Improving the Efficiency of Information Collection and Analysis in Widely-used IT Applications

Sergey Blagodurov*
Systems Research Lab
Simon Fraser University
sergey_blagodurov@sfu.ca

Martin Arlitt
Sustainable Ecosystems Research Group
Hewlett-Packard Laboratories
martin.arlitt@hp.com

ABSTRACT

Modern IT environments collect and analyze increasingly large volumes of data for a growing number of purposes (e.g., automated management, security, regulatory compliance, etc.). Simultaneously, such environments are challenged by the need to minimize their environmental footprints. A general solution to this problem is to utilize IT resources more efficiently. This paper describes our work to systematically evaluate the inefficiencies in the information collection and analysis of several widely-used IT applications, to implement a more efficient solution, and to quantify the improvements. In particular, the logging of HTTP transactions by the Apache Web server and of network events by the Bro intrusion detection system are converted from text files to DataSeries [24]. The costs of recording, storing and analyzing the information in the different formats are thoroughly evaluated and compared. We converted the text logs to DataSeries online, with no discernable overhead on the logging applications. We achieved upto a 7x decrease in the logfile sizes relative to the sizes of the default text logs, and speedups of 3x-8.4x to analyze the logfiles.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

DataSeries, Log Analysis, Log Storage and Representation, Multicore systems

1. INTRODUCTION

Improving the environmental sustainability of IT is an important challenge. A popular way to target it is to use IT resources (e.g., multicore servers) more efficiently. Simultaneously, businesses have recognized the value of gathering and using digital information. As a result, organizations may want to collect more data, retain data longer and analyze it quicker, without paying for larger or faster IT systems.

DataSeries is a toolset developed at Hewlett-Packard Laboratories for manipulating large datasets [24, 15, 11]. It is intended for storing structured serial data, so it is similar to an append-only SQL database in that it stores data organized into *extents* (the DataSeries equivalent of tables in a database). DataSeries is different from an SQL database in that the rows of the tables have an order that is preserved. Each row consists of a number of fields (similar to SQL columns) of various types. DataSeries can use one of a number of different compression algorithms. The choice of compression algorithm can be made to either decrease creation time, read time or storage space, and also provides other options to further optimize performance. Since DataSeries compresses the entire extent as one entity, it is not efficient to append rows to an existing and packed extent in a DataSeries file. We implement a workaround to this to use DataSeries for online logging. DataSeries includes a C++ interface to quickly read and analyze DataSeries files.

Previous studies have shown that DataSeries provides significant performance and storage benefits for saving and analyzing of structured serial data [11, 23, 24]. This type of information is collected by numerous applications in many areas of computing and science (e.g., maintaining event logs or logging transactions) [24, 2]. In this paper we focus on two widely-used IT application groups: *Web servers* and *network monitoring*. In the first category, we specifically consider the Apache Web server (version 2.2.14) [2] which services and logs Web requests, and Webalizer (version 2.01), a popular open source tool for analyzing Web server logs [22]. In the second category, we examine the Bro intrusion detection system (version 1.5.1) [7] for monitoring and logging network traffic and GNU `awk` (version 3.1.6) for analyzing the Bro logfiles.[1]

The study described in this paper compares the cost of recording, storing and analyzing the information in the default text formats used by the aforementioned applications against the DataSeries format. A goal of the project is to stimulate adoption of DataSeries, by demonstrating its benefits for commonly used applications, and by providing exemplary integrations. We intend to share all the source code additions and improvements, developed within this project, with the respective open source projects. The modifications made in this work are available for download at [18]. For related work on DataSeries, readers are referred to [11, 23, 24].

The rest of the paper is organized as follows. Section 2 describes

*Work completed as an intern at HP Labs.

[1]In preparation for this study, we asked Bro developers which tools they used to analyze Bro logs. They indicated that `awk` was a tool they commonly used for this purpose.

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.
Copyright 2011 ACM 978-1-4503-0519-8/11/03 ...$10.00.

our experimental design for evaluating the benefits of DataSeries for several popular open source tools. Sections 3 and 4 illustrate the benefits of DataSeries through two case studies: Section 3 explains the DataSeries integration and experimental results for the Apache-Webalizer program pair, while Section 4 discusses the corresponding case study for the Bro-awk program pair. Section 5 briefly describes the lessons learned within this study and Section 6 concludes the paper with a summary of our project and a list of future directions.

2. EXPERIMENTAL DESIGN

2.1 HTTP workload generator

To perform the experiments with logging of Web transactions, we need a benchmark to generate the Web traffic. We sought a generator that satisfied the following requirements:

- It should be able to generate the requests with different frequency (number of requests per unit of time) and duration.
- It should generate *a meaningful workload*: that is, the pattern of requests should closely mimic the behavior of actual Web users visiting a Web site.

After considering several HTTP traffic generators (Table 1), we chose RUBiS [16] as our main workload generator since it addresses both requirements.

RUBiS is an auction site prototype modeled after eBay.com. RUBiS is used to evaluate application design patterns and a Web site's performance and scalability. It is widely used as a benchmark in network research [26, 27]. RUBiS implements the core functionality of an auction site: selling, browsing and bidding. It distinguishes between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during visitor sessions, a buyer session user can bid on items and consult a summary of their current bids, rating and comments left by other users. Seller sessions require a fee before a user is allowed to put up an item for sale. The seller can also perform other tasks, like specifying a reserve (minimum) price for an item [16].

Multiple implementations of RUBiS are available, based on several different technologies: EJB (versions 2.0, 2.1 or 3), PHP, Servlet or Servlet_Hibernate. Additionally, the distribution can be downloaded as RUBiSVA 1.0 (RUBiS Virtual Appliance) [17]. We installed and configured the RUBiS version based on PHP.

One challenge we encountered with RUBiS is that it is not deterministic: RUBiS generates a slightly different workload every time it is launched. The number and order of requests is slightly different for the runs with the same experimental setup and duration. We addressed this challenge by running RUBiS three times for every experimental configuration and comparing average values and standard deviation of the results. Step-by-step installation instructions for RUBiS PHP and other programs used in our work are available in [25].

2.2 Experimental configuration

We created an *experimental design* that allows us to simultaneously evaluate two different applications that *record* logs and two different applications that *analyze* logs. The design is illustrated in Figure 1. First, we use *a workload generator* that generates HTTP requests. These requests are served by *a Web server* that records logs about which requests are served. The traffic between the workload generator and the Web server are observed by *a network monitor* that collects and records data on network activity. We then perform an offline analysis of both types of logs collected during the previous step.

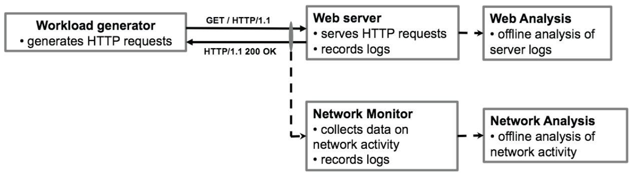


Figure 1: The experimental design used in this study.

Figure 2 shows the testbed we ran our experiments on.

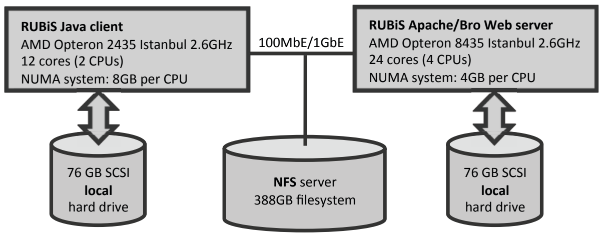

RUBiS Java client
AMD Opteron 2435 Istanbul 2.6GHz
12 cores (2 CPUs)
NUMA system: 8GB per CPU
100MbE/1GbE
RUBiS Apache/Bro Web server
AMD Opteron 8435 Istanbul 2.6GHz
24 cores (4 CPUs)
NUMA system: 4GB per CPU
76 GB SCSI local hard drive
NFS server
388GB filesystem
76 GB SCSI local hard drive

Figure 2: The experimental configuration used in this study.

One server, which we refer to as the **RUBiS Java client**, generated the HTTP workload. This server has two AMD Opteron 2435 Istanbul 2.6 GHz CPUs, each with six cores (12 total CPU cores). It is a NUMA system: each CPU has an associated 8 GB memory block, for a total of 16 GB main memory. Each CPU has 6 MB 48-way L3 cache shared by its six cores. Each core also has a private unified 512 KB 16-way L2 cache and a private 64 KB 2-way L1 instruction and data caches. The client machine was configured with a single 76 GB SCSI hard drive.

A second server ran the Apache Web server and Bro intrusion detection system. The **RUBiS Apache/Bro Web server** has four AMD Opteron 8435 Istanbul 2.6 GHz CPUs, each with six cores for a total of 24 CPU cores. It is a NUMA system: each CPU has an associated 4 GB memory block, for a total of 16 GB main memory. Each CPU has 5 MB 48-way L3 cache shared by its six cores. Each core also has a private unified 512 KB 16-way L2 cache and a private 64 KB 2-way L1 instruction and data caches. The server was configured with a single 76 GB SCSI hard drive.

The servers were configured with Linux Gentoo 2.6.29 release 6. Both of these servers could access a 388 GB NFS filesystem. Initially, these servers were inter-connected via a 100 Mb/s Ethernet network. During our experiments, we determined this network link became a performance bottleneck, so we upgraded to a 1 Gb/s Ethernet.

A goal of our study is to determine how the efficiency of the two application groups described above can be improved through more effective use of multicore servers, such as those in our testbed (depicted in Figure 2). The efficiency of logging and analysis in the two application groups will be improved in two ways. First, logs will be created using DataSeries rather than plaintext. Second, the analysis tools will be parallelized to take advantage of the multicore processors available on the modern servers. The improvements are quantified using two metrics: the decrease in the amount of disk space required to store the logfiles; and the speedup in analysis time.

	Varying workload	*Meaningful workload*
Pre-collected traces [15, 20]	No	Yes
Pktgen, Http Traffic Generator, etc.	Yes	No
httperf [13]	Yes	Partially (can be emulated with the user defined sessions)
RUBiS [16]	Yes	Yes

Table 1: The comparison of web traffic generators.

Another important consideration is whether the DataSeries logging implementation works properly. For offline conversion of a plaintext log, this is straightforward to check. However, with online creation of DataSeries logs, verification is more difficult, owing to the nondeterministic nature of the RUBiS workload generation. Therefore, we validate the conversion of plaintext logs to DataSeries using the following two metrics: the time for the application to log a similar number of items, and the amount of data logged for a similar workload. In both cases, we expect to see similar numbers with and without the integration of DataSeries.

2.3 DataSeries configuration

DataSeries has a variety of options that can be selected, to give users flexibility in how they use it [11]. We performed several offline experiments with DataSeries to determine which parameter settings to use in our online experiments. For these offline experiments, we converted an existing Apache logfile into DataSeries format. The logfile was created by the Apache Web server and a RUBiS workload that lasted 10 hours with 1,000 RUBiS clients. This resulted in almost 25 million log records in the Apache `access.log` file, which required about 3 GB of storage space. We found out that the DataSeries conversion tool csv2ds creates files with a very small default extent size (64 KB, which is approximately 500 rows from an Apache log), resulting in many small extents of the same type being included in the DataSeries file (once the current extent was full, csv2ds wrote it into the file and created a new one). We investigated how the size of the extent affects the size of the resulting DataSeries logfile. Figure 3 shows that if the size of the extent is kept relatively large (10,000 rows or higher), the difference in log filesize is kept within 10%.With larger extent sizes, slightly better compression can occur. However, the tradeoff is more log data must be buffered before the extent can be written to disk. We selected 10,000 rows per extent as a reasonable tradeoff to use in our online experiments.

DataSeries files can be compressed by one of four compression algorithms (lzf, lzo, gzip and bzip2) [11], with lzf being the least efficient in terms of storage space, but the fastest one in terms of access time, and bzip2 vice versa. We consider both lzf and bzip2 in our online experiments.

The ideal choice of compression algorithms and extent size depends on the intended use of the data. While we found an extent size of 10,000 rows and the lzf compression algorithm enabled quick analysis results, if our priority was to minimize storage space we could use larger extent sizes and bzip2. If we wanted to quickly generate a report on a dataset but then archive the dataset, we could also store the data initially using lzf, but then convert the lzf-compressed data set to a bzip2-compressed data set (potentially with larger extent sizes) before archiving the dataset.

3. APACHE-WEBALIZER CASE STUDY

In this section we describe how we integrated DataSeries into the Apache Web server and Webalizer, and quantify the benefits that it provides.

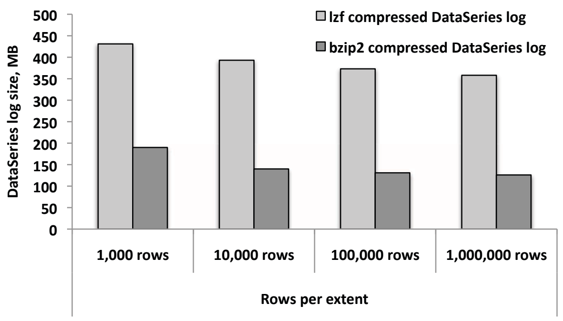


Figure 3: DataSeries filesize with different extent sizes.

3.1 Apache Web server logging

Mod_log_config [3] is the module that allows the Apache Web server to log client requests. Logs can be written in a customizable format directly to a file or to an external program. Each string in the logfile corresponds to a distinct Web request. The format for the logfile can be specified as a string argument to the *LogFormat* or *CustomLog* directives in the module's configuration file (/etc/apache2/modules.d/00_mod_log_config.conf on Linux Gentoo). This string is used to specify what information will be included in each log record and how it will appear in the logfile.

The log format we use is called *combined_plus*. In comparison with the default *common* format, it has three additional fields: referer, user-agent and request time (Table 2). The first two fields are usually included since they can provide useful information about the Web site audience (where visitors came from and what browsers they are using). The last field is included to test the efficiency of logging with DataSeries. Specifically, we want to verify that the time it takes for the modified Apache Web server to service a request does not change relative to logging in plaintext. The *combined_plus* log format appears in the configuration file as follows:

```
LogFormat "%h %l %u %t \"%r\" %>s %b
\"%{Referer}i\" \"%{User-Agent}i\" %D"
combined_plus
```

A corresponding sample string from the logfile:

```
127.0.0.1 - - [11/Jul/2010:13:13:39
-0700] "GET /PHP/index.html HTTP/1.1" 200
2149 "-" "Java/1.6.0\_17" 69
```

To create a new module for Apache, a tool called apxs [12] can be used. apxs allows compiling, linking and installing of the module into the Apache framework. The main difficulty we encountered is that the module must be written in C, while DataSeries uses C++. To build C++ source as a valid Apache module, the commands used by apxs need to be modified as is shown in [12].

Field	Description
host	The IP address or fully qualified domain name (FQDN) of the client that made the HTTP resource request.
user ID	A unique identifier for the user making the HTTP request. If no value is present, a "-" is substituted.
username	The username provided for authentication. If no value is present, a "-" is substituted.
date:time	The date and time the HTTP request was received.
request	The request field contains three pieces of information: the HTTP method (e.g., GET), the requested resource (e.g., index.html), and the HTTP protocol version (e.g., 1.1).
statuscode	A numeric code indicating the success or failure of the HTTP request.
bytes	The number of bytes of data transferred in the HTTP response, not including the HTTP response header.
referer	The page from which the user issued the current request.
user-agent	The user-agent (browser) that issued the request.
request time	The time taken to serve the request, in microseconds.

Table 2: The description of the fields used in the default *common* (in italic) and *combined_plus* (in italic and italic bold) log formats.

Table 3: Verification results (Apache).

(a) The number of strings in the Apache logfile.

	1 hour, 240 clients		2 hours, 240 clients	
	Mean	*Stdev*	*Mean*	*Stdev*
Plaintext log (mod_log_config)	324,966	2,315	645,814.5	1,034
DataSeries log (lzf compressed)	323,611	2,254	646,302	93
DataSeries log (bzip2 compressed)	327,156	1,310	645,846	903

(b) Apache Web server service times.

1 hour, 240 clients		2 hours, 240 clients	
Mean	*Stdev*	*Mean*	*Stdev*
255.838	287.517	253.392	283.396
251.02	290.091	244.359	333.965
244.274	370.158	240.071	333.917

The ability of DataSeries to compress the data of the created extents is very useful for maximizing storage space savings. However, it makes it very difficult to append new rows to the existing packed extent, a feature that would be useful when logging Web transactions. The issue is the extent would have to be first read from the DataSeries file, decompressed into main memory and then compressed back with the new rows appended to it. We use the following logging scheme to avoid this:

- The data about each serviced request is first saved into the usual text logfile (the way mod_log_config would do it).

- Once the number of strings in the logfile reaches a threshold number K, the data in the logfile is converted into an extent by the DataSeries toolset. The extent is then compressed and appended to the end of the DataSeries logfile. On one hand, K should not be too small, as in this case there could be conflicts between the threads that write into the temporary text log and the thread that reads from it, dumps the data into the DataSeries logfile and truncates it. On the other hand, K should not be too big, as that would cause an unnecessarily large temporary text log, and would increase the time of conversion it to DataSeries format. We experimantially chose $K = 10,000$ as a trade-off value (as discussed in Section 2.3).

- After the DataSeries extent is saved to disk, the small text logfile is truncated and the cycle repeats. One issue with writing a temporary file to disk and then writing the DataSeries file also to disk is that it actually increases the amount of disk traffic in comparison with the default Apache setup (where only one textfile is being modified). Although the increase in the disk traffic is not high (DataSeries compresses the extents so the amount of data being transferred is low), the disk traffic increase might be crucial on a busy Web server or when using nfs filesystem as a storage space. An alternative we tested is writing the temporary file to tmpfs[2] [19], a filesystem that exists in the main memory of the machine. This substantially decreases the disk traffic (even in comparison with the default case as only packed DataSeries files will be written to disk). The disadvantage of this solution is that, during unexpected crash of the machine, the data in such a temporary file might be lost. As the local disk was not a bottleneck in our experiments, the results we present in the remainder of the paper use the local disk approach for storing the temporary file.

The resulting DataSeries module, which we call mod_log_dataseries, allows for logging of HTTP transactions in both DataSeries and plaintext as well as DataSeries only. It uses the following schema [24] when creating DataSeries logfiles:

```
<ExtentType name="apache2ds" version="1.0">
  <field type="variable32" name="ip_address"
  pack_unique="yes" />
  <field type="variable32" name="client_identity"
  pack_unique="yes" />
  <field type="variable32" name="userid"
  pack_unique="yes" />
  <field type="variable32" name="request_time" />
  <field type="variable32" name="request_data"
  pack_unique="yes" />
  <field type="variable32" name="status_code"
  pack_unique="yes" />
  <field type="int32" name="object_size" />
  <field type="variable32" name="referrer"
  pack_unique="yes" />
```

[2]Another alternative is to create an in-memory buffer to store the temporary log file. However, as we are unfamiliar with Apache's memory management, we did not attempt this approach. However, should Apache developers adopt DataSeries, they should consider the in-memory approach.

```
  <field type="variable32" name="user_agent"
  pack_unique="yes" />
  <field type="int32" name="time_to_serve" />
</ExtentType>
```

The schema is used by the DataSeries module to determine the name and structure of the extent that holds the data within the DataSeries logfile, i.e., which fields each row of this extent will contain. In our case, the extent is called "apache2ds" and each of its rows contains ten fields that correspond to the log format fields from Table 2. DataSeries currently supports six data types: bool (0 or 1), byte (0-255), int32 (signed 32 bit integer), int64 (signed 64 bit integer), double (IEEE 64 bit floating point) and variable32 (up to 2^{31} bytes of variable length data, such as strings) [24]. The information contained in most of the fields can be described as a string (variable32), with two fields (object_size and time_to_serve) presented as 32 bit integers. For most of the string fields, the field-level option pack_unique is also specified. This option enables each unique variable32 value to be packed only once within that extent. For fields with many repeated values this option increases the effective compression ratio. Object_size and time_to_serve were chosen to be represented as integers due to its numerical nature and also because storing integer values were empirically proven to consume less space than variable32.

After integrating DataSeries into Apache, we needed to verify that it worked properly. Table 3(a) shows the number of records logged by the DataSeries module (in its lzf and bzip2 implementations) as well as the number of records in the plain Apache log created by mod_log_config for different durations of the experiment. Each combination of Apache module and experiment duration was tested three times. Due to the non-deterministic workload, there are minor variations in the results. However, the consistency indicates that the DataSeries module captures the same information as the default Apache mod_log_config.

The second validation experiment examined the overhead of online logging in DataSeries on the Apache Web server. Table 3(b) shows that the request service time when using mod_log_dataseries is quite similar to those recorded by the original mod_log_config. The small variations in the average service time are negligible in comparison with the high persistent standard deviation of the request times that exists even with the default Apache setup.

Satisfied that our integration of DataSeries into Apache worked correctly with negligible performance overhead, we then turned our attention to quantifying the benefits DataSeries provides. Figure 4 shows the improvements in terms of storage space when logging with the DataSeries module for the different experiment durations and number of RUBiS clients. It shows that the lzf-compressed DataSeries logs are on average 7 times more compact than the original Apache logfiles. The improvement in terms of disk space is even greater when using bzip2 compression. The size of DataSeries logfiles is close to that of the plaintext Apache logs, compressed with gzip *as a whole*. For the latter case however, the logs would first have to be collected in plaintext and then compressed *offline* (e.g., with gzip), which takes additional disk space during logging phase and compression time afterwards, whereas our DataSeries Apache module is able to gather compressed logs *online*, as Apache services Web requests.

The average number of lines in the DataSeries files for experiments with various duration is provided on Figure 5. In the shortest of our experiments, ~1 million records were logged while the longest experiment had approximately 189 million records. One concern that system administrators may have in adopting DataSeries is how to work with a binary format rather than the text format they are accustomed to. Fortunately, it is straightforward for system administrators to convert DataSeries files back into plaintext, if they so choose. It can be done with the special parallelization library from HP Labs called Lintel (available for download from [15]). The DataSeries distribution includes a tool called ds2txt that uses Lintel and will utilize all available CPUs on a multicore machine to speed this process up. It also allows user to skip unnecessary DataSeries internal information in the output log (i.e., names, types and position of extents in the input DataSeries file):

```
ds2txt --skip-index --skip-types \
--skip-extent-type --skip-extent-fieldnames \
access.ds > access_log_from_ds
```

Figure 6 shows the time it takes to convert the DataSeries logfiles back into plaintext (the sizes and the average number of records in each DataSeries file is given on Figures 4 and 5). The results obtained with ds2txt tool suggest that even for logfiles with almost 200 million entries, it only takes 11-12 minutes to regenerate the complete plaintext logfile.

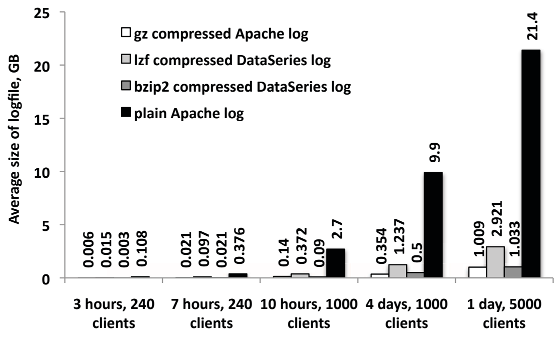


Figure 4: Effect of format on logfile size.

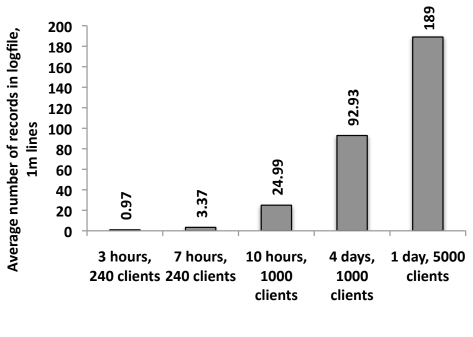


Figure 5: Number of records (Web requests) per logfile.

3.2 Analysis of Apache DataSeries logs

In this section we demonstrate the improvements that logging in DataSeries format can provide to the analysis time of the Apache logfiles. We tried several different analysis tools (awstats [4], Analog [1], etc.) before choosing Webalizer [21] as our log analyzer. Webalizer is a Web server logfile analysis tool created by Bradford Barrett and distributed under the GNU GPL. Webalizer produces usage statistics for different time periods, along with the ability to display usage by site, URL, referer, user agent (browser), search string, entry/exit page, username and country. We chose Webalizer for the following reasons:

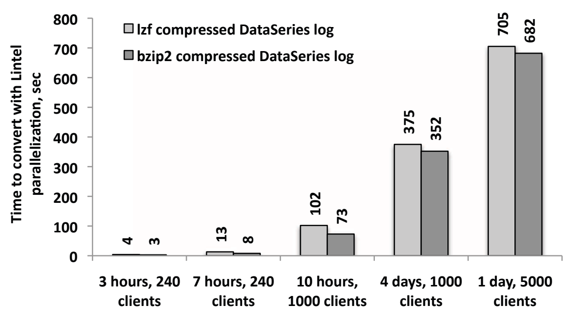


Figure 6: DataSeries to plaintext conversion times (Apache).

1) It is widely used [26, 27] and is included in all main Linux software package trees so can be easily installed and configured on almost any Linux distribution.

2) It is written in C (as opposed to Perl for awstats), which makes it easy to modify for working with DataSeries (C++). Note that working with C++ classes in Perl is also possible with the special modules for Perl called Inline-CPP [14], but, since the latter generally takes more time to configure and debug, we decided to use Webalizer instead.

3) It has a special mode (if enabled at compile time) in which it can analyze logfiles compressed with gzip. Any logfilename that ends with a ".gz" extension is assumed to be in an archive format and is uncompressed as it is read. This makes it possible to compare the analysis time of DataSeries logs with not only plaintext logs, but also with the logs stored in a popular compression format.

To take full advantage of DataSeries and multicore servers, we first needed to parallelize the single-threaded Webalizer. The important details when parallelizing Webalizer are:

Each thread in our multi-threaded version of Webalizer works independently on its own piece of the input DataSeries file. The threads exchange data only when the results are aggregated into the final report, as explained below. In this way we can maximize the parallelization of processing the logfile. We use the following scheme to break the logfile into pieces:

- We first obtain the number of extents of type "apache2ds" from the footer of the DataSeries file (with 10,000 rows in each extent). We refer to this as *extent_num*.

- We then divide *extent_num* by the number of analyzing threads *thread_id_max* specified as a #define directive during Webalizer compilation. This determines the fraction of the data (i.e., the number of extents) that will be processed by each thread.

- We obtain the initial and the last extent that each DataSeries thread will process.

- We open the DataSeries logfile for reading in each thread, move the read pointer to the start of the corresponding piece and start reading/decompressing the data.

Webalizer produces several reports (html) and graphics for each reporting period (e.g., one month) processed. The main report is written in a file "index.html" to the directory that can be specified as a configuration parameter OutputDir in /etc/webalizer.conf (by default /var/www/localhost/htdocs/webalizer) and is a summary page for the current and previous periods. The various totals shown in the report are explained in Table 4. When parallelizing the analysis stage (report generation), we considered the following issues.

Since the initial version of Webalizer processes strings from the logfile in a single-threaded mode, all of its internal structures must be transformed into arrays to make the parallelization possible. For example, there is a temporary variable called `t_hit`, which contains the total number of hits in the currently processing month. For the parallelized version of Webalizer we need to turn it into an array `t_hit[MAX_THREAD_NUM]` that contains a distinct copy of that variable for each processing thread (otherwise, all the threads would have to work with the same variable, which would be very costly in terms of access synchronization as the number of threads grows).

The initial version dumps all the total information for the reporting period into the report, once all the log records for the current period have been processed. In the multi-threaded version we need to detect when the period has been processed in all of the threads and only then generate the overall report. The difficulty here is that we need to combine the totals from each processing thread. For hits, files, pages and Kbytes this is straightforward (e.g., just sum `t_hit[t]`). Total unique request senders (sites) and the number of sessions (visits), however, are not simple summations of the corresponding variables from every thread. For example, the total number of unique addresses per the whole period (the "sites" total) is not simply a sum of unique addresses from each processing thread, since each thread worked only on its own piece of data and hence has the information about only a subset of the total records. As a result, the sum of all the `t_site[t]` values will always be greater than or equal to the entire site's value.

Although each analyzing thread will be able to correctly determine the number of unique visits within its piece of the logfile, the total number of visits could be less than the sum of all `t_visit[t]`, because the time of the last records in one piece can be no earlier than within 30 minutes of the time the next piece starts (30 minutes is the default visit timeout in Webalizer, which can be changed in /etc/webalizer.conf).

Due to space limitations, we refer the interested reader to our technical report [25] for the description of the multi-threaded solution.

Besides the varying level of parallelization (the number of parallel threads of execution), we also tested several different schemes of parallelization as described below.

Plaintext, single-threaded is the initial sequential version of Webalizer that works on the plaintext logs obtained with mod_log_config. The scheme *compressed plaintext, single-threaded* is the same as the *plaintext, single-threaded*, except we first compressed the input text log with gzip.

We designed and implemented four parallelization schemes. In the *unified* scheme (Figure 7(a)), each thread of the multi-threaded Webalizer performs the entire cycle of processing for each row in its log chunk: it reads the data from the DataSeries log, decompresses it, and performs the analysis. This is the most straightforward parallelization scheme. The schemes *unified* and *unified with parser* (Figure 7(b)) are essentially the same, except *unified* does not take advantage of the pre-parsed fields already present in the DataSeries file (each string from the Apache logfile is stored in the DataSeries file as a row of fields in an extent). Instead, *unified* first converts all the fields to one string and parses it with the standard Webalizer function `parse_record()`, while *unified with parser* takes DataSeries fields and incorporates them directly into the We-

Parameter	Description
Hits	Any request made to and logged by the server during the specified reporting period is considered a 'hit'.
Files	Some requests require the server to send an object, such as an html page or image, back to the requesting client. When this happens, it is considered a *file* response, and the files total is incremented.
Pages	Any request that retrieves an HTML object or causes an HTML object to be generated is considered a *page*. This does not include the other content that is embedded in a Web page, such as images, audio clips, etc. The default action is to treat anything with the extension '.htm', '.html' or '.cgi' as a page.
Sites	Each request made to the server comes from a unique *site*, which can be an IP address or FQDN. The Webalizer maintains a cache of DNS lookups to reduce processing the same addresses in subsequent runs. The final value of *sites* indicates how many unique sites made requests to the server during the reporting period.
Visits	Whenever a request is made to the server from a given site, the amount of time since a previous request (if any) by the address is calculated. If the time difference is greater than a pre-configured *visit timeout* value (or the site has never made a request before), it is considered a *new visit*, and this total is incremented. The default timeout is 30 minutes.
KBytes	This parameter shows the amount of data, in KB, which was sent out by the server during the specified reporting period.

Table 4: The description of the parameters presented in the total report generated by Webalizer.

balizer structure `log_rec[thread_id]` that describes the currently processed log record. *Unified with parser* skips the Webalizer parsing. By comparing the performance of these two schemes we can determine the usefulness of keeping the fields separately for each row in DataSeries file.

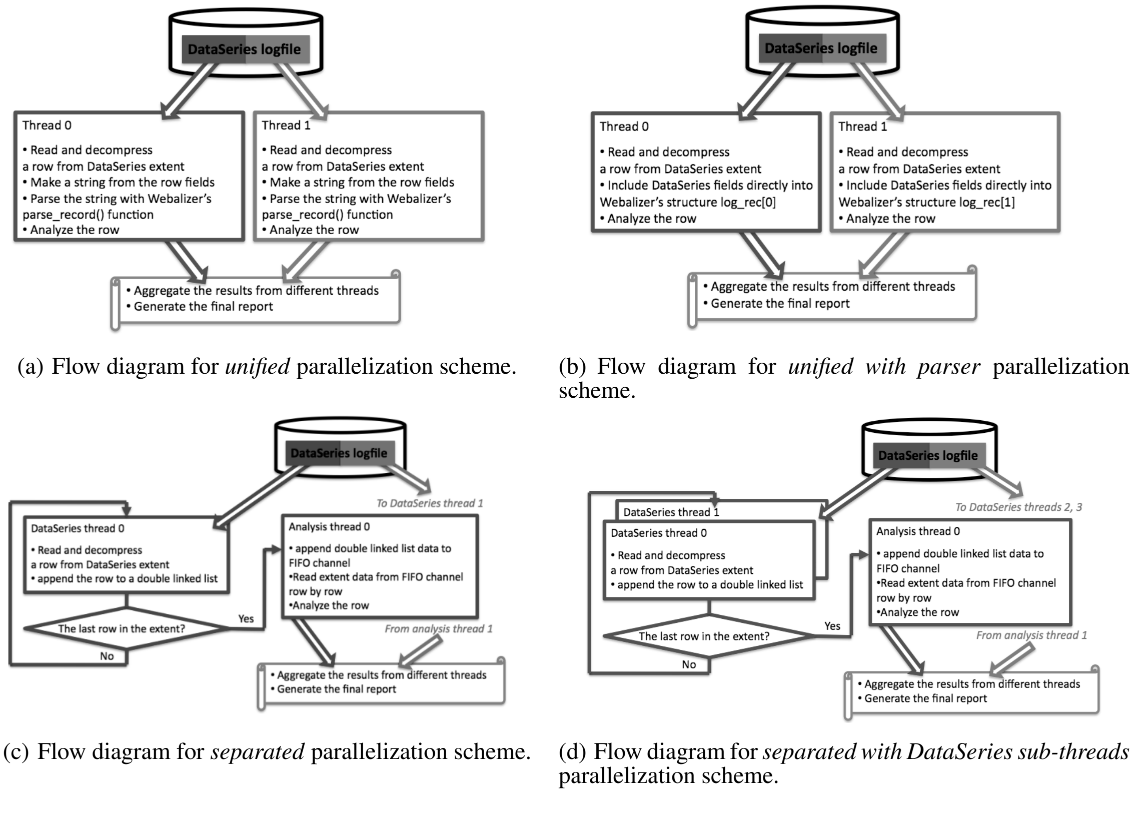
A third parallelization scheme we examined is *Separated* (Figure 7(c)). In this scheme there are two kinds of threads. The first type we call *DataSeries threads*. These threads read the DataSeries file, decompress the extents and supply the information to the analysis module (by taking advantage of the pre-parsed DataSeries fields, similar to *unified with parser*). The second type we call *analysis threads*. These analyze the data provided by DataSeries threads, save the intermediate results and create the resulting report.

The *separated* scheme works as follows. Once the current extent in a DataSeries thread has been decompressed, the thread starts to read rows of the extent one by one. The row data then goes into a temporary buffer organized as a doubly-linked list. There is one such buffer dedicated for every DataSeries thread. Each element in the buffer contains all of the fields of a log record (a row from the extent currently being processed by that DataSeries thread). Once all rows from the current extent are processed, the buffer with the extent data is then appended to a FIFO channel. Rows from the FIFO channel are then read by the corresponding *analysis thread* (keeping the buffer and the FIFO channel separate reduces synchronization overhead). As with the buffers, the number of FIFO channels is equal to the number of DataSeries threads. Each analysis thread works independently on its own channel, so the number of analysis threads is also equal to the number of DataSeries threads. The purpose of this scheme is to further parallelize the processing of each record: now obtaining the data from the logfile and analyzing it can be done concurrently.

A fourth scheme called *Separated with DataSeries sub-threads* (Figure 7(d)) is similar to the *separated* scheme. The difference with this fourth scheme is there are two DataSeries threads per each analysis thread: one DataSeries sub-thread works on odd rows of the currently processed extent, while the other works on the even rows. Both sub-threads dump information into the same doubly-linked list, which is in turn appended to the FIFO channel of the single analysis thread. We added this scheme after we observed that analysis threads in *separated* tests occasionally stalled due to a lack of data (i.e., DataSeries threads can be a bottleneck in the separated scheme).

The results of testing these different parallelization schemes on an lzf-compressed DataSeries logfile (1 day, 5,000 RUBiS clients, approximately 189 million hits) are provided in Figure 8. By default, Webalizer can parse plaintext logs and text logs compressed with gzip. The analysis of the Apache plaintext log by the default Webalizer takes ~646 seconds. It does not benefit from the multicore system, as Webalizer is a single threaded application. If we compress the plaintext log with gzip, this reduces the storage requirement but increases the analysis time by 9% if we ignore the time to create the gzip compressed file, and more than doubles the analysis time if we take into account the time it takes to convert the plaintext logs into gzip format. The results for DataSeries parallelization suggest that *unified with parser* is the fastest parallelization scheme with more than 8x speed up over the naive single-threaded approach, when all 24 cores of the multicore server running the analysis are fully utilized. *Unified* is slower by 48% on average than *unified with parser*, since the *unified* approach does not take advantage of the already parsed DataSeries fields. We expected *separated* might be faster than *unified with parser*, since it parallelizes obtaining the data from the logfile with analyzing. It gives in fact almost the same performance results as *unified with parser* when the number of analysis threads is small due to the additional overhead of data exchange between the DataSeries thread and its analyzing thread within the same multi-threaded application. As the number of analysis threads increase, this scheme becomes slower than *unified with parser*. This is because there are twice as many threads in separated as in *unified with parser*. Thus, when the number of analysis threads is high, the *separated* approach has twice the contention for the shared resources of the multicore system (shared last-level caches, memory controllers, etc.) This results in the corresponding slowdown. For the *separated with DataSeries sub-threads* case, the additional DataSeries thread increases the speed with which the data is supplied for analysis, but it also increases the synchronization costs of accessing the doubly-linked list buffer by two DataSeries threads. Nevertheless, it gives the fastest results (by up to 16% relative to *unified with parser*) until the number of analysis threads reaches six, after which the overhead of the increased number of threads results in the slowdown for this parallelization scheme as well. In fact, the overhead becomes even greater for the *separated with DataSeries sub-threads* approach, due to the extra DataSeries thread used.

Our next set of experiments read data from an NFS filesytem rather than from local disk on the server. Figure 9 shows the results for both the local disk and NFS experiments. The results are normalized to the analysis time of the default set-up of a single threaded Webalizer parsing uncompressed plaintext logs, read from



DataSeries logfile
Thread 0
• Read and decompress a row from DataSeries extent
• Make a string from the row fields
• Parse the string with Webalizer's parse_record() function
• Analyze the row
Thread 1
• Read and decompress a row from DataSeries extent
• Make a string from the row fields
• Parse the string with Webalizer's parse_record() function
• Analyze the row
• Aggregate the results from different threads
• Generate the final report
DataSeries logfile
Thread 0
• Read and decompress a row from DataSeries extent
• Include DataSeries fields directly into Webalizer's structure log_rec[0]
• Analyze the row
Thread 1
• Read and decompress a row from DataSeries extent
• Include DataSeries fields directly into Webalizer's structure log_rec[1]
• Analyze the row
• Aggregate the results from different threads
• Generate the final report
DataSeries logfile
To DataSeries thread 1
DataSeries thread 0
• Read and decompress a row from DataSeries extent
• append the row to a double linked list
The last row in the extent?
Yes
No
Analysis thread 0
• append double linked list data to FIFO channel
•Read extent data from FIFO channel row by row
•Analyze the row
From analysis thread 1
• Aggregate the results from different threads
• Generate the final report
DataSeries logfile
To DataSeries threads 2, 3
DataSeries thread 1
DataSeries thread 0
• Read and decompress a row from DataSeries extent
• append the row to a double linked list
The last row in the extent?
Yes
No
Analysis thread 0
• append double linked list data to FIFO channel
•Read extent data from FIFO channel row by row
•Analyze the row
From analysis thread 1
• Aggregate the results from different threads
• Generate the final report

(a) Flow diagram for *unified* parallelization scheme.

(b) Flow diagram for *unified with parser* parallelization scheme.

(c) Flow diagram for *separated* parallelization scheme.

(d) Flow diagram for *separated with DataSeries sub-threads* parallelization scheme.

Figure 7: Flow diagrams for the parallelization schemes studied.

local disk. For the DataSeries tests, both lzf- and bzip2-compressed DataSeries logfiles were used. The duration of the experiments was the same as for the tests depicted on Figure 8 (1 day, 5,000 RUBiS clients, 189 million requests). The results from Figure 9 suggest that:

The highest performance was obtained using the parallelized Webalizer with DataSeries (lzf compression), read from local disk (the solid grey line with circles, up to 8.4x speedup). This configuration was parallelizable until approximately 24 cores, at which point the server system becomes fully loaded.

We discovered that the speedup of the corresponding NFS analysis depends on the speed of underlying Ethernet network. With the relatively slow 100 Mb/s network, the link quickly saturates, as the 100 MbE link connecting the analysis server to the NFS filesystem becomes a bottleneck. The effect of this bottleneck can be seen from the dotted grey line in Figure 9. To verify that the network was indeed the bottleneck in this case, we used capstats [9], a package that is shipped with Bro to measure the network bandwidth used by a TCP connection. This problem disappeared as soon as we switched to the 1 Gb/s network (the solid black line with circles). Here the analysis time was very close to that of the local analysis, with up to 8.0x speedup.

We also observed that, while the DataSeries bzip2 curves are not affected by the Ethernet speed within the networks tested (the data that needs to be transferred via the network in case of a tightly packed bzip2 files is significantly less than that of the lzf files), the analysis speedup actually *decrease* for large numbers of threads. The reason is that the decompression threads that DataSeries spawns for bzip2 are significantly more computationally expensive than the decompression threads for lzf. Hence, when the number of analysis threads is high and the system is fully loaded, bzip2 decompression takes CPU time from analysis threads.

Lastly, we consider the experiments where the data is accessed via NFS. In this case, the plaintext analysis of the default single-threaded Webalizer experiences more than 2.5x slowdown if a 100 MbE rather than a 1GbE network is used. Similar to DataSeries bzip2 curves, the analysis of the plaintext logfiles, compressed with gzip as a whole, is not affected by the Ethernet speed and is close to the uncompressed plaintext analysis over 1GbE. Note that the speedups for "plaintext gzip (single-threaded)" are given for analysis stage only and do not include the time to compress the logfiles with gzip during their offline creation. If we were to include it, the plaintext gzip curves of the default Webalizer would be more than 2x slower (DataSeries logs are created online and hence do not have such an overhead).

4. BRO-AWK CASE STUDY

As further evidence of the benefits DataSeries provides to data collection and analysis processes, we integrate it into tools used in a different domain, and quantify its benefits over current practices.

4.1 Bro Intrusion Detection System Logging

Bro is a Unix-based Network intrusion detection system (IDS) [8]. Bro monitors network traffic and detects intrusion attempts based on the traffic characteristics and content. Bro may record summaries of different network events (e.g., *connection* characteristics), or generate alerts to potential intrusions.

Bro can read live traffic from a machine's local network interface by specifying the -i flag and the interface's name. Bro stores the information about the traffic it has observed in several log-

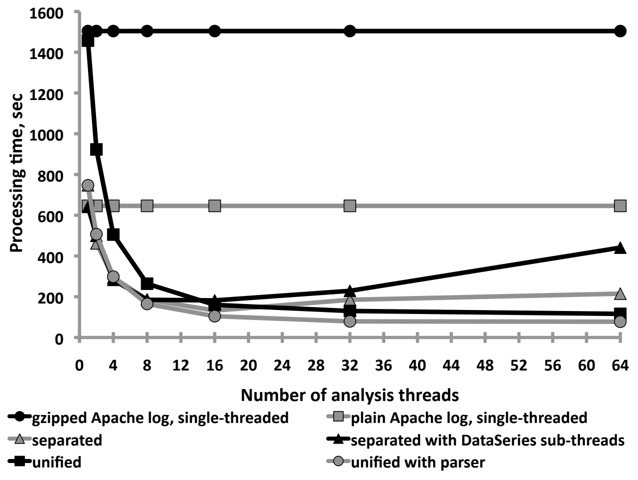


Figure 8: Webalizer processing times (189M requests).

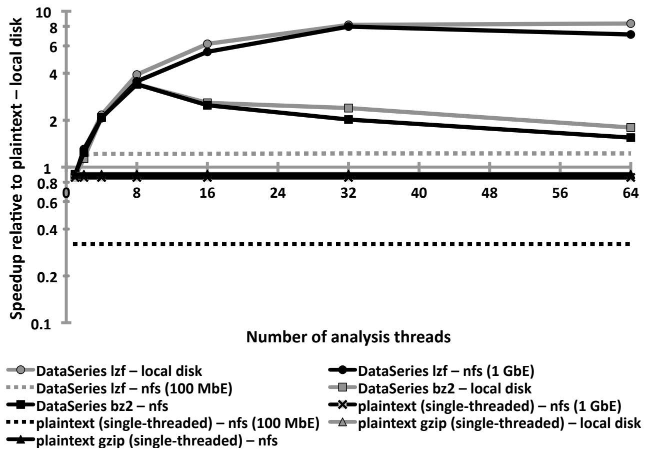


Figure 9: Webalizer analysis speedup (189M requests, logarithmic scale).

files. A commonly used Bro module called *conn* produces a log that contains one-line ASCII summaries of each "connection" [5]. The summaries are produced by the `record_connection()` function, and have the following format (all reported on a single line):

```
<start> <duration> <local IP> <remote IP> \
<service> <local port> <remote port> \
<protocol> <org bytes sent> \
<res bytes sent> <state> <flags> <tag>
```

These fields are explained in Table 5.

Here is an example of a conn.log connection summary:

```
1280879507.515707 3.533462 192.168.1.1 \
10.0.0.1 ssh 55092 22 tcp 538 1713 SF \
X @47-4187-11
```

The connection began at timestamp 1280879507.515707 (Aug 3 16:51:47 2010 as is shown by cf utility [6]) and lasted 3.533462 seconds. The service was SSH (this conclusion is based just on the responder's use of port 22/tcp). The originator sent 538 bytes, and the responder sent 1,713 bytes. Because the L flag is absent, the connection was initiated by host 10.0.0.1, and the responding host was 192.168.1.1. When the summary was written, the connection was in the SF state. The connection had neither the L nor U flags associated with it, and there was additional information, summarized by the string @47-4187-11.

Adding DataSeries logging to Bro is more straightforward than with Apache, since Bro does not use modules to log the information about transactions on the monitored network interface. The source file that is responsible for logging is located at <Bro_ source>/src/File.cc. This file contains the method `BroFile::Write()` of class BroFile (each instance of that class describes one logfile). The method writes strings to the appropriate Bro log depending on the specific BroFile object has called it. To add the ability of logging in DataSeries format, we added the function `convert_log_to_ds()`, which is invoked once the number of strings in the plaintext log `conn.log` reaches 10,000. As is the case with Apache module mod_log_dataseries (Section 3.1), this function creates a new extent, populates it with the data from the temporary conn.logfile and then appends the extent to conn.ds. `Convert_log_to_ds()` uses the following schema [24] when creating DataSeries conn.ds file:

```
<ExtentType name="conn2ds" version="1.0" >
  <field type="int64" name="start_time" />
  <field type="double" name="duration"
  opt_nullable="yes" />
  <field type="variable32" name="local_IP"
  pack_unique="yes" />
  <field type="variable32" name="remote_IP"
  pack_unique="yes" />
  <field type="variable32" name="service"
  pack_unique="yes" />
  <field type="int32" name="local_port" />
  <field type="int32" name="remote_port" />
  <field type="variable32" name="protocol"
  pack_unique="yes" />
  <field type="int32" name="org_bytes_sent"
  opt_nullable="yes" />
  <field type="int32" name="res_bytes_sent"
  opt_nullable="yes" />
  <field type="variable32" name="state" />
  <field type="variable32" name="flags" />
  <field type="variable32" name="tag" />
</ExtentType>
```

The DataSeries extent name is "conn2ds" and each of its rows contains 13 fields that correspond to the log format fields from Table 5. The description of each field type and the field-level option `pack_unique` can be found in section 3.1. The information contained in most of the fields can be described as a string (variable32), with some fields (start_time, local_port and remote_port) presented as 32 or 64 bit integers. Fields such as org_bytes_sent, res_bytes_sent or duration are typically numeric, but occasionally include a "?" when Bro cannot determine the corresponding value. For this reason, such fields could be described as variable32 values. Since only one non-numeric character will ever be present in these fields, an alterative approach is to replace the "?" in those fields by null values (field-level option opt_nullable) and represent them as either double or int32. We chose this approach because, according to our observations, numerical fields consume less space than do the corresponding strings.

Table 6 shows the number of records logged by the Bro intrusion detection system with DataSeries modifications (logs were com-

Parameter	Description
Start	The connection's start time, in seconds since the beginning of Unix epoch. (`cf`, a Bro utility, can convert this to human readable format [6]).
Duration	The connection's duration, in seconds.
Local IP, Remote IP	The *local* and *remote* IP addresses that participated in the connection, respectively.
Service	The connection's service (e.g., http, ftp, etc); this is based on well known port number mappings.
Local port, Remote port	The transport-level ports used by the connection.
Protocol	The transport protocol that was used (e.g., tcp, udp, icmp).
Org bytes sent, res bytes sent	The number of bytes sent by the *originator* and *responder*, respectively.
State	The state of the connection at the time the summary was written. `SF` state means the normal connection establishment and termination was observed ([10] describes other possible connection states).
Flags	Reports a set of additional binary state associated with the connection: L indicates that the connection was initiated *locally*; U indicates the connection involved one of the networks listed in the neighbor_nets Bro variable. X is used to indicate that neither the L nor U flags is associated with this connection.
Tag	Reference tag to log lines containing additional information associated with the connection in other logfiles, (e.g., http.log).

Table 5: The description of the fields in conn.tag.log Bro logfile.

pressed with either lzf or bzip2 compression algorithm) and without them. Due to the non-deterministic workload, we conducted three experiments for each testing combination and experimental duration. The results are generally similar to those presented in Table 3(a). This indicates that our implementation is functioning as intended.

Since the Bro IDS only *passively monitors* the data on the network interface (i.e., it does not affect actual network traffic), we do not provide the results showing that Bro implementation with DataSeries does not slow down the usual system work (similar to those in Table 3(b)). However, as future work a comparison of the overhead of logging the default text logs versus the compressed DataSeries logs should be conducted.

Figure 10 shows the improvements in terms of storage space when logging the *conn* network transactions with DataSeries support for different experiment durations and number of RUBiS clients. It shows that the lzf-compressed DataSeries conn.log is on average 2.6x more compact than the original plaintext conn.log. The improvement in terms of disk space with bzip2-compressed DataSeries, `conn.log` is higher, reaching 4.3x. According to these results, the DataSeries benefits for Bro conn.log is less than that for the Apache logs (7.3x for lzf- and 21.4x for bzip2-compressed logs, see Figure 4). The reason is that the conn.log data in general is less compressible than the Apache access.log. Even for the case when the entire conn.log file was compressed with gzip -best (something that is slow and not achievable online), the resulting .gz file was only 25% smaller than the bzip2-compressed DataSeries file. The average number of lines in DataSeries conn.log files for experiments with various duration is provided in Figure 11. The number of records per file is noticeably smaller than for the corresponding Apache log, as the Bro log contains records of TCP connections, each of which can carry multiple HTTP request/response transactions. Due to the smaller number of records, the conversion times for these logfiles from DataSeries back to plaintext (shown in Figure 12) is also much lower than was seen in the corresponding Apache tests (Figure 6).

4.2 Analysis of conn.log DataSeries logs

To describe the performance benefits that DataSeries provides to the analysis of the Bro transaction conn.log, we needed an analysis application as a baseline to compare DataSeries results with. While Webalizer is a popular tool for analyzing Apache logs, we

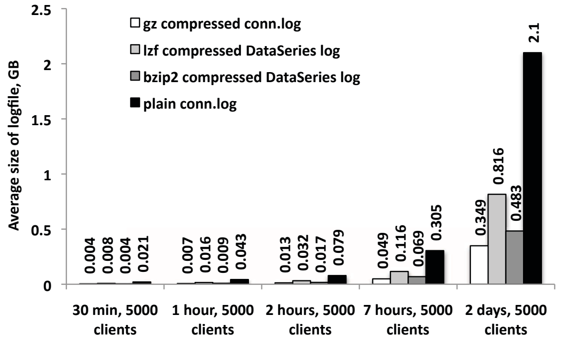


Figure 10: Storage space by logfile format (Bro).

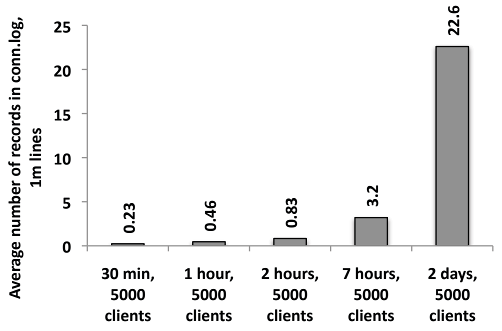


Figure 11: Number of records per DataSeries file (Bro).

are not aware of any specially crafted analysis tools for Bro. Instead, we asked the Bro developers which tool(s) they use to analyze conn.log. They indicated the `awk` processing tool was commonly used.

To compare analyses done with `awk` on plaintext logs with DataSeries, we performed the following analysis to look for potentially malicious traffic in the Bro logs. For each record in conn.log file we:

1. Check if the IP address of the originator is not in the list of

	30 minutes, 5,000 clients		1 hour, 5,000 clients	
	Mean	*Stdev*	*Mean*	*Stdev*
Plaintext conn.log	230,435	491.4	458,171	205
DataSeries conn.log (lzf compressed)	231,557	566	455,521	336
DataSeries conn.log (bzip2 compressed)	235,967	1,360	460,002	113

Table 6: The number of strings in the conn logfile.

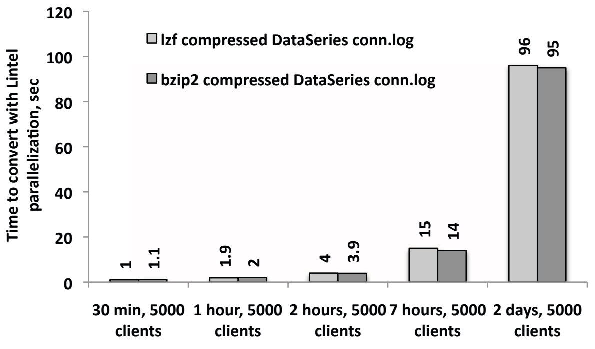


Figure 12: DataSeries to plaintext conversion times (Bro).

allowed IP addresses. The list of allowed IP addresses/networks is stored in the neighbor_nets Bro variable.

2. If the value of the Flags field is "X" (see the description in Table 5), then a possible network intrusion attempt may have occurred.
3. If so, check the State field: the values "S1" or "SF" mean that an untrusted connection was established.
4. If so, record the IP addresses, ports and duration of this connection.

After all the records are processed, return the untrusted connections, sorted by duration.

In `awk`, the malicious traffic analysis performed over the plaintext logs looks like this:

```
time awk '{ if (   ($12 == "X") && \
($11 == "S1" || $11 == "SF")    ) \
print $2, $3, $4, $6, $7 | \
"sort -n +0 -1"}' conn.log > conn.log.reported
```

We implemented the same analysis in C++ using the RowAnalysisModule DataSeries class [24], which is provided by the DataSeries API. RowAnalysisModule performs an analysis a row at a time. It handles the issues of iterating over the rows in each extent, and calling preparation and finalization functions.

The performance results for the experiments with RowAnalysisModule/`awk` over Bro `conn.log` are provided in Figure 13. For DataSeries tests with RowAnalysisModule, lzf- and bzip2-compressed DataSeries `conn.log` collected from a two-day long experiment with 5,000 RUBiS clients was used. The main results are similar to those for the Apache log analysis (Section 3.2). For the Bro case study, DataSeries' RowAnalysisModule provided over 3x faster analysis time for lzf-compressed and over 2.3x for bz2-compressed DataSeries files relative to the default `awk` analysis. The results differ from Apache log analysis in the following ways:

For the single-threaded mode, DataSeries analysis consistently provides faster results in comparison with the single threaded `awk` (up to 38%). The reason is the robust implementation of the malicious traffic analysis directly in C++ using the DataSeries toolset.

The NFS analysis over the DataSeries and plaintext files (the solid black lines) is significantly slower (by up to 57% for the fastest results) than the analysis of the same logs performed locally. The reason is that, unlike with the concise Apache analysis, the malicious traffic report may contain a lot of connections and so requires writing data to the NFS directory. Even with 1GbE connection that corresponds to the slowdown mentioned. The dotted lines correspond to the experiments when the initial conn.log was stored on the nfs server, while the report was saved locally on the analysis machine. As can be seen, the speedup in this case closely resembles that of the local analysis.

We also observed that the RowAnalysisModule analysis is getting slower for large numbers of threads. The reason for this slowdown is the consolidation of data prior to generation of the final malicious traffic report. In order to sort the connections according to its duration (part of the analysis), we need to first unite all the small arrays with connection data from every thread. As the number of threads goes up, the overhead of making and sorting one big array is also increasing.

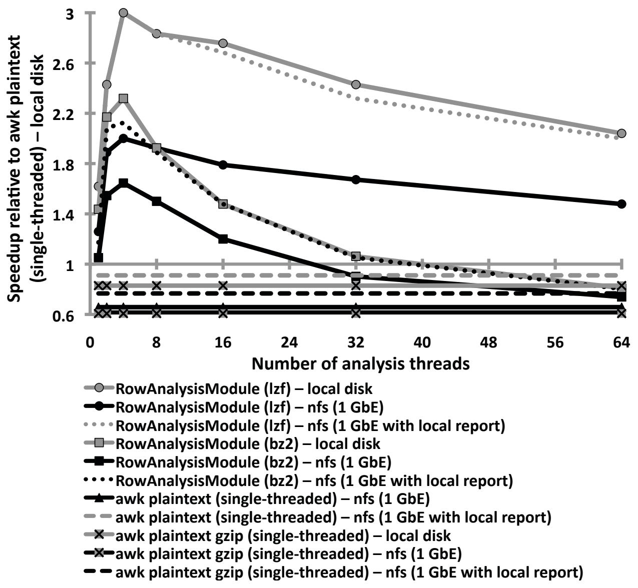


Figure 13: RowAnalysisModule/awk speedup (Bro).

5. LESSONS LEARNED

The main goal of the project described in this paper was to demonstrate how reuse of an existing software toolset (DataSeries) could enable more efficient and quick analysis of large data sets. Our experiences in completing the project revealed that a variety of common performance engineering properties continue to hold. We briefly describe several of these.

Exploiting parallelism is challenging. While we were able to improve performance by up to an impressive 8.4x, this is still well below the potential speedup of 24x that one might have expected, moving from a single-threaded application to a multi-threaded implementation running on a 24 core server. In several cases, the performance actually decreases as additional cores are used, owing to contention amongst the various threads. Clearly, the opportunity remains to improve performance further.

Bottlenecks shift. When we began our work, the performance of the single-threaded application was limited by the use of a single CPU core. When we introduced a multi-threaded implementation, that bottleneck was eliminated, but new ones emerged. In our NFS experiments, the bottleneck then became the 100 Mb/s network link. We eliminated that by upgrading to a 1 Gb/s link. The bottleneck is now contention amongst the decompression and analysis threads. Addressing this bottleneck is left for future work.

6. CONCLUSION AND FUTURE WORK

In this paper we described how we used DataSeries as the online logging format for two popular open source applications, the Apache Web server and the Bro intrusion detection system. We modified the Webalizer tool to efficiently analyze Web server logs in DataSeries format. We also demonstated how to efficiently analyze the Bro Intrusion Detection System logs in DataSeries format. We quantified the benefits of storing and accessing information in DataSeries format relative to the default log format of the chosen applications.

Our experimental results showed significant benefits are possible from leveraging DataSeries and multicore servers. The sizes of the Apache logs decreased by up to 7x (2.6x for Bro), and the time to analyze the Apache logs decreased by 8x (3x for Bro). Our work verifies that DataSeries is beneficial for online logging applications, and that it facilitates efficient analysis. A motivating goal of our work is to have our initial implementations serve as examples that others can use to integrate DataSeries into the applications we examined, or into other applications that generate or analyze structured serial data. We have shared our results and source code with the developers of Bro, who are considering adding DataSeries to a future release of the IDS, as improving the efficiency of Bro is one of their key goals. As future work, we will perform a comparison of the performance of binary log formats and logging libraries with the best applicability areas for each. We intend to communicate our results and source code with the Apache and Webalizer groups, and to continue to search for opportunities to improve the effective use of IT infrastructure.

7. REFERENCES

[1] Analog: a free logfile analyzer. *[Online] Available: http://www.analog.cx/.*

[2] Apache HTTP server project. *[Online] Available: http://httpd.apache.org/download.cgi.*

[3] Apache logging module mod_log_config. *[Online] Available: http://httpd.apache.org/docs/2.0/mod/mod_log_config.html.*

[4] Awstats: a free logfile analyzer. *[Online] Available: http://awstats.sourceforge.net/.*

[5] Bro IDS Reference Manual: Analyzers and Events. *[Online] Available: http://www.bro-ids.org/wiki/index.php/Reference_Manual:_Analyzers_and_Events.*

[6] Bro IDS Reference Manual: Getting Started (the cf utility). *[Online] Available: http://www.bro-ids.org/wiki/index.php/Reference_Manual:_Getting_Started#The_cf_utility.*

[7] Bro intrusion detection system. *[Online] Available: http://www.bro-ids.org/download.html.*

[8] Bro Quick Start Guide. *[Online] Available: http://www.bro-ids.org/Bro-quick-start.pdf.*

[9] Capstats: a quick hack to get some NIC statistics. *[Online] Available: http://www.icir.org/robin/capstats/.*

[10] Conn.log connection summaries. *[Online] Available: http://tinyurl.com/bro-conns.*

[11] DataSeries technical report. *[Online] Available: http://tesla.hpl.hp.com/opensource/DataSeries-tr-snapshot.pdf.*

[12] How do you create a new Apache module? *[Online] Available: http://ivascucristian.com/how-do-you-create-a-new-apache-module.*

[13] Httperf: a tool for measuring web server performance. *[Online] Available: http://www.hpl.hp.com/research/linux/httperf/.*

[14] The Inline::CPP module: put C++ source code directly "inline" in a Perl script. *[Online] Available: http://search.cpan.org/ neilw/Inline-CPP-0.25/lib/Inline/CPP.pod.*

[15] Open Source software at Hewlett-Packard Laboratories. *[Online] Available: http://tesla.hpl.hp.com/opensource/.*

[16] RUBiS: an auction site prototype. *[Online] Available: http://rubis.ow2.org/.*

[17] RUBiSVA: a virtual appliance of the RUBiS benchmark. *[Online] Available: http://rubis.ow2.org/download/rubisva_v1.0.pdf.*

[18] Source files with modifications done within this work. *[Online] Available: http://www.sfu.ca/ sba70/files/dataseries/.*

[19] Tmpfs: a temporary file storage facility. *[Online] Available: http://en.wikipedia.org/wiki/Tmpfs.*

[20] Traces from the Internet Traffic Archive. *[Online] Available: http://ita.ee.lbl.gov/html/traces.html.*

[21] Webalizer: a free logfile analyzer. *[Online] Available: http://www.webalizer.org/.*

[22] The Webalizer: free web server log file analysis program. *[Online] Available: http://www.webalizer.org/download.html.*

[23] E. Anderson. Capture, conversion, and analysis of an intense NFS workload. In *FAST '09*, pages 139–152, 2009.

[24] E. Anderson, M. Arlitt, C. B. Morrey, III, and A. Veitch. DataSeries: an efficient, flexible data format for structured serial data. *SIGOPS Oper. Syst. Rev.*, 43(1):70–75, 2009.

[25] S. Blagodurov and M. Arlitt. Improving the efficiency of information collection and analysis in widely-used IT applications. *HPL Technical report [Online] Available: http://www.hpl.hp.com/techreports/2010/HPL-2010-164.html.*

[26] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *SIGMETRICS '05*, pages 291–302, 2005.

[27] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy. Profiling and modeling resource usage of virtualized applications. In *Middleware '08*, pages 366–387, 2008.