SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **1**:1–10 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

High Performance Benchmarking with Web Polygraph

Alex Rousskov[1] and Duane Wessels[1]

[1] *The Measurement Factory, Inc., PO Box 380, Boulder, Colorado, 80306, USA*

SUMMARY

This paper presents the design and implementation of Web Polygraph, a tool for benchmarking HTTP intermediaries. We discuss various challenges involved in simulating Web traffic and in developing a portable, high performance tool for generating such traffic. Polygraph's simulation models, as well as our experiences with developing and running the benchmark, may be useful for Web proxy developers, performance analysts, and researchers interested in Web traffic simulation.

KEY WORDS: performance, benchmarking, HTTP, proxies, intermediaries

1. Introduction

The market for Web caching products experienced significant growth in the past few years. Many companies made performance claims regarding their products. However, such claims were essentially meaningless due to the lack of widely available tools, and standard workloads. Attempts to adopt existing origin server benchmarks (such as SPECWeb) failed due to the specifics of proxy cache workloads. Of particular importance are a virtually infinite data set, a very large number of interacting clients and servers, and high request rates. Furthermore, origin server benchmarks generally focus only on throughput, and do not care about hit ratios and object freshness, both of which are important factors for proxy cache performance.

Our group is one of the research and development teams that works on a high quality testing tool for the Web caching community. This paper describes the result of our efforts, the benchmark called Web Polygraph.

Web Polygraph is a versatile tool for generating Web traffic and measuring proxy performance. The benchmark can be configured to produce a variety of realistic and unrealistic workloads, suitable for macro- and micro-level benchmarking. Most of the traffic properties can be changed independently from each other. For example, changes in request rate do not necessarily affect hit ratios.

Received February 10, 2003
Revised August 5, 2003
Accepted August 8, 2003

We dedicate a significant amount effort to the development of industry-standard workloads. Standard workloads, developed in cooperation with caching companies and research groups, make product comparisons feasible and meaningful. These standard workloads are also the recommended starting point for new Polygraph users. A novice user can configure a sophisticated, standardized test* by specifying just a few key parameters such as peak request rate and cache size. Experienced testers can fine-tune standard workload parameters or define new workloads from scratch, usually as a result of many test trials and errors.

Development of Polygraph, and introduction of new workloads, is an interactive process. The complexity of real-world traffic (and hence our models) drives the development of the Polygraph software. Applying the workloads to real proxies gives essential feedback and generates desire to add new features. This paper covers about five years of Polygraph development and experimentation.

Web Polygraph was the benchmark of choice for several industry-wide benchmarking events. The collection of standardized Polygraph-based results is already quite comprehensive and continues to grow. The benchmark is routinely used by companies who market HTTP intermediaries, and by network engineers around the world. It is important to note that political and organizational issues related to developing and maintaining a benchmark of this scale did affect some of our design decisions, but they are beyond the scope of this paper.

1.1. Web Polygraph's Contribution

A good benchmark must generate traffic with realistic fundamental characteristics, such as the distribution of file sizes and request inter-arrival times. Extracting important parameters and patterns from various sources of real traffic constitutes a well established Web characterization activity (see Section 5). We do not claim to have made any contribution in that area, but simply use known characterization results in parameterizing Web Polygraph models. For example, standard Polygraph workloads use a mix of content types ("markup," "images," "downloads," and other), with various distributions of object sizes, including heavy-tailed distributions.

This paper discusses the problems we have encountered while developing a comprehensive performance benchmark, and describes our solutions to those problems. Our contribution is in integrating basic isolated results, adjusting known simulation models, and making them work in a real, high performance production environment. As we learned, integrating simple models is often more complex than characterizing or modeling isolated traffic patterns. In fact, direct application of existing models is often impossible due to conflicts with other models, or incurred performance penalties.

A good benchmark must come with a collection of well designed and tested workloads as well as a database of past results. We have started building the set of workloads to be used with Polygraph. Designing and testing new workloads is a complex and time consuming process that deserves a stand-alone research study. In this paper, we will discuss several workload

*The words "test," "run," and "experiment" are used interchangeably in this paper.

related problems and their possible solutions. We will focus on the relation between desired workload properties and their simulation in a high performance benchmark.

2. Benchmark Architecture

This section presents a high-level overview of the major Polygraph components. Before delving into design details, we outline the objectives that guided our work.

2.1. Design Goals

Several characteristics can be considered standard for any quality performance benchmark: realistic workloads, repeatable experiments, meaningful and comprehensive measurements, and reproducible results. These characteristics are well understood and create the foundation of Polygraph's design. The specifics of Web proxy benchmarking, and our ambition to develop an industry standard, led to the following additional goals:

Scalability One should be able to test any single Web cache unit without changing workload parameters, except for claimed peak request levels and/or cache capacity. Individual caching units may support anywhere from ten to ten thousand requests per second.

Flexibility The tool must be able to produce a wide range of workloads, from low level *micro*-tests to comprehensive *macro*-benchmarks. The tool should easily support new workloads related to proxy caching, such as workloads for server accelerators (a.k.a., surrogates or reverse proxies) and load-balancing L7 switches.

Portability The tool should be usable in a variety of environments, such as different operating systems and hardware platforms.

Efficiency The tool should utilize the available hardware to the greatest extent possible. Some benchmarking software requires excessive amounts of hardware in order to generate sufficient load. This limits the number of users who can perform their own tests.

As we shall see, these design goals were paramount in most of our implementation decisions.

2.2. Architectural Overview

The Web Polygraph benchmark consists of virtual clients and servers glued together with an experiment configuration file. Clients (a.k.a. robots) generate HTTP requests for simulated objects. Polygraph uses a configurable mix of HTTP/1.0 and HTTP/1.1 protocols, optionally encrypted with SSL or TLS. Requests may be sent directly to the servers, or through an intermediary (proxy cache, load balancer, etc.). As Polygraph runs, measurements and statistics are saved to log files for detailed postmortem analysis.

Polygraph generates synthetic workloads, rather than using real client or proxy traces. We feel that use of trace-based workloads does not allow us to meet our scalability and flexibility design goals. A real trace must be treated as a constant object. Any modifications such as scaling (i.e., intensifying request rate by playing the trace faster or by interposing trace fragments) result in a new, synthetic trace. Since Polygraph users generally require varying request rates, we would end up with a synthetic workload even though it is derived from a real trace. Altering some other parameters, such as the duration of an experiment or offered hit ratio, has the same effect.

Some may argue that a representative collection of real traces can be used to reproduce a variety of environments without trace modification. Unfortunately, using different traces may make most performance comparisons unfair or difficult to analyze because we end up changing many parameters at once when switching traces. Moreover, many micro-level tests require workloads that correspond to no real trace, and relying on trace-driven simulation makes such testing awkward if not impossible.

The information in this paper is based on Web Polygraph version 2.7.6, except where noted otherwise. Below we describe the major architecture components.

2.3. Robots

Polygraph robots are responsible for generating client-side Web traffic. A robot is implemented as a logical thread within a *polyclt* process. Robots may be configured individually, or collectively, and a single process can manage thousands of robots. We do not recommend running more than one *polyclt* process per CPU due to context switching and other overheads.

Depending on its configuration, a robot can emulate an end-user surfing the Web with a browser, a child cache sending requests to its parent, or a stream of aggregate Web traffic seen on a busy network link. This versatility of a robot explains its name. Initially, robots were called *clients* and then renamed to *users,* still causing misunderstanding and confusion. Robot is a neutral name that does not create any strong or misleading associations.

When simulating single caches and aggregate network traffic streams, a robot must be able to generate hundreds of HTTP requests per second and maintain thousands of concurrent connections for hours. Such workloads put stringent efficiency requirements on the robot implementation. As robots become smarter, the performance requirements become more severe. Polygraph robots are implemented using non-blocking Unix socket I/O, with a minimum per-request overhead. Creation and parsing of HTTP message headers is also highly optimized. We stopped short of implementing HTTP message parsing code in assembly language, a fairly common approach for high-performance caching proxies.

Realistic workloads may require simulation of tens of thousands of end-user browsers. Distributing robots among a similar number of machines would make such simulations prohibitively expensive. Most Polygraph users cannot afford to dedicate more than a few PC's for testing purposes. Consequently, a single *polyclt* process (machine) must be able to support thousands of robots. While each robot in this configuration emits a very slow request stream—far less than one request per second—robots are required to maintain a lot of state information to emulate browser behavior (see Section 3.8). In order to support a large number of robots per process, we must reduce the per-robot memory usage to an absolute minimum.

The two scenarios above represent somewhat extreme (but common) ends of a wide spectrum of possible workloads. These workloads require the robot implementation to be smart while remaining very fast and memory efficient. In the following sections we show some of the algorithms and techniques that we have used to satisfy these requirements.

2.4. Servers

Polygraph servers generate HTTP responses for Polygraph robot requests. Currently, Polygraph supports many characteristics of Web traffic such as content types, size distributions, object modification and expiration times, cachability, and embedded objects. The initial server implementation was relatively simple, as compared to the code for robots. However, with the introduction of embedded objects and increasingly realistic content simulation, the server-side code is also becoming very complicated.

Servers are implemented as logical threads within a *polysrv* process. As with robots, we often have to use many logical servers per process. For performance reasons, we recommend running just one *polysrv* process per machine. Individual servers bind to specific network addresses and ports so that running thousands of servers on a single machine is not a problem.

Compared to robots, it may seem that servers experience a lighter load. Indeed, with a non-zero hit ratio, a proxy handles a part of the load, and many requests (hits) never reach the servers. Occasionally, we run tests without a proxy cache, or even a workload that only has cache misses. In these cases, the robots and servers are equally burdened.

2.5. Proxy Caches

Polygraph considers a proxy cache under test as a “black box.” This means that all necessary measurements are made by Polygraph. Any other approach is unrealistic. There is no standard interface by which we can probe or query the proxy cache. Even if we could, test participants in a competitive environment could be tempted to give incorrect answers. The black box approach means, for example, that we can not measure or report CPU and disk utilization levels for a proxy cache under test. Users in a trusted environment can utilize vendor-specific or third-party tools to extract and record performance data reported directly by a proxy cache.

In some cases, minimal information about proxy configuration is still required. For example, some workloads require that the cache start out completely empty. Polygraph must fill the cache, and then begin its measurement phase. This, obviously, requires knowledge of the disk cache capacity.

2.6. Experiment Configuration

Setting up an experiment involves specifying a particular workload (i.e., the behavior of robots and servers), binding robots and servers to particular processes, and configuring supplementary activities such as logging. Most of the configuration is done using our own domain specific language, which we call PGL. Figure 1 shows some sample PGL fragments for robot, server, and content model configurations.

```
Robot R = {
    // servers to send requests to
    origins = [ '10.0.1-8.1-250:80' ];

    // various traffic parameters
    recurrence      =  55%;
    embed_recur     = 100%;
    public_interest =  50%;
    pop_model = pmZipf(0.6);
    req_rate = 100/sec;
    pconn_use_lmt = zipf(64);
    ...
};
```

```
Content cntHTML = {
  size = exp(8.5KB);
  cachable = 90%;
  may_contain = [ cntImage ];
  embedded_obj_cnt = zipf(13);
};
Server S = {
  contents = [ cntImage:65%, cntHTML:15%,
    cntDownload:0.5%, cntOther ];
  xact_think = norm(3sec, 1.5sec);
  pconn_use_lmt = zipf(16);
  ...
};
```

Figure 1. PGL code fragments. On the left, some robot configuration parameters. On the right, parameters that define servers and the content they serve. Note that "zipf(64)," "exp(8.5KB)," and "norm(3sec,1.5sec)" specify probability distributions for some parameters.

The robot configuration is interpreted by Polygraph when individual robots are created. There may be many robot or server configurations present at the same time to model a variety of environments. For example, an experiment can be configured such that 30% of servers are fast, while the remaining 70% are slow and poorly connected. Most simulation models described in this paper can be configured in a similar fashion.

Earlier versions of Polygraph used command line options to configure all aspects of an experiment. We soon discovered that the complexity of workloads makes command line options extremely messy. On the other hand, a PGL configuration is sufficiently flexible and creates self-documenting workload files that are relatively easy to understand.

PGL was original designed to be a very simple and intuitive language. The entire PGL configuration was declarative and could be interpreted once, at the start of a test. To our surprise, we have received requests for such advanced programming features as pointers, references, branching instructions, etc. In response to user requests, we also have added such run-time features as performance watchdogs (i.e., pieces of PGL code that must be interpreted based on current test conditions). At the time of this writing, the language is still evolving and some advanced programming features are being introduced to satisfy user needs.

3. Simulation Details

This section gives a detailed treatment of the most interesting or innovative algorithms and simulation models in Web Polygraph. The ideas presented here are the result of three generations of Polygraph code and many months of simulation time. Where possible, we give the genesis of an algorithm to illustrate possible alternatives and explain how a model achieved its current form.

$$\text{GET http://}\ \overbrace{\text{10.11.5.1:80}}^{\text{server location}}\ \text{/w}\overbrace{\text{01bf5245.ff0353:000002}}^{\text{world-id}}\text{/t}\overbrace{\text{0000011}}^{\text{type-id}}\text{/\_}\overbrace{\text{00000104}}^{\text{object-id}}\quad \text{HTTP/1.1}$$

Accept: */*
Host: 10.11.5.1:80
X-Xact: 01bf5245.ff0353:0000004c 01bf5245.ff0353:000000b2
X-Loc-World: 01bf5245.ff0353:000002 -1/14 7

Figure 2. HTTP headers from a typical Polygraph request. The URL contains a lot of important identifiers. A request may also include additional request headers not shown here.

Three major factors shape Polygraph's algorithms: design goals, limitations of the benchmarking environment, and lack of development time. It is important to analyze our approach with the first two factors in mind. Many algorithms could be implemented differently if we were developing a pure research tool rather than a versatile benchmark that is used both for real industry tests and research projects.

3.1. Message Structure

Knowing the format of Polygraph's HTTP messages, especially the URL structure, helps to understand the domain in which all simulation models are operating. Figure 2 shows the HTTP headers of a typical *polyclt* request. As you can see, the URL is filled with numbers that serve as various identifiers. The server location part identifies the origin server for a particular request. This part is absent when robots are talking directly to servers, although it is still present in the *Host* header.

Polygraph uses the notion of a *world* as a collection of URLs that have something in common. Different worlds are used to support both sharable and non-sharable Web objects. Usually, the simulated traffic contains $(r + s)$ worlds, where r and s correspond to the number of robots and servers in the experiment (see Section 3.7 for details).

World-ids and many other identifiers generated by Polygraph may appear excessively long and complex. Those long identifiers are essentially random numbers, except that they must be *unique* across all robots and servers, and even across independent Polygraph runs that occur close in time. To avoid initial synchronization and cumbersome inter-experiment state management, Polygraph uses many external variables (time, process-id, etc.) to build unique identifiers and minimize the chance that two of them collide.

Object type-id is an index into an object type description table built from the PGL configuration. In this context, the type identifies a group of objects with similar properties. Objects with the same HTTP content type (e.g., *text/HTML*) may actually belong to different type groups (e.g., big HTML pages and small HTML pages).

The object identifier uniquely specifies an object within a given world. Note that Polygraph pads object-ids with zeros to ensure that URLs do not increase in length over time. A gradual increase in HTTP header lengths could prevent an experiment from reaching a steady state, and may cause hard-to-explain performance anomalies.[†] Object-ids will be discussed in detail shortly.

Figure 2 also includes two extension fields: *X-Xact* and *X-Loc-World*. The *X-Xact* header field uniquely identifies the transaction (a request-reply sequence) that generated the header. It is especially useful for knowing whether a particular response was served as a cache hit or as a miss (see Section 3.4). The second extension field is used for sharing global experiment information as described in Section 3.7.

The URL format includes various tags to mark identifiers (w, t, _). These tags are used to optimize Polygraph's URL parsing. While Polygraph robots can use real URLs and can send requests to real origin servers, such a setup is not typical. We use full control over Polygraph-specific URL space to optimize the common case.

3.2. Object Identifiers and Properties

A simulated Web object has many properties or characteristics that are needed at different stages of processing. For example, object size is needed when a server generates a response to a request for an object. Many of these properties should remain consistent for a given object every time it is requested. In other words, some properties must be deterministic, rather than random. Because Polygraph must support millions of unique objects for some tests, it is infeasible to store per-object state data "as is."

Polygraph addresses this dilemma by deriving the object properties from an object-id. When the value of an object property is required, Polygraph generates that value from scratch, using the corresponding distribution or model. To avoid random value changes, Polygraph *seeds* the random number generator with a value derived from an object-id. Thus, the only per-object information that has to be remembered is the object-id itself.

Polygraph must remember which object-ids have been previously requested to simulate page revisits and generate cache hits. Keeping a list of unique object-ids is also prohibitively expensive. A four hour experiment with 1000 requests per second requires approximately 60 megabytes of RAM just to store the identifiers. Polygraph is often used in much longer tests with higher request rates. To keep memory requirements reasonable, Polygraph allocates object identifiers sequentially and remembers only the *number* of object-ids allocated within a world (world size). Thus, to repeat a request for a page, Polygraph chooses a random object-id with the value smaller than the current object-id allocation level. This object selection algorithm is controlled by the Popularity model described in Section 3.4.2.

The sequential allocation of object-ids makes them poor seeds for random number generators. Instead, Polygraph maps an object-id (possibly along with other tags) into a table of pre-generated, "good" seeds to avoid the problem. The latter approach limits the number

[†]We observed this behavior with an old Polygraph version while running some no-proxy tests. The response time was great for the first 99,999 requests. As soon as we hit 100,000, however, the response time increased significantly. The reason was that FreeBSD's TCP stack would delay packets that are slightly larger than the size of a single mbuf, and longer URLs crossed that threshold.

of possible random values to several millions, but the limit seems to be high enough to avoid any significant problems.

3.3. Load Generation and Throughput

The load generation module described here is responsible for pacing the request stream. Most Web Polygraph users start their performance analysis with the basic question: “How fast can a proxy under test go?” Consequently, the load generation algorithm is usually the first thing that Polygraph users learn. Of course, there is no single correct answer to this question as maximum proxy throughput depends on many workload parameters.

Polygraph supports two basic load generation models: *best-effort* and *constant request rate* (a.k.a., open and closed loops). Both models are configured on a per-robot basis. When the best-effort model is used, a robot submits its next request only after receiving a response to the previous one. The resulting request rate is determined by the proxy’s response rate and the number of robots that are submitting requests. In our experience, there exists an optimal number of robots that results in a peak throughput measurement. With too few robots, the proxy is underutilized. With too many, the proxy is overloaded due to a large number of concurrent sessions.

It is very important to note that in a best-effort scenario, the request rate and response time are tied together: request rate is the reciprocal of response time. This tight connection between request rate and response time is rare in real Web based systems. Best-effort workloads are sometimes useful for low level performance analysis, but are usually inappropriate for macro-benchmarks because they do not represent a realistic load and may lead to confusing results. See Section 4.2 for further discussion.

The *constant request rate* model is based on a Poisson request stream with a given constant mean. The name of the model is somewhat unfortunate because the Poisson stream contains bursts and the short-term request rate may not be constant. The burstiness of the Poisson model explains, in part, the fact that constant request rate workloads are usually harder on a proxy than their best-effort counterparts. For example, if you use a best-effort test to determine a peak throughput, then try to use that value for a constant request rate test, chances are good that the test will fail.

The important characteristic of the constant request rate model is that new requests are emitted regardless of the reply stream. Consequently, throughput and response time are not tied together as with the best-effort model. Even if a proxy stops responding to requests, robots continue to generate new ones until they run out of resources.

A true constant request rate (i.e., using the same delay between any two consecutive requests) is also supported. It is best to avoid this kind of request stream during macro-tests because it may lead to lock-step behavior due to unfortunate timing collisions (e.g., each request would arrive just when an expensive hardware interrupt or network event has occurred).

Studies on Web traffic characterization suggest that real request streams have a self-similar component visible at both micro- (milliseconds) and macro- (minutes) levels [CB96, GB97]. The Poisson stream used by the constant request rate model is not self-similar. However, the introduction of embedded objects and browser emulation (Section 3.8) may create a necessary self-similar component. Analyzing the self-similarity of resulting Polygraph streams is our future work.

3.3.1. Varying the Load

The constant request rate model allows us to study a proxy under steady load conditions with constant mean request rate. While this may be useful for finding the sustained peak performance, real proxies experience significant variation in request rates over time. For example, one may wish to simulate the daily bell-shaped load of a typical corporate proxy. Moreover, we usually want to gradually increase the load placed on a proxy, rather than suddenly subjecting it to a very sudden burst.

To support customized load patterns, Polygraph introduces the notion of a simulation *phase*. A phase may be of arbitrary duration, and an unlimited number of consecutive phases can be configured and scheduled using PGL. The phase configuration contains two *load factors*: load factor at the beginning of a phase ($load\_fact\_beg$), and the factor at the end of a phase ($load\_fact\_end$). During a test, Polygraph gradually adjusts the *current* load factor according to the configured beginning/end factors of the current phase and the time spent in that phase.

It turns out that a gradual adjustment of the current load factor is not trivial. Our first implementation used the following approach. Let d be the duration of a phase, and k be the slope of the line connecting $load\_fact\_beg$ and $load\_fact\_end$. Then, naturally, the load factor at time t is simply:[‡]

$$load\_fact[t] \quad = \quad load\_fact\_beg + t * k \tag{1}$$

However, using equation (1) leads to a problem much like Zeno's "Achilles and the tortoise." The problem is best illustrated with a single robot configuration, and $load\_fact\_beg$ being zero or very small. When a robot requests $load\_fact[t]$ at the beginning of a phase, the small value of t yields small value for $load\_fact[t]$. A robot then takes an exponentially distributed random variable (with a large mean equal to $\delta = 1/load\_fact[t]$), to simulate the Poisson request stream. The large mean leads to a long inter-arrival gap, and a lot of time passes until the robot emits its next request at the new $load\_fact[t + \delta]$ level. Note that no requests are submitted while the robot is waiting. The result is a staggered load pattern, with much lower actual request rates than mandated by the configuration. At the extreme (but still realistic) case, a robot may generate only one or two requests during the entire phase in which it was meant to emit hundreds of requests.

Our current implementation eliminates the problem. The load factor is calculated as a function of the number of requests (n), instead of time:

$$load\_fact[0] \quad = \quad load\_fact\_beg \tag{2}$$

$$load\_fact[n+1] \quad = \quad load\_fact[n] + \delta[n] * k \tag{3}$$

$$load\_fact[n+1] \quad = \quad 1/\delta[n] \tag{4}$$

[‡]For simplicity, the formulas talk about load factors rather than actual request rates. The calculations for request rates are very similar.

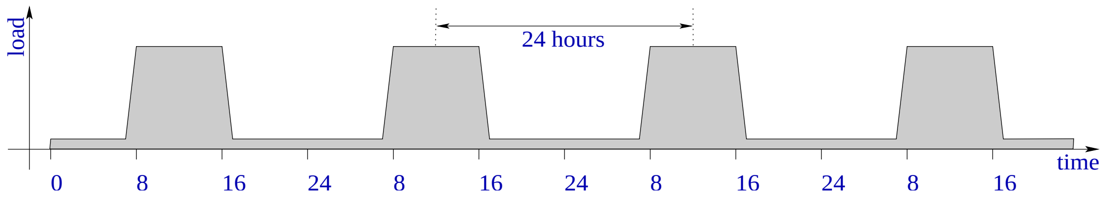

Figure 3. Load fluctuation on a corporate proxy.

Equation (4) is based on the fact that a robot submits exactly one request within inter-arrival time δ. It provides an alternative way to compute $load\_fact$ via δ. The three equations (2–4) have three unknowns ($load\_fact[n]$, $load\_fact[n+1]$, and $\delta[n]$) and can be solved to find $load\_fact[n+1]$. The resulting algorithm changes the load level smoothly from $load\_fact\_beg$ to $load\_fact\_end$.

The current load factor is used by Polygraph robots to calculate their request rate. The constant request rate model specifies a mean request rate (i.e., mean inter-arrival time) for a Poisson stream. A robot multiplies that mean by the current load factor to get the correct current mean. The adjusted mean is used to configure a random distribution that simulates the request inter-arrival time. This process is repeated for every request. Note that robots may be configured with different request rates, but they all share the same phase load factor.

Load factors may be used to produce workloads of almost any shape. Figure 3 depicts an approximation to the load pattern experienced by a typical corporate proxy.

In many environments, changes in proxy load are caused by changes in the size of robot population, rather than individual robot activity. For example, in a corporate environment, the load increases as more employees come to work and start surfing. The request submission rate from a single employee may not change much. To simulate the changes in robot population, Polygraph phases have a pair of $population\_factor\_beg$ and $population\_factor\_end$ fields that work exactly like load factors, but are applied to the number of active robots instead. To honor population factor settings, Polygraph adds or deletes robots participating in the test.

3.4. Hits, Misses, and Hit Ratios

Web caches add hit ratio measurements to the traditional metrics (throughput and response time). Hits may occur when a robot revisits an object. It is important to understand that revisited URLs do not always result in cache hits. Here are some of the situations when revisiting an object does *not* result in a hit:

- the object may be uncachable,
- an object has been modified since the last visit so a fresh copy of the object must be served to a robot,
- after the last visit, an object got purged from proxy's cache to free room for other objects.

Polygraph should never receive a hit on an uncachable object. Such an occurrence means that the proxy under test is not following the HTTP protocol. Polygraph counts hits on uncachable objects as errors. In some cases, HTTP does allow proxy caches to return stale responses as cache hits. Polygraph also reports stale responses as errors.

Both disk storage capacity and replacement policies may vary among products. In some situations, two proxies may achieve a different hit ratio for the same workload. To compensate for this, we introduce the notion of an *ideal hit* (or ideal hit ratio) and *measured hit* (or measured hit ratio). The ideal hit ratio corresponds to the behavior of an ideal proxy, with an infinite cache size, that stores all cachable responses. The measured hit ratio is the value that Polygraph measures for a real proxy cache. The ideal hit ratio is a convenient metric because it does not depend on the actual proxy in use. It also provides an upper bound for the measured hit ratio.

Polygraph uses its transaction identifiers to detect cache hits. When making a request, a robot inserts a unique transaction-id into the HTTP headers. If the request reaches a Polygraph server, the identifier is read and its *mutant* version is sent back in the response headers. When the robot receives a reply, it checks whether the transaction identifier is a mutant version of the current transaction-id. If so, it means the server was contacted during the transaction, and a miss is counted. If the received value is a mutant of some other transaction, then the robot received a hit. Transaction-ids are generated and mutated in such a way that there is a one-to-one mapping between mutants and original values. In other words, conflicts are not possible.

Polygraph generates request streams with a constant ideal hit ratio. However, ideal hit ratio is a derivative of several variables such as object freshness, cachability, and URL popularity. To allow fine-grained control, a robot is configured using what we call the *recurrence ratio*, which is the probability of re-visiting a particular object. With 100% cachable and fresh content, the recurrence ratio is equal to the ideal hit ratio. In practice, the recurrence ratio must be increased to account for uncachable or stale content, so the desired ideal hit ratio is offered.

3.4.1. Working Set

We define the working set as the set of all objects that have a non-zero probability of being accessed at a given time. To simulate cachable misses, Polygraph introduces new cachable objects into the working set (a so-called *fill stream*). If all previously introduced objects may be revisited, then the working set size will grow during an experiment. When the working set size increases, it means the probability of revisiting any particular object in the future decreases. Thus, the object popularity distribution may change over time.

A changing object popularity distribution violates one of the basic requirements for a good workload—that the workload must have a sufficiently long *steady state*, during which the global properties of the objects must not change. Individual objects may come and go, but distributions and other working set characteristics must remain constant.

To avoid problems associated with the growing working set, we artificially limit the working set *size*. Size limits do not imply that working set *contents* remains constant. New objects are simply added at the same rate that old objects are removed. Technically, enforcing the limit is

Cache Size (% of WSS)	**Measured Hit Ratio (%)**
2.0	1.3
2.5	1.7
5.0	3.4
10.0	6.7
20.0	13.3
50.0	31.1
100.0	51.0
130.0	55.0

Table I. Measured hit ratio versus relative cache size for a simulated LRU cache and 55% ideal hit ratio workload. An LRU cache must be larger than the working set size because some objects get requested just before leaving the working set, yet they stay in the cache for a longer time.

easy: we instruct Polygraph to stop requesting objects that were introduced into the working set some time ago. The hard part is selecting the appropriate size limit.

A natural working set size can be derived from proxy trace files. Unfortunately, we cannot use this approach because of the limitations of the benchmarking environment. A natural working set size would be at least 3–5 days worth of traffic. Most benchmarking activities cannot allow for tests of that duration. Thus, we are forced to use a relatively small working set size of at most 3–4 hours of peak traffic. The consequences of small working sets are discussed in Section 4.3.

For a proxy to achieve the ideal hit ratio during a test, it must be able to cache the entire working set. In practice, a proxy that implements an LRU-driven replacement policy must keep slightly more objects. This is because some objects are requested just before they are dropped from the working set. Such an object remains in an LRU cache for quite some time, even though it will never be requested again. Table I illustrates the relationship between LRU cache size and ideal hit ratio.

3.4.2. Object Popularity

To simulate a given recurrence ratio, a robot just needs to revisit any object from the working set. The algorithm that selects the object to be revisited defines object *popularity*. In the case of a uniform popularity model, all objects within the working set have equal chances of being revisited. A more realistic approach is to use Zipf-like distributions, which makes some objects more popular than others [BCF+99]. Polygraph implements both approaches. We discuss the controversy connected to using Zipf distributions in Section 4.4.

A skewed popularity distribution such as Zipf should be adjusted so that recently accessed documents have higher probability of being revisited. Polygraph achieves this effect by making popularity distributions relative to the last object added to the working set, allowing the distribution to “slide” along as new objects are being added.

Polygraph also simulates “hot subsets” — a configurable proportion of requests target a configurable portion of the working set. For example, 10% of all requests in a standard PolyMix workload access 1% of the working set. The hot subset jumps from time to time, simulating changing interest behavior known as “flash crowds.” With the exception of hot subsets, simulating reference locality is not supported by Polygraph. Efficiently supporting reference locality across URL worlds may not be possible, but introducing a locality biased towards recently accessed objects within the same URL world is our future work.

3.4.3. On Byte Hit Ratio

So far, we have only discussed document hit ratios (DHR). Many people are also interested in byte hit ratio (BHR) measurements. In most real caches, BHR is usually about 10–15% lower than DHR. Thus, an important characteristic of real traffic is that there are fewer cache hits for large files than there are for small ones. Polygraph models this phenomenon using the *bhr_discrimination* knob of the object popularity model.

BHR discrimination is applied after the popularity model selects a candidate object to be re-visited. Polygraph scans object-ids around the candidate and selects the object with the smallest size. The discrimination knob setting, a percentage, affects the probability of applying the discrimination algorithm. Determining object size from an object-id is a computationally expensive operation, involving generating several random variables using user-configurable distributions. To be efficient and to approximate object popularity distributions, the algorithm can only look at a few identifiers to make a decision. With such a small optimization space, we have found it necessary to set BHR discrimination relatively high, around 80%, for realistic workloads.

Note that there is no direct byte hit ratio parameter, only an indirect discrimination knob. As with document hit ratio, supporting user-specified BHR is awkward and is often impossible. Offered byte hit ratio depends on several user-configurable factors (e.g., response cachability, object popularity, and size distribution). It is impossible to deterministically map a user-specified BHR back into the right mix of those factors. The discrimination knob is one simple way to tilt BHR in the desirable direction. BHR discrimination setting alone does not guarantee any particular byte hit ratio. Its effect (in combination with other factors of a given workload) can be determined experimentally.

The discrimination algorithm described above is a result of several trials and errors. More complex algorithms were either too slow or required too many configuration parameters without clear physical meaning. The current algorithm is a good example of how difficult it is to efficiently combine two relatively simple but mutually dependent models (object popularity and size in this case).

3.5. Delays and Response Time

Adding artificial delays to the workload is essential for maintaining a realistic number of concurrent connections. High request rates alone may not result in high levels of concurrency if each request can be processed very fast. In real caches, various external delays in request processing are unavoidable and the number of concurrent connection usually increases with the request rate, presenting significant challenges to a proxy.

Polygraph simulates server-side latencies by delaying the reply using a given *think time* distribution. Once the server accepts a connection and reads the request, the reply is delayed by the think time amount. Once the think time has expired, the reply is written to the network as quickly as possible. Note that cache hits are not delayed in this manner. Thus, a cache's hit ratio is also reflected in its measured response time.

A more accurate delay model requires delaying individual TCP packets at the network level (for both client- and server-sides of the benchmark). Implementing such features in Polygraph, at the application layer, would be extremely awkward. Instead, we use third party kernel-level packet loss and delay simulation tools such as FreeBSD's *dummynet* [Riz97]. These kernel-level tools are usually specific to the operating system used. OS-independent hardware simulators are also available.

3.6. Object Life Cycle Model

Caching proxies may spend a significant amount of resources on validating the freshness of stored objects. For example, 10–20% of traffic through caches in the NLANR/IRCache Mesh is related to freshness validation [Wes03]. Thus, a realistic benchmark must model various freshness parameters. Polygraph handles freshness related characteristics using an Object Life Cycle (OLC) model described in this section.

The Object Life Cycle model is responsible for simulating object modification, expiration, and similar events in the "life" of a Web object. The model affects the outcome of validation (*If-Modified-Since*) requests and various prefetching or validation algorithms that depend on object freshness. The OLC model has three components:

- object creation time simulator
- object modification time simulator
- object expiration time simulator

Polygraph assumes that Web objects have a semi-periodic life cycle. For example, a daily news page may be modified every 24 hours, a personal home page may be stable for a month or so, and a page with old rock group lyrics might remain constant for years. Let's define a *cycle* as a time interval that contains exactly one modification of an object. Then a *cycle period* is defined as an average cycle length (Figure 4). The period of a cycle is object specific and can be configured using PGL on an object type basis.

While relying on a semi-periodic life cycle is certainly a limitation of the model, it may be the only way Polygraph servers can calculate past and future modification and expiration times. These calculations are already complicated by modification time variability (described below), and non-cycle-based models are likely to make them prohibitively expensive.

Standard Polygraph workloads use relatively simple OLC settings. Mean cycle lengths vary from one day to half a year, depending on the object category (e.g., HTML objects are modified more often than large downloads). Smaller cycles lead to more objects expiring in the cache during a test, increasing load on the caching proxy. Higher modification variability would make

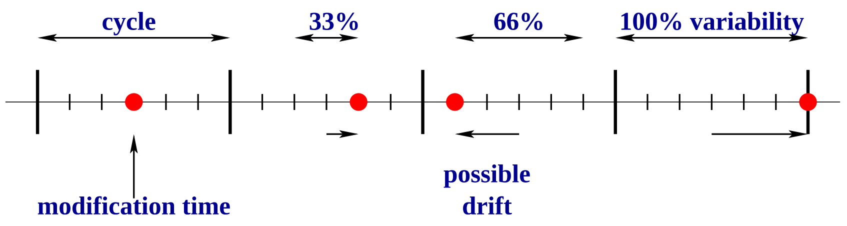

Figure 4. Polygraph's Object Life Cycle model. Exactly one modification occurs in every cycle. The exact modification time varies from cycle-to-cycle based on a user-defined distribution.

automated modification prediction more difficult (we are not aware of any cache using such predictions during a production test).

3.6.1. Object Creation Time

Every Web object is assumed to be created some time in the past. Originally, the *birthday* of an object was determined using a user-specified random distribution, and both absolute and relative (to the server process start) birthdays were supported. However, when it was necessary to add support for validation requests on the client-side, it became apparent that supporting relative birthdays was awkward because each Polygraph server process had different start times and those start times would have to be known to robots (or a global reference point would have to be negotiated run-time).

Current versions of Polygraph generate birthdays within the "first" object life cycle, using a uniform distribution. The first cycle is the one that starts at "system time zero" (usually January 1, 1970). While less flexible, The current model is simple and satisfies all current testing needs.

3.6.2. Object Modification Time

Polygraph assumes that the modification pattern of a given object is usually stable and often independent from other objects. Clearly, for many objects, modifications do not happen at constant intervals. Polygraph models variability in object modification times while keeping the cycle period constant. The variability is expressed in percents of a cycle period. Zero percent means no variability; all modifications happen exactly at the middle of a cycle. One hundred percent variability means that, for a given cycle, an object may be modified at any time (from the beginning until the end of a cycle). Variability higher than 100% may be used to simulate a problem at the server; modification events for an object could appear in the wrong order or in the future (from a robot's point of view).

Every object has a last modification time that is known to Polygraph. However, real Web servers sometimes do not include the *Last-Modified* entity-header field in replies. The OLC model allows us to specify the percentage of objects that announce their modification times. This property remains consistent between multiple requests of the same object. Note that HTTP/1.1 [FGM+99] recommends that responses without a cache validator (modification time) or explicit expiration time should not be cached. If a product follows that recommendation during a Polygraph test, it would not achieve the ideal hit ratio. However, none of the products tested so far appears to follow protocol recommendations by default.

Web Polygraph does not support entity tags yet. Thus, validations based on object's last modification time are the only ones possible. These kinds of validations are also the most common in real life.

3.6.3. Object Expiration Time

An object's expiration time is reported via the *Expires* entity-header field and, according to HTTP, indicates the time when a cached object should no longer be considered fresh without a revalidation. Since Polygraph knows the future modification times of objects, it would be very easy to report precise expiration times, thus eliminating the guesswork for proxies. However, having such an algorithm would lead to unrealistic simulations. Indeed, real Web servers cannot predict future modification times. Hence, in most cases, servers lie about the expiration time of objects.

A real origin server generates *Expires* fields based on several configuration parameters. Usually, there is a way to tell a server to compute the expiration value according to one of the following two formulas [Apa]:

$$last\_modification\_time + constant\_delta$$
$$current\_time + constant\_delta$$

Using the formulas above, one can request that an object expires delta seconds after it was last accessed or modified. The first formula expires all cached copies of a given object at the same absolute time. The second formula expires cached copies when they reach a given age (after the last revalidation). The Polygraph server implements both formulas, and also allows us to specify the portion of objects without any *Expires* fields.

3.6.4. Object Death Time

From the Object Life Cycle model point of view, Polygraph objects never die or disappear. The model can provide creation, last modification, and expiration dates for any Polygraph object. The Working Set Size model (Section 3.4.1) is responsible for introducing and removing objects from the test working set. Adding an explicit object lifetime knob may be useful to simulate realistic 404 (Not Found) responses.

3.6.5. Validation Requests

The Polygraph robot can be configured to generate validation (*If-Modified-Since*) requests, and the Polygraph server correctly responds to such requests (originating from a robot or a proxy). Polygraph always answers validation requests accurately. That is, it never lies about whether an object has actually been modified or not. Note that this rule includes objects for which the *Last-modified* reply header was omitted. Furthermore, an object's expiration time, if any, is irrelevant for responding to validation requests.

3.7. Global URL Space

In an early version of Polygraph, a *polyclt* process would emit requests destined for only one polysrv process. When multiple servers were used, the workload was effectively composed of multiple non-overlapping request streams. This serious limitation had to be removed. However, sharing requests streams and maintaining a single global URL space between multiple processes on multiple machines is a hard task.

The first challenge we had to address was how to distribute the information about the global URL space among robots and servers. In real life, and in many benchmarking environments, the network architecture prevents robots from communicating with each other. Just like in real life, individual robots can be placed on different networks isolated by firewalls or other routing restrictions. It would be unnatural to require global connectivity among robots (using HTTP or some other protocol). In the current version, all robots can request objects from all servers, which provides a mechanism for information sharing and synchronization.

3.7.1. Information Sharing, Second Generation

What kind of information must be shared among robots? To generate overlapping request streams and keep the ideal hit ratio constant, a robot needs to know which objects the other robots have requested. To provide this information, each Polygraph server remembers the highest object-id ($MaxOid_s$) that any robot has requested. In other words, it is guaranteed that all object-ids equal to, or smaller than $MaxOid_s$ have been requested by at least one robot. A server's $MaxOid_s$ value is sent with every response using an extension-header field. Robots collect these counters and update their tables.

The scheme outlined above allows robots to request a URL that was visited before, possibly generating a hit. However, knowing server's $MaxOid_s$ still does not allow robots to reliably request an object that no other robot has requested before! Indeed, if two robots start requesting what both believe to be a new object, one of them will inevitably end up requesting an old object and possibly even generating a hit. Such collisions would make reliable support of a given recurrence ratio impossible.

To avoid conflicts, earlier versions of Polygraph used the following object-id reservation scheme: Servers *reserve* new object-ids for robots (Figure 5). A $MaxOid_s$ counter described above is still maintained, but each robot also keeps a set of new (reserved) object-ids received from a server and uses only those ids to request new objects (i.e., generate cache misses). To reduce communication costs, servers give out new object-ids in chunks. Figure 5 shows a snapshot of the object-id allocation process with three robots and five object-ids per chunk.

Note that to preserve the $MaxOid_s$ semantics, a server can increment $MaxOid_s$ only when the $(MaxOid_s + 1)$-th object has been requested by a robot. However, reserved object-ids can be requested in no particular order (gray circles on Figure 5 indicate reserved, but not yet requested object-ids). Thus, a server must remember what object-ids (larger than $MaxOid_s$)

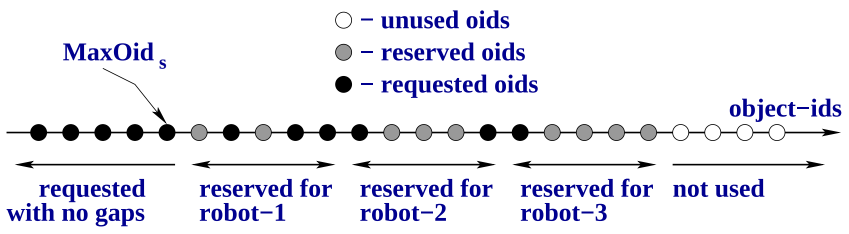

Figure 5. Original allocation of object-ids

have been requested so it can increment $MaxOid_s$ later without waiting for those object-ids to be requested again.

To save memory, a server maintains a bitmap of reserved object-ids. The bitmap has a fixed size. Consequently, if some *polyclt* process slows down (relative to its peers) and does not request reserved object-ids fast enough, a server will have to eventually give up on waiting for that process to request reserved oids and will re-assign them to another *polyclt*.

3.7.2. Information Sharing, Third Generation

The object-id reservation scheme works well for correctly configured tests, with typical request patterns. However, it lacks robustness when there is no or little (compared to other servers) communication between a *polyclt* process and a given server. In that case, reserved object-ids are not requested fast enough, and the server has to cancel reservations to move on and satisfy the needs of those *polyclt* processes that are faster, or that can communicate with the server. While the problems were mostly found in misconfigured experiments, users were frustrated that the test "does not just work" and generates strange errors about client-server communication, especially when the test reaches peak load levels. It's not just that the user sees strange errors, but that these transactions are counted as errors in the analysis of test results.

Our solution was to re-implement the information sharing algorithm. Instead of using one URL world per server and, relying on object-id reservations to avoid miss conflicts, the current code maintains one URL world per *polyclt*-server pair.[§] The server is still the point of synchronization and exchange of object-ids. Each *polyclt* process tells the server how far, in its world, it has advanced ($MaxOid_{sr}$). That information is propagated to all other *polyclt* processes. If a particular *polyclt* becomes slow or disconnected, other *polyclts* simply stop receiving new information about the troubled process, but are able to generate fresh hits using $MaxOid_{sr}$ of their own and those from other processes.

This third-generation algorithm is more robust, with a slight increase in memory usage. Whereas the previous version used a constant amount of memory, this one uses an amount

[§]That is, per *polyclt* process and server agent pair.

of memory in proportion to the number of *polyclt* processes. We estimate the memory requirements for this aspect of Polygraph to be about 32Kbytes per *polyclt*. Given that even most high-end caching products require less than 10 *polyclt* processes, the overall increase is not very significant.

3.7.3. Private Robot Interests

In real life, a Web surfer may have interests that are specific to her alone and do not overlap with other members of the same surfing community. Polygraph models this phenomenon by simulating private, robot-specific, worlds. Note that URLs in a private world may be revisited and generate hits. A private world is like a Web site that is known only to one surfer (robot).

All robots share public worlds, with one public world per logical server, and each robot has its own private world. The worlds are distinguished by their world identifiers (see Section 3.1) so that URLs from different worlds do not collide. The distribution of public versus private interest is configured using PGL on a per-robot basis.

The mixture of public and private worlds may be causing some undesirable side-effects. In recent test results, we often see a gradual decrease in measured hit ratio over time. For example, from 58% at the start of a test, down to 55% at the end. After analyzing the access log files from a proxy under test, we realized that the URLs from private worlds are significantly less popular than those from public worlds. This is mostly due to the fact that private worlds are private (each belongs to just one robot), and that individual robots have a very low request rate.

This means that cache hits in private worlds occur much less often than hits in public worlds. Assuming that a proxy under test uses basic LRU replacement, it may remove some of these infrequently requested URLs from private worlds, thereby decreasing the measured hit ratio. In fact, Polygraph does not place any restrictions on the amount of time that may pass, or total number of requests that may occur, between repeat accesses within a single world. Due to the probabilistic nature of our simulations, it's possible that relatively large amounts of time (or number of other requests) pass between repeated access in some robot worlds.

3.8. Browser Emulation

Web browsers have certain characteristics that we want to model in Polygraph. Currently, Polygraph has two major models related to browser emulation. The first model is responsible for managing the allocation and usage of TCP connections from a client to a proxy or server. Polygraph supports persistent HTTP connections as defined in the HTTP RFCs. The model follows Netscape's browser behavior as reverse-engineered by Zhe Wang [WC98]. A robot maintains a fixed-size connection pool (e.g., Netscape appears to be using at most 15 connections) and closes idle connections as necessary. Note that maintaining the per-robot pool not only requires extra memory, but significantly increases the number of file descriptors and other network resources tied to a robot, even if most of the connections in the pool are idle.

Polygraph also simulates the effect of embedded objects on browser behavior (and hence on the generated request stream). Servers may be configured to use content models that include embedded objects (see Figure 1 for a simple example). When a server generates a *container* object, it inserts some Polygraph-specific tags that identify the embedded objects that belong to the container. A robot parses the content of an object and extracts the tags. As tags get extracted, the robot may emit corresponding requests for the embedded objects. Real browsers may not request all of the objects embedded in a container (e.g., some objects may be already in a browser cache). Polygraph robots can be configured to simulate a small browser cache and/or just request embedded objects with a certain probability. A serious limitation of the current model is that embedded objects cannot be shared among containers.

4. Benchmarking Pitfalls

In this section, we describe a few of the interesting problems and insights we have discovered while benchmarking Web caches.

4.1. Half/Full Duplex

Almost all of our tests have involved 100BaseTX networks, and we have seen duplex mismatches cause confusion on numerous occasions. A fast Ethernet interface may be in either full- or half-duplex mode. The hardware is supposed to be able to auto-negotiate the speed and duplex settings. However, in our experience, auto-negotiation often fails. Only Ethernet switches (or a crossover connection) can support the full-duplex mode. Repeaters must use half-duplex.

Duplex problems often go undetected initially because the network works fine for applications like *telnet* and *ping*. When one interface is set to half-duplex on an otherwise full-duplex segment, some collisions do occur. These collisions are not a problem at low utilization because the interface will simply retransmit them. Problems begin to appear only when the network is heavily used.

For these reasons, it is critical to run TCP throughput tests with all devices connected to the network. Freely available software such as *ttcp*[¶] and *netperf*[‖] can easily find the maximum network throughput. A switched network should be able to achieve between 92 and 94 megabits per second. However, your system's ability to fully saturate a 100BaseTX network may also depend on other factors, such as the CPU, bus speed, and network driver. In the past, we found that FreeBSD's xl0 driver performed much worse (84 Mb/s) than the fxp0 driver (94 Mb/s). If the throughput is lower, the network interface probably reports a high number of errors or collisions. Throughput for a half-duplex, repeated network will be much lower of course. If using a full-duplex, switched network, it is a good idea to test the network in both directions at the same time. For example, by running two netperf tests at the same time — one in each direction. A unidirectional test may not reveal an interface with a half-duplex setting.

[¶]`http://ftp.arl.mil/ftp/pub/ttcp/`
[‖]`http://www.netperf.org/`

4.2. Best-effort Workloads

In Section 3.3 we discussed some of the differences between the best-effort and constant request rate submission models. The best-effort workload is a tempting choice because (a) it is almost guaranteed to succeed, and (b) it measures *some* kind of peak throughput with relatively little effort. The constant request rate workload, on the other hand, will fail if the proxy cannot keep up with the offered load. Finding the peak throughput with constant request rate is a time-consuming, and perhaps frustrating process.

If you plan to use a best-effort workload, you must understand its subtleties. The same is true if you are reading results derived from a best-effort test. As previously mentioned, the peak request rate depends on the number of robots and the proxy's response rate. Thus, you might be surprised to get a higher throughput with 100 robots than you would with 200.

One way that a best effort test can fail is if all of the robots become "stuck." Polygraph does not time out idle connections. Thus, if the proxy has a bug that causes connections to hang, some robots will stop submitting new requests. Of course, you can probably decrease the chance that a significant number of robots become stuck by using more to begin with.

A slightly more subtle problem exists with network latencies. For example, in an article published by Network Computing magazine [Yer99], a network-induced delay of 200 milliseconds resulted in a drastic decrease in performance of high-end proxies. Some of Network Computing's results were an order of magnitude lower than those measured for similar products at the IRCache bake-off only a few months before.[**]

4.3. Small Working-set Size

Our desire for a realistic benchmark is complicated by the need to run tests that take less than a day to complete. This conflict creates some unfortunate side-effects that you should be aware of.

One of the most frustrating consequences is that you need to be very careful if you want to use Polygraph results for capacity planning. Lets say that you read a benchmarking report that shows a product achieves 2000 requests per second. Furthermore, this product has 50 gigabytes of disk space. Notice that a 50 gigabyte cache holds about 4,000,000 objects, with a mean size of 13 kilobytes. Let's assume the report shows that the cache fill rate is 25% of the request rate, or in this case, 500 objects per second. This means that this product holds only about 2.25 hours worth of Polygraph traffic until it starts removing useful objects. This is much too short to be useful for a real stream of 2000 req/sec. Putting this another way, if you really need a solution that can sustain 2000 requests per second, then you need much more than 50 gigabytes of disk space.

We would prefer a workload that causes cache size (and replacement policy) to affect the achieved hit ratio. Unfortunately, to accomplish this with some realism, we need a working set size on the order of 3–5 days, which in turn requires tests that run for 6–10 days.

[**]For a detailed analysis of the incident, see `http://polygraph.ircache.net/Watchdog/netcomp.html`

4.4. Zipf vs. Uniform

When developing the first workload for Polygraph, we planned on using a Zipf distribution in the popularity model. This choice was primarily based on the fact that we observe Zipf-like[††] popularity distributions in real proxy traces. Our initial tests with Zipf revealed some undesirable side-effects, however.

The Zipf popularity distribution led to an unreasonably high memory hit ratio. That is, too many objects could be served from a cache's memory, rather than from its disk storage. Essentially the workload did not do an adequate job of stressing a proxy's disk system. A number of vendors were concerned that competitors could achieve exceptionally good response times by using additional memory. Some were also concerned that they would be *accused* of cheating because their response times were so low.

We tried a number of ways to force Zipf to give more reasonable memory hit ratios, but all attempts failed. Part of the problem was that we didn't have an efficient algorithm for implementing a Zipf-like probability distribution function. Another problem was that the test duration was relatively short (just one hour) and did not provide enough time to build up a reasonably large working set.

In the end, we changed the workload to use a pseudo-uniform popularity distribution. Here, repeated objects are selected uniformly from the set of previously requested objects. The set of objects increases with time, so its not strictly uniform. This approach gave us what we wanted—more disk hits and fewer memory hits.

Later, we implemented an efficient algorithm for simulating a Zipf-like distribution which allowed other researches to compare the two models [LML$^+$01].

5. Related Work

Web characterization efforts have received a lot of research attention in the past few years. Most basic Web traffic characteristics are well understood and can be easily modeled in isolation. For an excellent overview of fundamental results see [Pit98].

Several sophisticated benchmarks have been developed for testing origin servers. The Surge benchmark [BC98] concentrates on synthesizing a realistic origin server workload. Surge attempts to mimic end-user behavior by using an on/off process to simulate idle times of individual users. Object popularity, embedded references, and reference locality are among the most complex models implemented in server benchmarks available today. Surge developers were often faced with problems similar to the ones described in this paper. We have tried to learn from the authors experience and several Polygraph models (especially its early versions) were inspired by the ideas found in Surge.

Another interesting origin server benchmark is *httperf* [MJ98]. Httperf's design principles are similar to those of Web Polygraph—a robust, high-performance benchmark that can be

[††]Classic Zipf uses an exponent of 1, while Zipf-like may use any exponent between 0 and 1.

configured to simulate a wide range of micro- and macro-level workloads. The authors give several valuable insights on operating system tuning and how to correctly stress test an origin server to its real limits. Httperf provides an API that allows new features to be added without re-implementing the core functionality.

Perhaps the most well known origin server benchmark is SPECWeb [Sta], developed by the Standard Performance Evaluation Corporation. Unfortunately, the widely used SPECWeb96 has such a small data set that it creates an unrealistically low number of concurrent connections (see Surge versus SPEC comparison in [BC98]). When used, it mostly tests the TCP stack and RAM cache performance, but not origin server performance as a whole. SPECWeb99 attempts to address some of these issues. SPEC benchmarks may require significant hardware investment to generate high volume workloads and are not free, which complicates their analysis.

Unfortunately, applying existing origin server benchmarks to proxies is not appropriate. While some of the traffic patterns are similar for proxies and origin servers, many crucial proxy traffic characteristics are very different or completely absent at the origin server (e.g., hit ratio and object freshness). Most proxy benchmarking activities also require high performance origin servers. Naturally, origin server benchmark packages do not include software to simulate or generate origin server responses.

Proxy benchmarking also puts significant demands on the tool performance. These demands cannot be met by existing origin server benchmarks. For example, Surge limits the number of simulated users per machine to about 150, while Polygraph supports 5–10 times more on similar hardware. High performance requirements sometimes stand in the way of more sophisticated models; this paper has described a few solutions that we have implemented to balance the two.

Quite a number of tools have been developed to analyse proxy performance. A survey of relevant benchmarks can be found in [Dav99]. Many of those tools are, however, proprietary with little or no information available about their design and performance. Proprietary benchmarks were developed by CacheFlow, Inktomi, NetApp, Novell, and others. Caching vendors often tailor their tools to highlight features of their particular products. On the other hand, some proprietary tools are simply load generators rather than benchmarks.

Publicly available benchmarks dedicated to testing proxies include the Wisconsin Proxy Benchmark (WPB) [AC98, Cao98] and HTTP Blaster [Vöc]. Both benchmarks use simulated clients and servers. HTTP Blaster is a trace-driven tool. Cachability of objects and adjustable load levels are supported by appropriately tuning the trace. The Wisconsin benchmark uses a synthetic workload and models offered hit ratio, temporal locality, and server side delays, but the project is dormant now.

6. Conclusions and Future Work

We presented the design and implementation of Web Polygraph, a proxy performance benchmark with many advanced features (see Table II). We discussed Polygraph's design goals and showed how they shaped the implementation. Web proxy traffic poses many challenges to a benchmark, including high performance simulation of complex, realistic workloads. Our experience with Polygraph demonstrates that many of those challenges can be addressed, but a lot of knowledge, imagination, and skill are required to build the right models.

Feature	Description
Robot & server populations	Agents bind to interface alias addresses. Typically have 1000's of agents per CPU.
Packet loss & delays	Use operating system tools (i.e., FreeBSD's *Dummynet*) for fine-grained packet manipulation, as well as think-time distributions on agents.
DNS	Robot agents can query the DNS for server addresses.
IPv6	Polygraph version 2.8 release supports IPv6 addressing.
SSL/TLS	Polygraph version 2.8 supports SSL/TLS encryption.
Load generation	Robot agents use either best-effort, or Poisson request submission models.
Request methods	Support for GET, HEAD, PUT, and POST methods.
HTTP versions	Can specify the percentage of HTTP/1.0 and 1.1 requests and responses.
Persistent connections	Robots and servers may reuse HTTP connections, subject to configurable use limits and idle timeouts.
Aborted transactions	Robots and servers may be configured to abort transactions with a certain probability.
MD5 checksums	Responses may contain a *Content-MD5* header, which robots use to verify content integrity.
Usernames	Robots may add *Authorization* or *Proxy-Authorization* headers.
Cache validations	Robots may emulate a local browser cache and emit some percentage of validation (*If-Modified-Since*) requests. Of course, a caching proxy under test may also issue validation requests.
Content types	Responses generated by servers may contain configurable *Content-Type* headers and filename extensions.
File sizes	Each content type has its own file size distribution. Polygraph supports the following distribution functions: constant, uniform, exponential, normal, lognormal, Zipf, sequential, and tabular.
Hit ratios	Servers may generate a mixture of cachable and uncachable responses to meet specified hit ratio targets.
Working set sizes	The size of the working set is configurable. The contents of the working set change over time.
Object popularities	The object popularity distribution mimics web sites with broad appeal, such as Yahoo, CNN, Microsoft, etc.
Object life cycles	Polygraph objects have a life cycle, which determines when the object content is updated. It is important for generating validation requests.
Embedded objects	This configurable feature models HTML pages that contain some number of embedded images or other content.
Hot subsets	This object popularity parameter makes some objects extremely popular for a short time, simulating flash crowds.

Table II. Summary of Polygraph features.

We described some of the most interesting models implemented in Polygraph, and discussed their strengths and weaknesses. Several models illustrated the trade-offs between high performance and precise simulation. Where appropriate, we presented the genesis of the models so that readers may learn from our mistakes.

This paper can also be viewed as a collection of efficient and often unique benchmarking algorithms; a collection containing in-depth coverage of these techniques along with their alternatives and interactions. We hope that Polygraph users as well as future benchmark developers will build on this foundation, just as we used the ideas and warnings of the benchmarks developed before Web Polygraph.

A potentially promising direction for future work is building a distributed testbed using Polygraph robots and servers communicating via the Internet. Such a testbed may help us to validate network delay models and also provide for an on-demand tests that require no setup on the user side.

We are also working on extending Web Polygraph workloads to test various URL- and content-based filtering devices, authentication services, and L7 load balancers.

Web Polygraph software, documentation, and results are freely available [RW03].

REFERENCES

[AC98] . Jussara Almeida and Pei Cao. Measuring proxy performance with the Wisconsin Proxy Benchmark. In *Proceedings of the Third International Web Caching Workshop*, Manchester, UK, June 1998. `http://www.cs.wisc.edu/~cao/publications.html`.

[Apa] . Apache HTTP Server Project. `http://httpd.apache.org/`.

[BC98] . Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 151–160, Madison, WI, June 1998. `http://www.cs.gmu.edu/conf/sigmetrics98/`.

[BCF$^+$99]. Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE INFOCOM*, 1999. `http://www.cs.wisc.edu/~cao/papers/zipf-implications.html`.

[Cao98] . Pei Cao. Characterization of Web proxy traffic and Wisconsin proxy benchmark 2.0. In *Web Characterization Workshop*, November 1998. `http://www.cs.wisc.edu/~cao/publications.html`.

[CB96] . Mark Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, May 1996. `http://www.cs.bu.edu/faculty/crovella/papers.html`.

[Dav99] . Brian D. Davison. A survey of proxy cache evaluation techniques. In *Proceedings of the Fourth International Web Caching Workshop*, San Diego, CA, April 1999. `http://www.cs.rutgers.edu/~davison/pubs/`.

[FGM$^+$99]. Roy T. Fielding, James Gettys, Jeffrey C. Mogul, et al. RFC 2616: Hypertext Transfer Protocol — HTTP/1.1, June 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[GB97] . Steven D. Gribble and Eric A. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997. `http://www.cs.berkeley.edu/~gribble/papers/sys_trace.ps.gz`.

[LML$^+$01]. Johnson Lee, William Miniscalco, Meng Li, W. David Shambroom, and John Buford. Analysis of Web workloads using the bootstrap methodology. In *Proceedings of the The Sixth International Web Caching and Content Delivery Workshop*, Boston, MA, June 2001. `http://www.cs.bu.edu/pub/wcw01/`.

[MJ98] . David Mosberger and Tai Jin. Httperf — A tool for measuring Web server performance. In *Workshop on Internet Server Performance*, Madison, WI, June 1998. `http://www.cs.wisc.edu/~cao/WISP98.html`.

[Pit98] . James E. Pitkow. Summary of WWW characterizations. In *Proceedings of the Seventh International WWW Conference*, Brisbane, Australia, April 1998. `http://www7.scu.edu.au/`.

[Riz97] . Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, January 1997. `http://www.iet.unipi.it/~luigi/research.html`.

[RW03] . Alex Rousskov and Duane Wessels. Web Polygraph — a proxy performance benchmark, 2003. `http://www.web-polygraph.org/`.

[Sta] . Standard Performance Evaluation Corporation (SPEC). `http://www.spec.org/`.

[Vöc] . Jens-S. Vöckler. The Blast project. `http://statistics.www-cache.dfn.de/Projects/blast/`.

[WC98] . Zhe Wang and Pei Cao. Persistent connection behavior of popular browsers. `http://www.cs.wisc.edu/~cao/publications.html`, December 1998.

[Wes03] . Duane Wessels. A distributed testbed for national information provisioning, 2003. `http://www.ircache.net/`.

[Yer99] . Gregory Yerxa. Speedy performance, rock-bottom price put Squid freeware on top. *Network Computing Magazine*, May 1999.