ipbench — A Framework for Distributed Network Benchmarking

Ian Wienand

Gelato@UNSW, University of NSW

<ianw@gelato.unsw.edu.au>
http://www.gelato.unsw.edu.au

Luke Macpherson

School of Computer Science and Engineering, University of NSW

<lukem@cse.unsw.edu.au>
http://www.disy.cse.unsw.edu.au/

ABSTRACT

Existing network benchmark tools suffer from problems such asportability, poor repeatability and inability to perform accuratetesting of high-throughput networks.

We have built a distributed benchmarking tool which solves many of these problems. ipbench provides an extensible distribution framework and makes minimal demands of the device under test. We have fully implemented a latency test with many tunable parameters and demonstrated moving other tests into the framework.

1. Introduction

ipbench[1] is a distributed, extensible suite of tests for gathering reliable performance data from data networks. Presently, internet and ethernet protocols are supported, however the addition of other protocols is feasible.

There were four main factors as driving motivators for the creation of ipbench. The first reason was the scarcity of recognised benchmark software in our domain. We require targeted, highly tunable benchmarks that allow us to focus on evaluating the overheads of an operating system's handling of network packets. Many existing benchmarks are focused on userspace or underlying network benchmarking rather than on the network protocol stack and operating system implementation.

Secondly was the lack of scalability in existing tests. Gathering performance data often requires the generation of throughputs beyond which can be generated by a single host. To this end, some form of coordinated distributed operation is required. For example: gigabit Ethernet is currently mainstream, popularity of 10 gigabit Ethernet is growing and even faster optical technologies exist. Generating sufficient load with small packets on a single client is generally not possible at these speeds. Further still, with the high speed of modern processors overheads are often so low that useful and reliable results can only be acquired with **a lot** of data moving very quickly. Whilst the problem can often be worked around with existing tools using a combination of remote shell scripts, regular expression code on outputs and programmer persistence, it was felt this was not generally a good way to go about testing with multiple clients.

Thirdly was the API requirements of most existing benchmark suites. The general architecture of the existing tools is to use a client-server paradigm between the testing machines and the device under test (DUT; where appropriate we use terminology from [1]). Whilst we believe it is reasonable to expect the client to implement the full range of APIs expected of a modern operating system, this does not hold for the target device. For example, BSD sockets may not be provided on experimental operating systems, and they may not have anything near POSIX compatibility. Another significant requirement is architecture portability enabling easy testing in varied environments.

Finally was the lack of extensibility of existing benchmark tools. The authors had some ideas for a number of tests that could not easily be contained in existing network performance tools. Our goal was to make it easy to add new tests into the framework. Analysing the testing procedure of many existing tools by source code review is often a frustrating process, we wanted

1. This work is sponsored by The Gelato Foundation http://www.gelato.unsw.edu.au and National ICT Australia http://www.nicta.edu.au.

to avoid this frustration for others as much as possible.

2. Related Work

There are a number of existing benchmark applications that failed to meet our criteria.

At the lowest end of the scale, a tester can always use ad hoc testing with tools such as `ping` (with the associated flood option) and `time` to get some idea of network performance. This is not, however, accurate to any great degree. Often researchers interested in IP performance will construct their own small benchmarks suitable to their work. This reduces the repeatability of their experiments for other interested parties and raises the possibility of uncertainty in test results.

NetPIPE [7] is one well known benchmark that provides latency and bandwidth results for a wide variety of environments, one of which is TCP. The first major hurdle to using NetPIPE in our work was that even the latest released version was not 64 bit safe as it makes assumptions about the size of `unsigned long` - whilst a trivial and common problem it failed our requirement of portability. Even with these problems fixed we saw anomalous results (we did not fully investigate these further and they may not have been related to NetPIPE). Architecturally, NetPIPE is started in either a test or a listen mode; by requiring the full application to run at both ends of the test it also failed our API requirements. There is also no inherent remote operation in the code.

`Netperf` [4] is another well-known network benchmark suite. Copyrights in the source code reveal it has existed for at least 10 years and is still actively maintained.

Whilst `Netperf` is a widely used benchmark it did not meet our requirements in a number of areas. `Netperf` follows a client-server model and requires a separate `netserver` process to be run on the DUT, hence failing the aforementioned API implementation requirements. `Netperf` does not have any inbuilt distribution of operation and it is not generally suited to having multiple testers. Clever features such as running a test multiple times to get results that satisfy a particular confidence interval unfortunately fail when used with multiple testers as each tester lacks a global view of the results. `Netperf` also lacks some features we considered necessary for measuring OS performance such as tunable warm up and cool down periods.

SPECweb99 [8] is a common test of overall webserver performance with a similar multiple-client/single server architecture to `ipbench`, but we found it unsuitable for a number of reasons. While SPECweb99 indirectly tests the underlying network architecture, there are many other variables that affect its performance. Compontents such as choice of web server and unrelated parts of the kernel such as disk caching and memory utilisation code make significant contributions. It also has a coarse grained reporting mechanism, only providing the number of sustained connections over a time period. Thus it fails our first requirement of giving insight at the operating system level of network handling. It requires a fully functioning HTTP server with a large data set on the DUT, failing our API implementation requirements. The test is also not freely available, which we consider a disadvantage. This said, our experiences with the distributed client architecture of SPECweb99 has had a positive influence on our independent design of `ipbench`.

Other common tools for network benchmarking are a combination of `httperf` [6] and a web server such as `µserver` [2, 3]. `httperf` requires a server capable of accepting HTTP requests and recommends the use of multiple clients to generate sufficient and reliable loads on the server. Some of the ideas for `ipbench` came from our rudimentary patching of `httperf` for synchronised remote operation. We see our work as being complementary to these tools; whilst they are excellent at providing insight into the overall performance of a kernel with something approaching realistic work (i.e. a HTTP server) we hope to provide more details on the network stack implementation utilised by these tools.

3. Architecture

The overall goal of `ipbench` is to concatenate results from a number of synchronised remote testers each individually performing the same test at a single DUT (see Fig. 1).

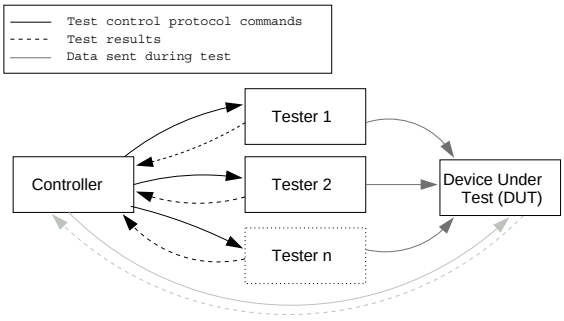

Figure 1: Overview of the `ipbench` architecture. Dark lines represent essential components of the test, lighter lines are optional components. Medium grey lines represent data sent during the test.

Specifically

1. The user interacts with the controller, specifying the test (plus any parameters) and the target machine to test.
2. The controller sets up and synchronously starts the remote testers.
3. Testers run and complete the specified test.
4. The controller receives test results from remote testers, aggregates the data and reports the results to the user.

We have designed the basic architecture of `ipbench` with three components: a controller, any number of testers and a target DUT.

The testers are controlled by the controller, and execute their test procedures targeted at the DUT. On each tester an instance of `ipbenchd` is started which listens for incoming test requests, sets up and performs the test and finally relays its results to the controller. `ipbenchd` is targeted at running on a fully featured client as it may have significant memory or processing requirements. Ideally the tester would have two interfaces — one to the controller network and one to the DUT network, however no information is passed between tester and controller during testing so not having this setup should not adversely affect results.

The controller is also expected to be fully featured as it will be aggregating the data from the (potentially many) clients.

In our earliest revisions both the testers and the controller have all tests inbuilt statically. Later revisions allow the loading of tests via prebuilt shared objects.

We have made it possible for the DUT to execute a "companion" testing procedure and return results as part of the test (for example, to measure used CPU time on the DUT). For this case, the controller will need a physical interface able to talk to the DUT and the DUT must be running the `ipbenchtd` test daemon (illustrated by the lighter lines in Fig. 1). This is an optional component and whilst it may provide useful extra information, there is no requirement that it be used. Thus the range of services required by the DUT varies; our existing test requires nothing more than a standard `echo` service, other tests may require more complex services such as HTTP.

We found this architecture to be very useful in practice; persons interested in testing can run the controller on their local PC, receiving the result data directly and using it as they wish. The testers, as remote daemons, once started require no further intervention.

3.1 Protocol

The controller and clients use a simple protocol to interact. A typical session begins with the controller sending an `IPBENCH_SETUP` message to the testers `ipbenchd` daemon containing the index of the test to run, the DUT hostname and port, and any test arguments. Each tester responds with either `IPBENCH_SETUP_OK` or one of two error conditions, `IPBENCH_SETUP_ERR` or `IPBENCH_BUSY` (if a test is already running).

If the user specifies the companion test should be run on the DUT during testing, a similar procedure is followed with the `ipbenchtd` daemon running on the DUT.

Once all components have reported in, the controller sends an `IPBENCH_START` command to all testers (and the DUT, if required). Initially, the option of passing a timestamp or holdoff time to begin the test was proposed, however during development it became clear that the implementation of such a feature should be left to each individual test (see the description of the warmup and cool down periods of the latency test on page 166 for an example). Each tester proceeds to execute its test, and when finished marshals its results to send back to the controller. At this point, some processing may be done by each tester, and if one decides the test was inaccurate for some reason (for example, the standard deviation of results was too high) it may flag its data as invalid.

Once the controller has received return data from each tester, if the DUT is running a companion test the controller will send an `IPBENCH_STOP` message. The DUT will stop its test, marshal its data and send it to the controller.

Once all data is received by the controller it is unmarshalled, analysed and any required calculations performed. Finally, the controller will output final results ready for further analysis.

3.2 Test Interface

We designed the test interface to be as simple as possible. Such an open framework moves programming work to test authors, but maximises the extensibility of the program.

The interface below must be implemented by each test[2]. The interface is generally straightforward, the only moderately complex parts are the marshalling/unmarshalling of arguments and the eventual passing of arguments to the output function. By clever use

2. Later versions implement tests as loadable shared objects and use a slightly updated structure with some extra header information.

of structs and casting the resulting code can be quite simple.

```
struct client_data
{
  void *data;
  size_t size;
  int valid;
};

struct test
{
  char *name;         /* Test name    */
  int id;             /* Test ID      */
  char *descr;        /* Description  */
  int default_port;   /* Default Port */

  int (*setup)(char *hostname, int port,
      char* arg);

  int (*start)(struct timeval *start);
  int (*stop)(struct timeval *stop);

  int (*marshall)(void **data,
      size_t *size, double running_time);
  void (*marshall_cleanup)(void **data);
  int (*unmarshall)(void *input,
      size_t input_len, void **data,
      size_t *data_len);
  void (*unmarshall_cleanup)(void **data);

  int (*output)(struct client_data *target,
               struct client_data data[],
               int nelem);

  struct test_target_code *target_code;
};
```

The setup(), start() and stop() functions should behave as their names imply. The arguments to start() and stop() should be timestamped by the functions, the difference is passed to marshal() as the running_time argument of the test; this may be useful for calculations before sending back to the controller.

The client's marshal() should wrap up all data for sending back to the controller into its data argument. The client may do some analysis of its data and if it decides that the results are outside predefined limits may return a non-zero value to flag the testing data as invalid. The test should always marshal some data no matter how invalid — the information is still useful for debugging. Companion cleanup functions are provided to facilitate freeing of dynamically allocated memory.

The target_code pointer points to a structure very similar to the test structure which specifies that the companion test to be run by the DUT target daemon. At runtime, if the user specifies the companion test should be executed on the DUT (with the --target command line switch) it will be started, stopped and queried similar to a test client.

The controller will receive the data from each client and pass the data to the unmarshal() function; storing the result in an array (struct client_data data[]). The output() function is called with this array, a count of how many elements are in that array and, if available, results from the DUT companion test (struct client_data *target). The output() function thus has all the data from each client, and should analyse this information and construct output in a format suitable for further analysis.

The valid flag of the client_data struct will only be set for data that was flagged as correct by the marshal() functions; it is up to output() to decide on the overall validity of the test in some form or another. It may choose to ignore some invalid values, however if it decides all of the test data is outside acceptable parameters it may return a non-zero value and the test will be rerun up to a number of times (as given by the user with --test-retries).

4. Tests

4.1 Latency

The first test instrumented with our tool was a latency test. The standard method for measuring latency is to send a short request to the DUT and await a reply for that request, measuring the time delta. Our test requires only a standard echo service to get this information. The crucial part of the test occupies a single function of around 60 lines of C code. It is a single threaded test utilising non-blocking IO.

The test has a number of tunable parameters as shown in Table 1. We have not found any other network testing tool that makes it as straightforward to measure latency at varying throughputs, and we feel this is a major advantage of our tool.

Argument	Description	Default
bps|Mbps	Throughput to attempt	10Mbps
size	Size of messages in bytes	100
nodelay	Set TCP_NODELAY	yes
bufsiz	Send and receive buffer size	OS Default
warmup	Warmup time in seconds	5s
cooldown	Cooldown time in seconds	5s
socktype	Socket type (TCP/UDP/RAW)	TCP
sockopts	Socket Options	
iface	Interface to use (RAW only)	eth0
drop	Time before considering UDP packet dropped in seconds	2s

Table 1: Tunable parameters for the latency test

The test works with each client recording a large number of individual latency samples (statically selected at compile time). For each sample a packet of *size* is sent, and when that

packet is received the latency (elapsed time) recorded. The typical size of a result during our testing was a total of ~ 10MiB (7 clients with 200,000 individual latency results, each a 64-bit integer).

Throughput control is achieved by keeping track of the number of packets actually sent versus the number required to be sent to achieve the specified throughput *bps*. In the situation where the DUT is falling behind (i.e. not responding fast enough) the client will simply sustain sending packets at the requested throughput. In the advent the client gets ahead, it will busy loop until required to send the next packet, effectively throttling itself. We found this method more reliable than putting the process to sleep whilst waiting to send the next packet.

Each client will calculate the *actual* throughput it achieved and report this figure in its results. Ideally, this will be within a few percent of the requested throughput — if it is not it indicated the DUT could not keep up with the requests.

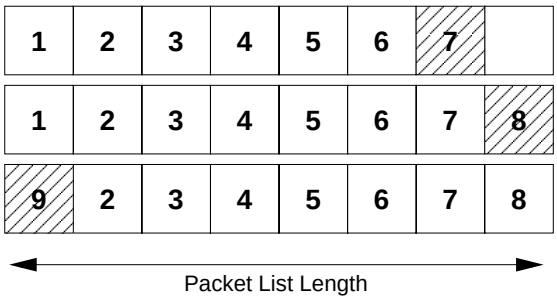

Figure 2: An example of filling the packet list to handle UDP packets. Numbers represent index of sent packets. Filled boxes represent the packet list head pointer.

To handle UDP (where guarantees are not made about packet delivery) a high performance method of calculating dropped packets was required. For this we implemented the *drop* parameter which influences the length of an outstanding packet list. As we know the packet size and the throughput, we can work out the length of the packet list required for a given *drop* value. Figure 2 illustrates the process. By keeping a sequential index in each echoed packet and a head of the queue pointer, any unreceived packets with $index_{head} - index_{received} \geq packet\_list\_length$ are considered timed out and dropped.

RAW packets are created by simply packing a specific protocol header, source and destination MAC address and a 64 bit stamp into a Ethernet frame. By utilising RAW packets the IP stack of the operating system should be bypassed, giving a direct reflection of the latencies incurred by the driver and underlying network. The receiving end must implement an echo service which takes any Ethernet frame with the specific protocol code and swaps the source and destination addresses and sends the frame back. The only requirement for running a raw test is that the target must be passed as a colon deliminated MAC address, rather than a hostname/IP address.

Warmup and *cooldown* times are critical to allow both the DUT and the network time to stabilise before running the test. Each client will run for the warmup time before starting to take latency results, and will continue to run without taking results for the cooldown time. This amortises any slight overlapping of clients starting and stopping their measurements.

The DUT can also be requested to keep track of its CPU usage thanks to a modified version of `cyclesoak` [5]. System usage is calculated with a low priority process that "soaks" all available idle usage. The CPU monitor will not record results during the warmup period, and discards any results taken in the last `cooldown` seconds from the time it is told to stop.

4.1.1 Test Examples

Some examples of data collected with the latency test are presented in Fig 3.

These examples were collected while testing the relative performance benefits of varying interrupt holdoff times for a network card driver. Interrupt holdoff describes the amount of time to wait before delivering another interrupt to signal an incoming packet. This allows the kernel to process more of the already received packets before being interrupted to handle any new ones, increasing throughput. The corollary is that latency (amount of time to respond) goes up, as the network card is deliberately holding back processing of packets.

The latency test can also be used to indirectly measure throughput (Fig. 4). Each client reports the *actual* throughput it achieved for each run, and these can be summed to get a measure of the total throughput for the network card. Measuring throughput like this requires multiple runs, each time increasing the throughput requested from each client.

4.2 Discard Test

The discard test is similar to the latency test, however the round trip time of the packets is not considered. At the DUT end, a special kernel module that simply counts the incoming packet and discards it is run. These values are reported back by the `ipbenchtd` invocation on the DUT. The clients collectively count their outgoing packets, and at the end of the test the two figures

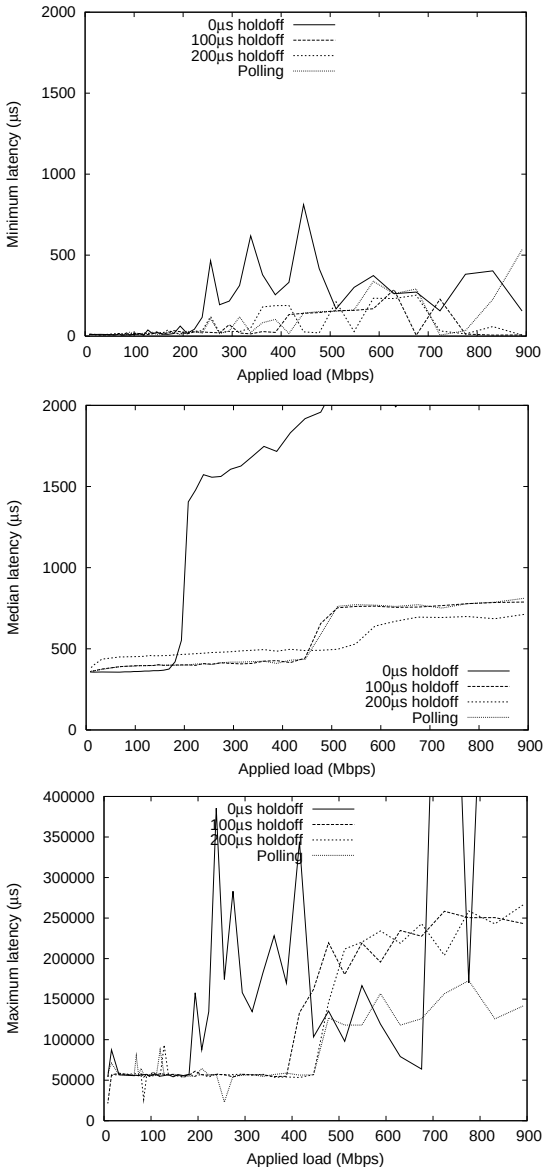

Figure 3: Minimum, median and maximum latencies for varying interrupt holdoff times, as measured by the `ipbench` *latency test.*

are compared to give a reference of how many packets were dropped.

4.3 `tbench` Throughput test

We have instrumented Andrew Tridgell's `tbench` (a subset of the `dbench` test [9]) as a form of throughput test. This test requires a fully featured DUT as it must run the `dbench` echo program.

Each of the clients runs the `tbench` test much as it would if run "out of the box"; however we have the added advantage that we can easily synchronise many distributed clients. The controller aggregates all results to give a total throughput.

This benchmark has not been extensively tested in this environment, however it was mostly instrumented as a proof of concept of our overall API architecture. It required less than a

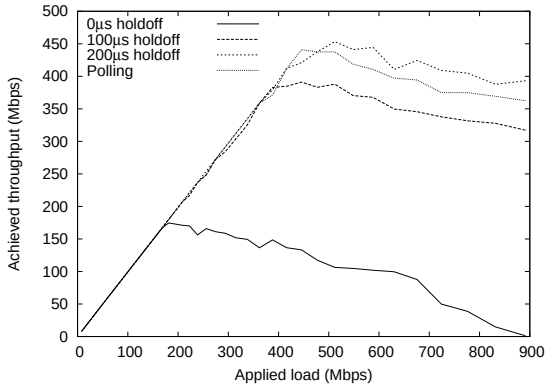

Figure 4: Achieved throughput for varying interrupt holdoff times, as measured by the `ipbench` *latency test.*

day of coding, and we feel it illustrates the flexibility of our approach.

4.4 Further Tests and Future Work

We have a number of ideas for further tests under various stages of construction.

- An explicit throughput test as either an expansion of the `tbench` test or re-implementing with something else.
- A SPECweb99 style HTTP server test. This would be a test that stresses a webserver.
- A distributed NFS benchmark where many clients can stress a single fileserver.

5. Experiences from implementation

After implementation, we have identified a number of areas where our implementation has succeeded and a number that require further thought.

5.1 Modularisation of tests

Allowing a fairly straight forward API for implementing a new test was a success and allowed rapid development and tweaking of tests.

5.2 Remote Invocation Protocol

During ongoing use it became apparent that it would be useful to bypass direct invocation by the controller and have the ability to manually interact with clients. This is not possible with the binary protocol `ipbench` implements. A simpler architecture is one such as that used by SPECweb99 where each client simply exports a control interface that is equally well utilised by the controller or a `telnet` session (assuming one knows the correct sequence of commands to send!).

Another important factor in designing the protocol, especially during early development,

is versioning. By making part of the protocol include a version stamp you can avoid the "Murphy's Law" case of mixing different development versions of clients and servers.

One criticism might be that we have implemented a protocol that could be realised with existing technologies such as RPC, CORBA, MPC or XML-RPC. These protocols tend to be aimed at at transferring complex states between distributed applications, something `ipbench` is not overly concerned with. The added complexity these protocols bring was decided to outweigh their advantages.

5.3 Error reporting

In a distributed system, reporting errors from many clients back to the controller is an important consideration. `ipbench` does not always handle this case well. Some way of flagging global exceptions and recovering all clients to a stable state is required for consistently reliable operation even after failure.

Additionally, `ipbench` was developed around a fairly ad-hoc state machine mostly designed in our heads. Formalising and implementing a state machine for client operation more concretely would also have helped with handling error conditions.

5.4 Division of tasks within the test

One area of partial success and partial failure is the division of work between the framework and the individual tests. For example, requiring data be marshalled into a simple array of bytes sent over the wire to the controller for unmarshalling was a successful design idea, as both test and client code was simplified by the assumption.

However, our initial design of having client code signal to the test to end after a specific period of time was not as successful. Not all tests run for a constant specified period of time; often they will be *measuring* the time to do a set amount of work. Those that do run for a constant time can easily implement their own alarm/signal handler to stop themselves, possibly taking timeout variables in their argument.

It is difficult to glean these insights before developing the tool, especially with the relative lack of prior work to go from.

5.5 Choice of language

C was probably the wrong choice of language for the `ipbench` daemons. The work they do is mostly confined to setting up communications (via sockets) and processing protocol commands. Many current scriptable languages such as Perl and Python make these sorts of tasks almost trivial with inbuilt libraries and simple string manipulation options.

However, writing tests in C is the correct choice. Tests such as the latency test are extremely performance sensitive, especially with regards to system calls. In this case the inbuilt convenience libraries of the scriptable languages become a liability as they often introduce unacceptable overheads. C also makes it straightforward to wrap existing tests (largely already written in C) into the `ipbench` framework.

6. Conclusion

We identified a number of problems with existing network benchmark suites such as lack of scalability, API requirements and portability concerns. We have implemented a new distributed testing framework which avoids these problems. We have fully implemented a latency test that has a number of unique options such as easily tunable throughput control and demonstrated easily extending the test. We have also demonstrated the ease of porting an existing test into the framework. At this time, our work is ongoing.

6.1 Code Availability

The code and documentation for `ipbench` is available at `http://ipbench.sourceforge.net`. It is released under the GPL.

References

1. Scott Bradner and Jim McQuaid. RFC 2544: Benchmarking methodology for network interconnect devices, March 1999. Status: INFORMATIONAL.

2. Tim Brecht and Michal Ostrowski. Exploring the performance of select-based internet servers. Technical Report HPL-2001-314, Hewlett Packard Laboratories, December 07 2001.

3. Abhishek Chandra and David Mosberger. Scalability of Llinux event-dispatch mechanisms. Technical Report HPL-2000-174, Hewlett Packard Laboratories, December 21 2000.

4. Rick Jones. Netperf — a network performance benchmark. Available at `http://www.netperf.org/netperf/NetperfPage.html`, cited Nov 2003.

5. Andrew Morton. cyclesoak— a tool for measuring system resource utilisation.,

December 2003. Available from http://www.zipworld.com.au/~akpm/linux/.

6. David Mosberger and Tai Jin. httperf - A tool for measuring web server performance. Technical Report HPL-98-61, Hewlett Packard Laboratories, March 30 1998.
7. Quinn Snell, Armin Mikler, and John Gustafson. Netpipe: A network protocol independent performace evaluator. In *IASTED International Conference on Intelligent Information Managment and Systems*, June 1996.
8. Specweb99 whitepaper. Available at http://www.specbench.org/web99/docs/whitepaper.html, cited Jun 2004.
9. Andrew Tridgell. dbench, May 2004. Available from http://samba.org/ftp/tridge/dbench/.