Exploiting Software Interfaces for Performance Measurement

Douglas P. Konkin
PMC-Sierra, Inc.
doug_konkin@pmc-sierra.com

Gregory M. Oster
Department of Computer Science,
University of Saskatchewan
oster@cs.usask.ca

Richard B. Bunt
Department of Computer Science,
University of Saskatchewan
bunt@cs.usask.ca

Abstract

Software performance measurement can be a difficult and tedious procedure, and this difficulty may explain the lack of interest shown in software performance optimisation in all but the most demanding areas, such as parallel computation and embedded systems. This paper describes the measurement shim, an approach to software performance which we have found to significantly reduce the effort required to make performance measurements. The measurement shim exploits the interfaces between software modules, and allows measurement at both data stream and procedure call interfaces. Experimental results indicate that the measurement shim provides high-quality data, and can be inserted with low impact on system performance.

1. Introduction

While there is much general interest in software performance, it appears that, like the weather, little is being done about it. If one is to do something about software performance, then one must begin by measuring performance; yet, despite the recent development of tools such as the Digital Continuous Profiling Infrastructure (DCPI)[1], software performance measurement seems little used in practice, outside of the parallel computing community. Most of the tools that exist to aid the programmer in attempting to tune an application, or in understanding the performance characteristics of that application, are not much evolved beyond the *prof* [15] call graph execution profiler that has been part of Unix operating systems for over 15 years.

Why is this so? Is software performance measurement so very difficult, or time-consuming? Is the pay-back so small that the effort is not worthwhile? Are the results so inaccurate that the time is wasted?

We don't claim to have answers to all of these questions. If one looks at commercial software products, there appears to be far greater interest in adding functionality than in improving performance. Perhaps Moore's Law ensures that faster and faster hardware will conceal the sins of the software producers, or at least create the situation where the perceived value is in features, not efficiency.

This does not say that performance measurement is not valuable; indeed, as evidenced by the work done in the parallel computing community, performance and software efficiency are an issue, and will continue to be an issue, in certain areas. In scientific computation, hardware performance does not improve quickly enough to meet the needs of the scientific computing community; here, efficient software is part of the approach to solving large problems. Similarly, embedded systems frequently require software that is space and time efficient, owing to hardware limitations caused by cost or power constraints.

What interest in software performance measurement as does exist in the general (sequential) computing realm exists primarily in the area of system software: operating systems, protocol stacks, and other components that support software applications. Interestingly, the performance of these system components is frequently much more difficult to analyse than is the case for application software. Here, at least, the cost (effort)/benefit ratio is viewed as positive.

It was in attempting measurements of system software that we came upon the approach described in this paper. We do not claim any sort of ownership of it; rather, it is the application of ideas that have been in use in other areas of computer science for decades. We see a range of related techniques of software performance measurement which we have found extremely useful, and which we believe can be profitably used by others in the computing community. Further, we do not view this approach as a replacement for the approaches that are in common use in the performance measurement community; instead, we view this approach as another "arrow in the quiver" of performance analysts, one which has several attractive characteristics.

The technique which we describe in this paper uses what we call a *measurement shim.* A shim is a thin piece of material inserted between two objects; in this case, it is a small ("thin") software module inserted at a defined interface between two other software modules. The measurement shim captures those performance metrics of interest to the performance analyst, without disrupting the flow of control or data between the modules on either side of the interface.

2. Background

Software performance measurement is not easy. Beyond measuring the time and space required to run a given application, significant complexity arises as the performance analyst strives to gain more detail regarding the performance of that application. What component of resource use is attributable to the application? How much is attributable to the operating system?

As we move beyond monolithic applications, the complexity increases. What are the costs of distribution? What are the effects of network traffic? How is performance related to protocol overhead? What is the impact of operating system and user load on network clients and servers?

Performance measurement tools and approaches can be considered to fall into two classes: *internal*, and *external*. Internal measurement approaches require that measurement take place within the application or module under test, typically through the addition of monitoring code to the application source code. External approaches attempt to capture measurement data without disturbing the application or module. External techniques view the software as a "black box", and observe the external behaviour only. Internal techniques operate within a "white box" environment, and therefore have the potential for much finer granularity of measurement, or, better, for precisely the level of specificity, and coverage, as is required by the measurement experiment.

2.1. Internal Techniques

A number of approaches are available to perform internal measurement. These are:

- Hand-modification of source code;
- Automated instrumentation of applications, and
- Instrumentation of runtime libraries.

2.1.1. Hand-modification of Source Code. A common way to take measurements is to modify the application source to insert performance monitoring code. This provides the ability to capture the metrics of interest, and the analyst has the luxury of taking the measurements in the most appropriate places. Unfortunately, this approach has a number of problems:

- For all but trivial systems, it is very labour-intensive. Furthermore, those people making the modifications must have a sound understanding of the design and implementation of the application to ensure that all appropriate measurement sites are instrumented, and that no bugs are introduced. It is not uncommon for timing problems to be introduced simply because the code path has been changed.
- The resultant system may no longer be representative of the unmodified application, particularly if subtle bugs have been introduced. Again, timing may be changed, or substantial overhead may be added in the process of instrumenting the code.
- If some measurement sites have been missed, the resulting data may be inaccurate.
- All of these issues are exacerbated in the case of an evolving software product or system, where modifications must be rolled forward into new releases, and measurement code changes may be necessary to cope with bug fixes and enhancements to the base code.
- All is for naught if the source code is unavailable.

2.1.2. Automated Instrumentation of Applications. Automated instrumentation is viable in a number of situations, particularly those in which interface definition languages (IDL) are used, or where the creation of an application from source code requires the machine generation of additional source code. In an application using Remote Procedure Call (RPC), for example, the RPC stub compiler could be modified to insert instrumentation code into the application. Blumson *et al.* [2] describe this approach applied to the Distributed Computing Environment (DCE) IDL compiler.

While automated insertion of monitoring code solves the problems of tedium and of tracking new releases, and reduces the probability of error, it does so at the cost of the fine control over measurement that hand instrumenting provides. Further, it adds the additional requirement that the source for the appropriate compiler or source code transformation tool be available for modification, or that such a compiler be created from scratch.

2.1.3. Runtime Library Instrumentation. A technique that is applicable in many cases is to instrument the runtime libraries used by an application, rather than the application itself. This has the advantage that the application source code is not necessary, and that new versions of the application can be measured "out of the box". With some manipulations of the dynamic linking/loading facilities of the host operating system, and a bit of care, instrumented libraries and normal ones can often coexist.

Aside from the above, the issues are still the same as for other internal measurement approaches: while the application source code is not being modified, the library source is being modified, and this must typically be done by hand. Depending upon the library being modified, this may be a significant challenge: the Digital Unix DCE 1.3 *libdce.so,* for example, includes some 560 functions.

2.2. External Techniques

External approaches cover a broad range of techniques, coverages, and granularities of data measurement. Typically, they may be viewed in several broad categories:

- Profiling systems, and
- Independent observers.

2.2.1. Profiling Systems. Profiling systems fall into two categories, those which require (automated) changes to the application, and those which do not. Those which require changes typically require only that the application be relinked with runtime infrastructure to perform the profiling task. Of the tools that require changes to the executable file, *gprof* [3] and *pixie* [8] are among the best known examples. They operate by annotating the binary code so that execution times and call statistics for functions (*gprof*) or basic blocks (*pixie*) can be recorded.[1]

The second category of profiling tool includes those based on statistical sampling of application execution. Notable here are *prof* [15], which has been in use for a very long time, and several new tools, *Morph* [16] and the Digital Continuous Profiling Infrastructure (DCPI) [1]. *Prof* uses program counter sampling to generate an estimate of the execution time and call statistics for the functions in the application. *Morph* and *DCPI* operate with sufficiently low overhead that they may be run continuously. *DCPI* goes further in its use of the CPU performance registers and randomised sampling to increase the accuracy and granularity of measurement, down to the ability to allocate the cost of cache misses to individual CPU instruction stalls.

[1] We recognise that this represents lowest common denominator capabilities of the many implementations of these utilities.

Profiling is a valuable tool, particularly for monolithic programs: for the purposes of finding hot-spots and determining where to expend effort in optimising a piece of software, it is a very valuable technique. Many profiling tools suffer somewhat in accuracy due to the statistical nature of data-gathering, though the performance of *DCPI* demonstrates what can be gained from a combined hardware/software monitoring facility. The advent of continuous profiling in *DCPI* and *Morph* greatly enhance the usefulness and applicability of profiling.

Tools which require relinking or binary annotation are limited in what they can measure: unless the program is linked against system libraries that have been compiled for profiling, information about the performance of library routines will be limited. These tools require that the application be available in source or object module format. Unless the kernel provides profiling capabilities, information about system calls on behalf of the process will also be limited.

These limitations become worse in distributed applications, as only the "local" portions of the executing application can be traced.

2.2.2. Independent Observers. Another approach is to position the measuring system at some point distinct from the executing application itself; for example, to observe the traffic on a shared-access network, and to determine the performance profile of the application or system software by inference from the measurements of the network traffic that it generates.

An example of a tool which uses this approach is the popular *tcpdump* [5], which passively monitors Ethernet traffic, and provides the ability to capture packet headers and data, and to time-stamp this data. It provides a good view of the network characteristics of a distributed application, but provides little information about the details of the performance of the components of an application. For example, if a server is slow in responding to incoming service requests, observing the transactions from the network perspective will not guide the performance analyst toward the cause of overhead, be it the protocol stack, the virtual memory system, or simply system load.

There is a strong analogy with hardware monitors here, and they too represent a good example of independent observation. It can be argued that *tcpdump* does a fair job of turning a Unix workstation into a network sniffer, which the experimenter "clamps" onto the network wires to observe the traffic. The analogy is complete in the sense that *tcpdump* is easy to employ, requires no changes to the system under test, and generates extremely useful performance data, despite its remoteness from the application under test. It also demonstrates that, as noted above, there is data which simply cannot be collected from a distant viewpoint.

This illustrates a common problem in performance evaluation: often, performance measurements are taken, not at the point in the system under test which would provide the most appropriate and accurate information, but rather where it is possible to make measurements, or, to quote one of the authors, "Where the light is good."

2.3. Measurement Shims

The measurement shim is a small module placed between two software modules at an interface point. The shim contains code which takes measurements of interest from the data stream that crosses the interface, or makes measurements of other phenomena, such as function calls, which cross the interface. It has no direct visibility into either of the modules that are parties to the interface.

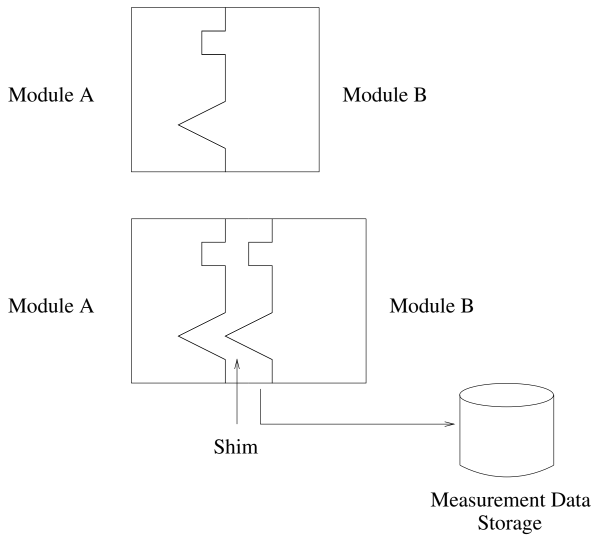


Figure 1: Inserting the shim.

Consider the diagram at Figure 1. In this case, the shim is visibly interposed between the two components that share the interface.

The Unix shell and utilities provide a simple example. Most Unix utilities are written to accept data on the standard input file descriptor and to output it on the standard output file descriptor.[2] This allows utilities to be composed into chains (Unix "pipelines"), each of which reads the output of its predecessor and sends its output to its successor, modifying the data stream on the way. To update an old example, the pipeline:

```
cat document | \
tr -cs "[:alpha:]" "\n" | \
sort | uniq > words
```

[2] A Unix file descriptor represents an input/output channel.

will reduce a text file first, into a list of words, one to a line (`tr`), sort them in ascending order (`sort`), then suppress all repeated lines (`uniq`) to create a dictionary list of the words in the original document. Unix *filters* can be composed in this fashion because of their adherence to the standard input and standard output interfaces.

If we wished to count the bytes passing through some portion of this pipeline, we could imagine writing a small program which would take a data stream as its standard input, count each byte as it passed, and send the data stream unmodified to its standard output. We would need some means to report the byte count, since it could not be written to standard output, but that could be arranged via a file or via the *standard error* facility.

This small program would be a measurement shim. It would invisibly count the bytes passing through the Unix pipeline: it would not modify the data stream, and its existence would not be apparent to either its predecessor or its successor in the pipeline. Because the interface is the same at all interface points, the shim could be inserted between any two filters in the pipeline. For that matter, a shim could be inserted at each interface point simultaneously, so that the differences in the size of the data stream could be measured at each point in the process of manipulating the dictionary list.

A second example of a measurement shim would be what is frequently called a *wrapper;* a procedure or object which encapsulates another. If Module A in Figure 1 calls Module B, then the creation of a new shim function, which accepts the procedure call or message from Module A, makes performance measurements, and passes the procedure call or message on to Module B, would achieve the same result for procedure call or object interfaces that the Unix pipeline shim achieves for data interfaces.

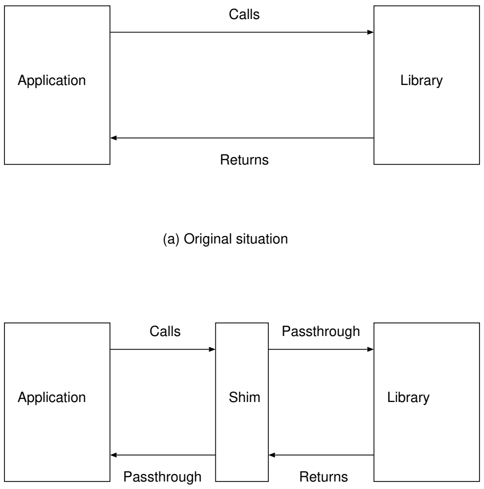
The remainder of this paper discusses some of the experience that we have had using this approach in measurement experiments, discusses the advantages and disadvantages of the shim technique, and describes related and future work.

3. Experience

3.1. DCE RPC

A successful application of the measurement shim has been the instrumentation of the DCE RPC libraries [10]. This has been done in a manner applicable to many systems which use dynamically loadable libraries. It is described more fully in [11]. This experiment provides a good example of using the measurement shim approach, as our DCE distribution does not include source code.

The technique used in this case is what might be described as a *dynamic library shim;* the shim exists as a dynamically loaded library which exports an interface identical to the dynamically loaded library which provides the DCE functionality. The subroutines in the shim library are coded to collect information, such as byte counts, on entry and exit to the library call. Aside from this performance data collection, the shim library is a pure pass-through to the actual library, which performs its normal function. This is illustrated diagrammatically in Figure 2.



(a) Original situation

(b) Shim inserted

Figure 2: Inserting a library shim

The implementation is remarkably simple: it requires extracting the interface definitions from the header files corresponding to the library functions, and creating appropriate function calls in the shim library. Much of this can be accomplished mechanically, though it is still necessary to understand the intent of the library call so that the appropriate performance data can be collected, and the specifics of data collection must be coded by hand.[3] An example of a shim library function is shown in Figure 3. Note the metric collection before and after the pass-through call to the actual DCE library function. (The measurement code is represented by the `getttimeofday()` and `printf()` calls in the source code.)

The first experiment we performed with DCE was to instrument some of the low-level RPC functions as a test of the feasibility of the approach and the accuracy of the data collected. We selected the RPC data marshalling and unmarshalling functions `rcp_ss_ndr_marsh_interp()` and `rpc_ss_ndr_unmar_interp()`, the connection setup function `rpc_call_start()`, and the client data transfer function `rpc_call_transceive()`. Instrumenting these functions allowed us to compare their relative processing costs. The experiment was performed using a simple "data exchange" application which moved blocks of data between a client and server. The data transferred was NULL-terminated character strings of various lengths, rangin from 0 to 64 kilobytes in multiples of 2 kilobytes. The machines used in the experiments were two DEC Alpha 2100A-500MPs, each having four 190MHz Alpha AXP21064 CPU's, 256MB of main memory, a 10Mbps Ethernet connection, and running Digital UNIX with OSF DCE 1.3. *Tcpdump* monitoring was performed using a DEC Alpha 3000/300 running Digital UNIX.

The results are shown at Table 3.1.. Note the relative overhead contributions of the different routines, and the large variance in the overhead of `rpc_call_transceive()`, which can be attributed to the search for an application server. It is worth noting again that these results, for functions at a very low level in the RPC call chain, were achieved without modifying any application or library source code.

A second experiment measured the round trip latency of data transfers as measured by the measurement shim and *tcpdump* simultaneously. In this case, only `rpc_call_transceive()` was instrumented via a library shim. A simple DCE client application sent 10,000 RPC data transfers varying from 0 to 64 kilobytes to a server. The averaged results for both measurement methods are shown in Figure 4. Note that the variation in the measurements made by *tcpdump* is much higher than in the measurements made by the shim. This is in large part due to lost packets on the interface of the *tcpdump* workstation. Note also the consistently higher processing load measured by the shim, attributable to the protocol processing in the network protocol stack below the RPC software, which *tcpdump* cannot measure from the network side of the interface.

Tcpdump is an excellent tool; it has been used in many performance analyses that have produced important results in the performance evaluation community. These results demonstrate that for the type of measurements that we made in this experiment, the measurement shim provides better data. The ability of the measurement shim to outperform such a good tool is one reason that we believe there is merit in the measurement shim approach.

Our experience with the DCE shim has been very positive. With relatively little effort, compared with the alternatives, we were able to capture performance data pertaining to RPC round-trip times (RTT) for a representative set of DCE RPC functions, and to apportion the processing cost of RPC calls across the functions that participate in the RPC call. [11] The data collected was more accurate than that which was collected concurrently via independent observation, and required no changes to the application or library

[3] A great deal of this work can be automated, particularly in the case where a small number of simple metrics (number of calls, bytes read/written) are being collected for all of the functions in a library.

```
void rpc_call_transceive(
      rpc_call_handle_t call_handle,
      rpc_iovector_p_t in_call_args,
      rpc_iovector_elt_t *out_call_args,
      ndr_format_t *remote_ndr_fmt,
      unsigned32 *status )
{
      struct timeval t1,t2;
      gettimeofday(&t1,NULL);
      printf("Start %s :  %.9f\n", "rpc_call_transceive",
           (double) t1.tv_sec + (double) t1.tv_usec / 1000000 );
      shim_rpc_call_transceive( call_handle, in_call_args, out_call_args,
           remote_ndr_fmt, status);
      gettimeofday(&t2,NULL);
      printf("End %s :  %.9f\n", "rpc_call_transceive",
           (double) t2.tv_sec + (double) t2.tv_usec / 1000000 );
}
```

Figure 3: Source code for the shim version of `rpc_call_transceive()`.

Table 1: Initial results from the DCE measurement shim.

Function Monitored	Mean Function Execution Time (msec)	Variance
`rpc_ss_ndr_marsh_interp()`	0.2229	0.0062
`rpc_ss_ndr_unmar_interp()`	0.1035	0.0017
`rpc_call_start()`	0.2477	0.0036
`rpc_call_transceive()`	64.7599	23.3057

source code, of which we had neither.

The overhead cost of this method is low. It consists of an additional function call and the cost of data collection. For simple data, such as instruction counts, this is very low. As the number of independent data items and the data volume increase, the cost of data collection increases.

What has been accomplished here is notable because the mechanism is so simple, and because the method is applicable to *any* dynamically loaded library; someone interested in the behaviour of X Window System applications could, for example, provide shims for any or all of the X11 library functions. Similarly, one could instrument the Unix libraries or those of other operating systems which use dynamic loading.[4]

[4]...once one had plumbed the vagaries of the particular operating system's dynamic linking and loading mechanisms, it must be said. We have found that what is trivial on one operating system may be difficult or impossible on others due to limitations in the tools provided by the operating system.

3.2. The Stackable File System

We have also employed a measurement shim in experiments with the Stackable File System (SFS) [4]. The file system is *stackable* in the sense that file system components can be stacked, one upon the other, to create a file system, rather than implementing the file system as a monolithic whole. This approach allows the user of the operating system finer-grained control over file system semantics, and allows, for example, the implementation of a single caching module which can be used by various file system types. Stackable file systems also allow novel file system configurations: for example, file systems which implement compression at the user level, but which allow maintenance operations (such as file backup and restoration) to be performed below the level of compression, avoiding the overhead of the compression/decompression layer during maintenance operations, and enhancing the security of user data.

The stackable file system we used is based directly upon the VFS/Vnode file system interface provided in modern UNIX implementations [14]. It differs in the details of certain kernel data structures and in the details of the *mount* kernel operation.

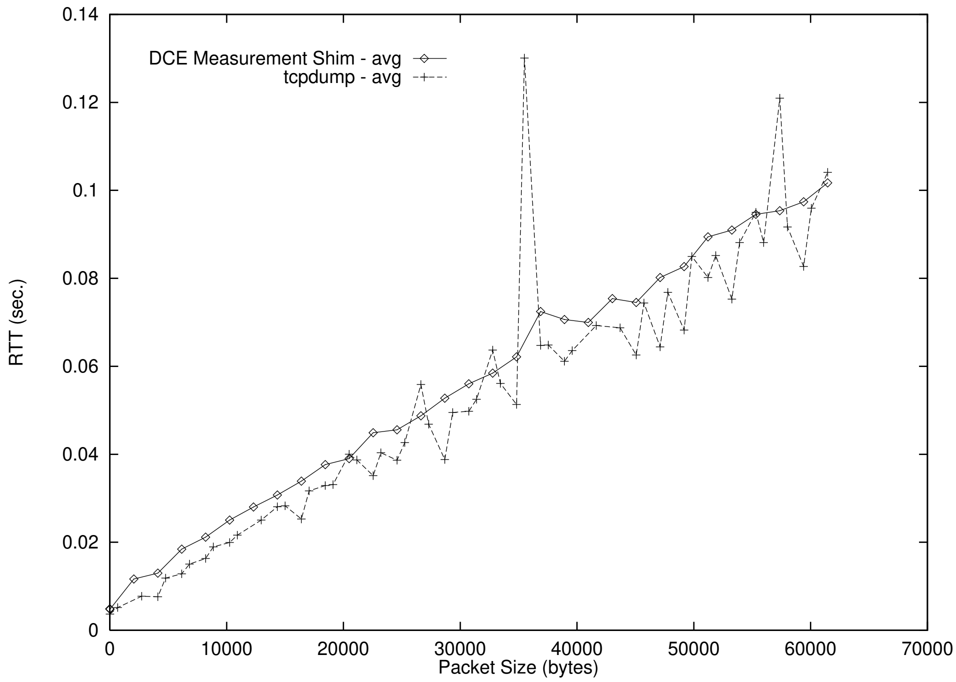


Figure 4: Average RPC round-trip times as a function of packet size.

For these experiments, we created a new file system type, *shimfs,* which was constructed so that it could be inserted at any point within the file system stack. The *shimfs* file system was easy to build, as it was based on several of the other file system types that are provided as part of the stackable file system implementation. Minor changes were also made to the *mount* command to recognise the new file system.

Of interest in this application was to determine the sort of information that might be accessible via the shim. The sole performance measurement made by the initial implementation was to count the operations (`read, write, seek, opendir, etc.`) performed on the underlying file system, and to report aggregate operation counts.

We inserted a measurement shim into a stackable file system on an HP 425t (25MHz 68040, 16 MB of main memory, 1 - CDC 94171 SCSI disk for test file systems, 1 - Fujitsu M2694ES disk for system files) running NetBSD 1.2, and performed multiple tests, including data transfer tests using the IOZone and Bonnie disk benchmarks to determine the overhead of adding the measurement shim, as as a means of ensuring that the shim was not having any negative impact on the underlying file system.

The results of running the Unix command

```
find /mnt -type s -print
```

against the "shimmed" file system are shown at Figure 3.2.; this is a representation of the data as output by the measurement shim. Each entry gives the name of the file system operation, its number in the table of file system operations, and the number of times that the operation was performed by the file system. This figure demonstrates that function call counts can be readily obtained by using the measurement shim. Without instrumenting either the kernel itself, or the individual file system components, these data would not be available. While the effort of creating the measurement shim was comparable to instrumenting a single file system type[5], the shim can be used to instrument all of the file system modules in a stackable file system.

4. Discussion

As can be seen from the experiment descriptions, the measurement shim has prove to be a useful technique for instrumenting software applications. In both of the cases described, we were able to collect high quality data with a small investment in effort.

We view the chief benefits of the measurement shim approach to be the following:

- The approach requires no source code changes to the system under test. This has a number of key subsidiary benefits:

[5]The effort was comparable, but lower, since the *shimfs* doesn't perform any file system operations, and is therefore much simpler than a "real" file system.

```
shimfs: shimfs counters:
default (0)=0
vop_strategy (1)=0      vop_bwrite (2)=0        vop_lookup (3)=24929
vop_create (4)=0        vop_mknod (5)=0         vop_open (6)=3729
vop_close (7)=3729      vop_access (8)=11187    vop_getattr (9)=21206
vop_setattr (10)=0      vop_read (11)=0         vop_write (12)=0
vop_ioctl (13)=0        vop_poll (14)=0         vop_mmap (15)=0
vop_fsync (16)=15815    vop_seek (17)=11219     vop_remove (18)=0
vop_link (19)=0         vop_rename (20)=0       vop_mkdir (21)=0
vop_rmdir (22)=0        vop_symlink (23)=0      vop_readdir (24)=7490
vop_readlink (25)=0     vop_abortop (26)=0      vop_inactive (27)=0
vop_reclaim (28)=0      vop_lock (29)=0         vop_unlock (30)=0
vop_bmap (31)=0         vop_print(32)=0
```

Figure 5: Representative SFS shim output

  - There is no risk of introducing bugs into the application source code;
  - There is little risk of impacting the application by changing timing relationships between components of the application;
  - There is typically much less effort required than hand modifying the code; and
  - The method can be used when source code is not available.

- The approach often provides the ability to take measurements closer to the point of interest than is possible except by modifying the source code. This reduces the amount of spurious information collected, and reduces the difficulty of allocating the metrics across the system under test and its environment.

- The approach is very flexible, and applicable to a large number of different application types, environments, and measurement data requirements. To return to the example using Unix filters and pipelines, anyone familiar with Unix knows the power of Unix pipes, which derives from the definition of, and adherence to, a very simple data interface. Unix pipes have very widespread application; we think that the shim approach is similarly flexible and useful.

- The approach is lightweight. While the amount and type of data required will have a significant impact on the overhead of the shim, we found the measured overhead of employing the shim to be very reasonable. The cost of measurement is comparable to that which would be incurred by hand-modifying the source code.

The approach is simple; it relies on exploiting well-defined interfaces between software components to insert measurement apparatus at that point. The shim itself is *semantically neutral,* so that neither component is aware of its existence. It behaves in a certain sense like a third party observer, watching the transfer across the interface and recording interesting information about the transfer. It differs from an independent observer in that it is directly interposed in the data stream or function chain, and its semantic neutrality is a function of careful design and implementation, not of distance.

The transfer across the interface is typically of data or of control; in measurement, it is usually data transfers that are of greater interest. So, for example, a shim between components in a protocol stack might measure bytes read and written, and with what latency of transfer. A shim between components in a file system might, in addition, count the frequency of calls to various directory and file operations. In either case, the shim can perform the measurement function without affecting the semantics of the interface specification.

Note also that the shim is dependent solely on the existence of a defined interface between the components; it need know nothing about the components themselves. In particular, there is no requirement to modify the source code of the components that communicate across the interface.

We note several limitations of the measurement shim:

- It relies on the existence, and maintenance, of well-defined interfaces. Without being trite, we do not view this as a major problem, since good modularity and information hiding are their own reward.

- While it can frequently allow measurement at or very close to the ideal point of measurement, the technique can never get closer than the nearest interface. Again, we do not view this as a major problem; our experiences suggest that the quantity and quality of data available at the interfaces is more than sufficient to provide good insights into the performance of the system under test.

- The measurement shim exists in an environment bounded by the interface it exploits. It is therefore blind to higher-level application context. This requires the measurement shim user to understand the system under test.

While powerful and effective, the measurement shim approach is not a silver bullet. It is a useful technique that, like most in performance measurement, must be applied with care to ensure the validity of the results produced.

5. General Principles

The soul of this technique is the exploitation of interfaces between computer components, be they hardware or software. Frequently, these interfaces are defined by *de jure* standards, such as DCE[10] standard for distributed computing. All too frequently, these interfaces are defined by *de facto* standards, such as that for the VFS/Vnode interface in 4.4BSD[7]. In the former case, the technique is generally more easily implemented, since these interfaces are more stable, and are often more mature. What is most important, however, is the existence of a standard interface to which the application or application modules adhere. As demonstrated by the example of a Unix pipeline, the interface need not be complicated to be useful.

The shim must have several notable characteristics to be useful:

- it must be semantics-preserving, in both directions;
- it must have low overhead.

5.1. Semantics Preservation

To correctly insert our shim between two sides of some arbitrary interface, we must not alter the semantics of the communication between them. For example, if we are inserting a kernel module shim between the kernel proper and a module which implements the network interface driver in order to measure the latency of network communication, the "kernel side" of the shim must be indistinguishable (to the kernel) from the network interface module itself; similarly, the "module side" of the shim must be indistinguishable (to the network interface module) from the interface that the kernel would present to the network interface module in the absence of the shim.

Our experience suggests that this is not difficult to achieve in practice; in reality, most of the semantics of most interfaces are are captured by mechanisms that are invisible to the shim. In the DCE library shim, for example, it is important that the library function be called with the function arguments unmodified, and that any return value from the library function be propagated through the measurement shim back to the caller. In both cases the hard work of doing so is handled by the procedure call mechanism, and the programmer of the shim need only ensure that the measurement code does not interfere with the transmission of parameters and return values. In the case of the SFS shim, the shim must ensure that the shim file system data structures are populated with the correct values from the underlying file system, and that operations and return values are propagated unchanged. This suggest that the watchword for measurement shim writers might be "Look, don't touch."

5.2. Overhead

While not as important as ensuring invisibility of the shim to the system software, creating a low overhead ("lightweight") implementation is essential to the technique.

Given the goal of effective performance evaluation, it is a truism that a measurement tool that affects the performance of the system reduces the validity of the measurements in question. Indeed, if the performance impact of the shim is great, it is possible to violate the semantics-preserving characteristic of an otherwise correct shim.

In almost all cases, it is not possible to measure a system without performance impact, particularly if the performance measurement subsystem relies on software components. It is important, therefore, to be able to minimise that impact, and to characterise it. The greater the detail and volume of data we wish the shim to gather, the greater will be the performance impact of the tool. What is important is to ensure that the impact is as low as can be reasonably implemented.

6. Related Work

We make no pretence to being the originators of this approach; what we intend is to systematise its use across a broad range of domains, and to provide both general and specific techniques and tools. As well, we are researching the foundations of the approach, to determine if there are theoretical or conceptual underpinnings which can aid our understanding.

There seems to be very little written in the specific area that we describe as *measurement shims.* Aside from anecdotes from other performance researchers, we have found nothing in the measurement literature which addresses this area directly.

There is, however, a solid body of material about shims in a more general context. One area is that which has been described as *Wrappers*. The term has been used in several contexts, notably in those where a procedure call or message passing interface is involved. The idea is generally to encapsulate something, such as legacy code being integrated into a new environment, or low-level interrupt code being "wrapped" in what amounts to syntactic sugar to make these low-level facilities easier to program. Specific examples include the C++ Wrappers, built to encapsulate Unix/C networking functionality[12], the ParcPlace PARTS toolkit for encapsulating relational databases to make them accessible by Smalltalk and Java programs, and various schemes to provide secure access to the Common Gateway Interface (CGI) built into most web servers from user-written web pages, *e.g. CGIWrap*[9].

A very general facility that was a part of the Interlisp-D[13] system is the concept of *Advice.* The Advice mechanism provided the ability to attach arbitrary functions to be evaluated *before, after,* or *around* any other function call. Obviously, this facility would provide the means to collect measurement data, although the actual use was frequently to modify parameters and return values of the advised function. Smalltalk allows a similar facility, called *Indirection Objects*[6] based on the `become:` method, which interchanges the identity of two objects. Any object which contained a reference to the first object now holds a reference to the second. This could be used like a measurement shim by first, creating a measurement object which gathers metric data and passes the message on; and second, interchanging the measurement object and the object being measured. This makes the measurement object the target of the incoming message instead of the intended object.

As previously noted, Interface Definition Languages provide a rich opportunity for shim style performance measurement. In particular, they provide an excellent means of automating the process. Blumson *et al.* [2] show such an approach for the DCE IDL compiler.

7. Future Work

There are three major directions in which we wish to take this work. The first is to apply the techniques to a broad range of measurement tasks, both to take advantage of the capabilities which we have uncovered, and to understand the breadth of application to which these techniques are applicable. The second is to develop a deeper understanding of shims, in both performance measurement and in general. Finally, we wish to determine the foundation aspects of the facility, and to determine if there are theoretical underpinnings which would be valuable to explore.

There seem to be many possibilities for measurement-related exploitation of interfaces, and we are interested in expanding our work to the areas of communication protocols, virtual machines, security monitoring, and program transformation applications.

8. Acknowledgements

This research was funded, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC) through Research Grant OGP0003707 and a CAP Grant. We are also grateful for support from Telecommunications Research Laboratories (TR*Labs*) and from IBM Canada's Centre for Advanced Studies (CAS).

References

[1] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monkica R. Jenzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandervoorde, Carl A. Waldspurger, and William E. Wiehl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 1–14. ACM SIGOPS, October 1997.

[2] Sarr Blumson, Mark Carter, and Daniel Hyde. Automatic insertion of performance instrumentation for distributed applications. Technical report, The Center for Information Technology Integration, University of Michigan, February 1995.

[3] Susan Graham, P. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software - Practice and Experience*, 13:671–685, 1983.

[4] John S. Heidemann and Gerald J. Popek. A layered approach to file system development. Technical Report 91-00007, University of California at Los Angeles, March 1991.

[5] Van Jacobsen, Craig Leres, and Steven McCanne. *TCP-DUMP*. Lawrence Berkeley Laboratory, University of California, Berkeley, CA., June 1989. UNIX Manual Page Entry.

[6] Wilf R. Lalonde and John R. Pugh. *Inside Smalltalk*. Prentice Hall, January 1991.

[7] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The design and implementation of the 4.4 BSD operating system*. Computer Science and Engineering. Addison-Wesley, Reading, Mass, 1996.

[8] MIPS Computer Systems, Sunnyvale, CA. *UMIPS-V Reference Manual*, 1990.

[9] Nathan Neulinger. *CGI-Wrap - Secure User Access to CGIs with httpd*.

[10] Open Software Foundation. *Introduction to OSF DCE*. Prentice Hall, 1992.

[11] Gregory M. Oster and Richard B. Bunt. Using shim technology to monitor DCE runtime performance. In J. Howard Johnson, editor, *Proceedings of CASCON '97*, pages 96–107. IBM Canada/National Research Council of Canada, November 1997.

[12] Douglas C. Schmidt. Systems programming with C++ wrappers: Encapsulating interprocess communication services with object-oriented interfaces. *C++ Report*, September/October 1992.

[13] W. Teitelman. *The InterLISP Programming Manual*. Xerox PARC, Palo Alto, CA, 1975.

[14] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, Upper Saddle River, NJ, 1996.

[15] Various. *NetBSD Reference Man*.

[16] Xiolang Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automated profiling and optimization. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 15–26. ACM SIGOPS, October 1997.