The Rubicon workload characterization tool

Alistair Veitch and Kim Keeton
Storage Systems Deparment, Hewlett Packard Laboratories,
1501 Page Mill Road, Palo Alto, CA 94304, U.S.A.

Abstract

We describe the design, implementation and usage of Rubicon, a tool for the characterization of I/O workloads. Rubicon provides a rich set of operations on I/O traces, and was designed to be easily extended through the addition of new analysis functions and reporting methods. The design of Rubicon makes it simple to adapt to multiple types of workload characterization, and allows the user to focus solely on the analysis task at hand, rather than the ephemera of the analysis framework. We describe several applications of the Rubicon tool that demonstrate its flexibility and ease of use. We believe that the structure and functionality provided by a tool like Rubicon should be applicable and useful to the analysis of any kind of traces, and that building such a tool is more useful than the series of "one-off" standalone analysis programs generally constructed for this task.

1 Introduction

Workload characterization is important for a variety of reasons. The I/O behavior and requirements of modern applications are poorly understood. The buyers of large, enterprise-scale applications often have very little idea of the storage requirements of these applications, even though storage is a major, and increasing, percentage of the total system cost. Buying decisions are often made using "rules of thumb", based on simplistic metrics such as capacity and expected number of I/O's per second. Compounding this problem, the vendors of storage systems may themselves have very little knowledge of the behavior of their systems under a variety of workloads. Applications may also vary extensively in their behavior, depending on the final use to which they are put (e.g. transaction processing vs. decision support in the case of a database system) or the configuration of other system components, such as the amount of available memory.

Because workloads are poorly modeled, the designers of disk array hardware have few means of validating their design decisions with realistic workloads. As an example, architecting for large cache sizes will not necessarily help all workloads, and it may be more worthwhile to invest in a faster backplane or disks. If it were possible to accurately characterize workloads from a variety of applications, then synthetic workload generators could be used to reproduce these workloads, in order to better test hardware designs [Ganger95].

Given a storage system such as a large disk array, it is very difficult to determine how to configure that system without accurate knowledge of the workload. For example, without information as to the sequentiality, mix of read/writes etc., present in a workload, it is hard to determine which RAID levels or stripe sizes [Patterson88] to use.

A final reason for workload characterization is that of monitoring. Once a system has been installed it is necessary to monitor it to ensure that it is meeting its performance requirements, or if the workload requirements have changed. If they have, then this information can be used in future tuning or reconfiguration of the system.

Figure 1 illustrates these uses of Rubicon, our workload characterization tool, and it's relationship to other storage system management tools. One of the goals of our research was to build a tool that was general enough to fulfill each of these roles, while providing good performance and maintaining flexibility for new applications. While several researchers have instrumented systems to measure and analyze I/O and file workloads, the programs to do the analysis have generally been custom-built, each performing it's own independant processing, or they do not support the extensibility of Rubicon. Rubicon contains a rich set of features that can be combined to perform many manipulations on I/O traces[1] as they are analyzed. The reporting of results is decoupled from the analysis itself, enabling the user to easily use the most convenient reporting format(s), whether it be a spreadsheet, graph, table or custom workload

1. Rubicon relies on the system being measured to be able to generate traces of all the I/O activity on that system. We have specified a trace format that we believe captures all the important information about any given I/O. While the exact details of the trace format and the methedology of how it is gathered are not important for understanding Rubicon, those interested can find further details in Appendix A.

description language. Rubicon is extensible, in that new analysis and reporting functionality can be easily added into the characterization framework. It is this definition of the framework for a configurable, extensible workload characterization tool which is the major contribution of this paper.

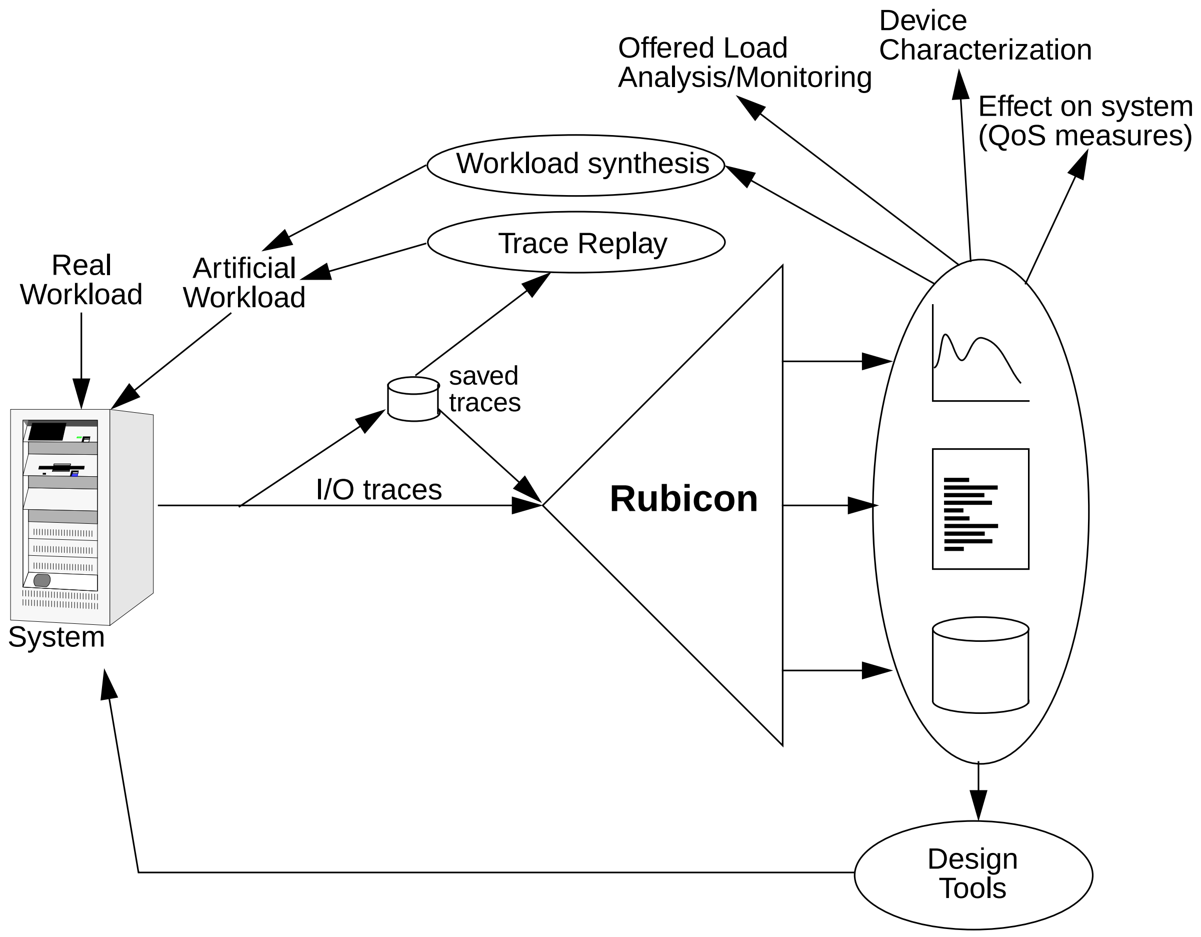


Figure 1: Some roles and uses of the Rubicon workload characterization tool

The remainder of this paper is organized as follows. Section 2 describes the issues in I/O workload characterization, and specifies the requirements for a system for accomplishing this task. Section 3 goes into detail on the design of Rubicon, the system we have developed to satisfy these requirements. Section 4 discusses how the system we made is configurable for a number of different analyses. Section 5 describes Rubicons extensibility. Section 6 summarises the design of Rubicon, and shows how the design goals and requirements are met. Section 7 describes some of the applications we have made of Rubicon. Section 8 describes the systems performance, and compares this to custom analysis tools. Section 9 discusses related work. Section 10 concludes with our thoughts on what we have learned in the development of the system, the lessons we believe are applicable to related systems, and directions for future work.

2 Issues in workload characterization

On the surface, the problem of workload characterization itself is simple – read a trace describing the I/O's performed by an application, analyze it in some way, and output results. Although this approach is sufficient for simple global characterizations based on all I/O's, it has many drawbacks when more complex analysis is desired.

The first problem is that of partitioning the trace into separate parts for processing. This can be necessary for many reasons, including studying the workload on each device, examining various time slices of the trace, or singling out the I/O caused by a single process. In practice, this is often done by filtering the trace prior to analysis, or by incorporating a set of `if... then...` statements into the analysis program. These methods tend to complicate the analysis environment and code, leading to a longer analysis process and multiple trace files that need to be organized, or poorly structured code that is often difficult to modify for a related, but different, application.

Another issue is how to incorporate new types of analysis. In this case, the new functionality must either be incorporated into the original program, or a new program developed. If at some later point in time yet another type of analysis is desired, the same decision must be made. Over time, this can result in unnecessarily large and complex programs, or in a multitude of smaller programs, each of which contains some functionality identical to, but separated from, it's parent. In either case, the maintenance and development problems associated with the code base are exacerbated.

Thirdly, one stage of analysis or processing might depend on a second. A simple example might be the production of a set of summary results from a larger set of results produced by the main analysis program. Once again, this is typically accomplished through the use of several different programs. Although this can be an effective technique, it also tends to complicate the analysis environment.

Another disadvantage is that most analysis tools provide only one type of output. Many users have a need for the same analysis presented and used in different ways. Most systems use separate programs for the conversion of results, when what is really required is a tool that can, when necessary, simultaneously generate different output formats from the same set of internal results. Once again, this raises the issue of one program vs. many.

Another problem is that of incorporating information about the details of the storage configuration for the machine on which the trace is gathered. Details about the logical and physical storage devices on the system are vital for the correct interpretation of trace records. For modern enterprise-scale applications, this system configuration can be extremely complex. Consider that large database systems may have many hundreds of tables, spread over a large number of logical volumes, which can in turn be composed from a number of LUNs[1]. In this case, several types of characterization are possible, depending on what level and type of workload is being examined. The database architect may be concerned with the amount of I/O to each of the individual tables, while the system administrator may wish to ensure that the database traffic is evenly spread over the logical volumes or LUNs. Because of this system complexity and range of demands, any characterization tool must be easily configurable for many different system layouts. Without this capability, the user is faced with the development of customized code for every system on which the workload analysis must be performed.

Based on these problems, we can specify the following requirements for a more generic workload characterization system:

- **Single image** – there must be a single program and source base.
- **Trace manipulation** – must have built-in primitives for manipulating I/O traces, particularly for filtering.
- **Configurability** – the system components can be easily and dynamically (re)configured for different target systems or analyses.
- **Multiple report formats** – the ability to report results in several different formats, independent of the analysis performed.
- **Extensibility** – new analysis algorithms and reporting formats can easily be incorporated.
- **Staged analysis** – the results of one analysis stage must be able to be fed into another.
- **Efficiency** – the final system must not be significantly more expensive in execution time than the comparable special-purpose program for the same analysis task.

The next section describes the architecture of Rubicon, a workload characterization system that satisfies all of these requirements.

3 Rubicon design

Logically, there are two separate stages to workload characterization. The first is the analysis phase itself, where the workload is examined, and various results computed. The second is the presentation of the results, which may vary depending on their ultimate use, e.g. visual display, further analysis or archival purposes. Rubicon provides a set of C++ objects that can be combined into different configurations for these tasks, and which together satisfy the requirements specified in the previous section. The following subsections describe each of these objects.

1. Disk arrays are typically split into a number of *logical units*, or *LUNs*; each LUN is a set of disks that is accessed independently of other LUNs in the array. High end disk arrays can potentially have several thousand LUNs.

3.1 Analysis objects

Each of the analysis objects in Rubicon performs a separate function on trace records. These objects are connected together in a directed acyclic graph (DAG), with the trace records entering at a designated root, flowing through each of the nodes, and terminating in analysis objects. There are four of these base object types:

- **Analyzers** do all the "work" in Rubicon – given a stream of records, they analyze those records and store their results for future use. The current version of Rubicon has approximately fifty different types of analyzers, that perform many different types of analysis ranging from simple rate measurement (I/O's per second) to correlations between I/O streams, spatial and temporal locality measures and self-similarity properties [Willinger95, Gomez98]. Rubicon is specifically designed so that the addition of new analysis modules is as easy as possible. In particular, the user is freed from worrying about how to incorporate new analysis functionality, and can concentrate solely on the analysis algorithms themselves.
- **Multiplexers** read trace records and multiplex these to several outputs (i.e. nodes in the DAG). Multiplexers are used to create multiple flows of records, so that different types of analysis can be done on each record stream. As such, they essentially serve as connectors between the other component types.
- **Filters** read a stream of trace records, and output selected records, based on a filter specification, which is written in a small, domain-specific language[1]. Filters are used to select out subsets of the full trace for analysis. A typical use is to combine Multiplexers and Filters to generate a number of logical trace record streams, say one for each disk on a machine.
- **Transformers** perform simple transformations on trace records. The most frequently used transformer in the current Rubicon system transforms the physical storage device offset specified in the trace to a logical volume offset[2]. Other examples of Transformers include their use for simulating the effect of application changes, such as coalescing several sequential physical I/O's into a single I/O or changing logical device identifiers.

Figure 2 illustrates a very simple Rubicon configuration using the above objects. For the sake of discussion, we assume a system with two disks, for which the following analysis is desired:

- IO rate (IOs per second) for both disks
- Bandwidth utilization (MB/s) for disk 1 and the I/O system as a whole

As should be clear from the figure, each of the various Rubicon objects can be configured together in a variety of ways. In particular, we wish to emphasize that Rubicon makes it possible to analyze for multiple properties, over multiple logical analysis domains (two disks and the I/O system in the example) simultaneously. This approach can be contrasted with the approach of building multiple tools for filtering and/or analyzing. Rubicon's design reduces the amount of both human and machine time required over such a system.

3.2 Result manipulation objects

While the objects described above handle the flow and analysis of records, another set of objects are used to store and manipulate the results:

- **Attributes** store the results calculated by Analyzers. Typically, though not necessarily, such results are named sets or tables of numeric results.
- **Flows** store the Attributes calculated by a set of related Analyzers. In the most common Rubicon configurations that we use, each flow typically represents the analysis for a different storage device. An example of this type of configuration is depicted in Figure 2, with three flows, one for each device, and one for the overall I/O traffic. Flows provide the interface through which Attributes can be communicated to different parts of the system. In particular, an Analyzer can read Attributes computed by other Analyzers and use these to compute new results. For example, a tool developed for storage system configuration [Alvarez01] requires information on the time overlaps between different flows. This can be accomplished by having an analyzer that computes when a flow is active or inactive, storing a vector of these times within an Attribute, and having other Analyzers read these Attributes after they have been computed.

1. See Appendix B for the full filter specification language.
2. Logical volumes are used in a number of operating systems to build up logical "disks" from real disks or LUNs. They provide the system administrator with a layer of indirection through which to manage the raw devices.

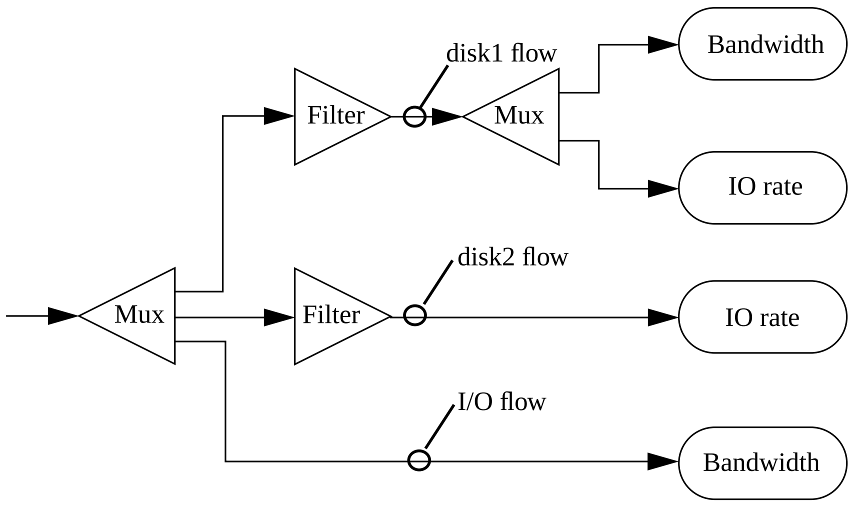


Figure 2: A simple analysis object layout. Arrowed lines represent the flow of trace records, left pointing triangles represent Multiplexers, right pointing triangles represent filters and ovals represent Analyzers.

- **Reporters** transform the results stored in Attributes into the desired output format(s). Given the raw data and structuring information stored in an Attribute, Reporters generate output in the desired format. We currently have reporters that generate flat data files, gnuplot [Williams98] input files (for graph generation), spreadsheets and a specialized workload description language. Just as adding new Analyzers is simple, so is the addition of a new reporter format. As arbitrary Reporters can be linked with each Attribute, it is possible to have the same analysis configuration generate multiple output formats from each set of results.

Separation of the concepts of result storage, calculation and reporting allows the independent development of modules for these purposes, and enables "mix and match" combinations, to suit the particular needs of the user at any time. Figure 3 shows this for the reporting of the results calculated by the analysis configuration depicted in Figure 2. Note that multiple reporters can independently report on the same Attributes. The interaction between Attributes and Reporters is two way, in that Reporters are configured to know which Attributes, in which Flows, to report, while Attributes call back to Reporters with their results. Their interfaces must be constructed in this way, as only Attributes know the type and structuring of the results. Examples of objects that use these interfaces are shown in Section 5.3.

4 Configuration

The various objects described in the previous section must have individual instances of each instantiated, that is the specific Analyzers, Reporters etc. must be specified. These objects must then be connected together into the DAG describing the record flow and the Reporters must be told which Attributes to report. All Rubicon configurations are built through a Tcl [Ousterhout94] script, which is evaluated at program startup. The script instantiates all the objects making up the analysis DAG, together with each of the Flows and Reporters. In this fashion, the Rubicon user gets the flexibility of a scripting language in how the overall system is configured, while retaining the performance of compiled code. Each of the various object types has a Tcl command associated with it. An example configuration file for the layouts depicted in Figures 2 and 3 is shown in Figure 4.

For any reasonably sized system, it is impractical to expect humans to be able to generate configuration files. This is because of the system complexity – configuration files for large systems can easily run to many thousands of lines. We have found that, in practice, two common styles of generating Rubicon configurations have developed. First are those configurations that are generated by other tools. In this case, the secondary tool (usually a script of some kind) examines the system and writes a configuration file, which can later be read by Rubicon. The second style is to have the configuration file itself be a true script, which can directly execute the Rubicon configuration commands. The difference between these two styles is subtle, but significant. The first essentially views the Rubicon input as a special purpose, domain-specific language for describing workload characterization configurations, and typically appeals to

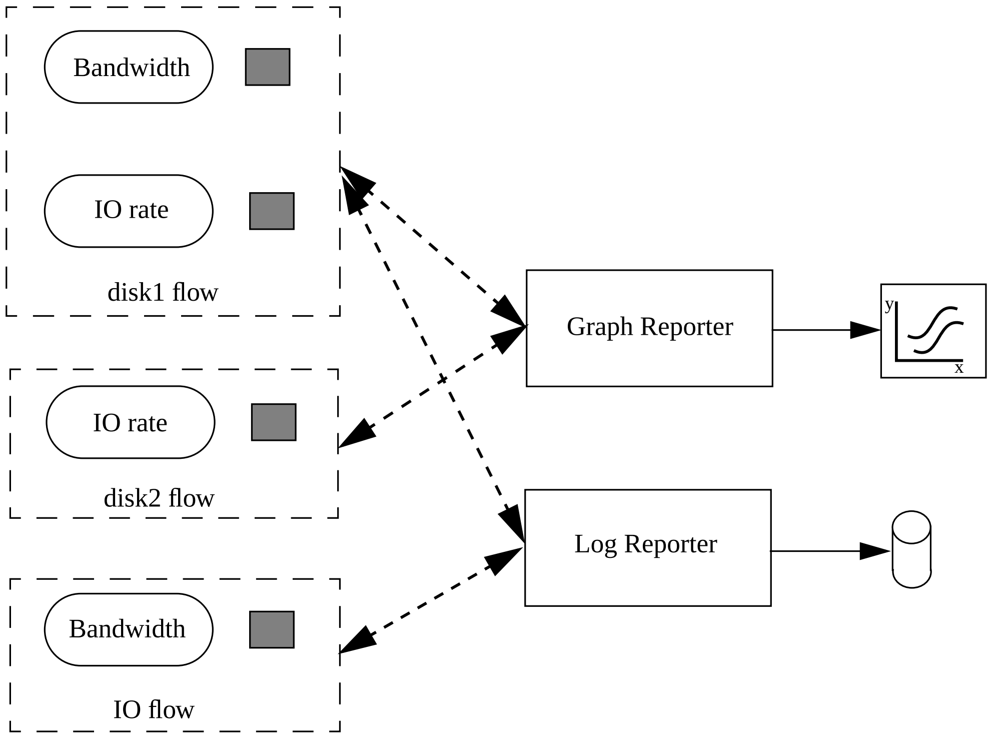


Figure 3: Flow of results and output in the Rubicon system. The diagram can be thought of as extending Figure 2. Each analyzer (oval) is bound within a logical Flow (dashed box) together with the results it computes (Attributes, shaded boxes). Through a generic reporting interface (dashed arrowed lines), Reporters (boxes) obtain these results, and report them. In the case shown, the Log Reporter logs its results to a disk file, while the Graph reporter will display a graph on the user's screen or generate a data file for plotting.

users who prefer not to work in Tcl or wish to record the actual configuration file used, while the second allows the maximum efficiency in generating configurations, as it removes the extra step of generating an intermediate file. In general, users tend to use both styles, dependant on their skills and current requirements. In either case, having the full power of a general scripting language available for configuration purposes allows for maximum flexibility, and has shown itself to be a valuable part of the overall system design.

5 Adding new functionality

One of the key requirements for Rubicon is the ability to add new functionality. Because of the separation of concerns in the design, Rubicon makes the addition of new analyzers, attributes and reporters extremely simple. Essentially, new objects of these types register callback interfaces, which are then linked to other objects as specified by the configuration script. The code to make new objects visible as configuration commands is automatically generated by a system we have developed for linking C++ objects into Tcl code, similar to those described in [Golding94, Heidrich95], although considerably more automated [Veitch01]. Note that although Rubicon contains the code for many different analyzers and reporters, only those specified in the configuration are instantiated by the system. Once instantiated, they exist independently of one another. The means for doing this are detailed below. For clarity and ease of exposition, some of the minor details of the class designs and Tcl interfaces have been left out of this discussion.

5.1 Attributes

Attributes must be calculated by Analyzers, and report themselves to Reporters. They are essentially only a store for data, with one method, `report`, which is used to accomplish the reporting. The `report` method will be called by a Reporter object (which passes a reference to itself as the first argument). When it receives this call, the Attribute can make a sequence of call-backs to the Reporter object with the desired values.

As an example, we will show the code for a very simple Attribute that contains information on the average request

```
# First create the three Flows. The argument to the Flow command is
# the name to give the flow.
set flow1 [Flow disk1]
set flow2 [Flow disk2]
set flowIO [Flow IOsystem]

# Now create the Analyzers. The second argument to each Analyzer is
# the Flow to which the Analyzer should be attached.
set disk1_bw [BandwidthAnalyzer $flow1]
set disk1_rate [RateAnalyzer $flow1]
set disk2_rate [RateAnalyzer $flow2]
set IO_bw [BandwidthAnalyzer $flowIO]

# A Mux object, for sending records to each of the disk1 Analyzers.
# The Mux command has one argument, a list of the destination
# objects.
set disk1_mux [Mux "$disk1_bw $disk1_rate"]

# two Filters, one for each disk. The first argument is the filter
# expression, the second is the output object.
set disk1_filter [Filter "deviceNo = 1" $disk1_mux]
set disk2_filter [Filter "deviceNo = 2" $disk2_rate]

# The head of the DAG is a Mux, which replicates the record stream
# three times (once per disk, once for all IO’s)
head [Mux "$disk1_mux $disk2_filter $IO_bw"]

# Finally, specify the reporters and the attributes to report.
# Reporters have two arguments, the list of flows, and the list of
# Attribute names.
ReporterGraph "$flow1 $flow2" "Bandwidth IORate"
set rl [ReporterLog "$flow1 $flowIO" "Bandwidth"]
# Set the output file to be used for the Log reporter.
$rl.setOutput "bw.log"
```

Figure 4: Sample Rubicon configuration file. The brackets [] force evaluation of the procedure contained within them, the `set` command sets the named variable to the specified value. Variables are dereferenced by prepending their name with a $.

size of the records in a flow. Essentially, the Attribute only contains a single value. The class would look something like the code shown in Figure 5. Because the class is so simple, containing only one inline method, it can be contained entirely in a header (.H) file. This is the case with the majority of Attributes. Also, note that the only data member of the Attribute is public. In general, since Attributes have such a simple purpose, this is acceptable when considered against the cost of requiring some number of methods to set and/or get simple data values. This may not be true for Attributes that store more complicated data. Since the Attribute only has a simple structure, the report method is similarly simple. In general, all that has to be done is to communicate the values computed, and their structure, to the Reporter. This is accomplished through the use of a number of `out` methods, which are overloaded to be capable of processing multiple types, including lists, arrays etc. We have found that this interface is rich enough to report complex data types in a number of different formats, while retaining the flexibility and ease of use of a generic system.

```
class AvRequestSizeAttribute : virtual public Attribute {
public:
     double average;
public:
     void report(Reporter &rep) const {
          rep.out(average);
     };
};
```

Figure 5: A simple Attribute class

5.2 Analyzers

Each analyzer object must implement three methods:

- `void processRecord(const SRTio *)`: process a single IO record
- `void startTrace(SRTtime_t)`: called when the trace starts, with the start time
- `void endTrace(SRTtime_t)`: called when the trace is finished, with the end time

The `processRecord` method will be called for each record that the analyzer is to process. The analyzer is free to maintain any statistics or information it needs from the record. The `startTrace` and `endTrace` methods allow analyzers which keep information on time-dependent variables to record these values. It is also expected that the `endTrace` method will trigger the Analyzer to calculate final information, and store this information in an Attribute.

Continuing the previous example, we will consider a simple Analyzer that will compute the AvRequestSize Attribute. This Analyzer might be implemented as shown in Figure 6. Note that the designer of the analysis system only has to worry about how to perform the analysis itself, as this code is isolated from other parts of the system by various object interfaces. Once this code has been developed, it is simply a matter of compiling and linking it into the Rubicon system to make it available to all users. We have found that the ease of developing new Analyzers enables new users of the system to quickly generate their favorite analysis routines, and proceed to experiment with new ones, often providing new insights into some aspects of workload characterization. In general, Analyzers form a kit of parts, from which the user can pick and choose as needed.

5.3 Reporters

Like Analyzers, Reporters can also be easily and efficiently added. A skeleton that shows the major details of the Reporter class is shown in Figure 7. Essentially, the interface is designed to be as generic as possible, with the Reporter being free to output the various result types in whatever way is appropriate.

The separation of concerns (between Analyzers/Attributes and Reporters) allows the user to concentrate solely on the important features of their desired functionality, rather than how it will interact with the rest of the system.

5.4 Summary

Our viewpoint is that Rubicon should handle the low-level details of how to connect objects together and leave the user free to develop their own analysis and reporting functionality. Essentially, users are supplied with a “kit” of parts, and are free to use these in any way they please, including adding new parts to the kit. Because the majority of control operations and the need to provide external configuration commands are hidden from the user, Analyzer and Reporter objects can be more easily developed, as only the core algorithms need be specified, and not the extraneous wrapping code to interface those algorithms to the external infrastructure. We have found that the ease of developing new Rubicon objects enables new users of the system to quickly generate their favorite analysis routines, and proceed to experiment with new ones.

6 Design summary

We previously described a list of requirements for a generic workload characterization system. Revisiting these requirements, we can see that Rubicon satisfies them as follows:

- **Single image** – Rubicon consists of a single program, containing a full set of functionality to manipulate I/O records, perform analyses and report results.

```
class AnalyzerAvRequestSize : public Analyzer {
private:
     AvRequestSizeAttribute attr;
     unsigned count;
     double sum;
public:
     //TIFmethod
     AnalyzerAvRequestSize(Flow &f) :
          Analyzer(f), count(0), sum(0) {
     };

     virtual void startTrace(SRTtime_t t) {
          count = 0;
          sum = 0;
     };

     virtual void processRecord(const SRTio *record) {
          count++;
          sum += record->length();
     };

     virtual void endTrace(SRTtime_t t) {
          attr.average = sum / count;
          setAttribute("AvRequestSize", &attr);
     };
};
```

Figure 6: Source code for the AvRequestSize Analyzer. The “TIFmethod” line marks the following line (the constructor) as being a method that should be exported as a Tcl procedure. The code that accomplishes this is automatically generated by other tools [Veitch01]. The `setAttribute` call registers the computed attribute with the Analyzers flow. The SRT trace record format is described in more detail in Appendix A.

```
class ReporterSimple {
public:
     // Initialization
     ReporterSimple(list<Flow *> flows, // list of flows we report for
                    list<string> attrs) // list of attribute names
          : Reporter(flows, attrs) {}; // base class does initialization

     // bracket the entire report. These methods will be called once as
     // each flow reports it’s attributes
     virtual void startFlowReport(const string &flowName);
     virtual void endFlowReport();

     // bracket the calls for one attribute
     virtual void startAttributeReport(const string &attrName);
     virtual void endAttributeReport();

     // Output data items. One report method for each type
     virtual void report(int);
     virtual void report(double);
};
```

Figure 7: A skeleton Reporter layout. Many of the “report” methods have been elided as they are not necessary for showing the structure (they exist for all primitive types, and many others such as lists, arrays and some often used statistical objects).

- **Trace manipulation** – Using Filter, Multiplexer and Transformer objects, trace records can be arbitrarily manipulated.
- **Configurability** – By using a Tcl-based front-end as a configuration language, Rubicon components can be connected in arbitrary ways. Configuration file generators can be used to produce custom configurations for many different system layouts, or the full power of having a general scripting language can be used.
- **Multiple report formats** – By separating analysis, result storage and reporting, Rubicon allows multiple reporting formats for the same set of data.
- **Extensibility** – New analysis algorithms and reporting formats can be easily incorporated, simply by writing the code implementing the desired algorithms. Tcl procedures for configuring the new functionality are automatically generated. The object interfaces are sufficiently simple that they do not require great programming skills to master.
- **Staged analysis** – Analyzers can read previously computed results from attributes.

In summary, Rubicon provides a flexible, configurable and extensible tool for workload characterization. The next section describes some of our experiences with the use of Rubicon that validate this.

7 Experience using Rubicon

We have successfully used Rubicon for several purposes that required many different types of analysis and reporting formats. We will briefly describe some of the uses to which we have put the system, concentrating on how a flexible workload characterization tool simplifies the user's work, enabling them to concentrate on results, rather than how to obtain them.

7.1 DSS system characterization

One of the things we have used Rubicon for is the characterization of a decision support system. The application we used was a variant of the TPC-D benchmark [TPPC96]. Instead of running each query in series, we selected 12 queries (2-4, 6, 8 and 11-17), selecting those that ran in less than an hour, and ran these in three parallel execution queues, with four queries in each queue. The assignment of queries to queues was done to roughly balance expected execution times. We believe that this is far more representative of real decision support systems, where multiple query types are executed simultaneously, than the baseline TPC-D benchmark. This system was run using Oracle 8.0.5, on an HP9000 K410 server with 4 200 Mhz PA-RISC CPUs and 1 GB of memory, connected to an HP FC/30 disk array [HP98] via a single FibreChannel controller. The system had 158 logical volumes, spread over 8 RAID-1/0 LUNs. Using a front-end utility for generating configuration files, we were able to quickly perform a variety of analyses on these systems without having to spend time customizing programs to match the system layouts, as might be required with other programs.

The first set of results – a sampling of some high-level attributes across the entire I/O system – are shown in Table 1. We can further characterize the workloads by examining the degree of sequentiality present. We can do this by examining the number of sequential "runs", i.e. accesses to consecutive addresses, to each of the logical volumes. Analysis of this type is easily done in Rubicon by filtering on the logical volume identifier in each trace record.

Attribute	Value
Request Size (KB)	43.8
Request Rate (IO/s)	281
Bandwidth (MB/s)	11.9
Read Proportion	0.970

Table 1: Average high level attribute values

Performing the sequentiality analysis reveals that the DSS workload contains large sequential components (the average number of sequential accesses is 145). Table 2 shows the breakdown of this and other attributes for selected logical stores. From this table, it can be seen that the DSS application varies according to the type of table being accessed, with indices (such as "idx2_2") being read in smaller sizes and in a far more random fashion, while general tables and transaction logs use larger I/0's and are far more sequential (the transaction log is entirely sequential, as would be expected).

Store	sequential run size (IO's)	request rate (IO/s)	request size (bytes)
customer	12.0	99.0	50480
lineitem	252	77.5	52880
orders	7.39	82.8	28310
parts	89.6	38.6	52300
idx2	1.17	73.6	8192
redolog	167.0	2.64	15400

Table 2: Average workload attributes for selected DSS stores

7.2 OpenMail system characterization

A second application that we have characterized is OpenMail [XXX]. OpenMail is an enterprise mail system, capable of handling extremely large volumes of email. The system we will discuss ran on an HP K-class server, using EMC 3430 disk arrays, and supported about 12,000 users, of whom approximately 3,000 were active at the time the system was traced.

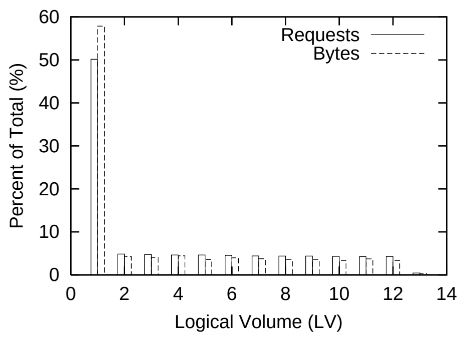
The first level analysis we performed was on the logical volume utilization, shown in Table 3. This shows that one of the OpenMail logical volumes has a far larger request rate than the others. Based on this information, we would

volume	sequential run size (IO's)	request rate (IO/s)	request size (bytes)

Table 3: OpenMail logical volume utilization

expect the system administrator to spread this store over many different physical storage devices. We can verify this hypothesis by examining the workload characteristics on a per-LUN basis. Again, the appropriate configuration file for this analysis can be automatically generated. Full results showing the total percentage of requests and bytes to each logical volume and LUN can be seen in Figure 8. While the logical volume load is skewed, the load on the LUNs is far more balanced, indicating that the system administrator has correctly configured the system to balance the I/O across the available devices (the first 8 LUNs in this case, with every other logical volume assigned to only one LUN). The logical volume and LUN with a very low number of I/O requests correspond to the '/' and '/stand' file systems, which are primarily used for system booting only.

These graphs were generated by running the same analysis on each of the two system viewpoints, and specifying the use of a graphical, rather than tabular, reporter object for these attributes. This illustrates the need for different reporting formats depending on the user requirements – it is easier to "see" that the I/O is balanced in the graph form than from a table.



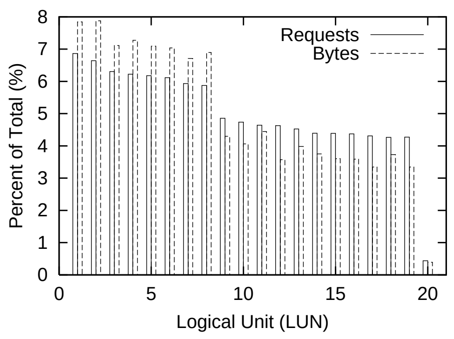


Figure 8: OpenMail logical volume and LUN usage by total requests and total data requested

7.3 Conclusions

In each of the characterization procedures described above (and in many that were not) Rubicon provided a convenient means of both doing initial analysis and making further measurements to verify various hypothesis about the system behavior. In conjunction with knowledge of the system itself, Rubicon has proved its worth as a flexible and versatile tool. Due to a large number of pre-existing analysis and reporting modules, and the ability to quickly build Rubicon configurations based on that of the systems being measured, we were able to rapidly produce relevant results in a variety of different formats.

8 Performance

One issue with incorporating extra features (such as filtering) into the analysis program is the performance overhead associated with these mechanisms. We have measured the overhead of both the filter and multiplexer objects on two systems, one an HP-UX 10.20 system, with 200 Mhz PA-RISC processors, the other a Linux 2.0.36 system on a 300 Mhz Pentium II. The results are shown in Table 4.

System	Filter	Mux
HP-UX	1900	20 + 260*n*
Linux	1200	330 + 50*n*

Table 4: Filter and multiplexer overheads (nanoseconds). Filter times are for a single equality check on a trace field. Multiplexer times are given as a base plus multiplier of the number of outgoing items multiplexed to. All times were measured using processor cycle counters.

As can be seen, these numbers are small enough that their overhead should not be expected to significantly impact the performance of the overall analysis, particularly compared to the alternatives. This conclusion is not as immediately obvious as the scale of the numbers in the table would suggest. To see this, consider the analysis of our DSS trace, which consists of 6.6 million records. If we wish to analyze this trace on a per-table basis, then 158 filters, fed by a multiplexer with 158 outputs, are needed. The analysis of this trace spends almost 90% of its time in code outside of the analysis routines itself, which seems to imply an unacceptable performance overhead. However, consider the alternatives to doing this analysis with a system that did not have these features built in:

- Create a custom program to read the trace file, do the analysis for each record, and split the analysis into 158 parts, based on the store identifier. Even assuming a library of convenient routines for this task, such a program would take some time to construct, and violates our single image requirement.
- Write a program to do the analysis, and a separate program that can split the original trace file. In this case the (94 MB compressed) trace file would have to be read multiple times, much reducing the efficiency of the

system when compared to Rubicon. As many trace analysis systems take this course, it is arguable that Rubicon can potentially give the user an effective performance increase of several orders of magnitude.

In either of these cases, the trace file still has to be split into multiple parts, both for the store filtering and the analysis routines. Each of these is going to consume a significant amount of CPU time, regardless of the system used – on the HP-UX system, a single trace field access method and "if" statement checking this field, takes 1250 ns, or 66% of the total filter time. We believe that the small amount of additional CPU time for a fully general filter object is well worth it, as it results in a more flexible system, which reduces the amount of human time required for experimentation and analysis.

A second consideration when considering performance is the nature of the analysis. For the example above, the analysis being performed was simple, involving only a small number of calculations per record. Many more complex analyses, such as those for spatial and temporal locality, are far more time-intensive, and involve a far larger proportion of the total time than that of filtering.

9 Related work

There are many examples in the literature of workload analysis and characterization based on file system traces [Ousterhout85, Miller91, Ramakrishnan92, Shirriff91, Roselli98] and network tracing [Caceres91, Paxson94, Paxson97], and I/O tracing [Bates91, Ruemmler93, Gomez98]. To the best of our knowledge, these efforts have used custom programs, built for the task at hand. Many projects have developed individual tools for such tasks as filtering traces, but none have integrated the filtering into the analysis system itself. In contrast, users can compose Rubicon's simple building blocks to construct a single system to do complex characterizations. Perhaps more importantly, Rubicon allows users to easily add new analysis functionality.

Despite the fact that a majority of workload characterization studies use ad hoc solutions for trace analysis, several trace analysis tools have been described in the literature. We first describe complete trace analysis systems, and then discuss work related to individual Rubicon components.

Several commercially available products provide offline trace analysis functionality similar to that provided by Rubicon [Grimsrud95, IBM99]. These tools gather traces, filter and analyze the trace data, and provide a facility for data visualization. Analyses range from simple metrics, such as average request size, to more complex "what-if" simulations, such as studying the effects of changing the cache size. In both cases, the types of analyses and data visualizations that can be performed are defined by the existing tool, and the user has no opportunity to extend this functionality to include other metrics or presentations. In contrast, Rubicon provides an extensible framework for analyzing and reporting trace data. Because we believe Rubicon's primary focus is as a workload characterization tool and not a simulation engine, Rubicon does not currently include the ability to perform "what-if" simulations.

Touati and Smith describe TRAMP, an extensible system for trace reduction and manipulation, which includes a subset of the functionality of the Rubicon system [Touati91]. Like Rubicon, TRAMP is structured as a runtime system that uses scripts to specify the desired analyses on the trace. In contrast to Rubicon, the TRAMP runtime invokes a separate standalone program to perform each analysis. New analysis functions can be added by creating new standalone analysis programs. This approach has the benefit of not requiring the tool to be recompiled each time new analysis functionality is added. Like the commercial offerings described in the previous paragraph, TRAMP supports simulation of simple policy decisions. Unlike Rubicon, TRAMP does not provide general filtering functionality, or a modular result reporting infrastructure.

The previously described products and projects provide comprehensive systems for trace analysis. There is also considerable related work for each of the building blocks of the Rubicon system, including filters, analyzers, and reporters.

Previous work on network packet filters [Mogul87, McCanne93] is analogous to the functionality provided by Rubicon's Filter objects. However, Rubicon contains this functionality as only part of a complete, generic, system, rather than as one component of a specialized program, which is the typical usage pattern for packet filters. Also, the common usage pattern of packet filters differs somewhat from the common usage of Rubicon filters. Generally, systems employ a single packet filter per application, which filters only those packets of interest to that application. In Rubicon, it is not unusual for us to build configurations with hundreds of filters, which allows us to carry out simultaneous analysis on many different flows of I/O requests.

Several network monitoring systems, such as HP OpenView, provide support for plug-in modules that can provide custom analysis, somewhat analogous to Rubicons Analyzer objects. These modules must typically perform their own filtering and output functionality – it is not possible to configure this externally to the module itself. In this respect, Rubicon offers much more control to the builders of an Analyzer, in that they only have to concern themselves with the analysis task at hand, rather than the requirements of obtaining the data or presenting the results.

Several papers have explored trace visualization, an area related to Rubicon's Reporters, as a means of exploring large data sets [Heath91, Malony91, Hibbard94, Eick96, Aiken96, Livny97]. These tools use hierarchical abstraction, color, shading, aggregation, and novel 3-D graphing techniques to illustrate data trends. A drill-down capability allows users to interactively examine data at successively more detailed levels. These tools are not generally restricted to displaying trace data, and have been used to visualize datasets as widely varying as weather data to cell images to financial histories. Most of the tools focus on data presentation alone, rather than including a combination of domain-specific analyses and results reporting. The techniques described in these studies could easily be employed as one or more Rubicon reporters to support more interactive exploration of the trace data.

A final area of related work is systems that perform online performance monitoring, such as Paradyn [Miller95] and Pablo [Reed93]. These tools measure the performance of large-scale parallel programs using dynamically generated instrumentation code. They also provide data analysis and visualization functionality.

In summary, we believe Rubicon differs from previous work in the respect that it offers a generic framework for workload characterization. Rubicon's modularized structure enables users to work on exactly the part of the system they need for their work, rather than having to build custom programs for each case. Moreover, its extensibility implies that users can easily add domain-specific analysis and reporting functionality to suit their needs.

10 Conclusions and future work

We have described Rubicon, a tool for workload characterization. Rubicon is different from other systems for this task, as it easily extensible and configurable, offering a large amount of flexibility to the user. In particular, Rubicon offers the following features:

- Set of objects for trace manipulation, including filtering, analysis and reporting methods
- Flexible configuration language, enabling the above objects to be connected together in arbitrary ways
- Easy addition of new analysis functionality
- Easy addition of new reporting (output) functionality

Using these features, it is possible to support many different analysis methodologies and techniques within a single system. We have found that it is better to have a single program that performs all analysis, while also providing support for separating and maintaining the dependencies between analysis phases, than a set of individual programs that are harder to maintain and develop. Similarly, we have found that rather than specify a fixed output format, and then develop tools for converting between these formats, providing support for the easy insertion of reporting modules can better meet the needs of a variety of users.

We have shown Rubicon's utility by characterizing applications on a variety of machines. We believe that our experience in these areas demonstrates that Rubicon is indeed well suited to its task. We intend to use Rubicon to build up an extensive library of workload characteristics, in order to enable a better understanding of the I/O requirements of modern applications. We have in place the facilities to collect and analyze workloads from a number of different enterprise-class applications, notably large mail servers, large data warehousing systems, and a variety of business-specific applications. Once we have gathered these workloads, we hope to use the detailed analysis that results for the design of new disk arrays and storage management software (particularly for capacity planning).

Rubicon is available for interested researchers. See the webpage at `http://www.hpl.hp.com/research/itc/csl/ssp/software/` for further details.

References

[Aiken96] A. Aiken, J. Chen, M. Stonebraker and A. Woodruff. Tioga-2: a direct manipulation database visualization environment. *Proc. Intl. Conf. on Data Engineering*, February 1996.

[Alvarez01] G. Alvarez, E. Borowsky, S. Go, T. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch and J. Wilkes. Minerva: an automated resource provisioning tool for large-scale storage systems. submitted for publication.

[Bates91] K. Bates. *VAX I/O subsystems: optimizing performance*. Professional Press Books, 1991.

[Caceres91] R. Caceres, P. B. Danzig, S. Jamin and D. Mitzel, Characteristics of wide-area TCP/IP conversations. *Proceedings of the 1991 SIGCOMM Conference*, September 1991

[Eick96] S.G. Eick and P.J. Lucas. Displaying trace files, *Software Practice and Experience*, 26(4), pp. 399-409, April 1996.

[Ganger95] G. R. Ganger, Generating Representative Synthetic Workloads: An Unsolved Problem. *Proceedings of the Computer Management Group (CMG) Conference*, pp. 1263-1269, December 1995

[Williams98] T. Williams and C. Kelley, gnuplot: An interactive plotting program, available from `http://www.cs.dartmouth.edu/gnuplot_info.html`

[Golding94] R. Golding, C. Staelin, T. Sullivan, J. Wilkes. "Tcl cures 98.3% of all known simulation configuration problems" claims astonished researcher! *Tcl Workshop*, New Orleans, May 1994

[Gomez98] M.E. Gomez and V. Santonja. Self-similarity in I/O workloads: analysis and modeling. *Workshop on Workload Characterization* (held in conjunction with the 31st annual ACM/IEEE International Symposium on Microarchitecture), Nov. 1998

[Grimsrud95] K. Grimsrud. *Rank disk performance analysis tool*. Intel Corporation White Paper, available from `http://developer.intel.com/design/ipeak/stortool`.

[Heath91] M.T. Heat and J.A. Ethridge. Virsualizing the performance of parallel programs. *IEEE Software*, pp. 29-39, September 1991.

[Heidrich95] W. Heidrich and P. Slusallek, Automatic Generation of Tcl bindings for C and C++ Libraries, *Tcl/Tk Workshop*, pp. 85-94, Usenix Association, 1995

[Hibbard94] W. Hibbard, B.E. Paul, D. Santk, C. Dyer, A. Battaiola and M.-F. Voidrot-Martinez. Interactive visualization of earth and space computation. *IEEE Computer*, pp. 65-72, 1994

[HP98] Hewlett-Packard Company, Model 30/FC High Availability Disk Array – User's Guide, 1998

[IBM99] IBM DFSSMS/MVS Optimizer. Product information available from `http://www.storage.ibm.com/software/opt/optprod.htm`

[Kuenning95] G.H. Kuenning. KItrace – precise measurement of operating-systems kernels. *Software Practice and Exerience*, 25(1), pp. 1-21, January 1995.

[Livny97] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki and K. Wenger. DEVise: integrated querying and visual exploration of large datasets. *Proc. of ACM SIGMOD Conference*, May 1997.

[Malony91] A.D. Malony, D.H. Hammerslag and D.J. Jablonowski. Traceview: a trace visualization tool. I*EEE Software*, pp. 19-28, September 1991

[McCanne93] S. McCanne and V. Jacobson, The BSD packet filter: A new architecture for user-level packet capture, *Proc. of the Winter 1993 Usenix Conference*, pp. 259-269, January 1993

[Miller91] E.L. Miller and R.H. Katz. Input/output behavior of supercomputing applications. *Proc. Supercomputing 1991*, pp. 567-576, November 1991

[Miller95] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam and T. Newhall. The Paradyn parallel performance-measurement tool. *IEEE Computer*, 28(11), pp. 37-46, November 1995.

[Mogul87] J.C. Mogul, R.F. Rashid and M.J. Accetta, The Packet Filter: an efficient mechanism for user-level network code, *Proc. of the 11th Symposium on Operating System Principles*, November 1987

[Ousterhout85] J.K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer and J.G. Thompson. A trace-driven analysis of the UNIX 4.2BSD file system. *Proc. of the 10th Symposium on Operating Systems Principles*, pp. 15-24, December 1985

[Ousterhout94] J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994

[Patterson88] D. Patterson, G. Gibson and R. Katz. A case for redundant arrays of inexpensive disks (RAID). Proc. of the *ACM SIGMOD International Conference on Management of Data*, pp. 109-116, 1988.

[Paxson94] V. Paxson and S. Floyd, Wide-area traffic: the failure of Poisson modeling. Proc. SIGCOMM '94, pp. 257-268, August 1994

[Paxson97] V. Paxson, Automated Packet Trace Analysis of TCP Implementations. *Computer Communications Review*, 27 (4), pp. 167-180

[Ramakrishnan92] K.K. Ramakrishnan, P. Biswas and R. Karedla. Analysis of file I/O traces in commercial computing environments. *Proc. 1992 ACM SIGMETRICS and PERFORMANCE '92 Intl. Conf. on Measurement and Modeling of Computer Systems*, pp. 78-90, June 1992.

[Reed93] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B.W. Schwartz and L.F. Tavera. Scalable performance analysis: the Pablo performance analysis environment. Proc. *IEEE Scalable Parallel Libraries Conf.*, 1993.

[Roselli98] D. Roselli, *Characteristics of file system workloads*. Technical Report UCB//CSD-98-1029, Computer Science Division, University of California, Berkeley, 1998.

[Ruemmler93] C. Ruemmler and J. Wilkes. UNIX disk access patterns. *Proc. of the Winter '93 USENIX Conference*, pp. 405-420

[Shirriff91] K. W. Shirriff and J. K. Ousterhout, A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System. In *Proc. of the Usenix Winter 1992 Technical Conference*, pp. 315-332, January 1991

[Touati91] H. Touati and A.J. Smith. Reducing and manipulating complex trace data. *Software Practice and Experience*, 21(6), pp. 639-655, June 1991

[TPPC96] Transaction Processing Performance Council. TPC benchmark D, standard specification, revision 1.2, November 1996

[Veitch01] The Tcl Interface Functions. HP Labs Technical Report XXX, in production

[Willinger95] W. Willinger, M. Taqqu, R. Sherman and D. Wilson. Self-similarity through high-variability: statistical analysis of ethernet LAN traffic at the source level. In *ACM SIGCOMM '95 Conf. on Communications Architectures, Protocols and Applications* (Cambridge, MA, USA, 1995)

Appendix A: Trace files

Currently, Rubicon uses I/O traces gathered from systems running HP-UX. The HP-UX kernel contains a number of internal instrumentation points. Through a specialized interface [Kuenning95], it is possible to obtain complete information on all system calls and on a number of events in important internal kernel interfaces. One of these trace points is located at the internal block I/O interface. We have developed a tool to read the information gathered at this interface, and use it to gather full information on all I/O that is performed, and store this information in a trace file. Each record in the trace file contains the following pieces of information:

- Time the I/O is enqueued to the device driver[1]
- Time the I/O is sent to the device
- Time the device responded to the I/O
- Type of device
- Device driver queue length
- Type of operation (read or write)
- Size of the I/O
- Device identifier
- Device offset (address of I/O)
- Logical volume identifier (if applicable)
- Logical volume offset (if applicable)
- Process ID of the process responsible
- Thread ID of the thread responsible
- Machine ID of the machine on which the trace record was generated
- A set of flags which are used for miscellaneous information, e.g. whether the I/O is synchronous or asynchronous, from a filesystem or to a raw device, etc.

Appendix B: Filter specification

Filter specifications are made in a simple predicate language. The following fields can currently be tested:

- `createTime` (time request was enqueued)
- `offset` (device address of request)
- `deviceNo` (disk device request is for)
- `lvDeviceNo` (logical volume)
- `operation` (read or write)
- `reqSize` (size of request in bytes)
- `pid` (process ID of process generating request)
- `thread` (thread ID of thread generating request)

1. All times are recorded as microseconds from the start of the trace

- `machine` (machine ID of machine generating request)
- `qlen` (device driver queue length)

Note that these form only a subset of the available fields. The rest will be available in future versions of Rubicon. All of these fields, with the exception of `createTime` and `operation` are specified as unsigned integer values, in either decimal or hexadecimal. There are two possible values for operation, `read` and `write`. Times are specified as `[ [ [ days ] hours: ] min: ] seconds`.

There is one unary operator, `not` (which can also be written as !). Binary operators are <, >, <=, >=, =, <> (can also be written as !=), `and` (can also be written as &&), `or` (can also be written as ||). One tertiary operator, `between ... and ...` is available.

Given this, an example filter might be (don’t ask if it’s a useful filter...):

```
CreateTime between 02:01:45 and 1 03:34:30.454635
and offset >= 50000
and not (operation = read or startAddress = 0xc0800)
```