for RRD files? For example, locality of data for updates would improve if RRAs with the same number of PDPs (but different consolidation functions) could be interleaved in the same block so that their corresponding data points are nearby when updated.

  Is there a way to avoid synchronized aggregations or consolidations across all RRD files? Perhaps we can introduce a stochastic component to skew those updates slightly in time. This is difficult to do without affecting RRD file read semantics and without introducing an independent thread to perform updates.

Conclusions

In conclusion, we've provided a general analysis method and two new tools, fincore (available at [18]) and fadvise (available at [16]), that expose readahead and buffer-cache behaviors in running systems. Without such tools, these performance-critical aspects of the operating system are hidden from system administrators and users.

By both modeling and simulation, we've provided a detailed analysis of the I/O characteristics of RRD file updates. We've shown how the locality of RRD file accesses can be leveraged, limiting page faults and disk I/O, resulting in improved performance and scalability for RRD systems. We've found that RRD buffer-cache utilization and page faults are defined by subtleties in the RRD file format and RRDTool's access pattern, rather than simply being defined by file size. This is advantageous because it means that larger RRD systems can be operated than would otherwise be thought.

We've outlined two effective methods to improve RRD performance. The first, RRDCache (available at [6]), is what we've called *application-level caching or buffering*. The second, for which we provide a patch to RRDTool (available at [17]), issues *application advice* to the operating system to select readahead and buffer-cache behavior appropriate for random RRD file I/O. While the two methods are starkly different, both eliminate the buffer-cache memory bottleneck that has been observed in large RRD network measurement systems. Conservatively, either technique *triples* the capacity of such systems. Together, these *complementary* techniques can be applied to maximize performance.

Finally, we've shown that system tuning and minor capacity-enhancing code changes improve Round Robin Database performance so that RRDTool can be used for even the largest managed networks.

[8] Apple's OS X with HFS+ file-system has an initial readahead of zero.

Acknowledgments

We thank Hideko Mills for her support, Michael Swift for his valuable input, and Robert Plankers and Kevin Kettner II for their assistance with system administration. Mark Plaksin provided helpful review feedback.

kSar [7] proved useful for conveniently visualizing sar data.

This work is a collaboration with the University of Wisconsin's Division of Information Technology.

Author Biographies

David Plonka is a graduate student and research assistant in Computer Sciences at the University of Wisconsin-Madison. He received a B.S. from Carroll College in Waukesha, Wisconsin in 1991. He can be reached at plonka@cs.wisc.edu .

Archit Gupta is a graduate student in Computer Sciences at the University of Wisconsin-Madison. His primary interest areas are Systems and Networking. He can be reached at archit@cs.wisc.edu .

Dale Carder is a senior network engineer for the University of Wisconsin-Madison and WiscNet. He can be reached at dwcarder@doit.wisc.edu .

Bibliography

[1] Allen, J. R., "Driving via the Rear-View Mirror: Managing a Network with Cricket," *Conference on Network Administration*, pp. 1-10, 1999.

[2] Arpaci-Dusseau, A. C., R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, and F. I. Popovici, "Transforming Policies into Mechanisms with Infokernel," *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October, 2003.

[3] Beverly, R., "RTG: A Scalable SNMP Statistics Architecture for Service Providers," *Proceedings of the 16th Conference on Systems Administration (LISA 2002)*, Philadelphia, PA, pp. 167-174, November 3-8, 2002.

[4] Burnett, N. C., J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Exploiting Gray-Box Knowledge of Buffer-Cache Contents," *The Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pp. 29-44, Monterey, CA, June, 2002.

[5] *Cacti*, http://www.cacti.net .

[6] Carder, D., *RRDCache*, http://net.doit.wisc.edu/˜dwcarder/rrdcache/ .

[7] Cherif, X., *kSar*, http://ksar.atomique.net .

[8] Gorman, M., *Understanding the Linux Virtual Memory Manager*, Prentice Hall, 2004.

[9] Hustace, D., *Queueing RRD*, http://www.opennms.org/index.php/Queueing_RRD .

[10] Johnson, T. and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases*, Santiago de Chile, Chile, pp. 439-450, September 12-15, 1994.

[11] Markovic, S. and A. Vandamme, *JRobin*, http://www.jrobin.org .

[12] Oetiker, T., *RRD Accelerator Design proposal*, http://oss.oetiker.ch/rrdtool-trac/wiki/RRDaccelerator .

[13] Oetiker, T., "MRTG: The Multi Router Traffic Grapher," *Proceedings of the 12th Conference on Systems Administration (LISA-98)*, Boston, MA, USA, pp. 141-148, December 6-11, 1998.

[14] Pai, R., B. Pulavarty, and M. Cao, "Linux 2.6 Performance Improvement Through Readahead Optimization," *Proceedings of the Linux Symposium*, 2004.

[15] Patterson, R. H., G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *SOSP*, pp. 79-95, 1995.

[16] Plonka, D., *The fadvise command*, http://net.doit.wisc.edu/˜plonka/fadvise/ .

[17] Plonka, D., *The fadvise random patch to RRDTool*, http://net.doit.wisc.edu/˜plonka/rrdtool_fadvise/.

[18] Plonka, D., *The fincore command*, http://net.doit.wisc.edu/˜plonka/fincore/.

[19] *fadvise*, http://www.opengroup.org/onlinepubs/009695399/functions/posix_fadvise.html.

[20] Redell, D. D., Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell, "Pilot: An Operating System for a Personal Computer," *Communications of the ACM*, Vol. 23, Num. 2, pp. 81-92, 1980.

[21] Shakshober, D. J., *Choosing an I/O Scheduler for Red Hat Enterprise Linux 4 and the 2.6 Kernel*, http://www.redhat.com/magazine/008jun05/features/schedulers/ .

[22] Simonet, P., *Post to rrd-developers mailing list*.

[23] Sinyagin, S., *Torrus: The Data Series Processing Framework*, http://torrus.org .

[24] Snyder, P., "tmpfs: A Virtual Memory File System," *Proceedings of the Autumn 1990 EUUG Converence*, Nice, France, pp. 241-248, 1990.

[25] Stonebraker, M., "Operating System Support for Database Management," *Commununications of the ACM*, Vol. 24, Num. 7, pp. 412-418, 1981.

Appendix: Performance Recommendations for RRD and MRTG Systems

- When building a very large RRD measurement system, dedicate the machine to this purpose. Since RRD is a file-based database, it relies on the buffer-cache that is shared across all system activity. Because of RRD's unique file access characteristics and buffering requirements, it is easier to achieve performance gains by tuning the system just for RRD.
- Use an RRDTool that has our fadvise RANDOM patch. On systems that have a fairly aggressive initial readahead (such as Linux), this will very likely increase file update performance by reducing the page fault rate and the buffer-cache memory required.
- Avoid file-level backups of RRD files unless the set of RRD files complete fit into buffer-cache memory. File-level backups read each modified file completely and sequentially; this can fill the buffer-cache and subsequently causes more page faults on RRD updates. Backups are essentially indifferentiable from application access, and thus unnecessarily populate the system's buffer-cache with content that won't be re-used soon. (Note that backup programs could call fadvise NOREUSE or fadvise DONTNEED to inform the operating system that the file content will not be re-used.)
- Split MRTG targets into a number of groups and run a separate daemon for each. In our system, we reconfigure daily and run a target_splitter script to produce a new set of ".cfg" files each with approximately 10,000 targets per MRTG daemon. Note that polling performance is also influenced by the SNMP agent performance on the network device polled. So, if the splitting results in grouping like targets together based on the model of device monitored, there could be quite a disparity in time to complete the MRTG "poll targets" phase.
- Do not create RRD files all at once. By staggering the start times, updates to like RRAs will cross block boundaries at different times, distributing the page faults that occur on block boundary crossings. As a network is deployed and grows, these RRD file start times would naturally be staggered, but this could be quite different when introducing measurement to an existing deployed network.
- Run a caching resolver or a nameserver on the localhost, i.e., the MRTG system itself. This reduces "poll targets" latency due to host name resolution; MRTG performs very many DNS name resolutions when hostnames are used (rather than IP addresses) in target definitions.
- Configure an appropriate number of forks for each MRTG daemon to minimize the time for the "poll targets" phase. On our system, 4 forks per daemon works well to keep polling in the tens of seconds for 10,000 targets. This might differ for a wide-area network.
- Place RRD files in a file-system of their own, ideally one associated with separate logical volumes or disks. This gives the system administrator flexibility to change mount options or other file-system options. It also isolates the system activity data (e.g., as displayed by sar) from unrelated activity.
- Consider mounting the file-system that contains the RRD files with the "noatime" and "nodiratime" options so that RRD file reads do not require an update to the file inode block. Of course the effect of this is that file access times will be inaccurate, but often these are not of interest for ".rrd" files.
- Consider enabling dir_index on ext file-systems to speed up lookups in large directories. MRTG places all RRD files in the same directory, and we've scaled to hundreds of thousands.