Middleware Enabled Fault Management for Commercial Operating Systems

Charlotte Rekiere

Advisor: Daniel P. Siewiorek

Master's Report

Abstract: Commercial computer systems have escaped fault-tolerant design requirements typically reserved for mission critical systems. As computer systems become an integral part of daily activities people are beginning to depend on and expect fault-free behavior. The implementation of a fault-management middleware layer to an existing operating system can prove to be an effective way to quickly and effectively add fault-management features to commercial computer systems.

This paper evaluates and defines a taxonomy of the implementations of four fault-management middleware layers in three commercial off-the-shelf Operating Systems: pSOS (embedded OS), Mach 3.0 (micro-kernel) and HP-UX (monolithic kernel).

The middleware development process for HP-UX is described and analyzed for performance and system overhead. Adding assertions shows the ease of implementing fault-management features to the HP-UX middleware. As a demonstration, assertions are used to protect an application from incorrect kernel behavior exposed in the unmodified operations system through running Robustness Benchmarks [Dingman96].

1.0 Introduction

Typically, the expectation and motivation for developing fault management techniques has focused on highly-specialized mission-critical systems. Commercial systems have escaped similar scrutiny and usually do not provide any type of fault management in the general computing environment. Computer systems have become an integral part of daily activities and people are beginning to depend on and expect fault-free behavior. One method to provide fault management policies on commercial systems is the implementation of a middleware layer with the necessary visibility and control for adding fault management techniques. A middleware layer can be described simply as software placed between two existing systems to facilitate their communication and interoperability. [Lee95].

This paper evaluates the implementation of four system call monitor middleware layers in three commercial off-the-shelf Operating Systems. Specifically, pSOS (embedded OS), Mach 3.0 (micro-kernel) and HP-UX (monolithic kernel). The middleware layers evaluated reside between the operating system and an application. The evaluation has produced a taxonomy of the architectural aspects that facilitate implementation of a fault management middleware layer.

The HPLR (HP Library Replace) middleware development process is described and analyzed for performance overhead. An assertion policy is evaluated on HP-UX with the execution of Robustness Benchmarks [Dingman96]. It is expected that the results of this work will provide a framework for understanding how middleware fault-management techniques can be applied within current operating system architectures. The attributes that are important to maximize the performance and capabilities of the mechanism defined can be applied to the development of future architectures.

2.0 Taxonomy

System calls provide the interface between a process and the operating system. Requests for operating system service can be made at many levels. At the lowest level, a system call is made directly. Higher-level requests, satisfied by a command interpreter or systems programs, are translated into a sequence of system calls [Silbershatz95]. A mechanism for monitoring system call requests can be developed when the method defined to service a system call for a particular operating system is understood. A log of system calls generated can be used to debug application or operating system behavior, analyze system performance and trend analysis. Three operating systems with monitoring mechanisms were evaluated to develop a taxonomy of operating systems features that facilitate system call monitoring: Mach 3.0 [Rashid89], a micro-kernel based operating system, pSOS [Integrated95], an embedded operating system and HP-UX [HPa95], a monolithic operating system.

The taxonomy begins with Figure 1 which defines a simple control flow model of a system call request in a generic operating system. Control switches from the application to the kernel when the system call is executed. This switch is defined by the enter path of Figure 1. After the system call is completed, control returns to the application as defined by the exit path of Figure 1. A middleware layer is added in Figure 2 that provides the system call monitor function. The figure shows the enter and exit paths both passing through the middleware layer. Depending on the operating system, the middleware layer may be restricted to only an exit or enter path. If the middleware is restricted to the exit path then checking arguments prior to entering the kernel would not be possible.

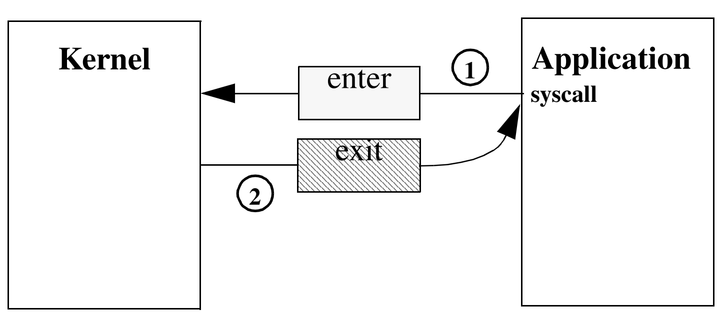

FIGURE 1. Generic Operating System Model

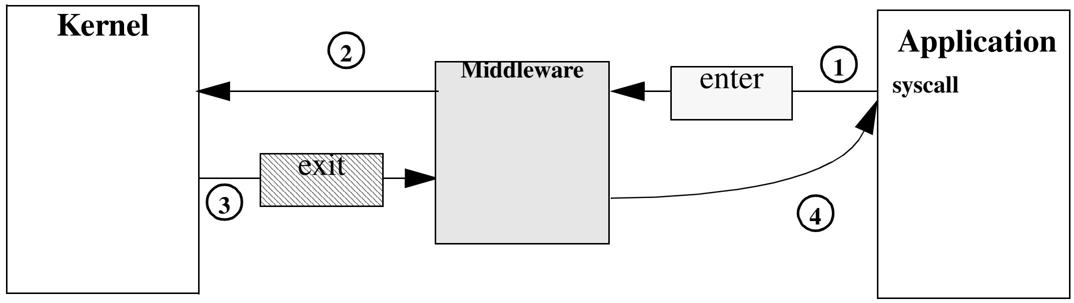

FIGURE 2. Generic Operating System model with Middleware

How the middleware layer is implemented for a specific operating system depends on the mechanism defined to switch control between an application and the kernel. The next set of figures will show three operating systems and four mechanisms to enable system call monitoring following the model established in Figures 1 and 2. A summary and comparison of the attributes of each middleware implementation is provided in Table 1.

The control flow of the pSOS operating system is represented in Figure 3. pSOS handles system calls through a single software interrupt. When the system is booting an entry in the interrupt vector table is set to point to the location within the kernel to execute the system call requested by an application. To execute a system call, the user application executes an interrupt instruction, and the processor jumps to the location specified in the interrupt vector table [Integrated95].

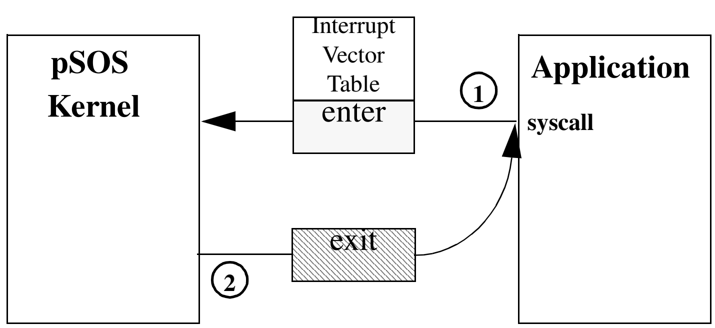

FIGURE 3. pSOS Operating System model

A middleware layer is added by mapping the entry in the interrupt vector table during the boot sequence to point to a routine in the middleware. This change forces all the system calls made by the user application to pass through the middleware prior to entering the kernel. Once control is passed to the kernel the system call is executed and the kernel returns control to the application as shown by the exit path of Figure 4.

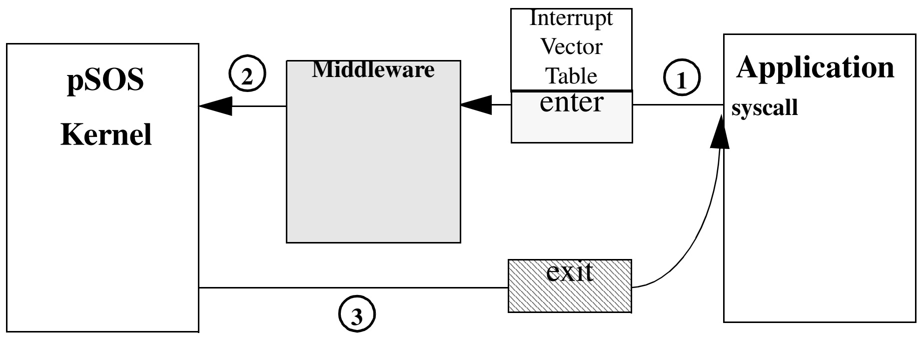

FIGURE 4. pSOS Operating System model with Middleware

Figure 5 shows the control flow of the Mach 3.0 microkernel. In microkernel operating systems, system services are divided between a user-level server, which provides much of the visible operating system interface, and a supervisor-level microkernel that implements low-level resource management. The microkernel provides enough low-level support so that various servers can be implemented to run on top of it [Russinovich94]. The server implemented for the system call monitor evaluation is UNIX 4.3 (UX) [Bach86]. In Mach 3.0, the microkernel provides an Inter-Process Communication (IPC) mechanism, virtual memory management, low level device drivers, and task/thread scheduling. UX implements all UNIX 4.3 system calls, signals, the UNIX File System, and UNIX process management.

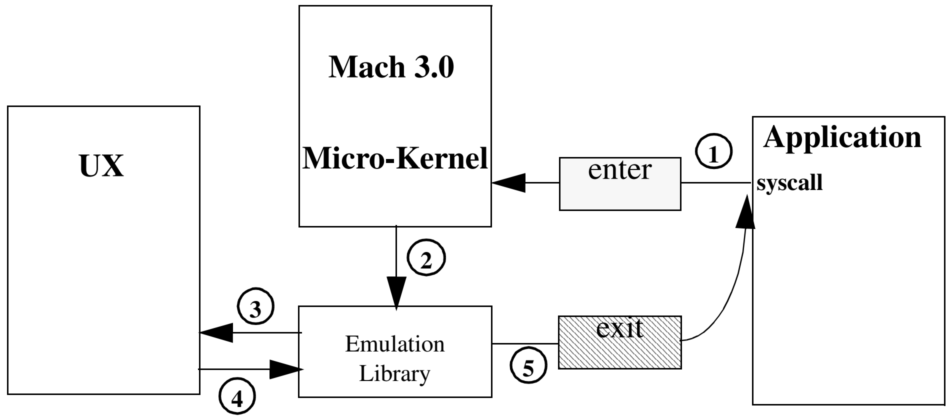

FIGURE 5. Mach Operating System Model

When a system call trap is made, Mach switches control into a special UX portion of the application address space called the emulation library. This library takes the system call parameters and number, and packages a Mach IPC message which is sent to UX. In UX, threads wait for incoming requests and after decoding the message, call the appropriate function. After the request has been serviced, UX packages a reply message and sends it to the application. The emulation library unpackages the reply parameters from the message and control is returned to the instruction following the trap [Russinovich94].

The middleware layer implemented for Mach 3.0 introduces the concept of a sentry mechanism. The central idea behind the sentry concept is that operating system entry and exit points provide sufficient visibility and control to support the majority of standard fault detection and tolerance techniques. These points, called sentry points, allow the encapsulation of operating system services such as system calls, page faults, interrupts, etc. [Russinovich94]. This encapsulation of a system call is represented in Figure 6. The same path as in a standard system call is followed until the UX thread receives a system call request message. At that time, the thread checks to see if a monitoring policy is enabled for the requesting process for that system call. If any are enabled, their entry sentries are executed. The entry sentry passes control to UX to execute the system call and control returns to the application through the execution of an exit sentry. The sentry mechanism middleware layer was used successfully to implement monitoring. Sentries also proved to be an extremely flexible and powerful mechanism upon which fault detection, recording, replay and management techniques can be built.

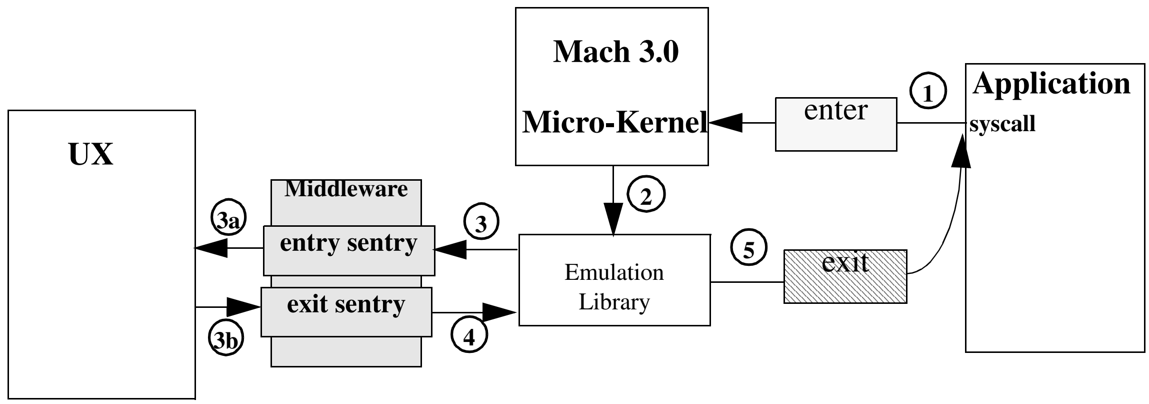

FIGURE 6. Mach Operating System Model with Middleware

Figure 7 represents the control flow for the execution of system calls in HP-UX. HP-UX is categorized as a monolithic kernel in which all system service and low-level control is contained in the kernel. All system calls are funneled through a single entry point in the system space, which is identified by space register 7 (sr7) [HPb95]. As with most Unix kernels, the system call code is contained in libc.a or libc.sl depending if archived or shared libraries were linked to an application.

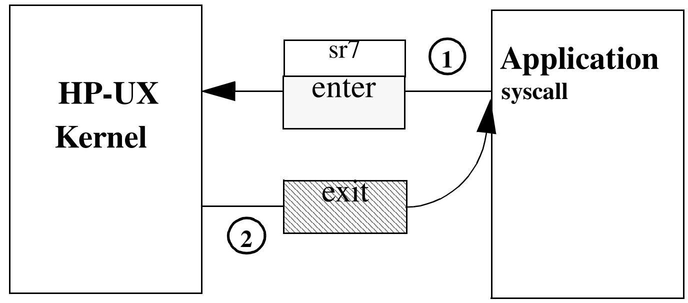

FIGURE 7. HP-UX Operating System model

The `ar` Unix command allows for code objects to be added or removed from archive libraries. To enable middleware for a system call, a custom libc.a containing objects with middleware functions is linked with the application. This method is called HP Library Replace (HPLR). The control flow for HPLR is shown in Figure 8. Details of how code objects are linked to the application are provided in section 3.0. Archive versus shared libraries on HP-UX are discussed in the Appendix.

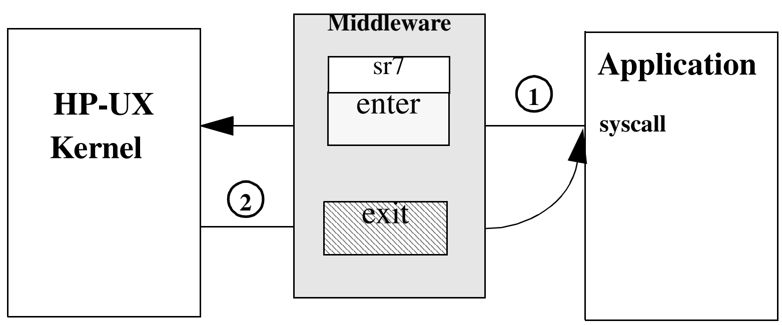

FIGURE 8. HP-UX Operating System model with Middleware (HPLR)

The last method investigated to build the taxonomy is represented in Figure 9. HP-UX is built with Kernel Instrumentation (KI) capabilities that are not enabled unless specifically requested by a user program.   A shareware program called trace1.6 was developed by Kartik Subbarao <Kartik_Subbarao@hp.com> which enables KI for the purpose of monitoring system and kernel calls. The trace program is available at the HP-UX Porting and Archive Centre (http://HP-UX.cae.wisc.edu).

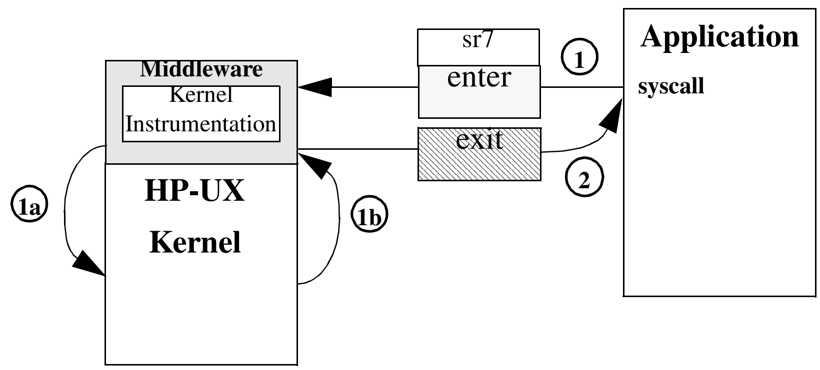

FIGURE 9. HP-UX Operating System model with Middleware (KI)

KI resides within the kernel and is enabled with code from a custom library provided with the trace program. The trace program was developed to provide a similar functionality to the PTRACE_SYSCALL function of `ptrace` available on other UNIX systems [Subbarao96]. When KI is enabled, data structures defined in the system are updated with the argument information and made available for the trace program to capture. A majority of the work done by the trace program is formatting the data output.

2.1 Middleware Attributes

Table 1 is a summary of the main features of each monitoring policy evaluated for the taxonomy. Each attribute is defined in the list following the table. All the middleware implementations provide the capability to monitor system calls. Table 2 is a summary of the fault management capabilities beyond system call monitoring available in the middleware evaluated.

TABLE 1. System Call Monitor Middleware Attributes Summary

Attribute	**pSOS**	**Mach3.0 sentry**	**HPLR**	**HP Trace**
Application (T-Transparent, D-Dependent)	T	T	D	T
Middleware Enable (K-Kernel Boot, U-User Program, L-Library Link)	K	U	L	U
System Call Visibility (E-Enter, X-Exit)	E	E,X	E,X	X
Log System Call (N-Name, A-Arguments, R-Return Value, E-Error Codes)	N	N,A,R	N,A,R,E	N,A,R,E
Libraries Required (S-Standard, C-Custom)	S	S,C	S,C	S,C
Attach to running process (Y-Yes, N-No)	N	Y	N	Y

Application (Transparent, Dependent): Middleware is application transparent if the application is totally unaware of the fault management being performed on its behalf. The application requires no modification to use the monitoring policy. The middleware is application dependent if the application participates actively in the monitoring policy. The HPLR implementation narrowly fits in this category because of the requirement of linking application object code to a custom libc.a to enable the middleware.

Middleware Enable (Kernel Boot, User Program, Library Link): This attribute describes how middleware is enabled for an application program. Kernel Boot: requires a system boot sequence to enable the policy or requires modification to kernel source code beyond what is commercially distributed. User Program: a separate user program must be executed to enable the middleware for an application or process and in some cases execute the application program. Library Link: custom middleware enabled libraries must be linked to the application program.

System Call Visibility (Enter, Exit): Enter visibility is achieved if the middleware has control between the application system call request and kernel execution. This is important for providing assertion policies and fault management as described in Section 3.5. Exit visibility is achieved if the middleware has control between the kernel return from system call and the application. This allows kernel return values and error codes from the kernel (if available) to be monitored.

Log System Call (Name, Arguments, Return Value, Error Codes): This attribute describes which monitored values the middleware logs.

Unique Libraries Required: The middleware requires access to unique libraries to execute monitoring specific system calls. The sentry activation mechanism in Mach 3.0 includes several system calls that have been added to UX, (un)guardserv, (un)guardproc, libarg, sentryon, sentryoff, ftexecve [Russinovich94]. This is also the case when the middleware requires access to non-monitored system calls for writing the log to non-volatile storage. HPLR requires a custom middleware enabled libc.a along with a custom logging function to write the monitored data to disk. HP Trace requires a KI library provided with the trace program.

Attach to Running Process: The middleware monitoring can be dynamically attached to a currently running process.

TABLE 2. Advanced Fault Management Middleware Attributes

	pSOS	Mach 3.0 sentry	HPLR	HP trace
Roll-back Visibility (Y-Yes, N-No)	Y	Y	Y	N
Roll-back Recovery (Y-Yes, N-No)	N	Y	N	N
Assertion Policy (Y-Yes, N-No)	N	Y	Y	N
Kernel Call Monitor (Y-Yes, N-No)	N	N	N	Y

Roll-back visibility: The middleware has visibility and access to buffers and data needed to create a log that can be used for roll-back recovery of the application. As an application executes, information that would not automatically be recreated in a repeat run, such as inputs from the user, must be saved in stable storage. When a failure occurs, this information can be used to re-execute the failed run, bringing the application up to the state that existed just before the failure [Russinovich94].

Roll-back Recovery: This feature requires middleware designed with roll-back visibility. Recovery of an application execution is possible when processes periodically save their states on stable storage during failure-free operation. If a failure occurs, the processes will use the information on stable storage to restore a global consistent state and restart execution from that state, instead of restarting the computation from the beginning [Elnozahy93].

Assertion Policy: Middleware has implemented assertions to protect the kernel from system calls that would have caused the system to hang or crash. The middleware can recognize a fault condition and return an error to the application rather than passing the malformed system call to the kernel.

Kernel Call Monitor: Kernel calls are operations executed in a kernel during application execution that are not requested directly by an application. In contrast, system calls are operations executed in the kernel at the direct request of an application. A kernel call monitor is capable of tracing kernel operations.

3.0 HP Library Replace (HPLR) Middleware Implementation

The development platform for the HPLR implementation is an HPPA RISC processor HP9000 series 750 workstation running the HP-UX 10.10 operating system. HP-UX is based on the UNIX System V Release 4 operating system[HPa95]. HPLR middleware is designed with two main objectives. First, to generate a log file of monitored system calls, arguments, return values and error codes. Second, to provide sufficient visibility for the implementation of assertions and roll-back recovery fault management features. The application and operating system features involved in executing a system call that are modifiable must be understood to achieve these objectives.

Three areas were considered for implementation of the middleware: kernel source, application binary, and the system library. The system library was selected and is reflected in the name of the method we chose, HP Library Replace (HPLR). Modifying kernel source was not an option because the kernel source is not available in the commercial distribution of HP-UX. Hard coding middleware into the application binary was not desired because it would result in a machine dependent solution. The HPLR implementation creates a middleware-enabled application by linking a custom library to an application. This is similar to hard coding middleware into an application binary but with the added benefit of machine independence. Machine independence is achieved because the binary is built at the link level rather than at the machine code level. The following four sections provide an overview of system library and linking on HP-UX, a detailed description of the middleware layer implementation, an overview of system call execution on HP-UX and an analysis of HPLR's impact on system performance.

3.1 Archive libraries and the linker

System call code is located in the system library libc. libc is available in two forms on HP-UX, archive or shared. The archive format of libc is called libc.a and the shared format is called libc.sl. A description of the differences between shared and archive libraries is provided in the Appendix. System libraries contain object files generated by compilers and assemblers. The Archive library format was selected for HPLR because the commands available to modify libc.a allow for individual object files to be replaced. This is contrasted to libc.sl which requires the entire library to be rebuilt to add or replace object files.

System call code is linked to an application to create an application executable during the link-edit phase of compilation. The compilation phases of HP-UX are shown in Figure 10. The compiler (`cc`) produces an object file (main.o) from the application source (main.c). An object file contains definitions of code and data in a format the linker uses to build the application executable (a.out). The definitions can be local, global or external.

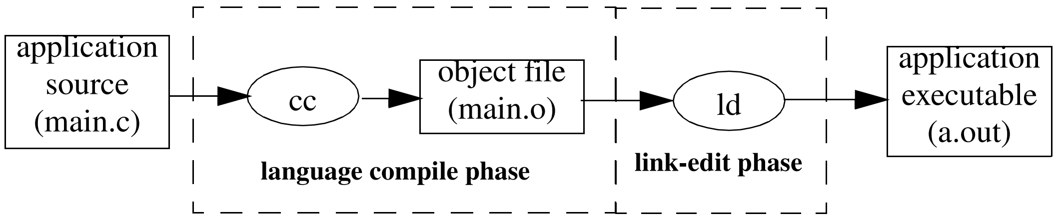

FIGURE 10. HP-UX Compiler Phases

A local definition of a routine or data indicates that it is accessible only within the object file in which it is defined. Global definitions contain the code for a named function, data or procedure. An external reference is the request for a local or global definition of a function, data or procedure in a program [HPc95]. System call code in libc.a is a global definition that an application requests with an external reference. Status information for the local, global and external definitions contained in object files is maintained by a symbol table in libc.a. Each object file can contain multiple definitions of any or all types. Object file contents is determined by the source code it was created from. In libc.a, a system call has one or two object files containing the definitions required to execute the system call.

Figure 11 shows an example of the symbol information for writecall.o, t_write.o and write.o. Writecall is a C program that executes a single `write` system call. Object files t_write.o and write.o contain the definitions to execute the `write` system call. The first column of the symbol table is the address of each symbol or reference. The second column denotes the symbol type. The last column shows the symbol name.

Object file symbols:

```
[writecall.o]:
0000000003 T  main
0000000000 U  write

/usr/lib/libc.a[t_write.o]:
0000000003 T  _write
0000000000 U  _write_sys
0000000003 TS write

/usr/lib/libc.a[write.o]:
0000000003 T  _write_sys
```

Symbol Types:

```
T     indicates a global definition
U     indicates an external reference
d     indicates a local definition of data
S     secondary reference
```

FIGURE 11. Symbols defined for dwrite.o, t_write.o, write.o

A representation of how the linker matches external references and global definitions during the link-edit phase is shown for standard archive libraries in Figure 12. The writecall.o object file contains an external reference to `write`. The linker matches this external reference to the global definition of `write` in t_write.o. The three symbols contained in t_write.o are: `_write`, `_write_sys`, and `write`. The `write` and `_write` symbols are different names representing the same function to support compiler external reference naming conventions. The `_write_sys` symbol is an external reference that the linker matches to the global definition in write.o.

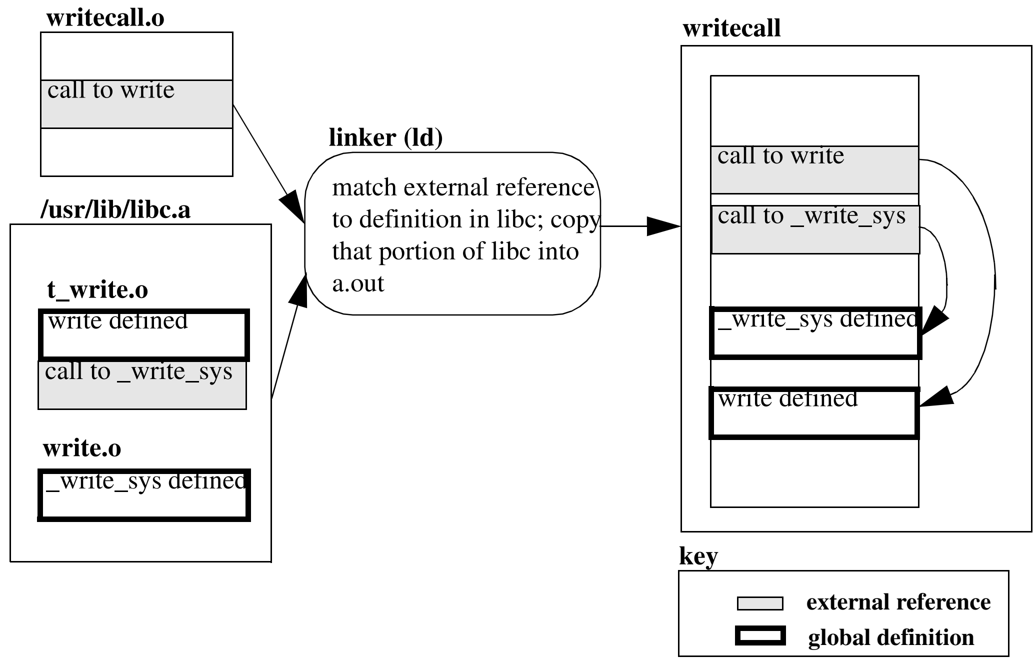

FIGURE 12. Matching external reference for write system call using standard libc.a

The `_write_sys` global definition contains code to enter the kernel and execute the actual system call. The symbol table in libc.a provides the necessary information for the linker to generate an executable output file. The linker generates the writecall binary executable file by matching external references to global definitions. When an application is linked to archive libraries, all code referenced by the application program is copied into the binary executable.

3.2 Creation of Middleware-Enabled libc.a

The middleware-enabled libc.a is created by replacing the global definition of system calls with middleware-enabled global definitions. This is accomplished for archive libraries with the `ar` command. The middleware enabled objects must have identical global definition names to the ones replaced so the linker will match them to the correct external references. When an application is linked with the custom libc.a, the middleware becomes part of the executable for the application. The application unknowingly enables the middleware by making an external reference to the middleware-enabled system call. Figure 13 shows an executable created with the modified libc.a for the `write` call.

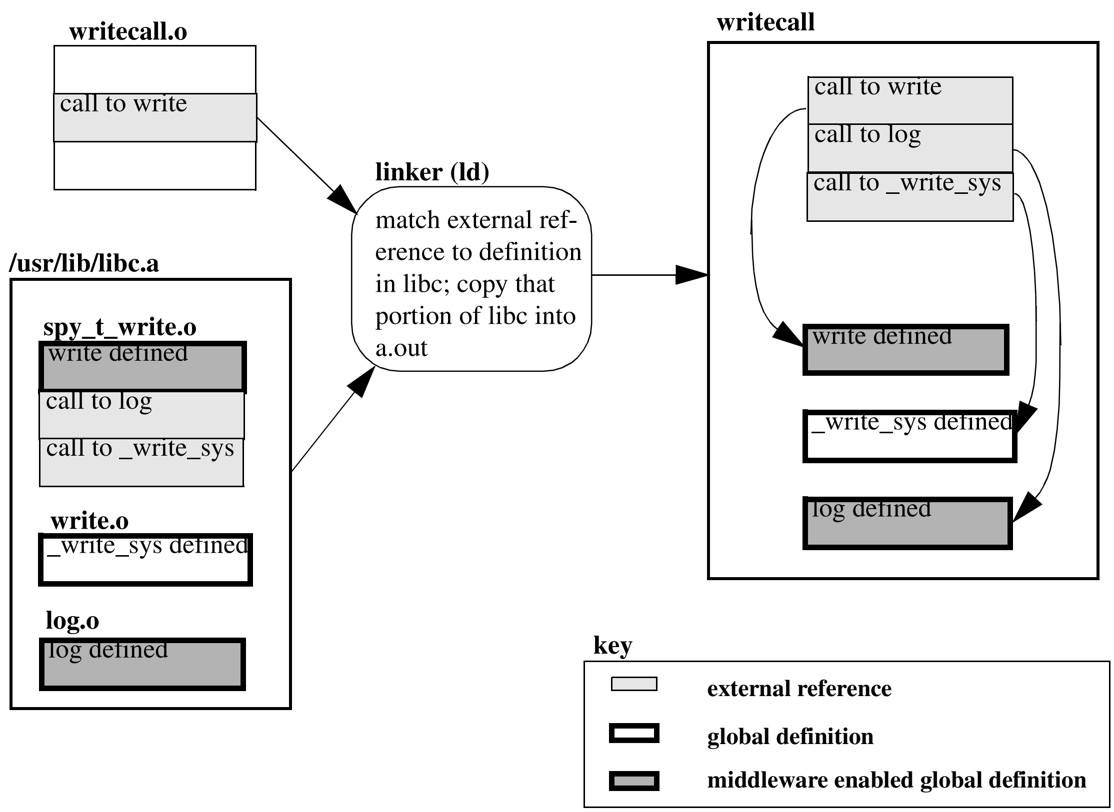

FIGURE 13. Matching external reference for write system call using middleware enabled libc.a

The main difference between Figure 12 and Figure 13 is the addition of spy_t_write.o and log.o object files. spy_t_write.o and log.o were added to libc.a using the `ar` command and contain the middleware functions to perform monitoring. Adding spy_t_write.o to libc.a causes a duplicate definition for `write` in libc.a. The symbol table maintains a date for the named definitions and by default the linker selects the most recent entry. The middleware-enabled object files for system calls were given unique names of the form spy_t_syscall.o to prevent writing over the original default definitions contained in libc.a. The log.o object code defines the `log` function used by all middleware-enabled system calls to create the log file. Object files added to monitor system calls for the benchmarks discussed in Section 3.4 are listed in the Appendix.

Each monitored system call must have the global definition for the system call replaced with a middleware-enabled version. The middleware monitoring code for the `write` system call is shown in Figure 14. Line 18 is where the actual system call is made by calling _write_sys. Lines 19 through 22 obtain the kernel error code if applicable. Lines 24 through 42 collect the monitored data. The monitored data is logged at line 43 and line 44 returns control from the middleware to the application. The format of the log file and examples are provided in the Appendix.

The HPLR middleware is invoked when the application executes a system call. The location of the middleware in the execution process provides sufficient visibility for the implementation of assertions and roll-back recovery fault management features. The code in Figure 14 could be easily enhanced for assertions by checking argument parameters and returning error to the application rather than making the actual system call. The HPLR middleware executes in user space and therefore has access to user input data buffers. [Russinovich94] defines two data sets necessary to perform roll-back recovery: user inputs and events leading to data request from users. Capturing events leading to the request for user data can be accomplished through enabling HPLR middleware for system calls that generate user data requests.

```
/*
 * spy_t_write.c
 * Replaces:_write, write (libc.a[t_write.o])
 */

#include <syscall.h>
#include <stdio.h>
#include <sys/unistd.h>
#include "log.h"
#include "errno.h"

int write(int fildes, int *buf, unsigned int nbyte)
{
  agent_record record;
  int argtype[MAXARG];
  int arglen[MAXARG];
  char *args[MAXARG];
  int i;

  /* actual write system call with parameters passed in from the application*/
  record.retval = _write_sys(fildes,buf,nbyte);

  if(record.retval == -1)
    record.err = errno;
   else
    record.err = 0;

  /* fill record and call log */

  record.sysnum = SYS_WRITE;
  record.ent_ex = 0;
  record.pid = _getpid_sys();
  record.nargs = 3;
  record.totarglen = sizeof(int) + sizeof(int) + sizeof(int);

  argtype[0] = INT;
  argtype[1] = PTRINT;
  argtype[2] = INT;

  arglen[0] = sizeof(int);
  arglen[1] = sizeof(int);
  arglen[2] = sizeof(int);

  for(i=0;i<record.nargs;i++)
  {
    if(!(args[i] = (char *)malloc(arglen[i])))
      exit(-1);
  }

  memcpy(args[0],&fildes,arglen[0]);
  memcpy(args[1],&buf,arglen[1]);
  memcpy(args[2],&nbyte,arglen[2]);

  log(record,argtype,arglen,args);

 return(record.retval);
```

FIGURE 14. Global definition code for the write system call (spy_t_write.c)

3.3 HP-UX System Call Execution

When the HPLR middleware is enabled, two symbols must be available in libc.a. One that matches the external reference made by an application that contains the middleware logging code. The second to execute the actual system call. Without unique symbols to distinguish the middleware from the actual system call in the modified libc.a, the middleware would loop infinitely with no way to execute the system call. Approximately 40% of the system calls in libc.a already have a separate symbol and code for the actual system call. In the cases where a separate symbol was not available one was created. Following the convention in libc.a, the actual system call symbols are of the form _syscall_sys.

Code for an actual system call is dependent on the target operating system's mechanism for entering system space, executing the call and returning control to the application. All system calls on HP-UX are funneled through a single entry point in the system space, which is identified by space register 7 (%sr7). An example of the assembly code used to enter the system space for HP-UX is shown in Figure 15.

```
; _exit_sys call
call         .EQU         1                          ;System call number

             .CODE
             .EXPORT _exit_sys, ENTRY

             .PROC
             .CALLINFO
_exit_sys    .ENTER
             LDIL         L'0xc0000004,%r1
             BLE          R'0xC0000004(%sr7,%r1)     ;Enter System Space
             LDI          call,%r22
             .LEAVE
             .PROCEND
```

FIGURE 15. Assembly code to enter system space and execute the exit system call

System space is a portion of the virtual memory space in HP-UX where the kernel executes system calls and performs low-level system services. Virtual addressing on PA-RISC is based on spaces. A virtual address is composed of a space identifier stored in the space register for that space and an offset within the space. A system call is made on HP-UX by executing a BLE (branch and link external) instruction to enter the system space and loading the unique system call number into general register 22. A list of the system call numbers as well as the location of the system call entry point is in the standard include file: /usr/include/sys/syscall.h.

3.4 Performance and Benchmarks

A set of benchmarks were run to verify the HPLR middleware implementation. Benchmarks were used for verification rather than an application program to ensure a repeatable test environment. Unlike an application, a benchmark executes a finite number of system calls in a consistent order. Table 3 provides a breakdown of the type and frequency of each system call used by the benchmarks selected. The benchmarks include the BYTE magazine benchmarks (dwrite, dread, fcalla, fcalle, fibo, float, iofile, loop, pipes, scall, sieve, bytesort), Usenet (iocall) and Dhrystone (dryr, drynr) benchmarks. The benchmarks were compiled by Rick Richardson of PC Research Inc. and are available in a file called miscbench.tar.gz (http://src.doc.ic.ac.uk/public/public/packages/unix-c/benchmarks/). System call and Benchmark descriptions are provided in the Appendix.

TABLE 3. Count of System Calls made by Benchmarks

Benchmark / System Call	fcalla	fcalle	loop	sieve	fibo	float	dryr	drynr	bytesort	dwrite	dread	pipes	iofile	iocall	scall
`brk`							1	1							
`close`										2		2	2	1	
`creat`										1			1	1	
`execve`	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
`exit`	1	1	1	1	1	1	1	1	1	1	1	2	1	1	1
`fork`												1			
`getpid`															25000
`ioctl`					1	1	1	1	1				1		
`lseek`											256		1000	2002	
`open`										1	1		1	1	
`pipe`												1			
`read`											256	1025	500	4004	
`times`							4	4							
`unlink`											1			1	
`write`					1	2	2	2	12	256		1024	631	1001	
diversity (d)	2	2	2	2	4	4	6	6	4	6	6	7	9	9	3
frequency (f)	2	2	2	2	4	5	10	10	15	262	516	2056	2138	7013	25002
RDF [ratio: d/f]	1	1	1	1	1	.80	.60	.60	.267	.023	.012	.0034	.0042	.0013	.0001
			a				b					a			b

Group 1 | Group 2

Table 3 is separated into groups based on call frequency, diversity and the ratio of diversity to frequency (RDF). Call frequency is the total number of calls for a given benchmark. Call diversity is a count of the number of unique call types for a given benchmark. Most UNIX benchmarks are designed to focus on CPU performance rather than operating system or user program performance. The groupings were established to determine which benchmarks most closely represent a typical user program to use for performance analysis of HPLR. System call diversity and frequency data for typical user programs from [Dingman96] is shown in Table 4.

The order of magnitudes of RDF and frequency data in Table 4 were used to split Table 3 into Groups 1 and 2. Table 4 frequency data includes order of magnitudes 10^2 and 10^3 and RDF data includes order of magnitudes 10^{-2} and 10^{-3}. The benchmarks in Group 2 of Table 3 have a similar range of magnitudes for RDF and frequency as Table 4. Table 4 contains two distinct groups based on the order of magnitudes of RDF and frequency between gdb and xv in the table. Using the RDF value for xv as the dividing line, the split of Table 4 was applied to Group 2 of Table 3 to form sub-groups 2a and 2b. Benchmarks with an RDF higher then 0.0028 comprise Group 2a and benchmarks with an RDF lower then .0028 comprise Group 2b.

TABLE 4. Typical Application Diversity to Frequency Ratios [Dingman96]

	emacs	**gcc**	**gdb**	**xv**	**bitmap**
diversity (d)	23	20	23	26	18
frequency (f)	851	434	927	9140	13027
ratio (d/f)	.0270	.0461	.0248	.0028	.0014

Figures 16 and 17 represent the execution times for Group 2a and 2b benchmarks with various degrees of buffered logging. The libc.a execution time is the benchmark run with standard archive libraries and no middleware enabled. The no log data value is the execution time with middleware enabled and no logging enabled. The middleware enabled without logging adds overhead from assigning argument values to local variables within the middleware. Execution times for various buffer sizes that require writes to disk are shown. When a buffer is full it writes the values to disk and proceeds to fill and write data to disk until the benchmark finishes execution. The buf=all data value is the execution time when the middleware and logging are enabled and all the logged data is written only to main memory. The entry log execution time is measured when the benchmark is run and the middleware is configured to write each log entry to disk immediately after each system call execution. A log entry contains a system call, arguments and return value. An example log file is shown in the Appendix. The parameter log execution time is the worst case logging in which the main data structure and each argument are written to disk immediately when they are available to the middleware. In summary, when logging is enabled to write to main memory only, there is a 1.5 times overhead compared to libc.a and writing to disk adds a 5 to 10 times overhead compared to libc.a.

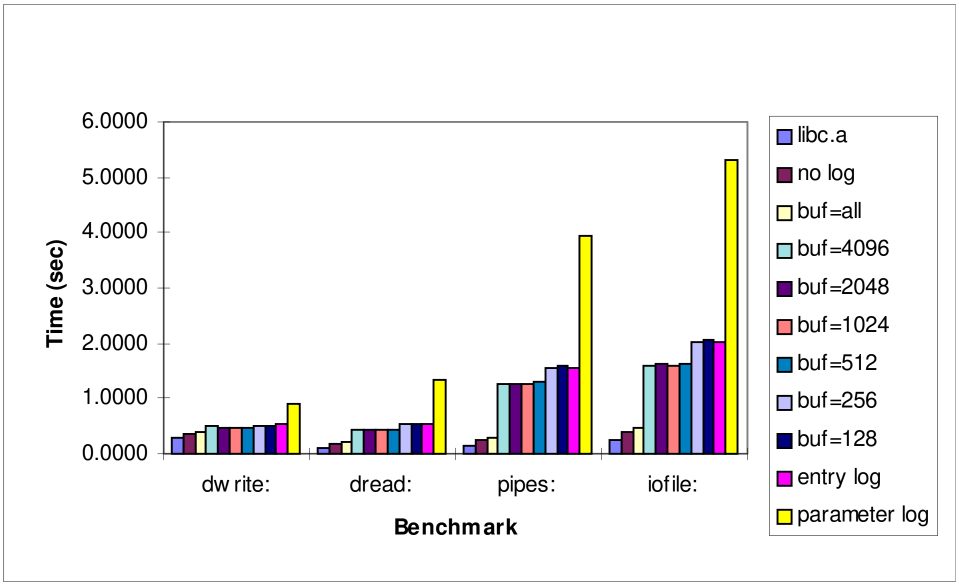

FIGURE 16. Group 2a benchmarks execution times

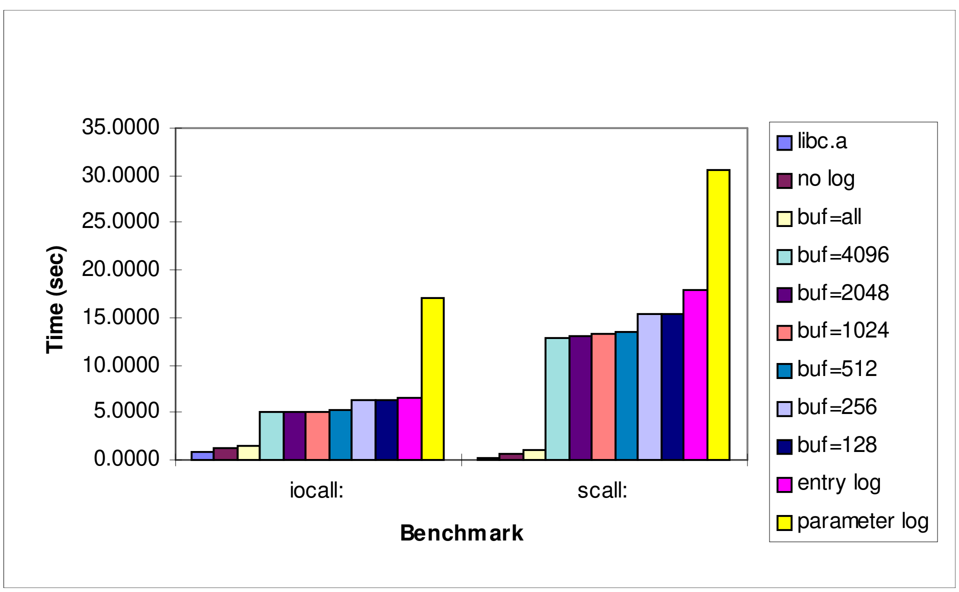

FIGURE 17. Group 2b benchmarks execution times

Figures 18 and 19 show the execution time with logging enabled to libc.a execution time ratio for selected buffer sizes. The ratio gives a representation of the overhead created by the logging of system calls. As the buffer sizes increase the overhead decreases and the ratio approaches one.

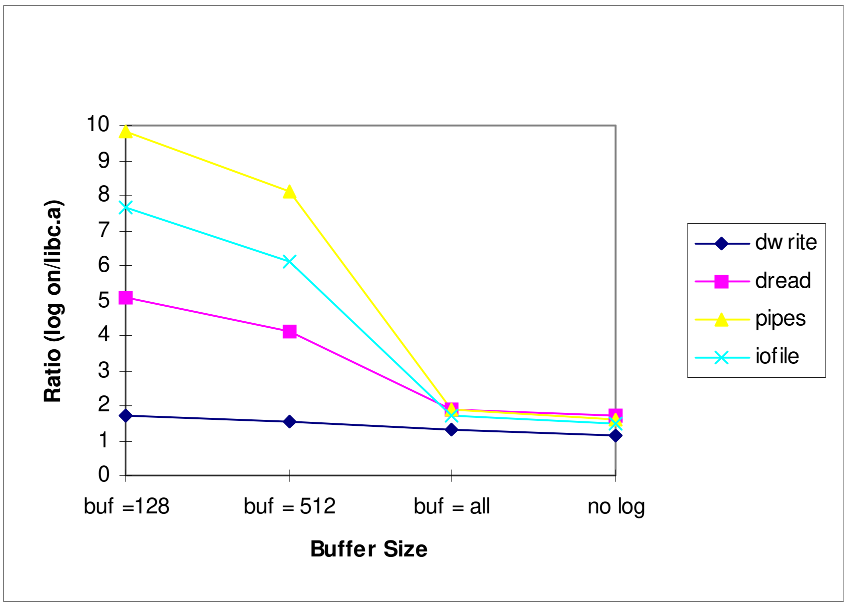

FIGURE 18. Group 2a benchmark execution time ratio: logging enabled to libc.a

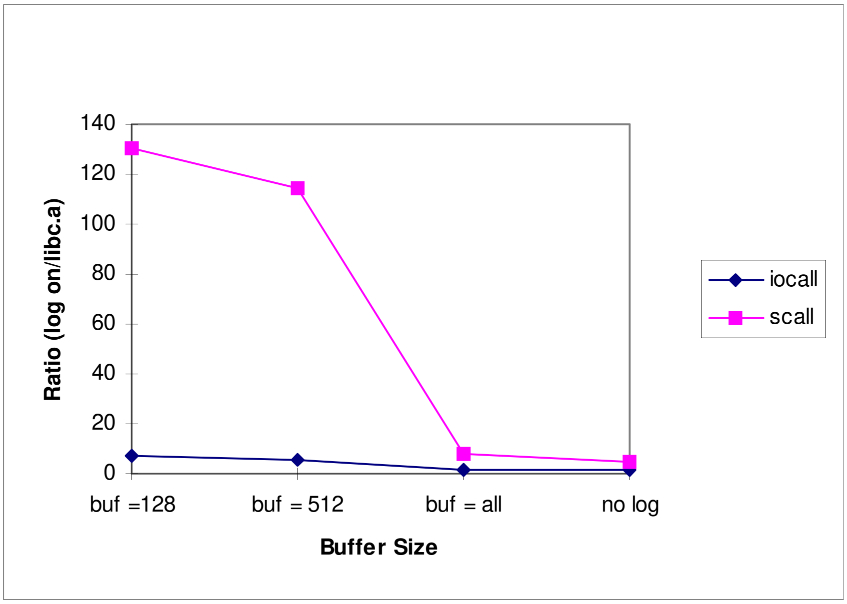

FIGURE 19. Group 2b benchmark execution time ratio: logging enabled to libc.a

The magnitude of the overhead is directly related to the number of log entries. Each log entry represents the execution of one system call. Overhead for system call monitoring can be attributed to storing arguments locally, passing arguments to the logging function and writing the logged data to memory. While overhead is high when writing logs to the disk, main-memory-only logging is effective. If the system call logging was expanded to include a Roll-back recovery mechanism every system call executed by an application would not be required to be logged[1]. Memory logging of system calls in an application is less than 0.07% per call on average.

1. The research done in [Russinovich94] describes a roll-back recovery mechanism that logs only user input and events leading to user input. This reduces the number of system calls and arguments logged compared to the system call monitor mechanism. The roll-back recovery mechanism was run with 9% total overhead.

Table 5 lists data logging rates to main memory for Group 2a and 2b benchmarks. The data logging rate varies by a factor of ten as a function of benchmark from 200 KByte/sec. to 1.4 MByte/sec.

TABLE 5. Data Logging rates to main memory for Group 2a and 2b benchmarks

Benchmark	MByte/sec
dwrite	0.20
dread	0.39
pipes	1.03
iofile	0.82
iocall	0.97
scall	1.42

3.5 HPLR Implementation Summary

The HPLR system call monitor mechanism has been shown to be an effective fault-management middleware layer for HP-UX and accomplishes the first goal for HPLR stated in Section 3.0. The second goal, to provide sufficient visibility for the implementation of assertions and roll-back recovery, is achieved based on the location of the middleware in the execution process. [Russinovich94] defines two data sets necessary to perform roll-back recovery: user inputs, and events leading to data request from users. User inputs are visible to HPLR when the application makes a system call request and passes control to HPLR. Capturing events leading to data request from users can be done by enabling the system call monitor. Verifying argument parameters with assertions is possible when HPLR is passed control from the application. If an invalid argument is found, HPLR can return an error to the application rather than executing the system call. Section 4.0 discusses the implementation of assertions in HPLR middleware.

4.0 Assertions and Robust Benchmarks

Robust Benchmarks stress a system by invoking system calls containing both valid and invalid parameter values. How the operating system handles those parameters is observed.

It is expected that the incorrect system calls are representative of one form of errors made by application designers or corrupted data [Dingman95]. The suite of Robust Benchmarks applied to Mach 3.0 in [Dingman96] was ported and run on HP-UX. The suite tests the `read, write, open, close, stat, fstat,` and `select` calls. These calls were selected because they were the most frequent functions in traces of user sessions[1] [Dingman96]. The robust test involving the `select` call is used to show assertions in HPLR because it resulted in fails on HP-UX.

A six-level classification system scale was defined in [Dingman96] to analyze the results of running robust benchmarks. Table 6 describes the six-levels of the classification system and results for the 405 cases tested.

TABLE 6. Robust Benchmarks six-level classification system [Dingman96]

Class	Description	select call test results
0	No fault (correct execution) or proper error code	378
1	Error indicated, but not the correct error code	0
2	Reports success, when error should have been reported	27
3	Process exits gracefully, with proper error code returned	0
4	Process “Hangs” or halts execution without exiting	0
5	Execution causes the system to halt or fail	0
	Total select call tests	405

Table 7 shows the combinations of arguments tested for the `select` call. A total of 405 tests are generated by building a select call with argument combinations of each type. Table 8 represents the Class 2 failure mode argument combinations. The fails are considered in Class 2 because the select call exhibits the correct behavior even though some arguments are invalid and should have resulted in a return error message.

1. System calls were monitored for multiple user sessions running the following applications: emacs (a text editor), gcc (the gnu compiler), gdb (the gnu debugger), bitmap (X-windows bitmap editor), xv (graphics file viewer). The robust benchmarks focused on the most frequent system calls. [Dingman96].

TABLE 7. Robust Benchmark Test Argument Combinations for the `select` call.

Number of file descriptors (nfds)	**Read file descriptors (readfds)**	**Write file descriptors (writefds)**	**Except file descriptors (exceptfds)**	**Maximum interval to wait for selection (timeout)**
0	NULL	NULL	NULL	NULL pointer
5	Valid pointer	Valid pointer	Valid pointer	Valid pointer
255	Bad pointer	Bad pointer	Bad pointer	-1 (bad pointer)
15000				
-1				

TABLE 8. Select Class 2 Fail Argument Combinations

nfds	**readfds**	**writefds**	**exceptfds**	**timeout**	**Class 2 fail count**
NULL	ALL 3	ALL 3	ALL 3	NULL	27

`select` enables synchronous I/O multiplexing for files and has parameters: nfds, readfds, writefds, exceptfds, timeout. nfds is the number of file descriptors to select for I/O. readfds, writefds and exceptfds are bitmasks containing bits from 0 to nfds-1. The bitmasks represent the status of the files as read, write or exception. timeout specifies a maximum interval to wait for file selection to complete. The Robustness Benchmark test is built from the list of values in Table 6. Each value is passed as an argument to `select` until all the possible combinations of parameters have been used resulting in 405 test cases.

When timeout is NULL, as it is for all the cases in Table 8, select is defined to wait for a signal. The 27 fail cases were classified as Class 2 failures because the select call waits even though other arguments are invalid. Class 2 fails are considered silent errors because the call may or may not cause a failure when select gets a signal. In cases where timeout and nfds are non-NULL, invalid parameters are correctly caught by the operating system and return correct error codes.

Monitoring middleware was generated for the select call using the HPLR method described in Section 3.0. An assertion policy was added to the middleware to detect the invalid arguments. Code to perform the assertion is shown in Figure 20.

```
if ((nfds == NULL) && (timeout == NULL))
{
  errno = 22;          /* error number for invalid argument */
  return(-1);
}
```

FIGURE 20. Assertion code for `select` Class 2 failure

The assertion checks if nfds and timeout are NULL and returns error to the application. Since this case is always invalid (nfds must be greater than 1 because the bit mask is built from 0 to nfds -1). The bit mask parameters are irrelevant to check for this assertion because the operating system correctly returns error for invalid readfds, writefds, and execptfds if nfds is not NULL.

5.0 Summary

A taxonomy has been defined representing system call monitor fault management middleware layers for three commercial off-the-shelf operating systems and four middleware implementations: pSOS (embedded OS), Mach 3.0 (micro-kernel), and HP-UX (monolithic kernel). The features of each implementation were compared and described. The taxonomy provides a framework for understanding how fault-management middleware can be applied to commercial operating systems.

The HPLR middleware implementation on HP-UX was described in detail. Benchmarks run on the HPLR method showed the overhead and performance impact of system call monitoring with various logging configurations. Logging to main memory rather than any of the main memory/disk logging combinations added the smallest percentage overhead per system call logged.

Robust Benchmarks [Dingman96] were run on HP-UX and results for the `select` call were described. Assertions were added to the HPLR middleware for `select` to avoid incorrect kernel behavior. The HPLR method proved to be an effective fault-management layer without requiring modification to kernel or application source code. The implementation of fault-management middleware by replacing system call object code in an archive format system library was introduced. This method can be applied to any UNIX-type operating system that provides the capability to modify or replace individual objects in system libraries.

Appendix

This appendix contains a discussion of archive vs. shared libraries in HP-UX, a description of the HPLR system call monitor log file, a list of the HPLR custom libc.a object files generated for the benchmarks described in Section 3.2, descriptions of the calls listed in Table 3, descriptions of the benchmarks from Section 3.2, and a glossary of terms.

Archive vs. Shared libraries in HP-UX

HP-UX supports two kinds of libraries: archive and shared. Like an archive library, a shared library contains object code. However, when linking an object file with a shared library, `ld` does not copy object code from the library into the binary executable file (a.out) file; instead, the linker notes in the a.out file that the code calls a routine in the shared library. An a.out file that calls routines in a shared library is known as an incomplete executable[HPa95].

When an incomplete executable begins execution, the HP-UX dynamic loader looks at the a.out file to see what libraries the a.out file needs during execution. The dynamic loader at run time loads and maps any required shared libraries into the process's address space. A program calls shared library routines indirectly through a linkage table. The dynamic loader fills the linkage table with the addresses of the routines as the routines are called. Shared libraries are built with position independent code (PIC). PIC makes sharing possible because it contains no absolute virtual addresses; only PC relative addressing is used. This allows PIC to be placed anywhere in a process's address space without addresses having to be relocated [HPa95]. Table 1 provides a comparison of archive vs. shared library attributes for HP-UX.

Archive libraries were selected for the HPLR middleware rather than shared libraries because of the mechanisms provided to modify the libraries in HP-UX. Unlike the `ar` command used to create archive libraries, the `ld` command used to create shared libraries does not provide a mechanism to modify single objects within a library. The entire library must be re-built when a single object is changed. Since neither the source code nor PIC objects for libc are available it is not possible to rebuild the entire libc.sl. If this research was taken further it would be reasonable to pursue opportunities with the dynamic linker and shared libraries. An immediate advantage would be enabling HPLR to be application transparent for a binary executable created with shared libraries which is the default compiler configuration on HP-UX.

TABLE 1. Comparison of Archive and Shared Libraries [HPa95]

Comparing	Archive	Shared
file name suffix	Suffix is .a	Suffix is .sl or .number representing a particular version of the library.
object code	Made from relocatable object code	Made from position-independent object code, created by compiling with the +z or +Z compiler option. Can also be created in assembly language.
creation	Combine object files with the ar command	Combine PIC object files with the ld command.
address binding	Addresses of library subroutines and data are resolved at link time	Addresses of library subroutines are bound at run time. Addresses of data in a.out are bound at link time; addresses of data in shared libraries are bound at run time.
a.out files	Contains all library routines or data (external references) referenced in the program. An a.out file that does not use shared libraries is known as a complete executable	Does not contain library routines; instead, contains a linkage table that is filled in with the addresses of routines and shared library data. An a.out that uses shared libraries is known as an incomplete executable, and is almost always much smaller than a complete executable.
run time	Each program has its own copy of archive library routines	Shared library routines are shared among all processes that use the library.

HPLR Log File Overview

The log files generated with the HPLR method are binary files. This design was leveraged from the log file format used in [Russinovich94] for Mach 3.0 system call monitor. Using a binary file format minimizes the data size and therefore minimizes overhead. Tools were developed to interpret the binary log file. Figure 1 shows the system calls monitored for the dwrite benchmark in text format. The redundant write calls were removed to decrease the size of the log shown.

```
 10069: creat( 4020d1c8 , 640  ) = 3
 10069: close( 3  ) = 0
 10069: open( 4020d1f8 , 1  ) = 3
 10069: write( 3 , 0X4020D228 , 512  ) = 512

# 254 write lines deleted

 10069: write( 3 , 0X402101F8 , 512  ) = 512
 10069: close( 3  ) = 0
 10069: exit( 0  ) = 0
```

FIGURE 1. HPLR log file for dwrite benchmark.

The first number is the process identifier (PID) for the benchmark when it was executed. Next the system call and arguments are list. The last value to the right of the equal sign is the return value. If the return value is -1 (indicating error) then the error value and description is listed as shown in Figure 2.

```
10328: creat( 4020d1a8 , 640  ) = 3
10328: close( 3  ) = 0
10328: open( 4020d1d8 , 1  ) = 3
10328: write( 44 , 0X4020D208 , 512  ) = -1  EBADF   9 -> Bad file number

# 254 write lines deleted

10328: write( 44 , 0X402101D8 , 512  ) = -1  EBADF   9 -> Bad file number
10328: close( 3  ) = 0
10328: exit( 0  ) = 0
```

FIGURE 2. HPLR log file for dwrite benchmark with force errors

The log file in Figure 2 was generated by modifying the dwrite.c benchmark to reference an invalid file descriptor as the first parameter of the `write` call.

HPLR Custom libc.a object files

Table 2 lists the original and added object files to create the custom libc.a for the benchmarks discussed in Section 3.4. The default object code for the system call in libc.a is listed in column two. For some calls multiple object files represents separate code for the system call referenced and the actual system call execution code as described in Section 3.3. Column three lists the middleware enabled objects created for each system call. If the default code did not provide a separate definition for the actual system call then an additional object file was added. The global definition names are the same as the symbol stored in the symbol table in libc.a.

TABLE 2. libc.a and middleware-enabled libc.a symbols and object code

System Call	default object code	middleware object code	global definitions
brk	_brk.o brk.o	spy_t_brk.o	_brk __brk brk
		spy_brk.o	_brk_sys
close	t_close.o	spy_t_close.o	_close close
	close.o		_close_sys
creat	t_creat.o	spy_t_creat.o	_creat creat
	creat.o		_creat_sys
execve	t_execve.o	spy_t_execve.o	_execve execve
	execve.o	spy_execve.o	_execve_sys
exit	exit.o strrchr.o _exit.o	spy_t_exit.o	___exit _atexit atexit exit exit _exit.o
		spy_exit.o	_exit_sys
fork	t_fork.o	spy_t_fork.o	_fork fork
	fork.o		_fork_sys
getpid	getpid.o	spy_t_getpid.o	_getpid getpid
		spy_getpid.o	_getpid_sys
ioctl	t_ioctl.o	spy_t_ioctl.o	_ioctl ioctl

TABLE 2. libc.a and middleware-enabled libc.a symbols and object code

System Call	default object code	middleware object code	global definitions
	ioctl.o		_ioctl_sys
lseek	lseek.o	spy_t_lseek.o	_lseek lseek
		spy_lseek.o	_lseek_sys
log[a]		log.o	log
open	t_open.o	spy_t_open.o	_open open
	open.o		_open_sys
pipe	t_pipe.o	spy_t_pipe.o	_pipe pipe
	pipe.o		_pipe_sys
read	t_read.o	spy_t_read.o	_read read
	read.o		_read_sys
times	times.o	spy_t_times.o	_times times
		spy_times.o	_times_sys
unlink	unlink.o	spy_t_unlink.o	_unlink unlink
		spy_unlink.o	_unlink_sys
write	t_write.o	spy_t_write.o	_write write
	write.o		_write_sys

a. log is the function defined to write the monitored data to a file. Unlike the other definitions in this table, log is not a system call.

System Call Descriptions

brk: used to change dynamically the amount of space allocated for the calling process's data segment.

close: close a file descriptor

creat: create a new file or rewrite an existing one

exit: terminate a process

fork: Creates a new process. The new process (the child process) is an exact copy of the calling process (the parent process).

getpid: Return process ID.

ioctl: Control device

lseek: move read/write file pointer; seek

open: open file for reading or writing

pipe: creates an I/O mechanism called a pipe and returns two file descriptors, fildes[0] and fildes[1]. fildes[0] is opened for reading and fildes[1] is opened for writing.

read: reads a specified number of bytes from a file and placed into a specified buffer.

select: synchronous I/O multiplexing. Examines the file descriptors specified by the bit masks readfds, writefds, and exceptfds.

times: Get process and child process times. Fills a time accounting structure defined in sys/times.h.

unlink: Remove directory entry; delete file.

write: writes a specified number of bytes of data from a buffer to a specified file.

Benchmark Descriptions

bytesort: sorts a 1000 entry buffer of random numbers.

dread: Reads 512 byte chunks of data from random byte offset locations in a_large_file (created by dwrite) and deletes the file when complete.

drynr/dryr: Contains statements of a high-level programming language (C) in a distribution considered representative:

assignments	53%
control statements	32%
procedure, function calls	15%

100 statements are dynamically executed. The program is balanced with respect to the three aspects:

-statement type

-operand type (for simple data types)

-operand access (global, local, parameter, or constant)

The combination of these three aspects is balanced only approximately. The program does not compute anything meaningful, but it is syntactically and semantically correct. drynr = no registers used, dryr = registers used. The program results in a drystone number that can be used to rank computer system performance.

dwrite: Creates a_large_file containing 256 512byte blocks.

fcalla/fcalle: Fcalla assigns a register 50000 times. Fcalle empties a register 50000 times.

fibo: Calculates the fibonacci number. fibonacci is a recursion test.

float: Performs 14 floating point operations 10,000 times

iocall: Tests the speed of Unix system call interface and speed of CPU doing common Unix io system calls. Benchmark is designed to cause only system buffer activity not any physical activity.

iofile: Creates files of random sizes for random reading and writing.

loop: For loop 1,000,000 times doing nothing

pipes: Creates two file descriptors, one for reading and one for writing and creates two processes with `fork`, one to write to the file and the other to read.

scall: Executes the `getpid` system call 25000 times.

sieve: Sets an array of flags to true if the array index is a prime number and false if the array index is not a prime number. The array is size 8191 and iterates 10 times.

Glossary

archive library: A library containing object code for subroutines and data that can be used by programs. Archive libraries are created with the `ar` command and contain one or more object modules. By convention, archive library file names end with .a.[HPa95]

assertions: A fault-tolerant mechanism used to guard an application from incorrect kernel behavior. This is done through checking arguments known to cause incorrect kernel behavior and returning a reasonable error message to the application and by-passing the kernel.

BLE: Branch and link external is an instruction for the PARISC family of processors to make interspace procedure calls. It places the offset of the return point in general register 31 and copies the space ID into space register 0. The return point is the location four bytes beyond the address of the instruction which executes after the branch. [HP94]

libc.a: archive library containing object code for system calls, standard C library routines and standard input/output routines that can be linked to a user program.[HPa95] global definitions: a definition of a procedure, function, or data item that can be accessed by code in another object file. [HPa95]

external references: a reference to a symbol defined outside an object file. [HPa95]

object file: a file containing machine language instructions and data in a form that the linker can use to create an executable program. [HPa95]

symbol name: The name by which a procedure, function, or data item is referred to in an object module. [HPa95]

symbol table: A table, found in object and archive files, which lists the symbols (procedures or data) defined and referenced in the file. Status and offset information for all symbols is maintained in the symbol table. [HPa95]

local definition: A definition of a routine or data that is accessible only within the object file in which it is defined. [HPa95]

shared library: A library, created by the ld command, which contains one or more PIC object modules. Shared library file names end with .sl [HPa95]

dynamic linking: The process of linking an object module with a running program and loading the module into the program's address space.

dynamic loader: Code that attaches a shared library to a program. [HPa95]

ar: create and maintain portable archives and libraries.

Linker (ld): takes one or more object files or libraries as input and combines them to produce a single executable file.

Virtual Memory: abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.

References

[Bach86] M.J. Bach, “The Design of the Unix Operating System”, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Dingman95] C. P. Dingman, J. Marshall, D. P. Siewiorek, “Measuring Robustness of a Fault Tolerant Aerospace System,” *Proc. of the Twenty-fifth Int’l Symposium on Fault Tolerant Computing*, June 1995, pp. 522-527.

[Dingman96] C. P. Dingman, “Robustness Benchmarking”, CMU Ph.D. Thesis, August 1996.

[Elnozahy93] E. N. Elnozahy, “Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication”, Rice Technical Report, Rice COMP TR93-212, October 1993.

[HP94] Hewlett-Packard Company, “PA-RISC 1.1 Architecture and Instruction Set Reference Manual”, Third Edition, February 1994. URL: http://hpcc997.external.hp.com:80/nsa/pa1.1/html/acd-1.html

[HPa95] Hewlett-Packard Company, “HP-UX Reference”, HP LaserROM HP-UX Release 10.0, June1995

[HPb95] Hewlett-Packard Company, “HP-UX Memory Management White Paper”, HP LaserROM HP-UX Release 10.0, June1995

[HPc95] Hewlett-Packard Company, “Programming on HP-UX”, HP Part No. B2355-90652, Hewlett Packard Company, January1995

[Integrated95] Integrated Systems Inc., “pSOS System / 386 Release 2.0 Manual”, March 1995

[Lee95] R.E. Lee, “Middleware on the HP3000”, *Interact*, August 1995,

[Rashid89] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr and R. Sanzi, “Mach: a Foundation for Open Systems”, *Proc. 2nd Workshop Workstation Operating Syst.*, Sept. 27-29, 1989.

[Russinovich94] M.E. Russinovich “Application-Transparent Fault Management”, CMU Ph.D. Thesis, August 1994.

[Silberschatz94] A. Silberschatz, P.B. Galvin, “Operating System Concepts”, Fourth Edition, Addison-Wesley Publishing Company, New York, 1994.