



Improving the efficiency of information collection and analysis in widely-used IT applications

Sergey Blagodurov, Martin Arlitt

HP Laboratories
HPL-2010-164R1

Keyword(s):

efficiency, collection, analysis, DataSeries, Apache, Bro, Web server, Intrusion Detection System

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. The goals of this paper are 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 will be converted from text files to DataSeries [1]. The costs of recording, storing and analyzing the information in the different formats are thoroughly evaluated and compared. In particular, we converted the text logs to DataSeries online, with no discernable overhead on the logging applications. We achieved a 7x decrease in the logfile sizes relative to the sizes of the default text logs, and speedups of almost 8x to analyze the logfiles.

External Posting Date: January 6, 2011 [Fulltext] Approved for External Publication

Internal Posting Date: January 6, 2011 [Fulltext]

A shorter version of the paper will appear in the 2nd ACM/SPEC International Conference on Performance Engineering (ICPE), March 14-16, 2011, Karlsruhe, Germany.

© Copyright the 2nd ACM/SPEC International Conference on Performance Engineering (ICPE), 2011.

Improving the efficiency of information collection and analysis in widely-used IT applications

Sergey Blagodurov and Martin Arlitt

Sustainable Ecosystems Research Group

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. The goals of this paper are 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 will be converted from text files to DataSeries [1]. The costs of recording, storing and analyzing the information in the different formats are thoroughly evaluated and compared. In particular, we converted the text logs to DataSeries online, with no discernable overhead on the logging applications. We achieved a 7x decrease in the logfile sizes relative to the sizes of the default text logs, and speedups of more than 8x to analyze the logfiles.

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 [1,9,10]. 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. Because DataSeries compresses the entire extent as one entity, it might be difficult to append rows to already existing packed extent inside DataSeries file, which might be useful when maintaining a logfile (we worked around this in our implementation). To access DataSeries files, a programmer needs to write a module to process the data, and then compile and link that module into a program. 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 [1,9]. This type of information is collected by numerous applications in many areas of computing and science (e.g., maintaining event logs or logging transactions) [1,3]. In this report 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 [4]. In the second category, we examine the Bro intrusion detection system (version 1.5.1) [6] for monitoring and logging network traffic and GNU awk (version 3.1.6) for analyzing

the Bro logfiles.¹

This study compares the cost of recording, storing and analyzing the information in the default text formats used by the aforementioned applications against the HP Labs 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 interested reader can download all the modifications at [7]). For related work on DataSeries, readers are referred to [1, 8, 10].

The rest of the report is organized as follows. Section 2 describes the choice of software that we used to generate the HTTP workload in our experiments, the experimental configuration that we used and the preliminary experiments we conducted with DataSeries. Section 3 explains the integration of DataSeries into the Apache-Webalizer program pair. Section 4 discusses the DataSeries integration into the Bro-awk program pair. Both of these sections also contain experimental results that quantify the benefits that DataSeries provides to these programs. Section 5 briefly describes the lessons learned within this study and Section 6 concludes the report with a summary of the project and a list of future directions.

2. HTTP workload generator, experimental configuration and preliminary results

2.1. HTTP workload generator

To perform the experiments with logging of Web transactions, we need a benchmark to generate the Web traffic. The following requirements should be satisfied by the workload generator program:

- 1) It should be able to generate the requests with different frequency (number of requests per unit of time) and duration.
- 2) 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 [11] as our main workload generator since it addresses both requirements.

	<i>Varying workload</i>	<i>Meaningful workload</i>
Pre-collected traces [5,6]	No	Yes
Pktgen, Http Traffic Generator, etc.	Yes	No
httperf [14]	Yes	Partially (can be emulated with the user defined sessions)
RUBiS [11]	Yes	Yes

Table 1. The comparison of web traffic generators.

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 [8,9]. 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 [11].

¹ In preparation for this project, 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.

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] that simplifies the installation, deployment and monitoring of the RUBiS benchmark. Essentially, RUBiSVA is an image disk of a preinstalled system that is ready to use. Although highly convenient, we did not use RUBiSVA in our experiments as it only provides the image for 32-bit Xen hypervisors, while our test servers are 64-bit. Instead, 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. Appendix A provides a step-by-step installation of RUBiS with PHP implementation on Linux.

2.2 Experimental configuration

We wanted to come up with an *experimental design* (Figure 1) that would allow us to simultaneously evaluate two different applications that *record* logs and two different applications that *analyze* logs. We decided on the following design. 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 data on network activity and additionally records logs about it. We then perform an offline analysis of both types of logs collected during the previous step.

The test-bed we used to perform the experiments for this work was the combination of two systems (Figure 2):

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

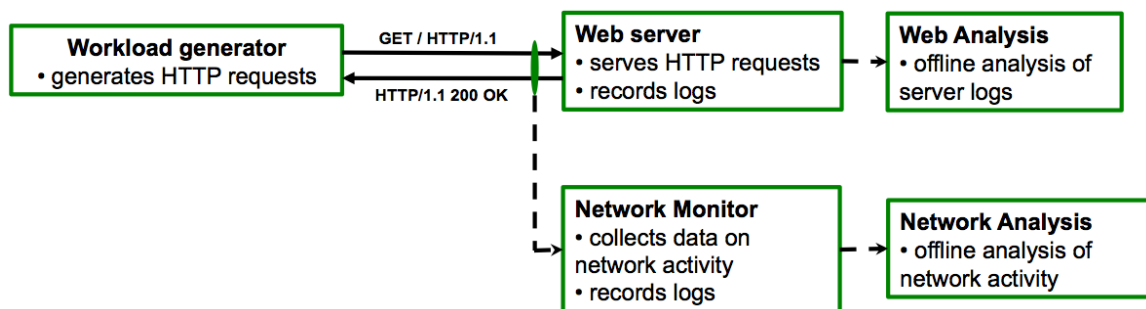


Figure 1. The experimental design used in this study.

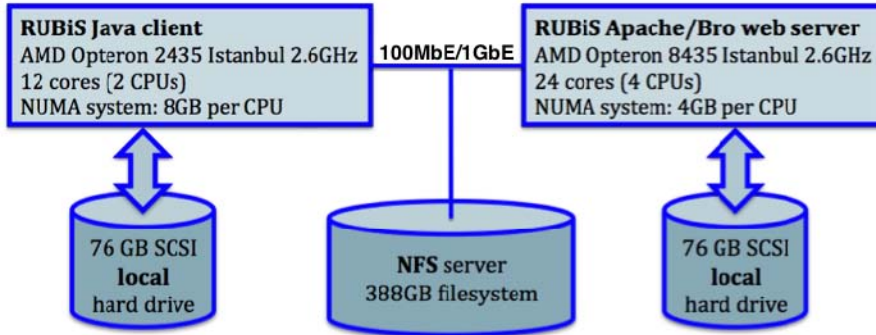


Figure 2. The experimental configuration used in this study.

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

Both machines could access a 388GB NFS filesystem over 100Mb/s and later 1Gb/s Ethernet. The machines were configured with Linux Gentoo 2.6.29 release 6.

A goal of this study is to determine how the efficiency of two application groups above can be improved through more effective use of the multicore setup depicted in Figure 1. The efficiency of logging and analysis in the two application groups will be improved with the following two methods:

- Logging in plain text will be converted to logging in DataSeries format.
- The analysis tools will be parallelized to take advantage of the multicore systems.

The improvements are quantified in the following two metrics:

- The decrease in the amount of disk space it takes to store the logfiles.
- The speedup in the analysis of the data saved in the logfiles.

The conversion to DataSeries stage is validated using the following two metrics:

- The logging program is not slowed down by the DataSeries add-on.
- There is no loss of data when logging with DataSeries.

To ensure that logging in DataSeries does not slow down the application that is logging data, we create the DataSeries file in parallel with the application's other functionality.

2.3. Preliminary experiments with DataSeries

Our initial experiment applied DataSeries to an existing logfile, without integrating DataSeries into the application. Figure 3 shows the results of this experiment. The logfile was created by our 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.

DataSeries files can be compressed by one of the four compression algorithms (lzf, lzo, gzip and bzip2), with lzf being the least efficient in terms of storage space, but the fastest one in terms of access time, and bzip2 vice versa. Using the DataSeries toolset, we then converted the plain text logfile into lzf- and bzip2-compressed DataSeries files. For this initial analysis, we used the DataSeries module csv2ds, slightly modified to parse the Apache log format, to convert the text logfile (access.log) into DataSeries format. When converting this lengthy plain text logfile into DataSeries format, we found out that the default csv2ds tool has very small default size of the extent being created (64KB which is approximately 500 rows from Apache log), resulting in too 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 then

investigated how the size of the extent affects the size of the resulting DataSeries logfile. Figure 3a 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 for this application.

Figure 3b shows the time it took to convert the plain text logfile into both a gzip-compressed text file and lzf- and bzip2-compressed DataSeries files. While the real time (the time it took to finish the conversion) for lzf and bzip2 is almost the same, the user time (which reflects the cumulative computational time on several cores of the multicore server) for bzip2 is significantly higher, due to its better packaging efficiency (and hence higher computational requirements) relative to lzf. We also measured the decrease in log size (Figure 3c). Compared to the default text logfile, the DataSeries files use almost 7x less storage space (for lzf-compression). With bzip2, the DataSeries file is even smaller, and is comparable in size to that of gzip-compressed plain log.

Our next preliminary experiment ran Webalizer on this set of logs. The average analysis time of calculating *the page rating* (one analysis Webalizer provides) for each type of file is shown in Figure 3d. Page rating is the list of “page name – the number of occurrences of this page in the logfile (number of hits)” sorted by the number of occurrences (referred to as *hits*). For the analysis of the DataSeries files, we used RowAnalysisModule [1]. This built in DataSeries general module performs analysis that operates a row at a time. It handles the issues of iterating over the rows in each extent, and calling preparation and finalization functions. With RowAnalysisModule the calculation of page rating is straightforward: we would need only to maintain the associative array of all the “page name – hits” pairs and then, after all the rows are processed, sort the array using hits as a key. The *real time* shown in the Figures 3a and 3c is the actual execution time of a program from the moment it starts till the completion. The *user time* on the same Figures reflects the cumulative CPU time that the program consumed on a multicore machine. This value can be higher than the real time when the program is multi-threaded and is being executed on several cores of the multicore server in parallel.

The results in Figure 3d show more than 6x speedup in the analysis time of DataSeries files in comparison with plain text and gzipped logfiles. The analysis time speedup of DataSeries logs originates from the following two factors:

- DataSeries exploits parallelism in operations, such as compressing/decompressing and writing/reading extents from disk in parallel, to increase the access efficiency on multicore systems. This improvement is very important for compression, which is usually slow, but can also be important for simple analyses that run faster than a single core can decompress [1].
- Since each Web request (a string in the Apache logfile) is stored in the DataSeries file as a row of fields in an extent, the time for parsing each string (row) during the analysis stage is significantly reduced in comparison with plain text analysis, where we first need to check the validity of each line and then extract the necessary fields from it.

It is important to note that the ideal choice of compression algorithms and extent sizes 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.

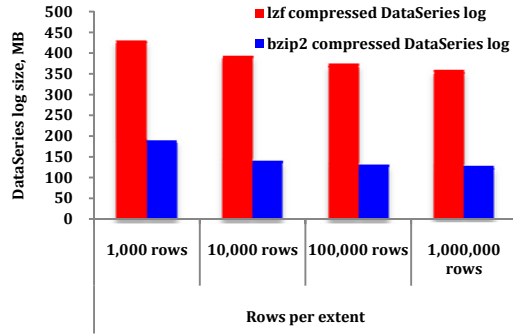


Figure 3a. DataSeries filesize with different extent sizes.

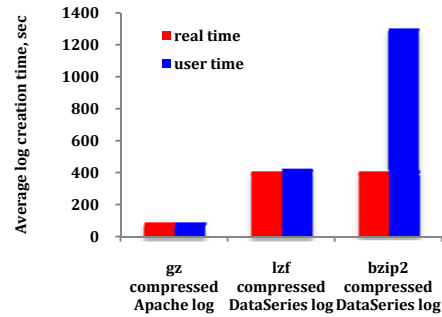


Figure 3b. Average log creation time.

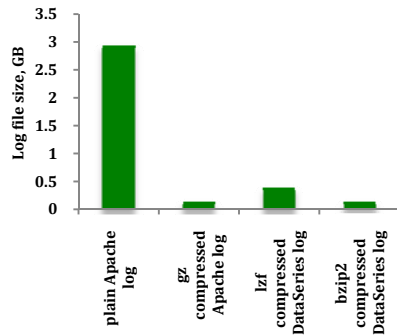


Figure 3c. Logfile size (10,000 rows/extent).

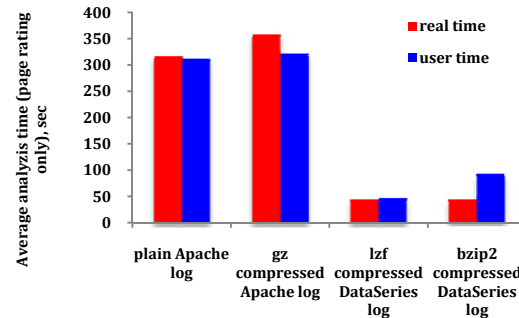


Figure 3d. Average analysis time.

Figure 3. The preliminary results for log creation and analysis with DataSeries.

3. Improving efficiency of information collection and analysis for Apache-Webalizer

In Section 2.3 we showed that DataSeries provides benefits both in terms of analysis speedup and storage space. In this section we describe how we integrated DataSeries into the Apache Web server and Webalizer, and quantify the benefits that it provides.

3.1. Logging of HTTP transactions by the Apache Web server

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 the 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. It can contain literal characters copied into the logfiles and the C-style control characters "\n" and "\t" to represent new-lines and tabs. Literal quotes and back-slashes should be escaped with back-slashes. The default log format for the Apache distribution is called *common* and looks like the following (see Table 2 for the description of the fields used):

```
LogFormat "%h %l %u %t \"%A\" %>s %b" common
```

A corresponding string in the logfile would look like:

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

The log format we use is called *combined_plus*. It has three additional fields that are absent in *common*: 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 required 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 plain text.

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

Field	Description
<i>Host</i>	The IP address or host/subdomain name of the client that made the HTTP resource request.
<i>user ID</i>	The identifier used to identify the client making the HTTP request. If no value is present, a "-" is substituted.
<i>username</i>	The username, (or user ID) used by the client for authentication. If no value is present, a "-" is substituted.
<i>date:time</i>	The date and time stamp of the HTTP request.
<i>request</i>	The HTTP request. The request field contains three pieces of information. The main piece is the requested resource (index.html). The request field also contains the HTTP method (GET) and the HTTP protocol version (1.1).
<i>statuscode</i>	The status is the numeric code indicating the success or failure of the HTTP request.
<i>Bytes</i>	The bytes field is a numeric field containing the number of bytes of data transferred as part of the HTTP request, not including the HTTP header.
<i>referrer</i>	The page from which the user posted the current request
<i>user-agent</i>	The user-agent (browser) that posted the request
<i>request time</i>	The time taken to serve the request, in microseconds.

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

To create a new module for Apache, a tool called apxs [18] 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, we modified the commands used by apxs like so:

```
/usr/bin/libtool --silent --mode=compile x86_64-pc-linux-gnu-g++ -prefer-pic -O2 -pipe -DLINUX=2 -D_REENTRANT -D_GNU_SOURCE -pthread -I/usr/include/libxml2/ -I/usr/include/apache2 -I/usr/include/apr-1 -I/usr/include/apr-1 -I/usr/include/db4.7 -c -o mod_log_dataserries.lo mod_log_dataserries.cpp
/usr/bin/libtool --silent --mode=link x86_64-pc-linux-gnu-g++ -lDataSeries -o mod_log_dataserries.la -rpath /usr/lib64/apache2/modules -module -avoid-version mod_log_dataserries.lo
/usr/lib64/apache2/build/instdso.sh SH_LIBTOOL="/usr/bin/libtool" mod_log_dataserries.la /usr/lib64/apache2/modules
/usr/bin/libtool --mode=install cp mod_log_dataserries.la /usr/lib64/apache2/modules/
```

Several additional details must be considered when creating a DataSeries logging module for Apache:

- A good starting point for creating a new logging module is to use the source of mod_log_config and rewrite it in C++ so that the creation and access of DataSeries class instances will be possible.
- 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 experimentally chose $K = 10,000$ as a trade-off value (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 (remember that the DataSeries provide at least 7x decrease of the filesize as is shown in Section 2.3), 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 tmpfs² [32], a filesystem that exists in the main memory of the machine. That would substantially decrease the disk traffic (even in comparison with the default case as only packed DataSeries files will be written to disk), but the disadvantage of this solution is that, during unexpected crash of the machine, the data in such a temporary file might be lost. Another solution is to store the small temporary file on the local disk of the machine, rather than on the nfs location.
- Since the Apache Web server is highly parallelized, the logging of the transactions with DataSeries must be thread-safe. To achieve this, we need to use the file handling functions and mutexes from the Apache Portable Runtime (APR). The APR Pools are a fundamental building block of Apache, and are the basis for all resource management. They serve to allocate memory, either directly (in a malloc-like manner) or indirectly (e.g., in string manipulation), and, crucially, ensure the memory is freed at the appropriate time. They also ensure that other resources such as files or mutexes can be allocated and will always be properly cleaned up [12, 13].

The resulting DataSeries module, which we call `mod_log_dataserries`, allows for logging of HTTP transactions in both DataSeries and plain text as well as DataSeries only. It uses the following schema [1] 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" />
  <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., what fields each row of this extent will contain. In our case, the extent is called «apache2ds» and each of its rows contains 10 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 32bit integer), int64 (signed 64bit integer), double (IEEE 64 bit floating point) and variable32 (up to 2^{31} bytes of variable length data, such as strings) [1]. 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 32bit 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

² Another alternative is to create an in-memory buffer to store the temporary log file. However, we are unfamiliar with Apache's memory management, so we deferred exploration of this approach.

storing integer values were empirically proven to consume less space than variable32.

Table 3a 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. There were three experiments for each combination of Apache module and experiment duration. This data indicates that the DataSeries module captures the same information as the default Apache mod_log_config. Table 3b shows that the request service time when using mod_log_dataserries 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. We used awk to calculate average and standard deviation (stdev) number of rows in each of the logfiles (\$13 is the column number in Apache logs that contains the request service time):

```
awk '{s+=$13} END{print "Sum: "s, "\nNumber of lines: "NR, "\nAverage: "s/(NR)}' file
awk '{sum+=$13; array[NR]=$13} END {for(x=1;x<=NR;x++){sumsq+=((array[x]-(sum/NR))^2);}print sqrt(sumsq/NR)}' file
```

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 plain text Apache log, compressed with gzip *as a whole*. For the latter case however, the logs would first have to be collected in plain text 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 190 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 plain text, if they so choose. It can be done with the special parallelization library from HP Labs called Lintel (available for download from HP portal [9]). Lintel includes a *batch-parallel* processing tool, which will utilize all available CPUs on a multicore machine to speed this process up. It uses a special ds2txt module to do that:

```
time batch-parallel -j 24 ds2txt -- access.ds
```

Alternatively, the DataSeries distribution includes a tool called ds2txt that also uses Lintel library and provides similar performance to that of batch-parallel, but additionally 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):

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

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

	1 hour, 240 clients		2 hours, 240 clients	
	Mean	Stdev	Mean	Stdev
Plain text log (mod_log_config)	324966	2315	645814.5	1034
DataSeries log (lzf compressed)	323611	2254	646302	93
DataSeries log (bzip2 compressed)	327156	1310	645846	903

Table 3a. The number of strings in the Apache logfile.

	1 hour, 240 clients		2 hours, 240 clients	
	Mean	Stdev	Mean	Stdev
Plain text log (mod_log_config)	255.838	287.517	253.392	283.396
DataSeries log (Lzf compressed)	251.02	290.091	244.359	333.965
DataSeries log (bzip2 compressed)	244.274	370.158	240.071	333.917

Table 3b. The request service time of Apache Web server.

The data for Apache experiments with two logging modules (mod_log_config and mod_log_dataserries) and different HTTP load.

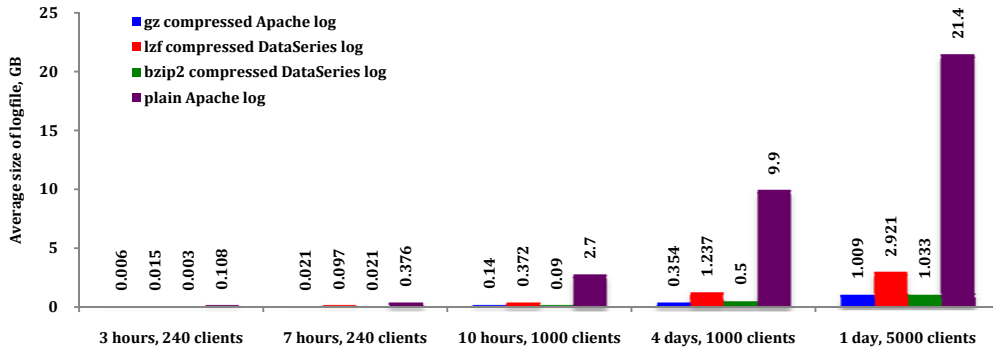


Figure 4. The Improvements in terms of storage space when logging with DataSeries.

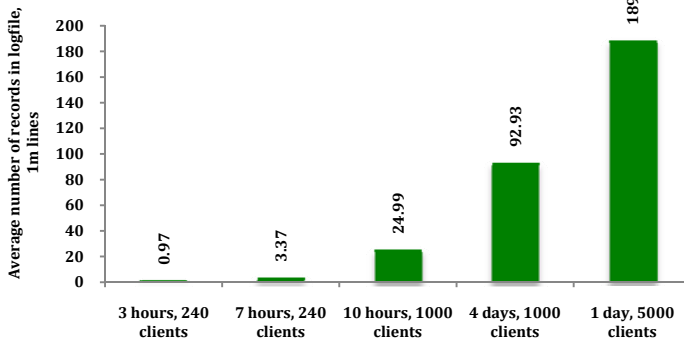


Figure 5. The average number of records (Web requests) in DataSeries files of various size.

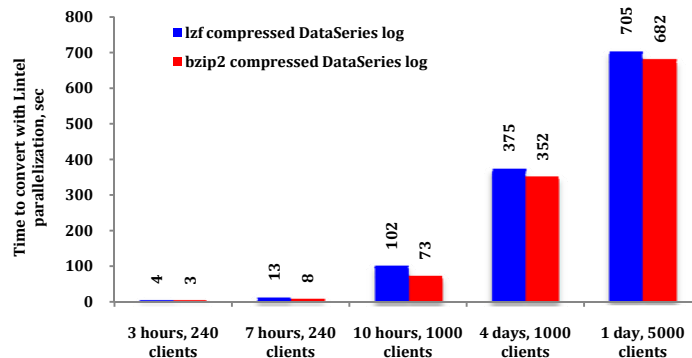


Figure 6. The time required to convert DataSeries logfiles to plain text logs.

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 [21], Analog [22], etc.) before choosing Webalizer [23] 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:

- 1) It is widely used [15,16] 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 [26], but, since the latter takes generally more time to configure and debug, we decided to use Webalizer's C source code 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 plain text logs, but also with the logs stored in a popular compression format.

To take advantage of multicore servers, we first needed to parallelize the single-threaded Webalizer to show the improvements in analysis of DataSeries logfiles. 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 the data only when the results are aggregated into the final report (see below). In this way we can achieve the maximum 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:


```
int initial_extent = (extent_num / thread_id_max) * thread_id;
int final_extent = initial_extent;
if ((thread_id+1) == thread_id_max) final_extent = extent_num + 1;
else final_extent = (extent_num / thread_id_max) * (thread_id+1);
```
- 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 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 months (up to 12). 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 parallel version 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 month into the report once all the log records for this month have been processed. In the multi-threaded version we need to detect when the month has been processed in all of the threads and only then generate the monthly 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 month (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 month records. As a result, the sum of all the $t_site[t]$ will always be bigger or equal than the entire site's value.

- Although each analyzing thread will be able to correctly determine the number of unique visits $t_visit[t]$ within its piece of the logfile, the total number of visits per month 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`).

Parameter	Description
<i>Hits</i>	Any request made to the server, which is logged is considered a 'hit'. This number represents the total number of requests that were made to the server during the specified report period.
<i>Files</i>	Some requests made to the server require the server to send an object back to the requesting client, such as an html page or image. When this happens, it is considered a 'file' response, and the files total is incremented. There can also be requests that won't result in an object being returned, but they will still generate a response (e.g., a redirection, not modified, client or server error). The relationship between 'hits' and 'files' can be thought of as 'incoming requests' and 'outgoing responses'.
<i>Pages</i>	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.
<i>Sites</i>	Each request made to the server comes from a unique 'site', which can be referenced by a name or ultimately, an IP address. The Webalizer fully supports both IPv4 and IPv6 DNS lookups, and maintains a cache of those lookups to reduce processing the same addresses in subsequent runs. The cache file can be created using either the stand-alone 'webazolver' program, or The Webalizer (DNS) Cache file Manager program 'wcmgr'. The 'sites' number shows how many unique sites made requests to the server during the reporting time period.
<i>Visits</i>	Whenever a request is made to the server from a given IP address (a site), the amount of time since a previous request by the address is calculated (if any). 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 (both for the site, and the IP address). The default timeout value is 30 minutes.
<i>KBytes</i>	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 monthly report generated by Webalizer.

To adjust the number of sites and visits for each month, we modify *the host hash table* maintained by Webalizer. During the processing of each record, Webalizer uses the host field of each log record to generate the hash value and then uses it as an index for the special array `sm_hstab[h]`. The program first checks if such a host exists in the table. If it is a new host (there is no element in the table with such a hash), it adds the new element with the specified hash, host string and the time of the corresponding record. If such a host is already in the table, the program updates the time of the found element (so that the element in the table always has the time of the latest visit from this host). When parallelizing Webalizer, we first created several host hash tables, one for each processing thread, and then added one more field to the table element: the earliest time of the host visit. This field is only assigned once when the element is being added to the table and then stays intact until its analyzing thread finishes the piece of logfile assigned to it. Using the hash tables created by each thread, we then adjust the number of sites and visits as follows:

- We first obtain the sums of $t_site[t]$ and $t_visit[t]$ for all analyzing threads for the current month.
- We then check the hash tables of all threads for the identical host values. If a pair of identical nodes is found in hash tables of two threads, then the total number of sites is decremented by 1.
- We also check the initial and the latest timestamps of the identical nodes. If they differ by less than half an hour, the number of visits is decremented by 1.

The experimental results of testing Webalizer are shown in Figure 7. Besides the varying level of parallelization (the number of parallel threads of execution), we also tested several different schemes of parallelization as described below.

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

In the *unified* scheme (Figure 7a), 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 7b) are essentially the same, except *unified* does not take advantage of the pre-parsed fields already present in the `DataSeries` file. 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 Webalizer 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.

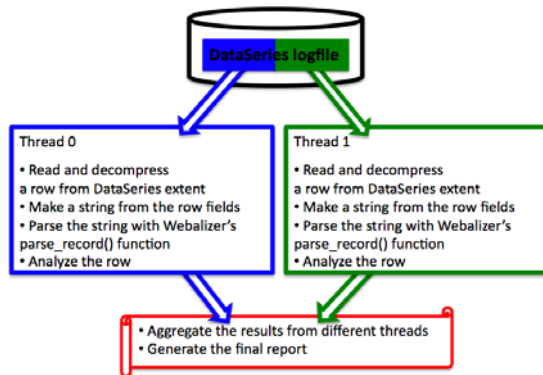


Figure 7a. Flow diagram for *unified* parallelization scheme.

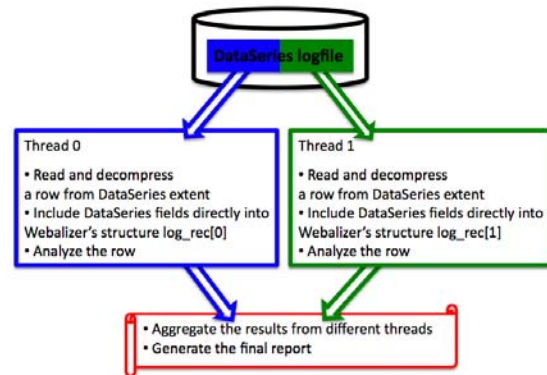


Figure 7b. Flow diagram for *unified with parser* parallelization scheme.

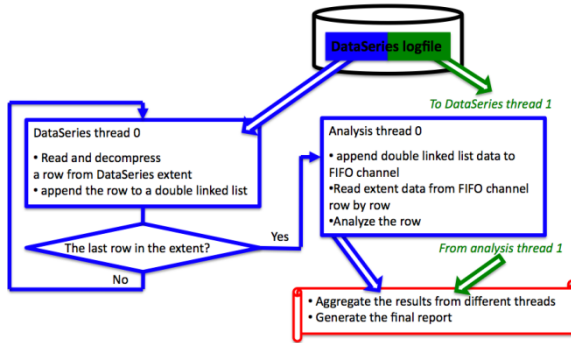


Figure 7c. Flow diagram for *separated* parallelization scheme.

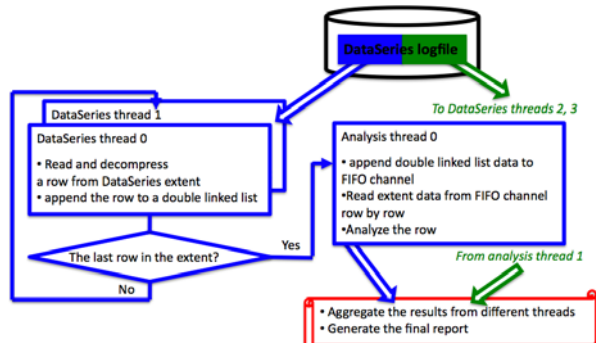


Figure 7d. Flow diagram for *separated with DataSeries sub-threads* parallelization scheme.

Separated (Figure 7c). In this scheme there are two kinds of threads:

- Threads that 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*). We call these *DataSeries threads*.
- Threads that analyze the data provided by `DataSeries` threads, save the intermediate results and create the resulting report. We call these *analysis threads*.

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.

Separated with DataSeries sub-threads (Figure 7d) is similar to *separated*, but this time there are two DataSeries threads per each analysis thread: one DataSeries sub-thread works on odd rows of the currently processed extent, the other one is working 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 decided to test this scheme after we observed that analysis threads in *separated* tests occasionally stalled due to a lack of the data from DataSeries thread (i.e., DataSeries threads can be a bottleneck).

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 plain text logs and text logs compressed with gzip. The analysis of the Apache plain text 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 plain text log with gzip, this reduces the storage requirement but increases the analysis time by 9% (if we don't consider the time it took to create gzip compressed file) and more than doubles the analysis time (if we take into account the time it takes to convert the plain 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 naïve single-threaded approach, when all 24 cores of the multicore server machine 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* would 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 filesystem rather than from local disk on the analysis machine. 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 plain text logs locally. For the DataSeries tests, 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). The results from Figure 9 state that:

- The highest performance was obtained using the parallelized Webalizer with DataSeries (lzf compression), read from local disk (the light green line, 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 (the dark green lines)

- depends on the speed of underlying Ethernet network. With the relatively slow 100 Mb/s network, the traffic quickly reaches saturation, as the 100 MbE link connecting the analysis server to the NFS filesystem becomes a bottleneck (the dotted dark green line). To verify that the network was indeed the bottleneck in this case, we used capstats [27], a package that is shipped with Bro to measure the network bandwidth used by a TCP connection. Appendix B provides the installation instructions for Bro and capstats. This problem disappeared as soon as we switched to the 1 Gb/s network (the solid dark green line). 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.
 - The plaintext analysis of the default single-threaded Webalizer experiences more than 2.5x slowdown when accessing a slow 100 MbE connection rather than a 1GbE one. Similar to DataSeries bzip2 curves, the analysis of the plaintext logfiles, compressed with gzip as a whole, is not affected by the Ethernet speed. 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).

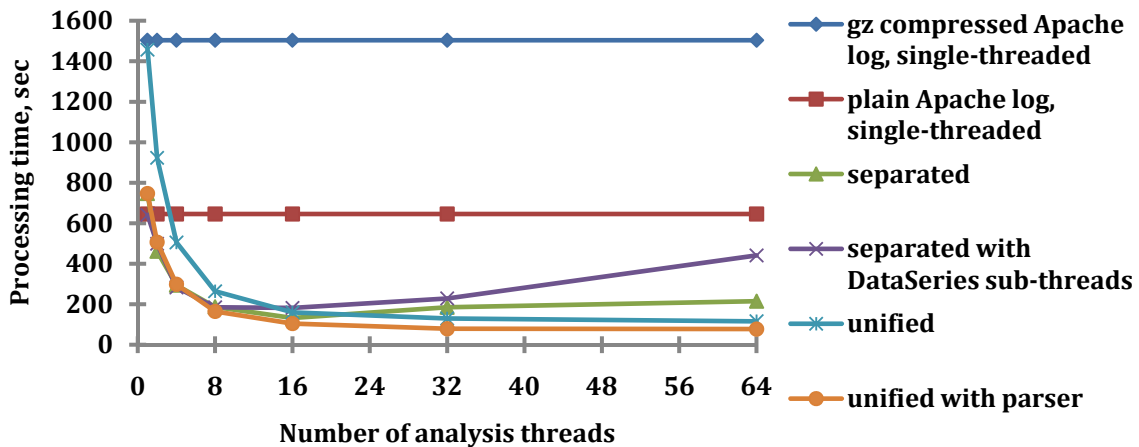


Figure 8. The processing times of Webalizer for 1 day and 5,000 RUBiS clients with different numbers of threads and parallelization schemes.

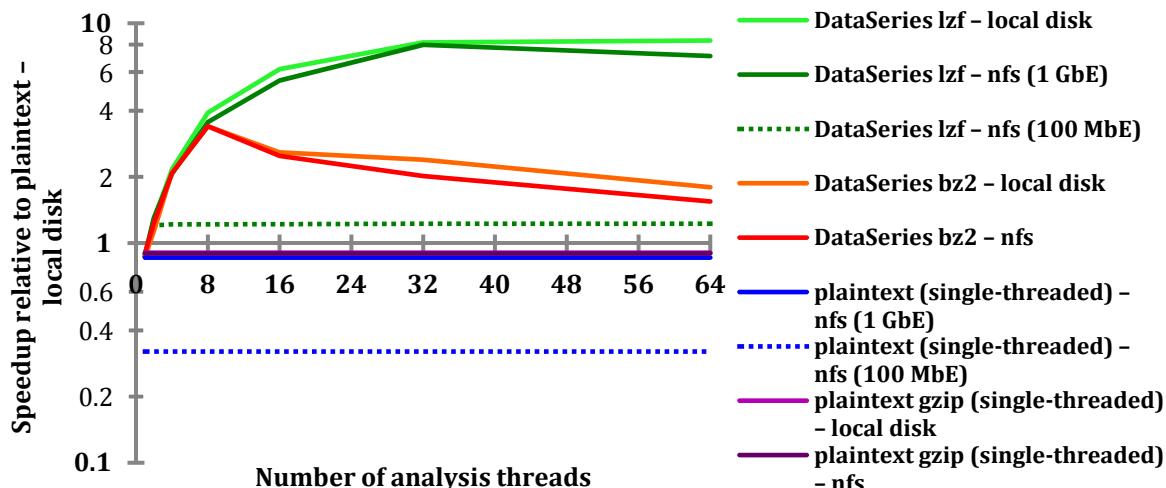


Figure 9. The processing speedup (on logarithmic scale) of Webalizer analysis relative to a single threaded Webalizer set-up parsing uncompressed plain text logs locally with different number of threads and reading from NFS filesystem against reading locally (*unified with parser* parallelization scheme).

4. Improving efficiency of information collection and analysis for Bro-awk

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. Logging of network transactions by the Bro intrusion detection system

Bro is a Unix-based Network Intrusion Detection System (IDS) [28]. 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. Appendix B provides step-by-step installation instructions for Bro on Linux.

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 logfiles. A commonly used Bro module called *conn* produces a log that contains one-line ASCII summaries of each “connection” [30]. 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.

Parameter	Description
<i>Start</i>	The timestamp that corresponds to the connection’s start time – in seconds since the beginning of Unix epoch (there is a Bro utility called <i>cf</i> to get the date and time in human readable format from the timestamp [31]).
<i>Duration</i>	Gives the connection’s duration, in seconds.
<i>Local IP, Remote IP</i>	Correspond to the <i>local</i> and <i>remote</i> IP addresses that participated in the connection, respectively.
<i>Service</i>	The connection’s service (e.g. http, ftp, etc); this is based on well known port number mappings.
<i>Local port, Remote port</i>	The transport-level ports used by the connection
<i>Protocol</i>	Indicates the transportation protocol that was used (e.g., tcp, udp, icmp).
<i>Org bytes sent, res bytes sent</i>	Give the number of bytes sent by the <i>originator</i> and <i>responder</i> , respectively.
<i>State</i>	Reflects the state of the connection at the time the summary was written (which is usually either when the connection terminated, or when Bro terminated). E.g. S0: connection attempt seen, no reply; S1: connection

Flags	established, not terminated; SF: normal establishment and termination; REJ: connection attempt rejected, etc. Reports a set of additional binary state associated with the connection: L indicates that the connection was initiated <i>locally</i> ; 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.

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

```
1280879507.515707 3.533462 163.21.148.176 199.60.16.246 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 [31]) 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 1713 bytes. Because the L flag is absent, the connection was initiated by host 199.60.16.246, and the responding host was 163.21.148.176. When the summary was written, the connection was in the SF state. The connection had neither the L or U flags associated with it, and there was additional information, summarized by the string @47-4187-11.

Adding DataSeries logging to Bro is actually easier than to 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 root>/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 what exact 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 plain test log conn.log reaches 10,000. As is the case with Apache module mod_log_dataseris (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 [1] 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. Some other fields (org_bytes_sent, res_bytes_sent or duration) could be described as variable32 values, since for many transactions Bro puts «?» into them if it cannot determine a numeric value. We decided to replace the «?» in those fields by null values (field-level option opt_nullable) and represent them as either double or int32, because, according to our observations, numerical fields consume less space than that of the corresponding strings.

Table 6 shows the number of records logged by the Bro intrusion detection system with DataSeries modifications (logs were compressed 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 3a.

Since the Bro intrusion detection system only *monitors* the data on the network interface without affecting 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 3b). 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 plain text *conn.log*. The improvement in terms of disk space with bzip2-compressed DataSeries *conn.log* is higher and reaches 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 bzip2-compressed DataSeries file. The average number of lines in DataSeries *conn.log* files for experiments with various duration is provided in Figure 11. Figure 12 shows the time it takes to convert the DataSeries *conn.log* files back into plain text (similar to the experiments depicted in Figure 6).

	30 minutes, 5000 clients		1 hour, 5000 clients	
	Mean	Stdev	Mean	Stdev
Plain text <i>conn.log</i>	230435,5	491,4	458171	205
DataSeries <i>conn.log</i> (lzf compressed)	231557	566	455521	336
DataSeries <i>conn.log</i> (bzip2 compressed)	235967	1360	460002	113

Table 6. The number of strings in the *conn* logfile.
The data for Bro experiments with default plain text logs and with DataSeries modifications and different HTTP load.

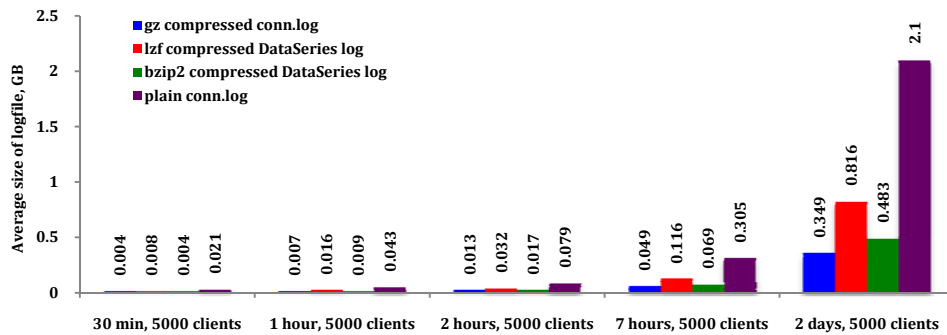


Figure 10. The Improvements in terms of storage space when logging Bro transactions with DataSeries.

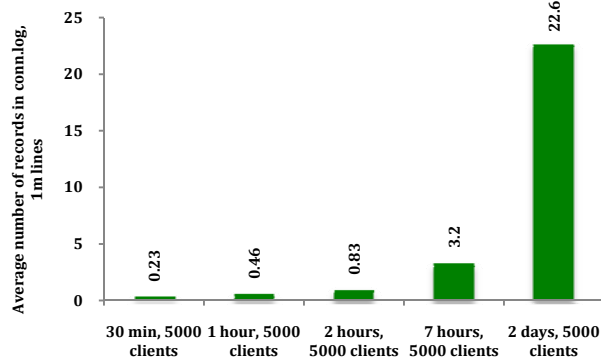


Figure 11. The average number of records (Bro transactions) in DataSeries files of various size.

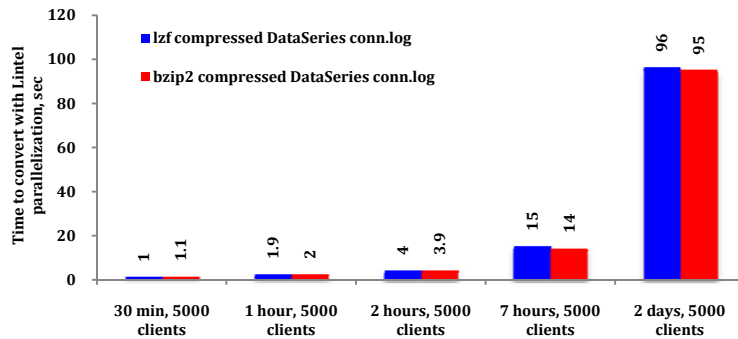


Figure 12. The time required to convert DataSeries logfiles with Bro transactions to plain text logs.

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 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 plain text logs with DataSeries, we performed the following analysis to look for potentially malicious traffic in the Bro logs. For each record in conn.logfile we:

1. Check if the IP address of the originator is not in the list of allowed IP addresses. The list of allowed IP addresses/networks is stored in the `neighbor_nets` Bro variable. For that get the `Flags` field and check its value.
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 plain text 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 (described in Section 2.3), which is provided within DataSeries API.

The performance results for the experiments with RowAnalysisModule/awk analysis over Bro conn.log are provided in Figure 13. For DataSeries tests with RowAnalysisModule, lzf- and bzip2-compressed DataSeries conn.log collected from two-day long experiments with 5,000 RUBiS clients was used. The main results generally repeat those for the Apache log analysis (Section 3.2) with DataSeries' RowAnalysisModule providing 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:

- In 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 files (the solid dark green and red lines) and plaintext files (the solid blue line) is significantly slower (by up to 57%) 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 the effect similar to the DataSeries bzip2 curves for Apache/Webalizer results, namely that the 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 one big array is also increasing.

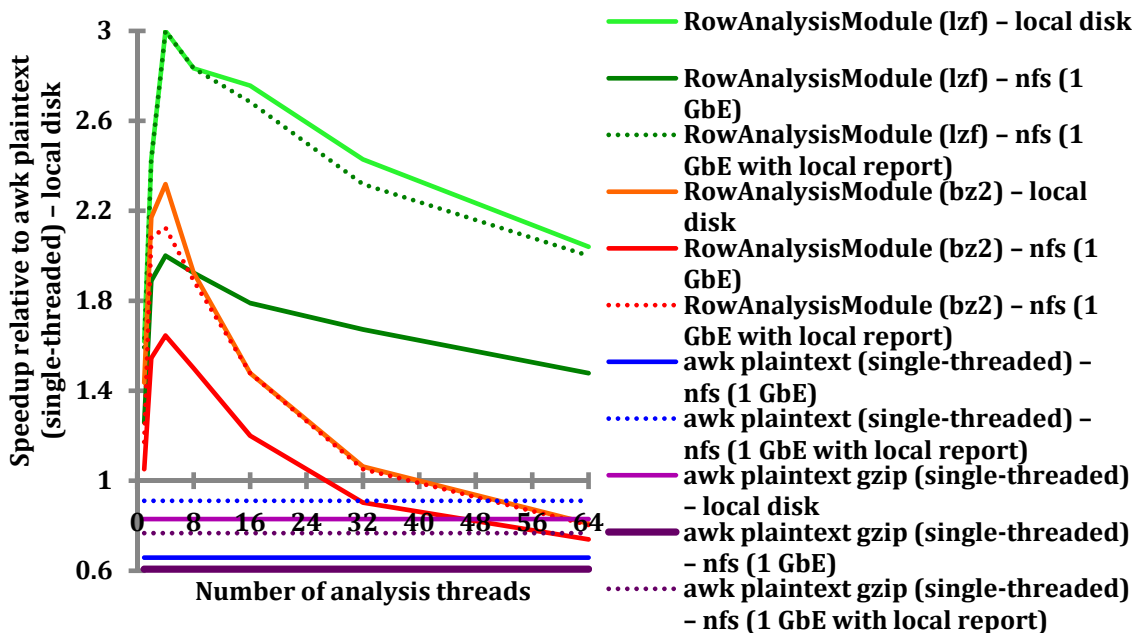


Figure 13. The processing speedup of RowAnalysisModule/awk analysis relative to a single threaded awk set-up parsing uncompressed plain text conn.log locally with different number of threads and reading from NFS filesystem against reading locally (*unified with parser parallelization scheme*).

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 work we used DataSeries as the online logging format for two popular open source applications: Apache web server and Bro intrusion detection system. We modified the Webalizer tool to be able to analyze Web server logs in DataSeries format. We used awk to perform an analysis on Bro transaction logs collected in DataSeries, and also utilized the RowAnalysisModule available in DataSeries toolkit.

We experimentally showed that online logging in DataSeries is a feasible alternative to logging in the current plain text formats. We quantified the benefits of storing and accessing information in DataSeries format relative to the default log format of the chosen applications. We achieved the following results:

- the sizes of the Apache logs decreased by up to 7x (2.6x for Bro) relative to the default plain text log format.
- the time to analyze the Apache logs decreased by more than 8x (3x for Bro).

Our work verifies that DataSeries is beneficial for applications such as the analysis of Web server or intrusion detection system logs. As there are endless possible collection, analyses and storage scenarios that may be of interest to application developers or users, one avenue of future work is to continue experimental studies of those deemed valuable. Similarly, 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.

References

[1] E. Anderson, M. Arlitt, C. Morrey and A. Veitch, "DataSeries: an efficient, flexible data format for structured serial data", ACM SIGOPS Operating Systems Review, Vol. 43, No. 1, pp. 70-75, January 2009.

[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] The Webalizer: free web server log file analysis program. [Online] Available: <http://www.webalizer.org/download.html>
- [5] Tcpdump command-line packet analyzer. [Online] Available: <http://www.tcpdump.org/mirrors.html>
- [6] Bro intrusion detection system. [Online] Available: <http://www.bro-ids.org/download.html>
- [7] Source files with modifications done within this work. [Online] Available: <http://www.sfu.ca/~sba70/files/dataseries/>
- [8] E. Anderson, "Capture, conversion, and analysis of an intense NFS workload", USENIX FAST, pp. 139–152, San Francisco, CA, February 2009.
- [9] Open Source software at Hewlett-Packard Laboratories. [Online] Available: <http://tesla.hpl.hp.com/opensource/>
- [10] DataSeries technical report. [Online] Available: <http://tesla.hpl.hp.com/opensource/DataSeries-tr-snapshot.pdf>
- [11] RUBiS: an auction site prototype. [Online] Available: <http://rubis.ow2.org/>
- [12] Traces from the Internet Traffic Archive. [Online] Available: <http://ita.ee.lbl.gov/html/traces.html>
- [13] Trace data provided by Hewlett-Packard Laboratories. [Online] Available: <http://tesla.hpl.hp.com/opensource/> ("Trace data" section)
- [14] Httpperf: a tool for measuring web server performance. [Online] Available: <http://www.hpl.hp.com/research/linux/httpperf/>
- [15] Uргаonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. 2005. An analytical model for multi-tier internet services and its applications. *SIGMETRICS Perform. Eval. Rev.* 33, 1 (Jun. 2005), 291-302.
- [16] Wood, T., Cherkasova, L., Ozonat, K., and Shenoy, P. 2008. Profiling and Modeling Resource Usage of Virtualized Applications. In *Proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference* (Leuven, Belgium, December 01 - 05, 2008). V. Issarny and R. Schantz, Eds. Lecture Notes In Computer Science, vol. 5346. Springer-Verlag, Berlin, Heidelberg, 366-387.
- [17] RUBiSVA: a virtual appliance of the RUBiS benchmark. [Online] Available: http://rubis.ow2.org/download/rubisva_v1.0.pdf
- [18] How do you create a new Apache module? [Online] Available: <http://ivascucristian.com/how-do-you-create-a-new-apache-module>
- [19] Basic Resource Management in Apache: the APR Pools. [Online] Available: <http://www.apachetutor.org/dev/pools>
- [20] Kew, N. 2007 *The Apache Modules Book: Application Development with Apache* (Prentice Hall Open Source Software Development Series). Prentice Hall PTR.
- [21] Awstats: a free logfile analyzer. [Online] Available: <http://awstats.sourceforge.net/>
- [22] Analog: a free logfile analyzer. [Online] Available: <http://www.analog.cx/>
- [23] Webalizer: a free logfile analyzer. [Online] Available: <http://www.webalizer.org/>

- [24] Moore, T. and Clayton, R. 2007. Examining the impact of website take-down on phishing. In Proceedings of the Anti-Phishing Working Groups 2nd Annual Ecrime Researchers Summit (Pittsburgh, Pennsylvania, October 04 - 05, 2007). eCrime '07, vol. 269. ACM, New York, NY, 1-13.
- [25] Cheng, Y. and Huang, C. 2008. A design and implementation of a web server logfile analyzer. WSEAS Trans. Info. Sci. and App. 5, 1 (Jan. 2008), 8-13.
- [26] 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>
- [27] Capstats – a quick hack to get some NIC statistics. [Online] Available: <http://www.icir.org/robin/capstats/>
- [28] Bro Quick Start Guide. [Online] Available: <http://www.bro-ids.org/Bro-quick-start.pdf>
- [29] Bro IDS Reference Manual: Getting Started. [Online] Available: [http://www.bro-ids.org/wiki/index.php/Reference Manual: Getting Started](http://www.bro-ids.org/wiki/index.php/Reference_Manual:_Getting_Started)
- [30] Bro IDS Reference Manual: Analyzers and Events. [Online] Available: [http://www.bro-ids.org/wiki/index.php/Reference Manual: Analyzers and Events](http://www.bro-ids.org/wiki/index.php/Reference_Manual:_Analyzers_and_Events)
- [31] 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](http://www.bro-ids.org/wiki/index.php/Reference_Manual:_Getting_Started#The_cf_utility)
- [32] Tmpfs: a temporary file storage facility. [Online] Available: <http://en.wikipedia.org/wiki/Tmpfs>

Appendix A

Step-by-step installation of RUBiS (PHP version, not through RUBISVA as its xen image works only on 32 bit machines) for Linux Gentoo. This guide assumes that Apache web server with PHP module and MySQL DBMS are already installed and configured on the server prior to RUBiS installation.

```
# performing sync to get the latest versions of packages from portage tree
emerge --sync
# installing necessary packages first
emerge -uD dev-java/javacc
emerge sysstat
USE="png" emerge -uD sci-visualization/gnuplot

# download RUBiS PHPdistribution from http://forge.ow2.org/project/showfiles.php?group\_id=44
# change DocumentRoot in /etc/apache2/vhosts.d/default_vhost.include to <rubis_root>
apache2ctl restart
# check that http://<machine>/PHP/ works

# in <rubis_root>/Client/rubis.properties change
# ## C-JDBC server information
# +cjdbc_hostname =
#
# to:
#
# # C-JDBC server information
# cjdbc_hostname =

# and update the following fields like so:
# httpd_hostname = <machine>
# ejb_server =
# servlets_server =
# httpd_use_version = PHP
# workload_transition_table = <rubis_root>/workload/default_transitions.txt
# database_server = <machine>

# Here we can specify different HTTP load: the number of clients (web users) connecting to the auction website
# workload_number_of_clients_per_node = 240
```



```

# The duration of the experiment can be controlled here (in ms)
# workload_session_run_time_in_ms = 7200000

# database_regions_file = <rubis_root>/database/ebay_regions.txt
# database_categories_file = <rubis_root>/database/ebay_simple_categories.txt
# monitoring_options = -n DEV -n SOCK -rubCw
# monitoring_gnuplot_terminal = gif

# change config.mk (in <rubis_root>) fields (described in http://rubis.ow2.org/doc/compile_and_run.html) :
# JAVA = /usr/bin/java
# JAVAC = /usr/bin/javac
# RMIC = /usr/bin/rmic
# RMIREGISTRY= /usr/bin/rmiregistry
# JAVADOC = /usr/bin/javadoc
# JAR = /usr/bin/jar

# from RUBiS root:
make client

mysql -u root -p
CREATE USER 'cecchet'@'localhost' IDENTIFIED BY '';
GRANT ALL PRIVILEGES ON *.* TO 'cecchet'@'localhost' WITH GRANT OPTION;
create user "cecchet"
# restart MySQL

# download RUBiS sql dump from http://forge.ow2.org/project/showfiles.php?group_id=44
mysql -u cecchet
mysql# create database rubis;
mysql# exit
mysql -t -vvv -D rubis -u cecchet < <rubis_root>/rubis_dump.sql

# RUBiS will need to be able to authenticate without a password on the machine, for that:
# in the command below agree to default dir and don't specify passphrase. Issue under root:
ssh-keygen -t rsa
# before issuing the following command first check if ~/.ssh/authorized_keys on the web server exists. If yes then just
# append ~/.ssh/id_rsa.pub to it.
# on the web server:
cp ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys
# in order to login without password you need to copy ~/.ssh/id_rsa to ~/.ssh/id_rsa on clients
# you also need to add this key to ~/.ssh/authorized_keys on client for launching local services on client
ssh root@<machine>
# In Linux there is a limit on the number of open file descriptors per process. To increase it for RUBiS under root:
ulimit -n 1000000
# run RUBiS from RUBiS root:
make emulator
# after the test will be finished you can see the results in the latest entry @ http://<machine>/bench/

```

Appendix B

Step-by-step installation of Bro intrusion detection system for Linux Gentoo.

```

emerge -up dev-libs/geoip
emerge -up net-analyzer/tcpdump
tar -xzf bro-1.5-release.tar.gz
In the following instructions:
  <Bro root install> is the root directory of Bro distribution (e.g. ~/bro-1.5.1)
  <Bro home> is the root directory of Bro installation (e.g. /usr/local/bro/)
cd <Bro root install>
./configure
make
if the following error occurs:
X509.cc: In static member function static int X509_Cert::verifyChain(Contents_SSL*, const u_char*, uint32):
X509.cc:195: error: cannot convert _STACK* to stack_st_X509* in initialization
X509.cc:236: error: cannot convert stack_st_X509* to _STACK* for argument 1 to int sk_push(_STACK*, void*)
X509.cc:262: error: cannot convert stack_st_X509* to _STACK* for argument 1 to void sk_pop_free(_STACK*, void
(*) (void*)
download the following patch:

```

<http://tracker.icir.org/bro/raw-attachment/ticket/247/x509-stack-fun.diff>
and apply it (you might need to specify the filenames explicitly at patch prompt when issuing the following command):
patch < x509-stack-fun.diff
make install

```
cd scripts/  
make install-brolite  
./bro_config  
miga scripts # ./bro_config  
Automode not enabled
```

Running Bro Configuration Utility

Configure settings in bro.cfg? (YES/no) [YES]
Values enclosed in '['s are the default value set if you hit return.

```
**** Detected previous bro.cfg file ****  
May I use /usr/local/bro/etc/bro.cfg for defaults? [Y/n]Y  
Sourcing /usr/local/bro/etc/bro.cfg for defaults.  
Detected an old local.site.bro, saving it to local.site.bro.save  
Log archive directory [/usr/local/bro/archive]  
User id to install and run Bro under [root]  
Interface name to listen on. The default is to use the busiest one found. [eth0]  
Site name for reports (i.e. LBNL, FOO.COM, BAZ.ORG) [SOMESITE]  
Starting time for a report run (0001 is 12:01 am and 1201 is 12:01pm) [0010]  
How often (in hours) to generate an activity report [24]  
Email reports? (YES/no) [YES] no  
error: "net.core.rmem_max" must be of the form name=value  
error: Malformed setting "="  
error: "16777216" must be of the form name=value  
ERROR: Can't change value, entry exists in /etc/sysctl.conf!  
May I guess your network configuration for you? [YES] YES  
Checking network  
Running localnets script to determine the local network range ...  
This will take about 20 seconds  
Capturing packets .... done.  
Analyzing dump file.....[1]+ Done tcpdump -n -i $BRO_CAPTURE_INTERFACE -w  
/tmp/bro_config.tcpdump.file.$$ > /dev/null 2>&1  
done.  
You don't appear to be running on a DMZ (found more then two (2) hardware  
address. Please edit local.site.bro to reflect your correct network parameters  
Bro Configuration Finished.  
Press any key to now to continue.
```

```
cd back to <Bro root install>/scripts  
vi bro.rc  
comment the following line (at the end of the script):  
exit $?
```

choose the starting policy for bro (some file with .bro extension, usually brolite.bro as it incorporates all the rest) from
/usr/local/bro/share/bro

```
then  
vi bro.cfg  
replace the following line:  
BRO_START_POLICY="miga.bro"  
with:  
BRO_START_POLICY="/usr/local/bro/share/bro/brolite.bro"
```

```
cp bro_user_id /usr/local/bro/etc/  
cp bro.rc-hooks.sh /usr/local/bro/etc/  
cp bro.cfg /usr/local/bro/etc/
```

```
start Bro:  
./usr/local/bro/etc/bro.rc start  
stop Bro:  
./usr/local/bro/etc/bro.rc stop
```

test analysis of an offline tcpdump trace (the RUBiS client should be executed on the machine, different from the one on which the Apache web server with the auction website is installed):

```
tcpdump -w ~/tcpdump_trace.out
```

```
BROHOME="/usr/local/bro"  
BROPATH="${BROHOME}/site:${BROHOME}/policy"  
/usr/local/bro/bin/bro -r ~/trace.out ~/quatchi.bro  
/usr/local/bro/bin/bro -r ~/tcpdump_trace.out /usr/local/bro/share/bro/http.bro
```

install capstats (shipped with Bro):
cd ~/bro-1.5.1/aux/broctl/aux/capstats
make install

measure the bandwidth on eth0 each second:
/usr/local/bro/bin/capstats -l 1 -i eth0