Profiling and Tracing Dynamic Library Usage Via Interposition

Timothy W. Curry
Internet: tim.curry@sun.com
Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043

Abstract

Run-time resolution of library functions provides a rich and powerful opportunity to collect workload profiles and function/parameter trace information without source, special compilation, or special linking. This can be accomplished by having the linker resolve library functions to special wrapper functions that collect statistics before and after calling the real library function, leaving both the application and real library unaltered. The set of dynamic libraries is quite large including interesting libraries like libc (the C library and Operating System interface), graphics, database, network interface, and many more. Coupling this with the ability to simultaneously trace multiple processes on multiple processors covering both client and server processes yields tremendous feedback. We have found the amount of detailed information that can be gathered has been useful in many stages of the project lifecycle including the design, development, tuning, and sustaining of hardware, libraries, and applications.

This paper first contrasts our extended view of interposition to other profiling, tracing, and interposing techniques. This is followed by a description and sample output of tools developed around this view; a discussion of obstacles encountered developing the tools; and finally, a discussion of anticipated and unanticipated ways those tools have been applied.

1. Motivation

The tools described in this paper were created to analyze performance of graphics applications. The application writers seldom has access to the graphics library source or profiled versions of the graphics libraries. The library and hardware provider seldom has access to application source and data files. Our goal was to get useful performance data without special requests placed on either the application or libraries.

We were also after more information than is typically available. While traditional profile tools might tell you how many times a line drawing function is called, they don't tell you what percentage of the lines are write-only versus read-modify-write operations; what the average length, width, and angle of the lines are; and what line styles are used. One can envision similar questions for a database, such as percentages of read versus write transactions, if access patterns were sequential or random, etc. This additional information significantly improves the ability to perform more detailed analysis and make more informed decisions.

Graphics libraries remain our group's primary interest but the tools are generic to any dynamic library and have been applied both internally and externally to profile, trace, and generally interpose on non-graphics libraries. Additionally, the data that can be collected has proven useful well beyond performance analysis.

2. Terminology

For detailed discussions on dynamic linking, the reader should refer to [1,2,3]. The techniques described in this paper presume the applications which are to be profiled and/or traced have already made the decision to use dynamically linked libraries and that the run-time linker/loader provides a means to perform interposition. System V release 4 (SVr4) UNIX and all versions of Solaris provide that means through an identical interface. This technology has been around for several years, is easily used, and commonly found in operating systems. A brief description of the terms used by this paper should quickly resolve various operating system terminology issues.

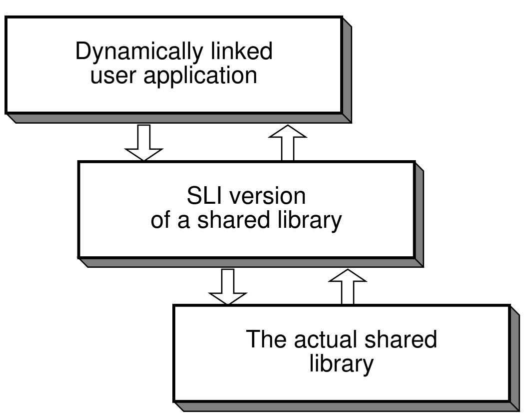
A *dynamic library* consists of a set of variables and functions which are compiled and linked together with the assumption that they will be shared by multiple processes simultaneously and not redundantly copied into each application. Because of this sharing, dynamic libraries are sometimes called *shared libraries* or *shared objects*. Compiler and linker flags assure the program sections and data sections are cleanly separated and the program sections are reentrant and reusable. The compile and link of the application can leave the dynamic library *symbols* (variable and function addresses) unresolved until *run-time* (or time of execution). This complicates the *loader* (the program which initiates an application) since the *linking* (resolution of all symbols) must be completed on execution rather than at compile/link time.

The process of placing a new or different library function between the application and its reference to a library function is called *interposing*. We specifically avoid placing any constraint on what the interposing function must do other than accept the parameters of the real function and return an appropriate value back to the application. The real function may or may not be called and new side effects may or may not occur due to the interposing function.

3. Profiling and Tracing Techniques

We have developed a set of tools under the umbrella name SLI (pronounced sly) which is an acronym for **S**hared **L**ibrary **I**nterposer. SLI contains programs and utilities that enable application and library developers to monitor and analyze calls to shared library functions. SLI is intended to augment, not replace, analysis tools such as *tcov* [4,5,6,7], *gprof* [4,5,7], and *analyzer* [10]. This section provides an overview of our technique and contrasts it to other techniques.

3.1. Overview of SLI's Technique



What distinguishes SLI from traditional analysis tools is *how* it collects information and *what* information it collects. The loader waits until execution of an application to resolve the shared library functions called by the application. SLI has the loader resolve the addresses to a wrapper of the library function, collects information in the wrapper, and calls the real library function from the wrapper. Several advantages arise from this technique:

- An accurate trace of call sequences can be logged.
- Parameter values are available to be logged and/or altered both before and after the real function call.
- The real function can be replaced if desired.
- Any subset of the library functions can be profiled instead of all functions in a library.
- Nesting levels into the libraries can be controlled.
- Different levels of profiling can be enabled or disabled while the application is running.
- Multiple processes and multiple library statistics can be logged to single or multiple locations.
- Profiling is available without application or library source and without requiring any specially compiled or linked objects. The only requirement is that the application must be dynamically linked to the shared libraries of interest.

Some of these advantages can be found in other tools or other tools can be altered to produce similar results, but SLI pulls it all together in an easily maintainable, dynamically controllable, and customizable package which remains independent of the application and library sources.

3.2. Comparison To Other Techniques

The *trace* command [4,5] of BSD UNIX and *truss* command [6,7] of SVr4 UNIX demonstrate some desirable features. These commands run an application or attach to an active process providing a trace of the system calls made by the application, showing the parameter values and return values or a summary count of all system calls and total time spent in each. *Truss* also allows restrictions on which calls are reported. No special flags are required when the application is compiled. With the exception of attaching to an active process, interposition of dynamic libraries allows all of these features to be applied to user level libraries. Additionally, our tools allow programmatic and interactive control of the data collection and multiple process data collection to a single file or per-process separate files for custom postprocessing reports. For some libraries, such as graphics or database libraries, all updates must pass through the library so parameter and return value capturing is sufficient to record and replay an application run. This has many positive ramifications on the project lifecycle as discussed in section 6.

System call interposition can significantly expand operating system functionality and transparently provide a number of new services to applications. The COLA [17] and nDFS [18] projects are examples of expanding file system functionality through system call interposition. The Interposition Agents Toolkit [16] presents a number of clever examples of system call interposition utilities and provides an environment to easily create new utilities.

Trace, *Truss*, and *Interposition Agents* are implemented through an operating system trap mechanism for system calls. The trap mechanism allows both dynamically and statically linked applications to benefit from the utilities and allows attaching to an active process. However, the trap mechanism incurs a heavier overhead and is not available for user level library functions. Inversely, interposing on dynamic libraries does not work for statically linked applications and must be selected prior to the application execution but introduces less overhead and allows more libraries to be interposed. Interestingly, the end results of any of these tools and utilities is independent of the interposing technique so informed decisions can be made in the selection of a technique.

Unfortunately, most tools for profiling and tracing require that the source code of the application and libraries be compiled and linked with options different from those of the release executable. That alone can change the executable enough to skew results significantly. The *prof* [4,5,6,7] command of UNIX and its variants *gprof* and *lprof* are the most common profile report generators in the UNIX environment. Special compilation flags cause code to be added to each function to maintain counts. When the application is run, it is interrupted on regular intervals and information about the currently active function is collected. This information, and counts of all function entries, are written to a file upon application completion. This technique has extremely low overhead but lacks detailed accuracy, is limited to the functions that were specially compiled, and only allows a single application per data file. Interposing on dynamic libraries overcomes these restrictions but only for library functions, not the functions of the application. The time interval between library calls can be monitored, giving some measure of application time versus library time, but no detailed profile of the application functions is collected. It is possible to add calls to our toolkit library directly in the application or library source or object but that defeats the concept of interposition to avoid altering application source code and linking.

More onerous than the potentially skewed results of special compilation is the requirement for source code, or at least specially compiled but unlinked object files. Operating system vendors often provide multiple versions of a library: an optimized dynamic version; an optimized static version; a profile static version; and internally there may be debug dynamic and static versions. The application writer then compiles and links appropriately. Debuggers such as *adb*[4,5,6,7], *dbx* [4,5,7], and *sdb*[6] require the source files be present to exploit their full power. Special interposing functions generally require relinking the application with new versions of the functions. It is extremely common to see alternate versions of the libc memory allocation routines *malloc*/*free* [4,5,6,7,8,9]. The precedence of linking allows interposition to occur either at compile/link time or run-time. Linkers resolve references on a first-come first-serve basis. If an application is linked with two libraries containing functions with the same name, the functions of the first library scanned are used. At compilation/link time, the order of the libraries listed on the link command specify the precedence. At run time, there are a number of environment variables that can alter the order and list of libraries scanned. See section 4.1 for our choice of technique. Some tools such as *Purify*, *Quantify* [9], and *Sentinel* [8] may alter the code contained in the application and libraries or link in new functions not previously included in the application. Relinking requires a new version to replace the function of the original library. The default behavior of our tool is to have a wrapper call the actual function which the application would have used, maintaining precise timing measurement and accuracy with the results of the function call. However, there is no requirement that our wrapper has to call the real routine. The same linker tricks we employ can be used to totally replace the real function or augment with new functionality as in [16,17,18]. More often, we still call the real function but output additional data such as hardware simulation streams.

The granularity of our library tracing technique is limited to the function level. Some profilers such as *tcov* and *Quantify* track hot spots of source code within functions. For a theoretical treatment of hot spot profiling and tracing with source code, see [14]. We consider the function level granularity quite sufficient for our needs, especially when combined with parameter and return value tracing. Lack of access to source code was considered part of our constraints.

4. The SLI Toolset

Tools are provided for varying levels of expertise. Our primary customer is interested in the same libraries that our group is and utilizes the interposing libraries we've already built and the report generators we've already written. The second tier user needs more or different information than we are providing by default and modifies the interposing library source or the postprocessing report generators to their own needs. Third tier users want to interpose on an entirely new library and so must create their own interposing library from scratch. SLI includes a collection of source, binaries, *awk*, and *perl* scripts to assist and serve as examples to help each level of user.

4.1. SLI Data Collection

Data is optionally collected to three locations. First, cumulative information is kept in shared memory. This cumulative information includes a count of function invocations, how much time was spent in the function, and how much time was spent in the function plus any descendants it may have invoked. Second, data can be written to standard-out or standard-error (stdout, stderr) providing trace and parameter information similar to the output of *truss* or customized output. Third, trace data can be collected to disk including the process-id (pid), the library, the function, the nesting level, how much time has elapsed since the last call into the library, how much time the call took, how much time SLI added in overhead, and parameter values and/or other interesting data. Multiple libraries from multiple applications can write to the same file or each process can create its own file.

The data collection can be controlled programmatically or through a terminal command line interface or through a graphical user interface (GUI). While we consider it desirable to be able to control data collection through scripts and without requiring the window system to be running, experience has shown 100% of the user base uses the GUI ignoring the other two methods. Perhaps this is a skewed sampling since our primary customers are graphics library users.

Data control includes clearing the cumulative information; starting and stopping the stderr output; and starting collection to disk through appending to existing data or rewinding/truncating the file and starting new. Additionally, data reduction can be controlled to reduce disk overhead; per-library flags can be controlled to alter wrapper functionality on the fly; and inner library nesting call tracing can be controlled (e.g. if the application calls a function in the library and that library function in turn calls another function in the same library, does the user want to capture that "inner library" call or only see what was directly invoked via the application). The figure below is a snapshot of the GUI which controls data collection.

The *ldd* [5,6,7] command lists the dynamic libraries used by a program. The list is generated at compile/link time but the path to locate the libraries can be altered at run-time. Following is an example of the *ldd* output of the *xterm* program (a terminal emulator program in the X-windows environment):

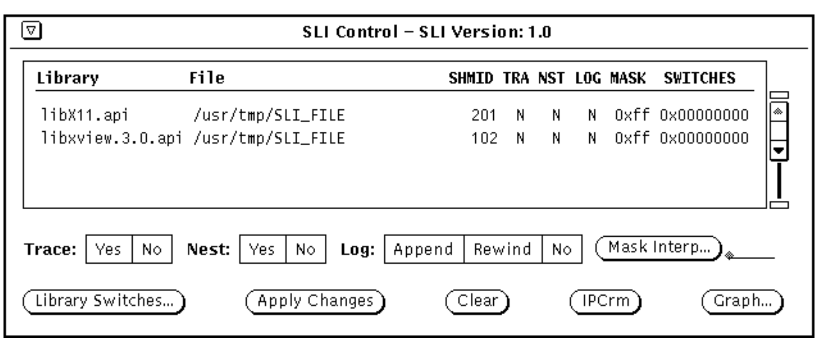
```
% ldd xterm
libXaw.so.5   =>/usr/openwin/lib/libXaw.so.5
libXmu.so.4   =>/usr/openwin/lib/libXmu.so.4
libXt.so.4    =>/usr/openwin/lib/libXt.so.4
libX11.so.4   =>/usr/openwin/lib/libX11.so.4
libdl.so.1    =>/usr/lib/libdl.so.1
libc.so.1     =>/usr/lib/libc.so.1
```

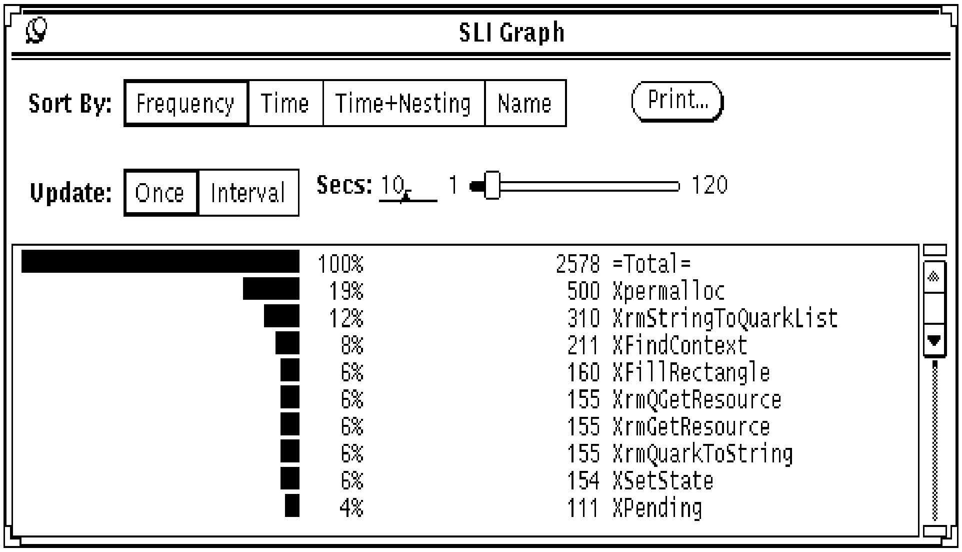
Each of these libraries can have all or any subset of their functions wrapped with data-collecting interposing functions. Furthermore, since *xterm* is an X-windows client application, it is possible to simultaneously profile the X11 server process and correlate interactions between the client and server.

The wrapper functions are invoked through the use of the LD_PRELOAD environment variable of the loader. The following example shows how the *ldd* output for *xterm* is changed by this environment variable.

```
% LD_PRELOAD="./libX11.api.so ./libsli.so"
% export LD_PRELOAD
% ldd xterm
./libX11.api.so=> ./libX11.api.so
./libsli.so   => ./libsli.so
libXaw.so.5   => /usr/openwin/lib/libXaw.so.5
libXmu.so.4   => /usr/openwin/lib/libXmu.so.4
libXt.so.4    => /usr/openwin/lib/libXt.so.4
libX11.so.4   => /usr/openwin/lib/libX11.so.4
libdl.so.1    => /usr/lib/libdl.so.1
libc.so.1     => /usr/lib/libc.so.1
```

You will note that there are now two versions of libX11 listed, our wrapper version and the system version. We've added *api* to the name because we have





three interposing versions of libX11. One version contains the Application Programmer Interface (API) functions where we monitor only those functions the application programmer has access to. A second version includes all functions contained in the libX11 source. A third version interesting to our group contains all the X11 functions with a graphics context parameter. Every symbol we've included in our wrapper library is resolved to us and every symbol we haven't included is resolved through the normal path. The additional library *libsli.so* contains SLI wrapper support functions.

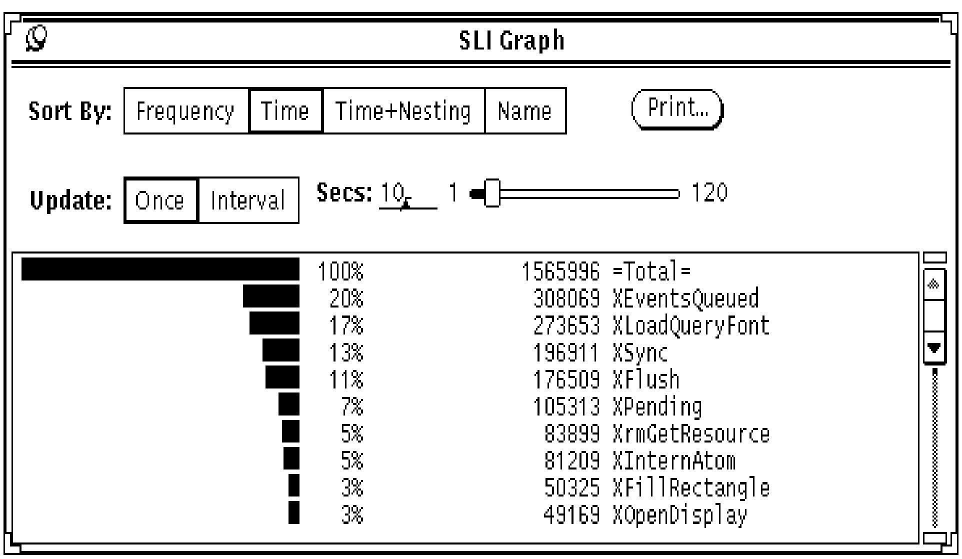
4.2. SLI Reports

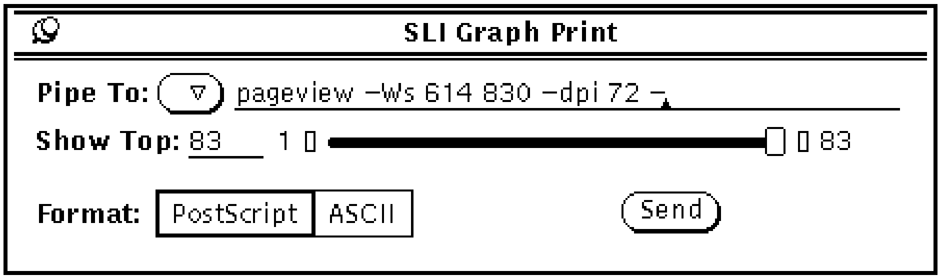
Default reports are provided on the cumulative data and the data written to disk. While data collection is generic, interesting reports on the collected data can be very library specific. Once the data has been collected, it is often necessary to write a custom postprocessing script in *sed*, *awk* [4,5,6,7], or *perl* [15] to glean the interesting information. We provide some postprocessor filters for our graphics libraries which also serve as examples.

The figure above shows a graph of the cumulative information collected on the start-up of an *xterm* before any character has been typed.

From this we note that 2578 calls were made into libX11 of which 500 calls (19%) were Xpermalloc. However, the function called the most doesn't necessarily consume the most time, as demonstrated in the figure below.

The times are displayed in microseconds. This tells us it took slightly over 1.5 seconds to get the xterm started. Even though Xpermalloc was the most frequently called function, it is not one of the top 9 consumers of total time.





The graph can be updated on regular intervals while the program is running, monitoring changes during execution. The graph could also be collecting the libX11 calls for all active applications in addition to the single *xterm*.

Two other sorts are provided by default. The time a function took plus all of the interposed descendants it invoked is sometimes more illuminating than the overhead of just the function itself. Second, you may want to find the call frequency or time for a specific function, and the sort-by-name simplifies finding the function.

The cumulative information can be processed through the print option. The default destination is a postscript preview program, but any command can be given to direct the output to a file, printer, or filter program. Generally, the list of interesting functions falls off very fast, so a threshold can be set on how many functions are reported. The format can be in ASCII, allowing postprocessing by custom filter programs.

The data collected to disk is kept in binary form to attempt to reduce the size of the files. A program called *sli_interp* is provided to convert the data to ASCII. A postprocessing program can then take that ASCII stream and generate interesting reports.The figure below shows some sample output from *sli_interp*.

The first column provides feedback for nesting information. If no function calls are nested, all of the information associated with a function is kept to one line starting with a vertical bar. If other function calls are made within a traced function, on open brace denotes the entry of a function and a close brace denotes exit from the function. A dashed field means this information can not be provided until the function exits, or it has already been provided on the function entry. This example starts with the application making a call to *xgl_inquire*. The *xgl_inquire* function, in turn, makes several libc calls. We also see that *calloc* makes two other libc calls, *malloc* and *bzero*. The *PID* column traces which process made the call. The *Library* column shows the library, and the *Function* column shows which function of that library. *Nest* shows detailed scope for which functions called which. *Appl* shows how much time passed since the last application call into this library. *Elapsed* shows how much time

```
Logfile version: SLI log file version 1.0
X   PID  Library           Function          Nest     Appl   Elapsed       SLI Data
{    541 libxgl.2.0.api   xgl_inquire         0      1600     -----         98
|    541 libc.4.api       malloc              0         0        24        314
|    541 libc.4.api       strcpy              0        11        10        188
|    541 libc.4.api       getenv              0         9        65        153
|    541 libc.4.api       getrlimit           0      1535        61        249
|    541 libc.4.api       malloc              0         9        18        179
|    541 libc.4.api       memset              0       119        24        164
|    541 libc.4.api       strlen              0      1288         8        211
|    541 libc.4.api       writev              0       312     80090        190
|    541 libc.4.api       read                0       462      1492        298
{    541 libc.4.api       calloc              0        38     -----        109
|    541 libc.4.api       malloc              1         0        29        186
|    541 libc.4.api       bzero               1         9        12        156
}    541 -----            -----             ---     -----        70         74
|    541 libc.4.api       ioctl               0       780       108        255
|    541 libc.4.api       getpagesize         0       631        39        229
|    541 libc.4.api       mmap                0         8       333        179
|    541 libc.4.api       mmap                0       497       687        205
|    541 libc.4.api       munmap              0       288       448        211
|    541 libc.4.api       munmap              0        16       558        194
|    541 libc.4.api       close               0        34        81        201
}    541 -----            -----             ---     -----    307711         78
{    541 libc.4.api       sscanf              0       839     -----       3361
|    541 libc.4.api       strlen              1         0        10        261
|    541 libc.4.api       strcpy              1        10        11        191
|    541 libc.4.api       ungetc              1       109        11        178
|    541 libc.4.api       _filbuf             1        24         8        154
}    541 -----            -----             ---     -----       205         77
|    541 libxgl.2.0.api   xgl_object_set      0      2864        22        314 S 3
```

```
Function              Blocks  Calls   Prims Segments   C/B   P/C   S/P
xgl_multimarker            0      0       0        0     0     0     0
xgl_multipolyline       3982   6121   15981    69715 1.537 2.610 4.362
xgl_multi_simple_pol       0      0       0        0     0     0     0
xgl_polygon              607    607     607     1214     1     1     2
xgl_triangle_strip        53    673     673    60174 12.69     1 89.41
xgl_quadrilateral_me       0      0       0        0     0     0     0
xgl_stroke_text            0      0       0        0     0     0     0
xgl_annotation_text        0      0       0        0     0     0     0

Totals:
Markers                                 0
Lines                                   69715
Chars                                   0
Triangles                               61388
Calls to xgl_context_post()             4374
Calls to xgl_context_new_frame()        124
Calls to xgl_object_get()               124

Timing:
appl+func+sli time:                      134.20
sli time:               6.14                 5%
appl+func time:                128.06       95%
appl time:             18.77      15%       14%
func time:            109.28      85%       81%
```

was spent on this function call. The *SLI* column shows how much time overhead SLI introduced to collect this information. *Data* optionally contains parameter or other interesting data values before and after the function was invoked.

All of these fields can optionally be omitted from the data collection in order to do up-front data reduction. If the user knows only one process is being profiled or only one library is being traced, then the user can select not to save those fields in the binary file. The *sli_interp* program knows how to handle the reduced data files.

Library specific postprocessors can produce useful reports from the *sli_interp* output. For example, the report in the figure above is a summary of what graphics primitives were used during an application run, how well the application merged multiple primitives into a single library call, and breaks out the percentages of time spent in the application versus the library versus the overhead introduced by SLI. In this example, the entire run took just over 2 minutes (appl + func + sli time = 134.20 seconds). Only 14% of the time was spent in the application and 81% of the time was spent in rendering the graphics. The overhead introduced by SLI only represented 5% of the total time (6.14 seconds). It is fairly obvious that the shared memory data produces the lowest overhead, but not so obvious that the binary data collected to disk is much faster than the formatted ASCII output sent to stderr. SLI memory maps the file and accesses it as if it were memory, leaving it up to the system to write it back to disk when necessary. Contrast that to a formatted print statement being output to a scrolling terminal and the reason for the different overheads becomes more clear.

4.3. Interposing Library Source

The second and third tier users want to go beyond the default reports and libraries that our group has provided. This implies they need to alter the interposing libraries we provide or create new interposing libraries. We provide the source to each interposing library we've already created and we also provide tools to automate the process of creating a new interposing library. There are several steps to creating a new interposing library that have taken us as little as 20 minutes for one library and as long as two weeks for a particularly difficult library whose default output format just wasn't what we wanted to report to our users. On an average, most of our customers have been able to get a useful interposing library in one half-day of effort.

The first step is to create what we call a *prototype* file. This is a file that consists of the function declarations for all functions to be traced. Generation of this file is generally quite easy. For an Application Programmer's Interface library, the declarations are already in a header file. Lint library declarations can serve as a source as well. If C program source is available (either K&R-C [11] or ANSI-C [12]), then the *cproto* program [13] quickly and easily generates the prototype file. Using *cproto* has been our primary method. C++ turns out to be quite difficult to generate interposing libraries for and libc provides some special challenges. Both of these are discussed in more detail in the "Obstacles" section.

Since libc is quite common to many operating systems, we will use a small example of *calloc* and follow it through to an interposing library. The prototype file would contain:

```
void *calloc( size_t num, size_t size);
```

We allow single line comments and preprocessor directives to be placed in the prototype file as well. These are passed directly from the prototype file to the generated C code. The prototype file is processed by an awk script that generates two files. One file is a “translation file” that tracks the total number of functions in the library, the length of the longest function name, and a translation table from a numeric assignment to the ASCII name of the function. Since this information is static once the library is generated, it aids the dynamic allocation of arrays during the data collection phase of profiling. The number-to-name mapping allows compact information to be written to the data file in binary. The second file from the awk script is the C source for the interposing library. We call this a working wrapper template. We use the term *working* because the generated code can compile and be useful right away, but we add the term *template* because truly interesting, detailed data collection generally requires customization of the generated code. Knowing the name and size of parameters is useful, but contextually understanding the contents of a complex structure and what’s interesting generally requires human intervention and customization. The generated C code for the calloc prototype is:

```
void *calloc( size_t num, size_t size)
{
    static char *func_name = "calloc";
    typedef void *(*real_func_type)
        ( size_t num, size_t size);
    static real_func_type real_func;
    void *return_value;
    int save_sli_active = sli_active;

    SLI_DECLARE

    if (sli_active)
    {
        if (!real_func)
            real_func = (real_func_type)
                (*sli_resolve(3, func_name));
        return((*real_func)(num,size));
    }

    sli_mark(SLI_MARK_SLI_ENTER);
    sli_active = 1;
    if (!sli_lib_info_3)
        sli_lib_info_3 = sli_find_info(3);
    if (sli_lib_info_3->tra_ctl == SLI_TRA)
        fprintf(SLI_STDOUT,
            "calloc( num=0x%0x size=0x%0x)\n",
            num, size);
    if (!real_func)
        real_func = (real_func_type)
            (*sli_resolve(3, func_name));
    SLI_PROLOG
    sli_send(SLI_ENTER, 3, 15, SLI_EOP);
    sli_active = 0;
    sli_mark(SLI_MARK_FUNC_ENTER);
    return_value = (*real_func)( num, size);
    sli_mark(SLI_MARK_FUNC_EXIT);
    sli_active = 1;
    sli_send(SLI_EXIT, 3, 15, SLI_EOP);
    SLI_EPILOG
    sli_active = save_sli_active;
    sli_mark(SLI_MARK_SLI_EXIT);
    return return_value;
}
```

This example serves to illustrate several points. First and foremost, strong typing must be followed for the return types and parameter types. Different compilers have different rules for data type sizes, calling conventions, and parameter promotion rules. The template is careful to cast all types. This template is for an ANSI-C compiler. The same awk script knows how to generate output for K&R-C and C++ with minor variations. The function name is placed in a variable per function so it can be symbolically referenced in the *SLI_DECLARE*, *SLI_PROLOG*, and *SLI_EPILOG* macros. These macros are null by default but provide a means for all functions to have common code added with ease.

You will see the constants 3 and 15 throughout the template. Both of these constants are the number-to-name mapping values generated in the translation file. The 3 is for libc and the 15 is for calloc in libc.

Since this function is in libc, there is some added code that is not typically found in the templates. The initial check for the global variable *sli_active* is a hook to avoid recursing on libc from our interposing code (that is to say, we want to trap libc calls from the application and the functions we are tracing but we don’t want to trap libc calls that our tracing software uses). A global variable lacks elegance but provided a quick solution to trace all libc functions. A better solution will be required to support multiple threads.

The *sli_mark* function is used to track the time durations of the overhead SLI has introduced and the duration of the real function when it is called. There are four *sli_mark* calls. First on entry to the wrapper, second just before the real function is called, third immediately upon return from the real function and fourth on exit from the wrapper. For non-libc wrappers, *sli_mark* is the first executable statement.

The first time any function is called in the library, some initial one-time overhead is incurred. The *sli_lib_info* is a shared memory page that is used for multiprocessing locking and run-time interactive control of profile and trace functionality. Likewise, the first time a wrapper function is called, we have to find the pointer to the real function. This pointer is saved so the overhead is only encountered once per function.

The *tra_ctl* structure member contains the current value for the “trace to stderr” option. This is the only data collection under complete control of the customizing user. The shared memory data collection and collection to the file is handled by the support library through the *sli_send* function. A variable list of parameters can be sent and stored as the data field in the binary file.

This is as much as can be automated without contextual knowledge associated with the functions. For instance, the parameters are always just printed as a hex value. Often, a parameter might be a complex structure requiring human intervention to know what information in that structure is interesting and what format it should be printed in. Similarly, it would be inappropriate to simply save all parameter values to the trace file causing a tremendous growth in size. It is more appropriate not to save the parameters by default and allow human intervention to decide what information is important to save and in what format.

5. Obstacles Encountered

There are a number of issues that get in the way of implementing an interposing library. Fortunately, nearly all are solved, although some require fairly detailed system knowledge.

5.1. Finding The Real Function

This is potentially difficult to figure out for an arbitrary operating system. The Solaris 2.3 operating system provides a simple interface to accomplish this. The *dlsym* function is used to find the address of a symbol in a dynamically linked library. Solaris 2.3 provides a special parameter to *dlsym* called RTLD_NEXT which indicates to "find the next address of this symbol in the list of libraries". That's all it takes. For standard SVr4 and earlier versions of Solaris, *dlsym* is available but does not support RTLD_NEXT. The solution we followed is to traverse the linker structures and locate the list of libraries through them. We then *dlopen* each library in the list and loop through the list with *dlsym* looking for the real function. This is not too difficult and is somewhat documented [1] but should not be considered a normal, supported user interface. There are two concerns with this approach. One, be careful not to find your own symbol and get caught in a recursive loop. Two, for the sake of efficiency, you only want to loop through the libraries once to find the first function used in a library and keep that handle around for subsequent symbols from the same library.

It is not uncommon for compilers to slightly alter the names of functions from the source to the object file. This usually takes the form of an underscore character placed at the front and/or back of the name. The *dlsym* function properly handles the underscore for the programmer. The C++ language adds considerably more information including the class and parameter type information in the object file function name. This presents a problem in specifying the name of the real function to *dlsym*. For C++ compilers that preprocess the C++ code into C and then use the C compiler, you can collect the "mangled" name from the C code. For C++ compilers that compile directly into object files, you may need to use the *nm* [4,5,6,7] command to determine the mangled names.

5.2. C++

There are a number of interesting problems that arise when attempting to interpose on C++. As just mentioned above, finding the real function with the name mangling schemes is one hurdle. Generating the list of function prototypes can be much more complex than with C. Interposing on a programmer's interface is still straightforward, but finding all of the functions in a C++ library can be quite difficult. A proper profile and trace of a library needs to know where all of the overhead comes from. Scanning the source is insufficient since C++ may generate a lot of functions for the programmer. Examples of generated functions are constructors, destructors, copy operators, function templates, and virtual functions. The solution appears to be to query the library object files for functions and reverse that back to function prototype declarations. However, this can still be missing critical information such as default parameter values and full type information and class member function declarations for compiler created functions. At this time, C++ still remains a challenge which we've only partially solved.

5.3. Interposing On libc

Our support library is written in C and uses many functions from libc. Furthermore, most of the libraries we interpose on are also written in C and make use of libc. When we finally decided to add libc to the list of supported libraries, we found ourselves with recursive looping problems. The COLA project [17] also uses LD_PRELOAD to interpose on system calls and reports a similar looping problem.

Our solution required two steps. First, the interposing version of libc checks a global variable to know if it is being called from an interposing or SLI internal function rather than an application or regular library function. If so, it calls the real function directly without collecting any statistics. As noted in section 4.3, the use of a global variable will prove to be a problem when we want to support multiple threads. This is our only global variable and might be fixed by making it a thread specific variable. Second, the routine to find the real function had to be made "libc clean". That is to say, it couldn't have any references to libc in that one function or it had to precisely resolve any libc functions it did use directly to the real libc library.

5.4. LD_PRELOAD Side Effects

LD_PRELOAD should be used with care. One side effect of this environment variable is that the interposing functions are loaded for all commands issued. You may end up collecting data from more processes than expected. Also, if the interposing functions reference symbols expected to be resolved by libraries of the application, other commands might not have included those libraries, leaving those symbols unresolved causing command execution failure. This can be overcome by linking each interposing library to the library it interposes.

5.5. Scoping Issues

All functions in a library made available to an application programmer have to be declared global. However, it is possible that the library may have internal support routines that it uses but does not expose to the application programmer. If a function is declared static, then it can not be interposed. We generate multiple versions of interposing libraries for each target library. One consists exclusively of the functions available through the Application Programmer Interface (API); a second for all non-static routines in the source; and sometimes a third containing a specifically interesting subset of functions.

Global variable references can be a problem if not considered carefully. Two functions within a library may share access to a global variable. The scope of that global variable and whether or not the interposing functions are interested in that variable raises some issues. If the variable is global to the entire library and application, then no problem exists. If the variable is shared between two functions in the same source file but not global to the library and application, then it may not be accessible. Similarly, we have encountered some compiler discrepancies on inner library calls. If two functions *foo* and *bar* are contained in the same source file, and compiled to the same object file, and *foo* calls *bar*, some linkers improperly resolve *bar's* reference in *foo* at link time rather than run-time which prevents interposition. This is actually a bug and if encountered, can generally be overcome through compiler or linker command line options.

5.6. Parameter Handling

It is reasonable to believe that a function found in a library is independent of the compiler it was generated from, but in reality, problems such as parameter promotion and variable parameter list handling can present particularly difficult problems to isolate and resolve.

A hard and fast rule to apply is: use the same compiler for your interposing function as the real library. K&R-C compilers [11] have different parameter promotion rules from the ANSI-C standard compilers [12]. If the interposing function does not properly pass the parameters down to the real function or properly pass the return value back to the application, the interposing function is useless.

Variable parameter list functions are an especially interesting problem to solve generically since it is the responsibility of the called routine to determine how much information to read from the stack. The interposing function has the responsibility to pass the correct amount of information down to the real function. We solved this two ways. One, for our automatically generated interposing functions that contain variable argument lists, we have some in-line assembly routines inserted that copy the entire frame of the calling routine into the stack for the real function. This can potentially copy too much information, generating unnecessary overhead, but guarantees the real function receives everything. The other solution is to customize the interposing routine to know how to parse the stack and pass down the correct amount of information. This too adds overhead since the stack must be both parsed and copied, but insures only the necessary amount of information is copied.

5.7. Multiple Processes and Processors, Threads, and Network Implications

Data collection and interpretation is straightforward if only one process is collecting data, but multiple process data collection is too valuable to ignore. For example, setting LD_PRELOAD to include all graphics libraries before starting the window system allows capture of all frame buffer activity for every application. The three data collection points plus the central control area must maintain atomic transactions for updates. For multiple processes on multiple processors on a single system, this can be handled fairly easily through an atomic read-modify-write semaphore in shared memory. Multiple threads of the same process on multiple processors adds complications. The same process-id may have mixed library function entry and exit flows. A thread identification needs to be included with the process-id to sort out data flow and maintain nesting stacks. Multiple processes running on separate machines in a network are very difficult to synchronize and we are only starting to tackle that problem.

Semaphore locking adds the potential for deadlocks. We only lock when we are ready to update shared information and then immediately free the lock.

This has only been a problem when the application being profiled is killed. Our solution is to have the request for a lock time-out and clear the lock itself, reporting that the data "may be corrupted".

5.8. Timer Overhead

The resolution of the timer can make a major difference in the usefulness of a profiler. Initially, we used the *gettimeofday* libc function, but found the overhead of a regular system call took on the order of 50 milliseconds when we wanted resolution on the order of nanoseconds. Under Solaris 1, we wrote a device driver to provide direct user reads of the system clock. Under Solaris 2, a new function *gethrtime* is provided. Both of these gave us around 2 microsecond clock resolution improving our accuracy considerably.

6. Application of the Tools

The tools have proven quite successful in quickly isolating performance bottlenecks in the use of graphics libraries and have provided the expected feedback to both the application writer and the library writer. What was unanticipated was the amount of information we could gather and how that information could be applied.

First, once the tools were in place, we found adding new libraries to be trivial (with the single exception of libc). In one case, the time between the request for a new interposing library and the time it was ready was only twenty minutes. In general, we are now surprised if it takes us more than two days to overcome any difficulties in creating a new interposing library. We originally anticipated looking at only a few libraries. The ease of adding new libraries has led to a quick proliferation of new libraries on demand and spread beyond graphics libraries. Sun customers who were shown the tool to assist in graphics performance have taken the initiative to create interposing wrappers for their own libraries.

The application developer uses the default reports and the postprocessing reports to be able to make better use of the library. The hardware and library developers get feedback on actual usage patterns in the library. That information can be applied in many ways. The library builder can sort functions that are often used together to provide cleaner paging. Application regression tests can show what primitives and attributes are used and which aren't (and thus candidates for eventual removal). Analysis of benchmarks, demonstrations, and actual application usage emphasize what functions are critical and with what attributes or parameter values, what functions are time consuming, and what functions deserve the most attention and possible hardware acceleration.

The ability to capture all of the calls and parameters has potential beyond simple playback. If an application records a session, encounters a bug, and that bug reproduces in the playback, then the odds are pretty good that the bug really is in the library or bad parameter values were passed to the library by the application. Either way, vendor support and bug reporting can reproduce and analyze the bug without having to acquire the application, data, and instructions. Additionally, the playback program (or even the wrapper) need not actually call the real library but can instead be used as a translator. The translator might emit simulation traces allowing developing hardware to test different schemes against real application data patterns. The translator might emit calls to an alternate or new version of the library testing the robustness and performance of the new library before the application has actually been ported. Furthermore, the playback is often considerably faster than the original application since the computations leading to the function calls are already made. This means that bug tracing is much quicker and easier. Additionally, having the source to the playback program rather than the original application means special interposers like Purify can be applied to the run by recompiling and linking the playback program rather than the application source. If the wrappers are compiled with debug flags, then a debugger can be used to provide functionality on library functions you wouldn't normally have access to. An example is conditional breakpoints based on contextual parameter values to obtain a callback stack on a system call.

7. Conclusions

Dynamic library interposition has been extremely successful for us. We have been able to exploit the detailed information in many different and useful ways for many different libraries. The value of tracing the parameters in addition to the functions, should not be underestimated. Initially developing the tools was nontrivial but with the tools in place, our development teams are able to make much more informed decisions based on real workloads and fewer guesses. We've generated approximately 40 interposing libraries of both Sun supplied libraries and third party libraries. Around a dozen applications have used SLI as the primary analysis tool with significantly improved performance. Playback to test pre-release hardware and software improved release quality, and hardware simulation trace files are currently being generated for projects in progress.

Acknowledgments

Recognition must go to Doug Gehringer and Dave Phillips for being the first users, major shapers in the directions of the development, and contributors to the playback, symbol resolution, and early timer code. Thanks to Rob Gingell and Rod Evans for quick fixes and knowledge dumps on the strange linker magic we've encountered. Timothy Foley has been our front line and primary evangelist for SLI, and lots of support and patience can be credited to Dean Stanton. Additionally, Brian Herzog, Ralph Nichols, Matt Perez, Jon Cooke, Mike Penick and Roger Day all had the foresight to take the long term view and support tool development despite the difficulty up front in demonstrating the difference tools can make to the bottom line.

References

[1] "System V Application Binary Interface", UNIX Press, Prentice-Hall Inc., 1990, ISBN 0-13-877598-2.

[2] "SunOS 5.3 Linker and Libraries Manual", SunSoft, Part No: 801-5300-10, November, 1993.

[3] Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, "Shared Libraries in SunOS", Summer Conference Proceedings, Phoenix, 1987, USENIX Association., 1987.

[4] "Unix User's Manual: Reference Guide", 4.2 Berkeley Software Distribution, March, 1984.

[5] "SunOS Reference Manual", SunSoft, Mountain View Ca., Part No: 800-3827-10, March, 1990.

[6] "System V Interface Definition", AT&T, Third Edition, 1989.

[7] "SunOS 5.3 Reference Manual", SunSoft, Mountain View Ca., Part No: 801-5297-10, October, 1993.

[8] "The SENTINEL Debugging Environment", Virtual Technologies, Inc., Dulles, VA. info@vti.com.

[9] "Purify User's Guide", Pure Software Inc., Sunnyvale, CA. info@pure.com.

[10] "Introduction to SPARCworks", SunPro, Mountain View CA, Part No: 800-7262-11, October, 1992.

[11] Kernighan, Brian W., Dennis M. Ritchie, "The C Programming Language", Prentice-Hall Inc., 1978. ISBN 0-110163-3.

[12] Arnold, Ken, John Peyton, "A C User's Guide to ANSI C", Addison-Wesley Publishing Co., 1992. ISBN 0-201-56331-2.

[13] Huang, Chin, "Cproto Manual", cthuang@zerosan.uucp or chin.huang@canrem.com, 1993.

[14] Ball, Thomas, James R. Larus, "Optimally Profiling and Tracing Programs", ACM, 089791-453-8/92/001/0059, 1992.

[15] Wall, Larry, Randal L. Schwartz, "Programming perl", O'Reilly & Associates, Inc. 1990.

[16] Jones, Michael B., "Interposition Agents: Transparently Interposing User Code at the System Interface", Proceedings of the 14th ACM Symposium on Operating Systems Principles. Asheville, NC, December, 1993.

[17] Krell, Eduardo and Balachander Krishnamurthy, "COLA: Customized Overlaying", Winter USENIX Conference Proceedings, January, 1992.

[18] Fowler, Glenn, Yennun Huang, David Korn, Herman Rao, "A User-Level Replicated File System", Summer USENIX Conference Proceedings, June, 1993.

Tim Curry is a Senior Staff Engineer with Sun Microsystems. He is currently working in the Technology Development group of Sun designing portable workstation hardware. He has been with Sun since 1985 primarily in windows and graphics software. Tim has a B.S., M.S., and Ph.D. in Computer Science from the University of Central Florida.