AutoPerf: Automated Load Testing and Resource Usage Profiling of Multi-Tier Internet Applications

Varsha Apte* T. V. S. Viswanath Devidas Gawali Akhilesh Kommireddy Anshul Gupta
Computer Science and Engineering Department
Indian Institute of Technology - Bombay
Powai, Mumbai 400 076, India

ABSTRACT

A multi-tier Internet server application needs to be analyzed for its performance before it is released. Performance analysis is usually done by (a) load testing of the application on a testbed and (b) building a performance model of the application. While there are a plethora of Web load-generator tools available, there are two problems with these tools: one, the tests have to be configured manually, which can lead to a time-consuming trial-and-error process until the desired performance charts in the appropriate load ranges are obtained; and two, the load generator tools do not produce output that is directly useful for creating a performance *model* of the application. In this paper, we present *AutoPerf*, a load generator tool designed to meet two distinct goals, named *capacity analysis* and *profiling*. The goal of capacity analysis is to run a comprehensive load test on a Web application, in an appropriately chosen range, at a minimal number of load levels, while still producing an accurate graph of throughput and response time vs load levels. The goal of profiling is to generate a detailed *server resource usage profile per request type*, without instrumenting the application code. This data (e.g. CPU execution time by Web server for one request) is crucial for parameterizing performance models of the application. AutoPerf intelligently plans and configures its load tests by using analytical results from queuing theory along with some heuristics. Results show that AutoPerf is able to run performance tests very efficiently while still producing an accurate chart of performance metrics.

CCS Concepts

•**General and reference** → **Performance;** *Measurement; Experimentation;* •**Software and its engineering** → **Software performance;**

*This work was supported by Tata Consultancy Services under the aegis of the TCS-IIT Bombay Research Cell.

ICPE'17, April 22-26, 2017, L'Aquila, Italy
 ISBN 978-1-4503-4404-3/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3030207.3030222

Keywords

Performance, Load generators, Profilers, Capacity analysis

1. INTRODUCTION

A multi-tier server system that supports a typical Internet application is usually subjected to *load testing* before its release. Load testing involves the synthetic generation of requests on the application deployed in a *testbed* at certain *load level* so that its performance under that load level can be measured. This load level is usually the *number of users* who are simultaneously engaged in a *session* with the application. A typical load test suite consists of experiments conducted at gradually increasing load levels until the application reaches *saturation*. Thus the primary purpose of load testing is to determine the *capacity* of this application on the testbed platform, usually in terms of the number of users of the type being emulated, that it can support.

Load testing is carried out using one of several *load generator tools* that are available freely or commercially [7]. These tools generally take as input a session description (list of URLs that a user accesses in a session), the user *think time*, and a test plan which typically specifies a starting load level, an increment and an ending load level. For each load level, the tools also need inputs such as warm-up time and overall test duration.

The important outcomes of such a load test are the graph of *throughput vs load level* and the graph of *response time vs load level*. The capacity of an application maybe determined either as the level after which throughput stops increasing, or by the level after which a response time target is exceeded. The nature of the throughput and response time curves is also of interest to a performance analyst so as to understand the behaviour of the application.

Load testing in this manner may require several iterations before the test is properly configured. For example, the range of load levels could be wrong - that is, the highest load level turns out to be too low with respect to application capacity, or the lowest load level could be too high with respect to the capacity. In both cases, if the performance-tester is not well-trained, performance tests could be misinterpreted, or it may take a long time and other resources to figure out and run the load test in the correct range.

The time duration of the load test is another parameter that can go wrong. It could be too short, in which case one may not get steady-state values of the performance metrics at each load level, or it could be too long, in which case the overall test may take a long time.

Another major pitfall of load testing, is running into *client bottleneck* - that is, the load generating client process itself reaches the maximum number of requests that *it* can generate on the local client resources. In our experience this is the most common cause of mis-interpretation of results. Typical server capacity is very high for modern servers, so a highly scalable load generator is required to "load" such servers. Sometimes the capacity of the application to serve requests exceeds the capacity of the load generator clients to issue requests. In such a case, when throughput flattens out, if a performance analyst is not an expert, the capacity of the *client* may get reported as the capacity of the server.

In this paper, we present *AutoPerf* - an automated load testing tool for *multi-tier* Web applications. AutoPerf requires no specification of load level ranges or test duration or warm-up time for performing a capacity analysis test suite that results in throughput and response time charts as a function of load levels. AutoPerf produces this chart with a minimal number of load tests, and ensures that the metrics represent steady-state values. Furthermore AutoPerf uses a platform-independent heuristic to be self-aware of its own bottlenecking, and reports such bottlenecking to the user.

The key to making AutoPerf work intelligently is to run *profiler agents* on each of the servers, which send server profiling data to the load testing controller which runs on the load generating client. The AutoPerf controller incorporates several intelligent mechanisms, some of which make use of the profiling data and apply well-known results from queuing theory, to run the load test efficiently. We summarize the novel mechanisms being used in AutoPerf in the following:

- *Minimal configuration*: The input provided to Autoperf just consists of the description of the URLs to be fired, a probabilistic user session in terms of these URLs, user think time, the IP addresses of the servers, and the identifiers of the server processes. No other configuration such as range of user load levels or duration of a test is required. It auto-determines parameters such as:
  - *Minimum Load Level*: AutoPerf ensures that load tests at an unnecessarily low load level are not run. It uses profiling data sent from the server to determine the load level that will utilize the bottleneck CPU to some minimum configured level and sets that as the minimum of the range.
  - *Maximum load level detection*: For CPU intensive applications, AutoPerf determines the maximum load level at which load tests need to be performed by using the CPU service demand at each tier, which is used in the *Kleinrock saturation heuristic for queuing networks* [9]. Thus, it works for a multi-tier application. For non-CPU intensive applications, where this heuristic can overestimate the maximum load level, AutoPerf detects such overload using a heuristic based on the "power of a queue", and breaks out of tests at that load level.
  - *Intermediate load levels*: In between the minimum and the maximum load levels, AutoPerf runs load tests at an "optimal" number of load levels - that is, it intelligently selects load levels in such a way that the "actual" shape of the curve is seen, yet load testing is not done at fine intervals.
  - *Auto-setting the duration at a load level*: AutoPerf auto-detects whether "steady state" has been achieved at any particular load level and terminates the test at a load level when the average values of performance metrics have converged.
- *Client bottleneck detection*: AutoPerf uses a novel indicator of client bottlenecking - if the *achieved* think time as estimated using Little's Law is much larger as compared with the configured think time, AutoPerf concludes that there is a client bottleneck. This is a more reliable indicator of client bottleneck than client CPU utilization.
- *Resource demand profile per request, per tier, per resource*: AutoPerf can be run in *profiling mode*, where it takes as input a list of URLs whose resource demands need to be profiled, and the load level at which they need to be profiled. It runs the suite of tests required to give the following per-request resource demands for each URL, at each tier: CPU ms per request per server process, Network bytes read and written per request per host, disk bytes read and written per request per host. This type of data is crucial for *extrapolating* results from load tests to scenarios for which tests have not been carried out, by using models that need such inputs.

The rest of the paper is as follows: Section 2 presents the detailed design of AutoPerf and Section 3 presents experiments conducted to validate and evaluate AutoPerf's mechanisms. Section 4 discusses related tools and methodologies and Section 5 concludes the paper.

2. AUTOPERF DESIGN

One of AutoPerf's main goals is to automatically run a suite of load tests on a typical multi-tier Web application, so that graphs such as Throughput vs Number of Users, and Response Time vs Number of Users as shown in Figure 3 are generated without having to manually give inputs such as minimum load level, maximum load level, step size, duration of one load level, etc. We call this the *capacity analysis* mode of running Autoperf. Thus, AutoPerf was designed with the goal of producing the "curve" of these metrics vs load levels, up to saturation and a little beyond, with only the input from the user that is absolutely required to produce these curves.

Autoperf's second goal is to calculate resource demands such as CPU ms per request at each server process on each host, disk bytes read and written per host per request, and network bytes read and written per host, per request. We call this the *profiling* mode of running Autoperf.

The key components in AutoPerf architecture that support these goals are *profilers* that run on the server hosts.

Thus, AutoPerf's input specification (given in the form of an XML file) consists of only two required blocks: one containing the URLs and session description, and one naming the hosts (IP addresses) and server processes names which are possible bottlenecks in the Web application, which should be profiled for their resource demands. AutoPerf accepts user session description in the form of a probabilistic

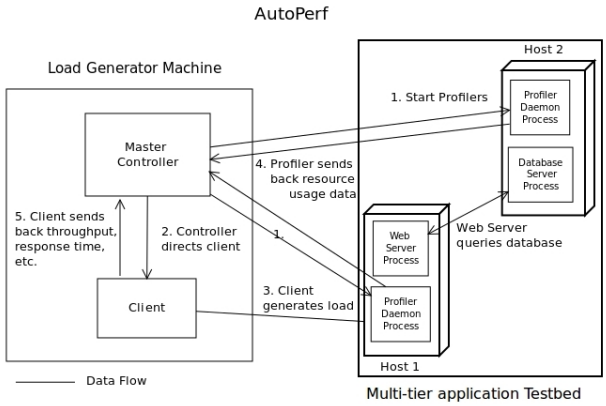


Figure 1: AutoPerf Architecture

navigation graph i.e. the navigation probability from one URL to the next, called a *Customer Behaviour Model Graph (CBMG)* [10] and the think time between getting a response and issuing the next request.

Apart from these inputs, there are some configuration parameters such as error thresholds and maximum number of requests or profiler reading thresholds that AutoPerf uses in its automation mechanisms, which are specified in a separate configuration file (*config.properties*). However, unless the user is an "advanced user", this file need not be edited for running a typical test.[1] In the rest of this section we describe the architecture and algorithms that achieve this automation.

2.1 AutoPerf Architecture and Algorithms

Figure 1 shows an overview of AutoPerf's architecture.

AutoPerf consists of two main modules: *Profilers* and the *Master Controller*. Profilers are daemon processes that run on the host machines on which server processes are running. A profiler is responsible for accumulating resource usage information for a particular process. Thus AutoPerf requires one profiler per server process. Profilers are instantiated before generation of load. Profilers start and stop collecting measurements when they receive instruction from the Master-Controller. The profilers which are currently build for Linux systems, use existing Linux measurement utilities such as *ps, vmstat* and *netstat* to sample and send these values to the controller.

The master in itself has two main components: controller and client (which generates load on the application).

In *capacity analysis* mode, AutoPerf has to determine two main settings using these components:

- The *load levels* (number of users) at which tests should be run. This involves determining the minimum load level, the maximum load level, and a *minimal* set of intermediate load levels such that a faithful throughput vs load level curve is produced, from low throughput, to saturation throughput.
- The *duration* of each load test. The throughput and response time that we obtain from each load test is expected to be at *steady state*, so AutoPerf needs to ensure that adequate "warm up" has happened. Furthermore, AutoPerf uses the CPU service demand of a request at each server in its load level selection algorithm, thus the duration of the test should be such that an accurate estimate of this parameter is obtained.

[1]AutoPerf does offer a "manual override" option, where we can specify the exact load levels at which to run load tests at, and the duration of each such test.

Before a capacity analysis run begins, the controller first simply initializes the profilers. Then, it starts its load level selection mechanism, which determines the load level and instructs the client to generate load (HTTP requests) at that load level. After a load test at a particular level starts, the Controller instructs the client to continue to generate load until *throughput convergence* has been achieved (this is a simple check based on the difference between consecutive time-averaged values of throughput falling under a certain threshold). At this point, the client stops sending throughput and response time measurements to the controller, but continues to generate load on the server for the purpose of calculating the *CPU service demand* per request. The controller now directs the profilers to start *profiling* the server.

The controller now starts receiving *cumulative service demand* measurements from the profilers (cumulative CPU ms used by each server process specified in the input file, network bytes per host, and disk bytes per host). It uses this and client-side request counts to estimate the average service demand per request for various server resources. Load generation continues further until *CPU service demand convergence* check has been satisfied. Note that in capacity analysis mode, this service demand is a *weighted average* of CPU service demands of the various requests in the specified session.

Now the controller uses this average service demand to estimate the minimum and the maximum load levels.

We note here that many of the mechanisms related to determining load levels in AutoPerf are currently centered around *CPU service demand per request* of the server processes. This is because AutoPerf is primarily built for typical transactional Web applications that tend to be heavier in their CPU usage. However, AutoPerf uses a different set of algorithms, not dependent on CPU service demand for determining these parameters when the bottleneck tier is not CPU-intensive.

In the following subsections, we describe the details of the various mechanisms for automated load-testing and profiling that are built into AutoPerf. Since the CPU service demand estimation is at the core of many other algorithms we describe that algorithm first.

2.1.1 Estimation of CPU service demand per request

The profiler agent deployed at the server periodically (every 1 second) sends the cumulative service time of the server process to the autoperf master. In Unix-based systems, it gathers this value using "ps" utility. The master keeps track of number of URLs $n_{urls}(t)$ processed till time t. Then, the service time estimate at any point of time t is calculated as:

$$\tau_{p,CPU}(t) = \frac{S_{p,CPU}(t)}{n_{urls}(t)}$$

where $S_{p,CPU}(t), n_{urls}(t)$ and $\tau_{p,CPU}(t)$ are the cumulative CPU execution time of the process p till time t, the number of URLs completed in this load test till time t and the CPU service demand estimate at process p at time t.

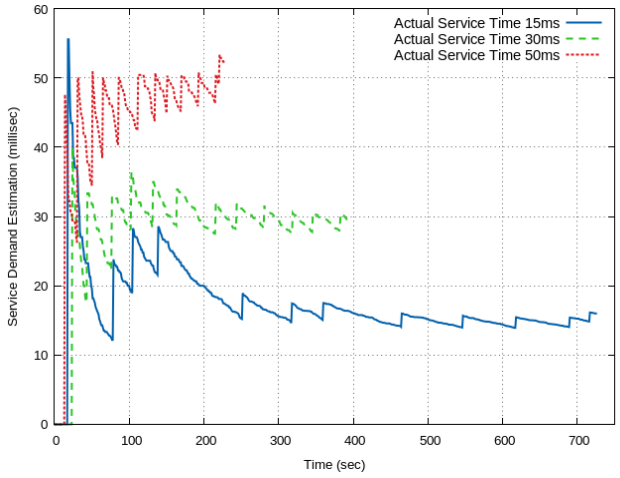


Figure 2: Service Demand Convergence Algorithm

While this basic method is obvious and straightforward, there is a challenge of *granularity* in this. The ps utility has a granularity of 1 second. Thus the cumulative service demand sent from the server is rounded off to the nearest integer value. Thus the service time estimate curve is a saw tooth curve (Figure 2) because until one second worth of execution time time is not accumulated, the cumulative execution time value $S_{p,CPU}(t)$ does not increment. The decreasing trend in the graph is when $S_{p,CPU}(t)$ remains the same but $n_{urls}(t)$ increases. The "jump" is when there is an increment of one second in $S_{p,CPU}(t)$. Thus, since the CPU service demand estimate keeps changing, a mechanism is required to check whether the value has converged.

The convergence detection has to avoid some pitfalls - because of the saw-tooth nature of the curve, we cannot use a routine check of the percentage difference between consecutive estimates decreasing below a threshold - since this method can easily lead to a false convergence detection just before a "jump" in the curve.

One way would be to detect convergence when the relative jump magnitude itself goes under a certain threshold. However it can be shown that, the relative jump magnitudes are not necessarily monotonically decreasing. Thus, we could again falsely detect a local convergence. To overcome this, we use an additional check of detecting a *global* convergence of the jump sizes, and only then check for *local* convergence of the service demand estimates.

Suppose consecutive readings of the profiler are $S_{p,CPU}(t)$ seconds of cumulative CPU service demand, with $n_{urls}(t)$ URLs completed, and $S_{p,CPU}(t+1)$ seconds of cumulative CPU service demand, with $n_{urls}(t+1)$ URLs completed. Suppose there is a jump in the service demand estimate; i.e.

$$\frac{S_{p,CPU}(t)}{n_{urls}(t)} < \frac{S_{p,CPU}(t+1)}{n_{urls}(t+1)}$$

The jump magnitude as a percent of the previous estimate is

$$\Delta\tau_{p,CPU} = \frac{\left(\frac{S_{p,CPU}(t+1)}{n_{urls}(t+1)} - \frac{S_{p,CPU}(t)}{n_{urls}(t)}\right) \times 100}{\frac{S_{p,CPU}(t)}{n_{urls}(t)}}$$

Since $n_{url}(t+1) > n_{url}(t)$

$$\frac{S_{p,CPU}(t+1)}{n_{url}(t+1)} < \frac{S_{p,CPU}(t+1)}{n_{url}(t)}.$$

Thus

$$\Delta\tau < \frac{\left(\frac{S_{p,CPU}(t+1)}{n_{url}(t)} - \frac{S_{p,CPU}(t)}{n_{urls}(t)}\right) \times 100}{\frac{S_{p,CPU}(t)}{n_{urls}(t)}}$$

Now, suppose the host on which the process p is running has N cores. It is obvious that the maximum possible difference between $S_{p,CPU}(t)$ and $S_{p,CPU}(t+1)$ on this server is N seconds (in the extreme case that all N cores are busy with the process p for the one second between t and $t+1$).

$$\Delta\tau = \frac{(S_{p,CPU}(t+1) - S_{p,CPU}(t)) \times 100}{S_{p,CPU}(t)} < \frac{N \times 100}{S_{p,CPU(t)}}$$

Thus a strict "global" convergence check can be that

$$\frac{N \times 100}{S_{p,CPU}(t)} < \epsilon \tag{1}$$

since this implies that $\Delta\tau < \epsilon$ where ϵ is a pre-configured threshold. Note that once this inequality is satisfied for t, it will be satisfied for all future values $u > t$ since $S_{p,CPU}(t)$ is monotonically increasing with t. Thus, this indicates a global convergence of jump sizes.

While this condition works well, we find that it is overly strict and convergence detection can take a long time. This is because at low load, accumulating an $S_{p,cpu}$ value that is large enough so that the "maximum" possible percentage difference goes below a threshold can take a long time. Instead we use a heuristic to arrive at an "expected" difference. Consider a load test at a user level L, the running average response time $R(L)$ and a thinktime γ. The expected number of URLs completed in one second is given by the expected throughput of requests. For a closed system the throughput is given by

$$\Delta(L) = \frac{L}{\gamma + R(L)}$$

Given this, we can argue that at most each of these requests could have kept one core busy for the entire second, in the one second profiler interval. Thus, the consecutive busy time values ($S_{p,CPU}(t+1)$ and $S_{p,CPU}(t)$) can then be expected to be separated by $\Delta(L)$. For a server with N active cores, we redefine $\triangle S$ as

$$\triangle S = \min(\Delta(L), N)$$

When the load level and thus the throughput is low, and N is high, we expect that the difference will be $\Delta(L)$. In our modified global convergence check we replace the N in Equation 1 by $\triangle S$. We flag global convergence as achieved when

$$\frac{\triangle S \times 100}{S_{p,CPU}(t)} < \epsilon.$$

After global convergence is achieved, a simple "local" convergence check is made where the variation in the values of a moving window of consecutive service demand estimates is calculated (maximum value - average value in the window). Local convergence is flagged as achieved when this difference goes below a configured threshold.

While this approach eventually does result in a good estimate of the average service demand, it may still sometimes take prohibitively long time for the service demand to converge, if the execution time is very small, the load level is low, and the think times large. This results in a low rate of requests being sent, which in turn results in the cumulative CPU execution time not increasing fast enough. Thus the profiler may keep reporting zero cumulative CPU time for a long time (5-6 hours in an experiment that we ran).

To address this problem, we use the following heuristic to break out of the service demand estimation loop: after a certain max number (p_{max}) of profiler readings have been received, if the maximum of the cumulative CPU execution times reported for all the server processes is still zero, we assume that the cumulative processing time on a bottleneck server was some value S_{low}, which is currently set to a value less than 0.5 seconds (since ps has 1-second granularity). If this break happens at time t, the CPU service demand is then estimated as

$$\tau_{p,CPU}(t) = \min\left(\frac{S_{low}}{n_{urls}(t)}, R(L)\right). \quad (2)$$

Here we use $R(L)$ as the estimate of bottleneck service demand at low load, if it is the lower of the two estimates. If the load test results in such "timing out" it sets a flag which indicates that "load level is too low", which indicates to the Controller that it should keep searching for the correct "minimum" load level at which to run a load test.

There is another case that has to be considered: this is where the maximum cumulative CPU execution time among all the server processes reported is positive, but a few server processes report zero cumulative CPU execution time, even after p_{max} readings. In this case, we set an "ignore" flag for each of these server processes, which indicates that the controller will not wait for convergence of service demand for these processes. Figure 2 shows three examples of varying service demands, estimated fairly accurately by AutoPerf.

2.1.2 The Load Level Selection Algorithm

AutoPerf's load level selection mechanism is designed to solve two distinct problems:

- Figuring out the range - i.e., the *maximum* and the *minimum* load levels that the load tests should be run for. The maximum level should ideally be the one where the application achieved its maximum throughput. However, we have designed AutoPerf so that it generates load just a little beyond this maximum. We set the minimum to whatever keeps the bottleneck server CPU busy at least ρ_{min} fraction of time, where ρ_{min} is a configurable tool parameter whose default value is 0.1.
- Once the range is known, *minimizing* the number of load levels in between the minimum and the maximum, at which a load test is run while still generating a precise graph of throughput vs load levels.

Determining the minimum load level

Our goal is to set the minumum load level to that which achieves a certain minimum CPU utilization ρ_{min} on the "bottleneck" CPU server. For this, AutoPerf needs an estimate of the CPU service demand per request on this bottleneck server. AutoPerf starts its capacity analysis experiments at minimum load level 1. As described in the above section, if the loadtest "times out" with a "load level too low" flag set, the controller uses the service demand estimate from that run to estimate the minimum load level N_{min} it should run at, by using Utilization law as follows:

$$\frac{Nmin}{R(N_{min}) + \gamma} \times \tau_{p,CPU}(N_{min}) = \rho_{min} \times n_{cores}$$

$$N_{min}(L) = \frac{R(N_{min}) + \gamma}{\tau_{p,CPU}(N_{min})} \times \rho_{min} \times n_{cores}$$

$$N_{min}(L) \approx \frac{R(L) + \gamma}{\tau_{p,CPU}(L)} \times \rho_{min} \times n_{cores}$$

where $R(N_{min})$ and $\tau_{p,CPU}(N_{min})$ are estimated by $R(L)$ and $\tau_{p,CPU}(L)$ respectively. Note that we assume here that no other system resource bottlenecks before the bottleneck CPU reaches ρ_{min} utilization.

Determining the maximum load level - CPU intensive application

AutoPerf uses Kleinrock's saturation number formula [9] as a heuristic to estimate the maximum number of users the server system can support. It can be shown that a closed queuing network with M queuing stations, where expected service demand at station i is τ_i, and average user think time is γ, can support approximately N_{max} number of users, where

$$N_{max}(L) = \frac{\gamma + \sum_{i=1}^{M} \tau_i(L)}{\tau_b(L)} \times n^b_{cores} \quad (3)$$

where $\tau_i(L)$ is the CPU service demand at tier i at load level L, and $\tau_b(L) = \max_i \tau_i(L)$ is the *bottleneck CPU service demand* at load level L and n^b_{cores} is the number of CPU cores at the bottleneck tier.

Thus, at the end of a load test at load level L, AutoPerf estimates all the service demands $\tau_i(L)$, based on measurement data sent by the profilers, and calculates the $N_{max}(L)$ using the above formula. Note that this estimate can keep changing after each test at different load levels, since the service demand estimates may keep changing.

The problem with the suggested method is that the result is valid only when the application has CPU bottleneck. When CPU is not the bottleneck, the N_{max} value determined by using CPU service demand is an overestimate, and results in AutoPerf running load tests beyond the *saturation* region, in the *overloaded* region of the capacity of a server system. We have a mechanism to detect this situation. We will first describe AutoPerf's basic load level selection algorithm, and then present the overload detection mechanism.

Minimizing the number of tests carried out

AutoPerf tries to minimize the number of experiments required to produce a "smooth" graph of throughput vs number of users. It does this by using a simple approach where the throughput curve is approximated by a linear segments. A recursive algorithm, is called after load tests at the two endpoints of a range L_{min} to L_{max} have been carried out. When called for this range, it firsts runs a load test at load level $L_{mid} = \frac{L_{min}+L_{max}}{2}$. Let $\Lambda(L)$ denote the throughput measured at load level L. Then if

$$\Lambda(L_{mid}) \approx \frac{\Lambda(L_{min}) + \Lambda(L_{max})}{2}$$

then no more load tests are required for this region. If not, then the algorithm is recursively called for ranges (L_{min}, L_{mid}) and (L_{mid}, L_{max}). After running tests in the

minimum to maximum range, AutoPerf runs tests at every load level from the estimated maximum to a level slightly beyond this maximum.

In summary, AutoPerf first starts at Load level 1, and then uses the minimum Load level selection algorithm to ramp up to the minimum load level (N_{min}). Using the CPU service demand estimates from the load tests run so far, it estimates the maximum load level N_{max} of this application. The recursive algorithm is called with this N_{min} and N_{max} as its L_{min} to L_{max} range. Note that since CPU service demand estimates are constantly updated, we may have a new estimate of N'_{max} at the end of this run. In this case, the recursive algorithm is called again in the (N_{max}, N'_{max}) range.

Determining the maximum Load Level - non CPU intensive application

The above algorithm works fine when CPU is indeed the bottleneck resource of this server system. However, when this is not the case, the N_{max} estimate can turn out to be much higher than the actual capacity of the system. When experiments are run in the overloaded range of a server, the throughput values can be highly unstable, resulting in a jagged curve (see Figure 4) and the algorithm that is attempting to find the linear segments to fit this jagged curve can take a long time to terminate. Thus, we need a mechanism to detect that (a) the load levels seem to be in overload region and (b) the experiments seem to be stuck in a jagged part of the curve.

We use a state-machine based approach to address this. The state machine detects whether the current series of experiments are in a "high load" region (load near saturation) and further whether they are in an "overload" region (load beyond saturation and system unstable).

The heuristic we use to detect "high load" uses the metric of *power of a queuing system* ($P(L)$) [9] which is given by

$$P(L) = \frac{\Lambda(L)}{R(L)}$$

This is combined metric which helps us determine the ideal load level at which a queue provides good throughput - too low would result in low throughput, and thus low power, and too high would result in high response time and thus low power. As load tests are done, we keep track of the current maximum power (P_{max}) and the load at which that was achieved (L_{Pmax}) and the current maximum throughput (Δ_{max}) and the corresponding load ($L_{\Delta_{max}}$). The state machine starts in a normal load state (state 0). If the following conditions are detected at load level L:

$$P(L) < P_{max} \quad \text{and} \quad L > L_{P_{max}}$$
$$\Delta(L) < \Delta_{max} \quad \text{and} \quad L > L_{\Delta_{max}}$$

then the state machine moves to the next state (+1), which indicates possible unstable state. The conditions essentially tell us that the load level L is an "overloaded" load level, since both Power and throughput at that level are lower, even though the load level is higher than those levels at which these metrics had higher values. If this happens K_{ovld} consecutive times, this indicates being "stuck" in the jagged part of the curve and an overload state is confirmed. Otherwise, the state machine moves back to the normal load state.

If overload state is confirmed, we terminate the experiment run and report the summary of results. Note that because of the way the recursive algorithm described earlier progresses, where the left side of the range is called first, we can assume that when we are in the saturated region of the curve, the tests corresponding to the load levels to the left of this region have already been carried out.

2.1.3 Client Bottleneck Detection

One of the biggest pitfalls of closed loop load testing is that often it is the load generating *clients* that are bottlenecking. Thus as we see in later the experiments in Figure 5, the throughput flattening could indicate either a client bottleneck, or server bottleneck. We have developed a heuristic independent of the hardware on which AutoPerf is run, to detect client bottleneck. At the end of a run at a load level, the think time achieved in the experiment is calculated using Little's law for closed loop systems [9] as follows:

$$\gamma_{achieved} = \frac{L}{\Lambda(L)} - R(L)$$

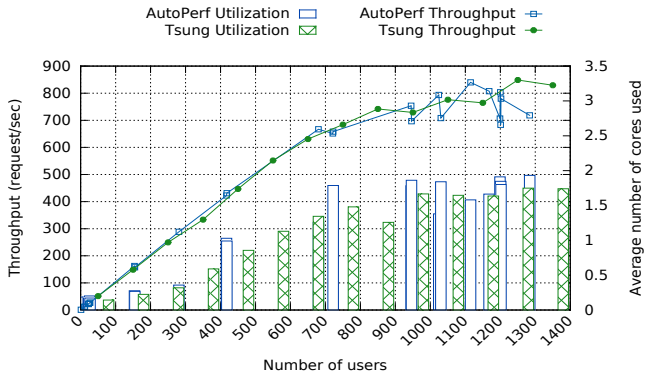
and is compared to the think time value provided as input. Our claim is that if the achieved think time is much higher than configured think time, this indicates that the tasks related to generating the requests in a load generating client were queuing for resources and not issuing requests in a timely manner. This indicates a client-side bottleneck. In AutoPerf, if achieved think time is higher than the configured think time by a given threshold, we declare a warning to the user on the screen. This alerts the user in case the throughput is flattening out, that it could be due to client capacity limit, rather than server capacity limit.

2.1.4 AutoPerf Profiling Mode

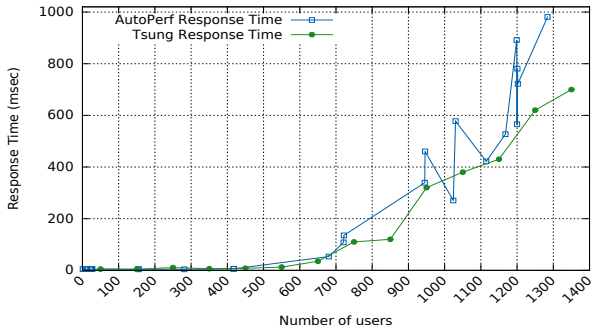
In this mode, the purpose of running an experiment is not to find saturation throughput of an application. Instead, the user specifies a list of URLs that (s)he would like to *profile*, i.e. find per-request resource demands for. AutoPerf profiles requests by generating load of only one type of request. E.g. if a "login" request has to be profiled, AutoPerf issues only logins, and uses the CPU service demand convergence algorithms to output an accurate estimate of this parameter for all the server processes being profiled. It also calculates disk bytes read and written per host per request, and network bytes read and written per host per request. Note that if in a session based application a request requires, say, a "login" before it can be executed, this can be specified to AutoPerf. AutoPerf executes the "pre-requisite" requests before it profiles the subsequent requests.

3. EXPERIMENTAL EVALUATION

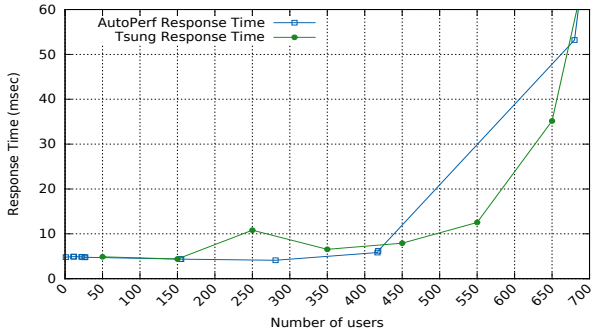
In this section we present results of experiments done to evaluate the various automated mechanisms built into AutoPerf. We also evaluate the scalability of AutoPerf itself, since the ability to generate high load is one of the most important requirements of a load generator tool. We do this by comparing AutoPerf with popular load generator tools such as Tsung [15] and JMeter [5]. On the server end, we evaluate AutoPerf under various scenarios such as a very high capacity ("infinte server") system, server system with CPU bottleneck, server system with non-CPU bottleneck, and under a well-known benchmark called "DellDVD", in which we validate AutoPerf's mechanisms while using a probabilistic navigation graph.



(a) Throughout and Web Server CPU Utilization

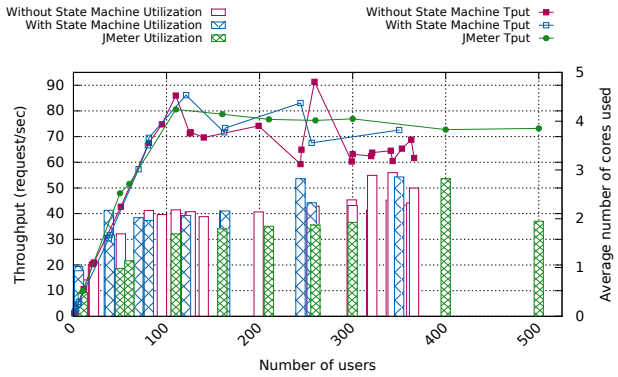


(b) Response Time

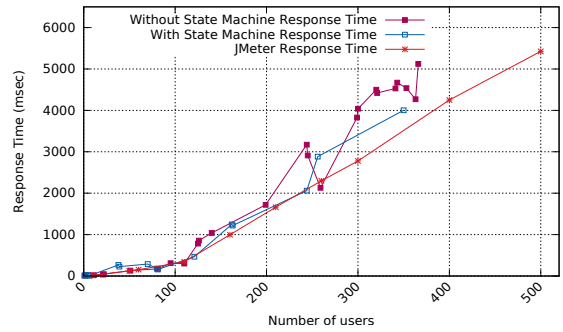


(c) Response Time Before Saturation

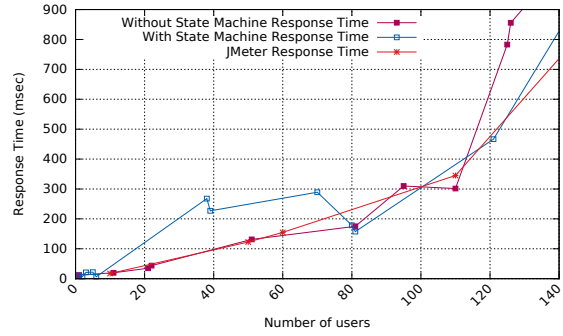
Figure 3: Correctness validation - CPU bottleneck with AutoPerf-Capacity Analysis mode and Tsung



(a) Throughout and Web Server CPU Utilization



(b) Response Time



(c) Response Time Before Saturation

Figure 4: Correctness validation - non CPU bottleneck with AutoPerf-Capacity analysis mode and JMeter

3.1 Accuracy of AutoPerf's mechanisms

To validate AutoPerf's ability to faithfully recreate a throughput and response time graph that a correctly but manually configured tool would produce, we run it in "capacity analysis" mode to automatically run load tests for two types of applications: CPU intensive, and non-CPU intensive and compare the results with existing load generators Jmeter [5] and Tsung [15].

3.1.1 CPU intensive Application

The application we used here is *DellDVD*. This is a two-tier DVD store application with a Web server tier and a database server tier. The testbed consisted of a 16 core Intel(R) Xeon(R) CPU E5-2650 v2 with 16GB RAM, with only *two* cores enabled which hosted the Web server and an Intel(R) Core(TM) i5 CPU 650 host having 4 cores as the database server. The DellDVD database was populated with 5000 users. The client was a 16 core AMD Opteron(TM) Processor 6212 machine with 16GB RAM. The client and servers were connected by a 1 Gbps switched network. In the user sessions, each user navigated probabilistically between four DellDVD URLs, with a think time of 1 second. We ran AutoPerf in capacity analysis mode for this application, and compared it with running Tsung with DellDVD request probabilities that corresponded to AutoPerf's CBMG values, with the same think time, in a range of 50-750 users in steps of 50. The duration of each load level was set to 180 seconds. We verified manually that this time duration was giving steady-state results.

Figures 3 shows the comparison between the throughput and response time graphs produced by AutoPerf and Tsung respectively. Figure 3a also shows the Web server CPU uti-

lization. We see that the CPU utilization (shown in the units of average number of busy cores) reaches 2, which shows that the Web server CPU is the bottleneck in this application (the DB server CPU had very low utilization).

We can see that AutoPerf is able to largely match the graphs that Tsung produced, thus basic correctness of AutoPerf measurements is validated. Note that all the parameters such as minimum and maximum load levels, and test duration, were arbitrarily chosen and manually configured for Tsung, whereas AutoPerf would have determined everything automatically. We can see that AutoPerf is able to correctly estimate the maximum load level and achieves its goal of performing experiments until throughput saturates. Note that this would have required the service demand convergence algorithm to work properly. Its minimal intermediate load levels selection algorithm reproduces the shape of the manual curve faithfully. At the low load end we can see that AutoPerf searches for the minimum load level for some time. Thus, this experiment validates AutoPerf's saturation number based maximum load level heuristic for systems with a CPU bottleneck, and validates its minimal intermediate load levels selection algorithm. It also confirms that in every load test run, AutoPerf is giving steady state results close to Tsung's thus validating its warm-up detection mechanism.

AutoPerf finished this capacity analysis run in 832 seconds (14 mins) while Tsung finished in 45 minutes. Thus, AutoPerf took much lesser time than Tsung and still gave almost the same results.

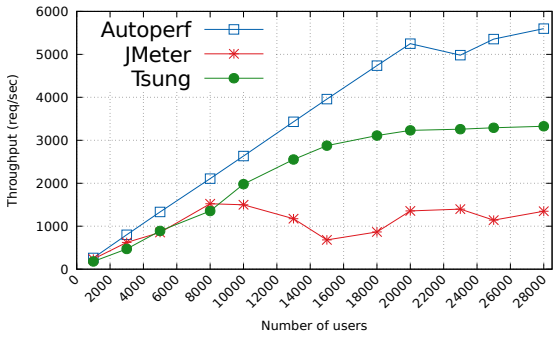
3.1.2 *Non CPU-intensive Application*

In this experiment we used a simple php script on the Web server, which was writing random characters onto the disk to behave like a I/O bound application. The user session consists of just this URL with a think time of one second. The server side script uses an exclusive lock mechanism, so only one user can write at a time. Thus, the writes are serialized and represent the bottleneck and the CPU is not the bottleneck. The Web server tier was hosted on an Intel(R) Xeon(R) CPU E5-2650 16-core machine with 16 GB RAM, with 4 CPU cores enabled and load generator client was hosted on Intel(R) Core(TM) CPU i7-4790 8-core machine with 8 GB RAM. The client and the servers were connected by 1 Gbps switched network.

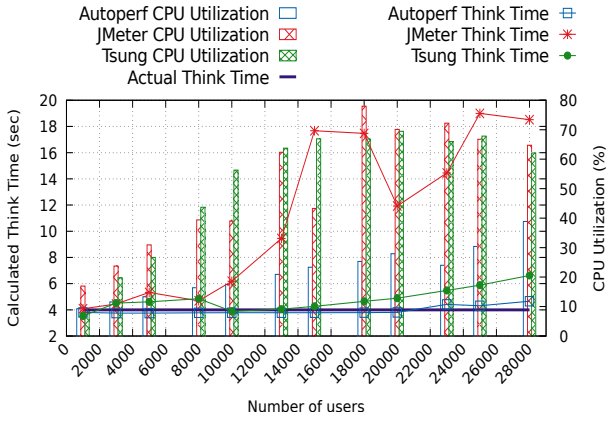
Figure 4 shows AutoPerf experiments with the power and throughput heuristic-based state machine enabled and disabled respectively. Figure 4a also shows the bottleneck (Web server) CPU utilization in units of average number of busy cores. We can see the the cores were not fully busy even when throughput of the application flattened, thus the CPU is indeed not the bottleneck in this application. Without the state machine, AutoPerf carries out many experiments in the overloaded region. With the state machine enabled, AutoPerf was able to detect the situation when the experiments got "stuck" in the jagged area in the overloaded region of the curve, and ran the capacity analysis at fewer load levels to generate the curve in Figure 4.

3.2 Scalability Analysis

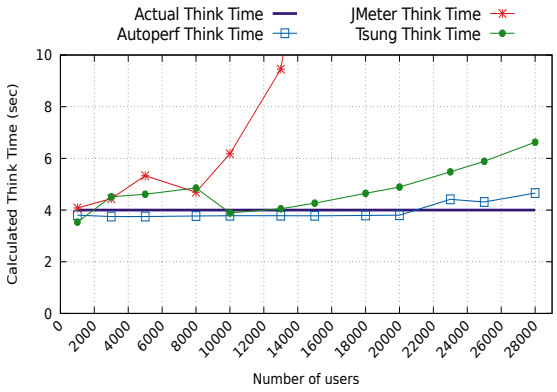
For these experiments the application used is a Web server serving a simple blank HTML file. This was done so that we had a server with very large capacity, that would exceed the capacity of all the load generators, and therefore would reveal *their* capacity. We call this server the "infinite server". These experiments are divided into three progressive parts: i) First, where each load generator simply fires one static URL at the server; ii) the second, where the load generators create dynamic URLs which have name-value pairs where the values are read from a file and iii) dynamic URL with name-value pairs read from a file with a probabilistic URL generation where possible. AutoPerf does this using a CBMG, and Tsung takes unconditional URL probabilities as input. This allows us to characterize the load generator scalability when it has to do varying degrees of work. We expect



(a) Throughput vs Load Level

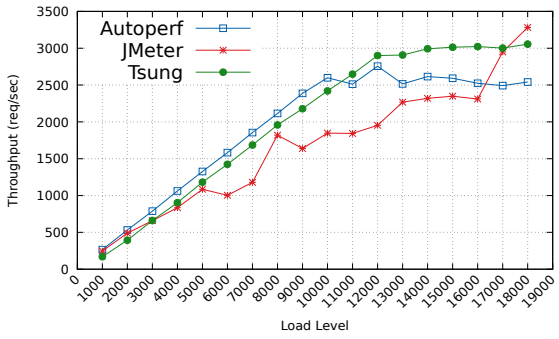


(b) Achieved Think Time and Client CPU Utilization

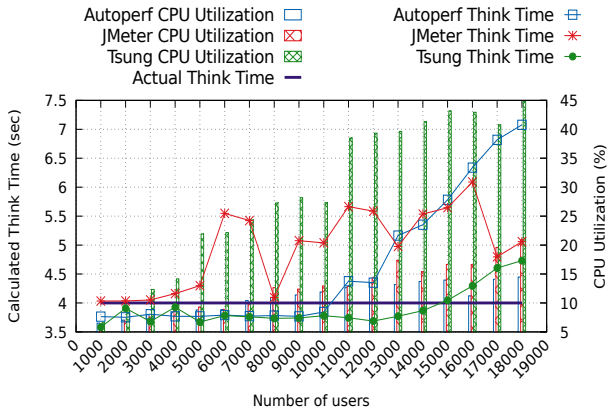


(c) Achieved Think Time

Figure 5: Experiments for scalability with AutoPerf, Tsung and JMeter with "Infinite server" and configured think time 4 sec, single URL of blank HTML file

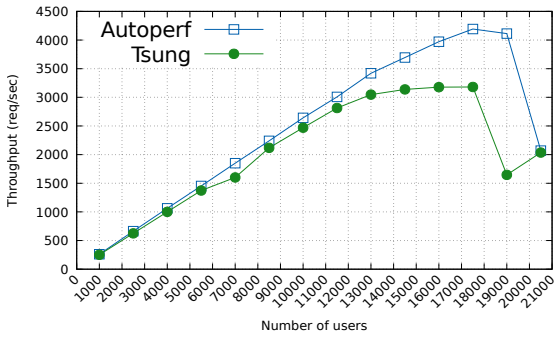


(a) Throughput vs Load Level

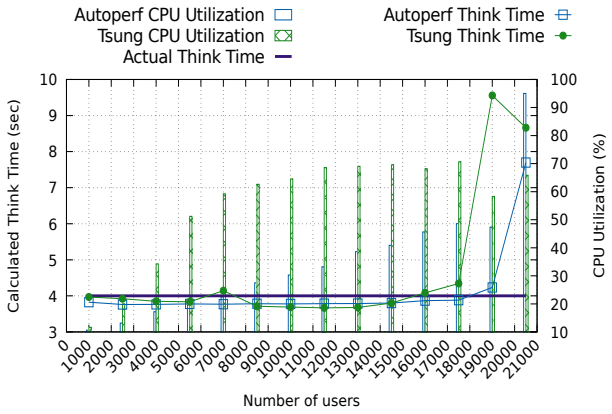


(b) Achieved Think Time and Client CPU Utilization

Figure 6: Experiments for scalability with AutoPerf, Tsung and JMeter with "Infinite server" and configured think time 4 sec, Reading Name-Value pairs from a file



(a) Throughput vs Load Level



(b) Achieved Think Time and Client CPU Utilization

Figure 7: Experiments for scalability with AutoPerf, Tsung and JMeter with "Infinite server" and configured think time 4 sec, Name-values from file with probabilistic URL generation

the static URL firing to be least client resource-intensive and (ii) and (iii) to be more client resource intensive.

The client machine used was 4 core AMD Opteron(TM) Processor 6212 (max Frequency 2.6 GHz). We had 10GB of memory allocated to java heap size for the runs. The server machine was 24 core Intel(R) Xeon(R) CPU E5-26200 (max Frequency: 2 GHz). The power governor settings at both the client side and the server side were set to performance. The apache2 version 2.2.22 was deployed on the server. The prefork module was enabled with MaxClient value set to 100000. We ran the experiments at various load levels from 1000 to almost 20000 users. Each test at a load level was for one minute.

Figures 5, 6 and 7 show the results of the experiments done with tools AutoPerf, JMeter and Tsung with configured think time of 4 seconds and for the three respective workload scenarios described earlier.

Figure 5a shows the throughput vs load level graphs for each of these load generators for the static single URL case. Note again that since the server is of "infinite" capacity, these graphs show us the *load generator* capacities. For this particular case, AutoPerf achives the highest maximum throughput at user load level of 20,000 with a request throughput of approximately 5000 requests/second. Tsung also bottlenecks at 20,000 users with a maximum throughput of about 3000 requests/second. Jmeter does most poorly in this experiment, bottlenecking at 8000 users, with a maximum throughput of just 1500 requests/second. Figure 5b shows the load generator ("client") CPU utilization and also the "calculated" think time for the three load generators, which we claim is the actual *achieved* think time by applying Little's Law to this closed loop system. Since Jmeter's achieved think time values dominated this graph, Figure 5c shows the think time at a zoomed in scale so that AuoPerf's and Tsung's achieved think time values can be observed properly. As discussed in Section 2.1.3, we claim that the bottlenecking of closed loop load generators can be detected in terms of this achieved think time. We observe that around the load level at which each load generator starts bottlenecking and throughput starts flattening out, the achieved think time also start inflating as compared to the configured think time of 4 seconds. Jmeter displays a rather inconsistent value of achieved think time for lower load levels also, however AutoPerf and Tsung show think time inflation around the level that their througputs flatten. However, it is important to note that *client CPU utilization* in all cases remains below 70%, which by *does not* indicate exhaustion of local CPU resources and would not have indicated client bottlenecking. Thus, think time inflation is a much more reliable indicator of client bottlenecking, or some other misbehavior of the load generator tool.

Figure 6a shows the same comparison for the scenario of reading name-value pairs from a file. In this case, Tsung

Table 1: Request resource demand of web and DB tier after profiling of an application using AutoPerf

Transaction	Service Demand (ms)		N/W Packets Read(KB)		N/W Packets Written		Disk Blocks Read		Disk Blocks Written	
	Apache	MySQL	Apache	MySQL	Apache	MySQL	Apache	MySQL	Apache	MySQL
Login	0.5875	0.4197	343930	161480	364076	124637	0	8	2236	631
SearchActor	1.5443	0.0922	202556	69400	233046	53424	4	0	4264	162
SearchCategory	2.4188	0.0568	176440	43245	182050	33375	0	0	292	139
SearchTitle	1.5952	0.0948	190662	63798	218086	49193	0	0	492	146

achieves the highest throughput, but AutoPerf is not too far behind. Jmeter again displays the most unstable curve. Figure 6b shows the achieved think time vs load level for the three load generators, which again is able to track the client bottlenecking fairly closely. We can see that the load levels at which the clients bottlenecked were lower for all the load generators than those for the static URL case. E.g. AutoPerf could scale to 20,000 users with the static URL vs about 14,000 users with dynamic URL generation.

Figure 7a shows the throughput graph in the case where AutoPerf is using a CBMG, and Tsung is using direct equivalent URL probabilities. Jmeter is not included in this comparison as it does not have probabilistic session generation capability. In this scenario, AutoPerf achieves significantly higher throughput than Tsung. The corresponding think time graph in Figure 7b again shows inflation in think time at high loads, around the same load levels that the throughputs were flattening out, even when the client CPU was not bottlenecking. Surprisingly enough, AutoPerf scales to 18,000 users with 4200 requests/second throughput in this (more resource intensive) case as opposed to 14,000 users with 2500 requests/second in the previous case. At this point we do not have a good explanation for this anomaly.

We conclude from these experiments that AutoPerf scalability is in a similar range or in some cases better than existing high-capacity load generators. Our experiments also confirm our think time heuristic for client bottleneck detection in closed loop load tests. We believe this is a powerful check that every load testing exercise should do to ensure that the client is behaving as it has been configured to behave, and eliminate the possibility that the throughput capacity determined in a load test, is of the *client* rather than the server.

3.3 AutoPerf Profiling Mode results

AutoPerf can be used to determine the per request resource demands at each tier, for each type of request in a multi-tier system. Table 1 shows the profiling results for four profiled request types in DellDVD at the Web tier and the Database tier. We note that doing this task manually for each request for each tier is extremely tedious and would require huge amount of manpower resources.

AutoPerf is the only tool that we know of that can produce this result. These results expand the use of AutoPerf greatly, from an efficient capacity analysis tool, to a tool that can be used in a pipeline of measurement and modeling tools to *predict* the performance of an application using *models*. For example, per request service demands are a critical input required for parameterizing queuing models of such systems [6].

4. BACKGROUND AND RELATED WORK

Performance tests can be carried out on multi-tier Web applications using one of a very large number *load testing* tools available either free or commercially [7]. Jiang and Hassan [8] present a comprehensive survey and classification of all aspects of load testing tools. In this section, we discuss a few popular tools, and a few other tools that have comparable goals to those of AutoPerf.

One of the most used load testing tool is Apache JMeter [5]. JMeter is a GUI-based tool written in Java. The JMeter test plan for a web application consist of a session of HTTP requests defined with other workload parameters such as number of users, running time, user ramp up time, think time etc. JMeter fires the requests deterministically in a sequential manner, completes a session and collects the result. The reports collected by the "listener component" of JMeter show standard metrics such as total number of request generated, average response time, throughput etc. The JMeter architecture depends on plugins which allow it to be extended to offer many additional features. E.g. a plugin for JMeter server agent is available, which can be used to monitor the server and report server metrics such as CPU usage, memory and disk statistics, etc.

Tsung is another feature-rich tool which provides great flexibility in defining the test environment (client and server configuration), user behaviour and load distribution. The test plan for a Web application consist of sessions configured with a probability value, where each session is composed of transactions and each transaction consists of a group of HTTP requests. Each virtual user chooses a session based on the probability value and fires URLs accordingly. Tsung runs for a predefined time and provides average response time and highest server throughput achieved for all requests, transactions as well as each session defined in an XML file which is given as input. It also retrieves server activities i.e. CPU activity, load average and memory usage by communicating to server agents or monitoring through Munin or SNMP.

There are numerous of other load generator tools that can be found on the Internet [7], however JMeter and Tsung are among the most feature-rich. However, these tools are also essentially built to be manually driven by a performance tester, who by trial and error can figure out the correct range in which to run the tests for, and the duration of the test. When the server capacity is not known, this can be a time and resource intensive process. The application in this testbed could have very high capacity - say 10,000 users and the analyst could spend a lot of time running tests in a low range. On the other hand, the tester could run tests in too high a range, and keep getting only saturation results.

Finally the range could be correct, but the step size could be unnecessarily small, or too large.

The duration of the test is another pitfall. Assuming that steady-state results are desired, the test could be run for too short a duration, resulting in non-representative results, or it could be run for too long a duration, which will result in too much time for the overall experiment. If the tester is not a trained performance engineer, such errors could result in misinterpretation of the application performance.

Some existing methodologies have been proposed that address this problem. The CLIF benchmarking service [14] automatically determines the saturation level and aims to never run an experiment at a level higher than server capacity. CLIF uses response time at the single user load test as an indicator of the service time at the server, and models the server as an $M/M/K$ queue, where the algorithm tries to guess the number of servers K. It uses a specified "fineness factor" to determine the number of load levels at which the load tests should be run in the range between single user and saturation level. CLIF can also "scale itself" when it detects that its own CPU usage is exceeding a certain threshold, and use new virtual machines to generate additional load.

A methodology which also addresses the problem of static and manual setting of load testing parameters was proposed by Shivam et al [13] in the context of NFS server benchmarking. They propose various mechanisms such as binary search for "searching" for the saturation load level. They determine the duration and number of repetitions of load tests using statistical techniques of confidence determination. The primary goal of their work though, is "mapping response surfaces" - i.e. generating the performance surface as a function of workload, server configuration and amount of resources available.

BenchLab [4] is a load testing framework that addresses the problem of making load tests much more "realistic". It uses real web browsers to emulate users with automated load injection tool, uses traces to replay realistic workload and uses cloud resources to place clients in geographically distributed locations. There is minimal interaction between these clients.

Rain [2] is load generation toolkit that can generate "open" load, closed load as well as a mix. It is also the only other tool we know that accepts a probabilistic navigation graph to describe a user session. It can also generate time-varying load.

While the above tools and methodologies address some of the problems in efficient running of load tests, they still have some drawbacks. Tsung, JMeter, BenchLab, Rain and a host of other load generator tools are clearly not made for automated load testing runs. While CLIF does automate saturation detection, it is not clear how CLIF's response time heuristic works in the case of multi-tier applications - where response time is not equal to the bottleneck server's service time, and in the case when service demand changes with load levels - which is very often the case with real-life applications. While Shivam et al's methodology [13] addresses many issues raised here, their goal is different from our goal of generating throughput and response time vs load level curves for multi-tier Web applications.

A second very important need that is not met by any existing tool that we know of is that of collecting data that can be used to parameterize *performance models* of an application. While a load test reveals important characteristics of an application, performance modeling is an important step after performance testing, which requires very different quantitative inputs. Queuing models are parameterized by *service demands* of each different class of requests. The service demand is the resource usage requirement of *one request* at any server resource. Queuing models are used so that performance can be predicted for different request mixes, and different server platforms. Collection of service demands requires very different kinds of test and measurement.

There are some methodologies proposed to address the problem of measuring CPU service demand. Several make use of request logs and utilization data from production systems to estimate the CPU service demand per request type, using Utilization Law [3, 11, 16]. Some estimate the service demand by measuring response times and throughputs [1]. However, to our knowledge no existing *load testing tool* provides this information, and none gives per-request resource demands for resources other than the CPU.

5. SUMMARY AND CONCLUSIONS

In this paper, we presented an intelligent load generator and resource usage profiling tool AutoPerf, which completely takes the tedium of trial-and-error out of the process of load testing of multi-tier Web applications. It additionally offers a unique feature not offered by any other load testing tool - that of discovering *service demands of individual requests* at various server resources. This is immensely useful in creating performance models of multi-tier applications.

AutoPerf does this by working symbiotically with the theory of queuing systems - it uses some simple results such as Utilization Law, Little's Law, the Saturation Number Heuristic, the "power of the queue" metric to inform its load testing process so that it can be run automatically with very little configuration, and is self-aware of its own bottlenecking. On the other hand, it produces measurement data that can be used to parameterize queuing models of the application.

Out results showed that AutoPerf can run an efficient load test that produces throughput and response time graphs that are very close to those produced by a conservatively configured load test - i.e. one with a fine step size and a long test duration.

There are several directions we plan to expand AutoPerf in, in the future. Firstly, the self-bottleneck detection can be trivially employed to have AutoPerf auto-scale to use a bank of client load generator machines, if one machine is bottlenecking. This work is in progress. Further currently, AutoPerf only reports server-side resource usage profiles. We can build an additional capability of AutoPerf carrying out *diagnosis* regarding performance bottlenecks using these server-side measurements. We also plan to enhance AutoPerf so that it explores server-side configurations that improve performance. Lastly, while a tool *VirtPerf* [12] was built using AutoPerf, for performance benchmarking of virtualized applications, it needs to be enhanced so that it can provide additional parameters required to create performance models of applications on virtual machines.

6. REFERENCES

[1] M. Awad and D. A. Menascé. On the predictive properties of performance models derived through input-output relationships. In *European Workshop on*

Performance Engineering, pages 89–103. Springer, 2014.

[2] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, and D. A. Patterson. Rain: A workload generation toolkit for cloud computing applications. Technical Report UCB/EECS-2010-14, EECS Department, University of California, Berkeley, Feb 2010.

[3] G. Casale, P. Cremonesi, and R. Turrin. Robust workload estimation in queueing network performance models. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 183–187. IEEE, 2008.

[4] E. Cecchet, V. Udayabhanu, T. Wood, and P. Shenoy. Benchlab: an open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 4–4. USENIX Association, 2011.

[5] H. H. Emily. *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Limited, Birmingham, 2008.

[6] D. Gawali and V. Apte. The m3 (measure-measure-model) tool-chain for performance prediction of multi-tier applications. In *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*, pages 30–35. ACM, 2016.

[7] R. Hower. Web site test tools and site management tools. http://www.softwareqatest.com/qatweb1.html#LOAD.

[8] Z. M. Jiang and A. E. Hassan. A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, 41(11), Nov 2015.

[9] L. Kleinrock. *Queueing systems, volume II: Computer applications*. Wiley interscience, 1976.

[10] D. A. Menasce, V. A. Almeida, L. W. Dowdy, and L. Dowdy. *Performance by design: computer capacity planning by example*. Prentice Hall Professional, 2004.

[11] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi. Cpu demand for web serving: Measurement analysis and dynamic estimation. *Performance Evaluation*, 65(6):531–553, 2008.

[12] P. Patil, P. Kulkarni, and U. Bellur. Virtperf: a performance profiling tool for virtualized environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 57–64. IEEE, 2011.

[13] P. Shivam, V. Marupadi, J. S. Chase, T. Subramaniam, and S. Babu. Cutting corners: Workbench automation for server benchmarking. In *Proc. USENIX Annual Technical Conference*, pages 241–254, 2008.

[14] A. Tchana, B. Dillenseger, N. De Palma, X. Etchevers, J.-M. Vincent, N. Salmi, and A. Harbaoui. Self-scalable benchmarking as a service with automatic saturation detection. In *Proc. Middleware 2013*, pages 389–404. Springer, 2013.

[15] Tsung is an open-source multi-protocol distributed load testing tool. http://tsung.erlang-projects.org/.

[16] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*, pages 27–27. IEEE, 2007.