Device driver routines (`leintr()`, `lerint()`, `leread()`, `leget()`) copy the received Ethernet packets from regions of the network interface buffer to *mbuf* chains. `Lerint()` first attaches a profile data buffer to a region of the network interface buffer and sets `svdevent`. That is, an event identifier for a packet is the address of the region. `Leread()` then copies the contents of the region to an *mbuf* chain by calling `leget()`. After copying, `leread()` detaches the profile data buffer from the region and re-attaches it to the *mbuf* chain. `Ethernet_input()` places the *mbuf* chain in the IP packet input queue and posts a software interrupt. This sequence is repeated until all Ethernet packets are processed. Then `interrupt()` restores the last value of `svdevent` and resumes normal execution.

When software interrupt is acknowledged, the network and transport layer routines (`ipintr()`, `udpinput()`, etc.) process *mbuf* chains to assemble, in this case, a UDP packet. `Ipintr()` sets `svdevent` when it removes an *mbuf* chain from the IP packet input queue. When `ipintr()` concatenates two *mbuf* chains, it merges the contents of the profile data buffer for the second *mbuf* chain into those for the first *mbuf* chain. This merging actually chains both the profile data buffers because of their structures (See Figure 4.4). Assembled packets are put into the receive queue of a socket and processes sleeping on that queue are woken up.

When the process waiting for the packet is resumed, it removes the packet from the receive queue of the socket. `Soreceive()` merges the contents of the profile data buffer attached to the packet into the profile data buffer of the running process pointed out by `svdproc`. In this way, resource consumption caused by processing of the packet is attributed to the process receiving the packet.

References

[1] S. L. Graham, P. B. Kessler and M. K. McKusick, “An Execution Profiler for Modular Programs”, Software – Practice and Experience, Vol. 13, 671-685, (1983).

[2] S. L. Graham, P. B. Kessler and M. K. McKusick, “gprof: a Call Graph Execution Profiler”, ACM SIGPLAN Notices 17(6), 120-126, (1982).

[3] R. J. Hall and A. J. Goldberg, “Call Path Profiling of Monotonic Program Resources in UNIX”, 1993 Summer USENIX Technical Conference Proceedings, 1-13, (1993).

[4] M. K. McKusick, “Using gprof to Tune the 4.2BSD Kernel (May 21, 1984)”, Distributed with 4.4BSD UNIX (1993).

[5] M. K. McKusick, “Measuring and Improving the Performance of Berkeley UNIX (DRAFT April 17, 1991)”, Distributed with 4.4BSD UNIX, (1993).

[6] M. Loukides, *System Performance Tuning*, O’Reilly & Associates, Inc., (1990).

[7] A. Eustace and A. Srivastava, “ATOM A Flexible Interface for Building High Performance Program Analysis Tools”, 1995 USENIX Technical Conference Proceedings, 303-314, (1995).

Author Information

Shingo Nishioka is a research scientist at the Advanced Research Laboratory, Hitachi, Ltd. His interests include programming languages and operating systems. He received his B.S., M.S., and Ph.D. degrees from Osaka University. He can be reached at nis@harl.hitachi.co.jp.

Atsuo Kawaguchi is a research scientist at the Advanced Research Laboratory, Hitachi, Ltd. His interests include file systems, memory management system, and microprocessor design. He earned his B.S., M.S., and Ph.D. degrees at Osaka University. He can be reached at atsuo@harl.hitachi.co.jp.

Hiroshi Motoda has been with Hitachi since 1967 and is currently a senior chief research scientist at the Advanced Research Laboratory and heads the AI group. His current research includes machine learning, knowledge acquisition, visual reasoning, information filtering, intelligent user interfaces, and AI-oriented computer architectures. He holds B.S., M.S., and Ph.D. degrees from the University of Tokyo. He was on the board of trustees of the Japan Society of Software Science and Technology and on the board of the Japanese Society for Artificial Intelligence. He was also on the editorial board of Knowledge Acquisition and on the editorial board of IEEE Expert. He can be reached at motoda@harl.hitachi.co.jp.

Alternately, all authors can be contacted via mail:

Advanced Research Laboratory, Hitachi, Ltd.
Hatoyama, Saitama, 350-03 Japan.

Availability

PKPROF and related documents are available at http://www.harl.hitachi.co.jp/~nis.

difficult to add a pkprof facility to an existing operating system. We believe that it is necessary for operating system developers to support pkprof in their operating systems and to consider its implementation during the early stages of their development.

Appendix Some details on asynchronous event profiling

PKPROF functions

Table A.1 summarizes PKPROF functions used for profiling asynchronous event processing. When an interrupt handler allocates a new internal data object, it calls `pkprof_hash()` to attach a PDB to the object. When the handler allocates a new internal data object and discard the old one, it calls `pkprof_pass()` to detach the PDB from the old object and to attach the PDB to the new object. When the handler concatenates two internal data objects, it calls `pkprof_merge()` to merge contents of the PDB of one object to the PDB of the other object. If a process is found to benefit from an internal data object, `pkprof_attribute()` is called to merge contents of the PDB of the object to the profile data buffer for that process. If the process is not being profiled, the contents of the PDB are merged into the profile data buffer for "OTHER" processes.

Profiling disk read

Figure A.1 shows the call flow for the kernel routines of a SCSI disk read operation. The process issues a `read` system call. The kernel routine named `read()` handles this request and starts setting up a disk read data transfer operation. `Bread()` allocates a *buf* structure and a data buffer for the *buf* structure. `Biowait()` calls `tsleep()` on the *buf* structure to wait for data to become available. A SCSI interface eventually interrupts normal execution after DMA transfer from a disk to the DMA buffer. `Interrupt()` is then called. `Interrupt()` saves the last `svdevent` value and sets it to NULL.

`struct PDB *` `pkprof_hash(void *)`	Allocate a PDB and associate it with the argument by storing the PDB into the hash table using the argument as a hash key. Return a pointer to the PDB.
`struct PDB *` `pkprof_lookup(void *)`	If the argument is associated with a PDB, return a pointer to the PDB.
`struct PDB *` `pkprof_pass(void *old, void *new)`	De-associate a PDB with old and re-associate the PDB with new.
`struct PDB *` `pkprof_merge(void *old, void *new)`	`Merge contents of a PDB associated with old into a PDB associated with new.`
`void` `pkprof_attribute(void *, struct proc *)`	Merge contents of a PDB associated with an internal data object into a profile data buffer for a process.

Table A.1: PKPROF functions used for profiling ashynchronous event.

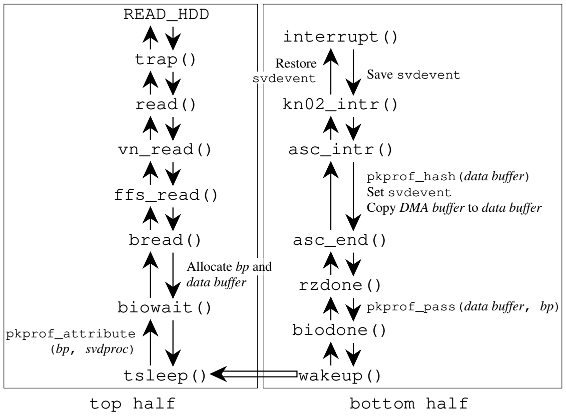

Figure A.1: Call flow of the kernel routines for SCSI disk read operation.

`Asc_intr()` is then called to copy data in the DMA buffer to the data buffer passed from `bread()`. `Asc_intr()` allocates and associates a PDB with the data buffer, sets `svdevent`, and starts copying data from the DMA buffer to the data buffer. `Rzdone()` detaches the PDB from the data buffer and re-attaches it to the *buf* structure. `Biodone()` then wakes up processes sleeping on the *buf* structure.

When a process sleeping on the *buf* structure is resumed, the contents of a PDB associated with the *buf* structure are merged into a profile data buffer for that process. The resources consumed in processing the read operation for that *buf* structure are attributed to the process in this way.

Profiling UDP packet receiving

Figure A.2 shows the call flow for the kernel routines of UDP packet receipt operation. The process issues a `recvfrom` system call. The kernel routine named `recvfrom()` handles the system call. `Tsleep()` is finally called from `sbwait()` to wait for a packet arrival. When Ethernet packets arrive, a network interface interrupts normal execution and `interrupt()` is called. `Interrupt()` saves the last `svdevent` value and sets it to NULL.

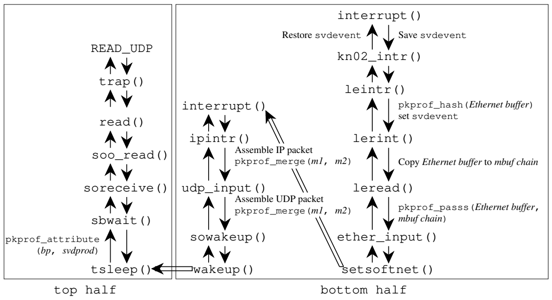

Figure A.2: Call flow of the kernel routines for UDP packet receive opearation.

File size (bytes)	Read throughput (Mbytes/s)				
	Normal kernel	Kernel with KPROF		Kernel with PKPROF	
		Disabled	Enabled	Disabled	Enabled
256K	0.80	0.80	0.78	0.79	0.78
1M	1.06	1.05	1.04	1.05	1.04
4M	1.21	1.21	1.21	1.21	1.19

Table 5.4: Performance impact on file read throughput.

space. These PDBs construct 106 PDB-lists on average, and they accounts for 1 363 968 bytes in total.

Overall performance degradation

Table 5.4 shows the performance impact of PKPROF on file read throughput. It also shows KPROF's impact for comparison. We measured read throughputs of files on a disk for various sizes. We arranged the measurements carefully so that the buffer cache never hit. The results show that both KPROF and PKPROF suffer less than 3% decrease on file read performance.

Table 5.5 shows the performance impact of both PKPROF and KPROF on UDP packet receipt throughput. We used two communication processes for measurement; a receiver running on the platform received UDP packets infinitely and a sender running on another computer sent UDP packets to the receiver via the Ethernet. We determined the maximum throughputs by changing packet transmit rate until the receiver began dropping the packets. The results show that KPROF suffers about a 60% decrease on 64 byte-data UDP packet receive performance and PKPROF about 80%[6].

5.2 Server process

Server processes that work for other user programs instead of the kernel are common on many of today's operating systems. PKPROF is able to show to what extent kernel resources are consumed by such servers, but cannot tell for what process the servers have worked.

Seamless profiling can be extended to deal with such cases. The kernel places a mark for every data packet passed via interprocess communication. When a server process is scheduled, the scheduler knows which data packet has triggered the server. It can identify for which user program the server is working by referring to the mark. As this method may cause degradation in server performance, markers (e.g. address of passed data) and reference methods should be carefully selected.

Packet data size (bytes)	UDP packet read throughput (packets/s)				
	Normal kernel	Kernel with KPROF		Kernel with PKPROF	
		Disabled	Enabled	Disabled	Enabled
64	3270	2080	1320	2240	670
4096	292	292	252	292	149

Table 5.5: Performance impact on UDP packet receive throughput.

5.3 Call-graph support

PKPROF reports on seamless profiling results in the same format as gprof. Current implementation can report on call-graph profiles across the user-kernel boundary passed through via system calls. We have modified the monitor codes of the system call trap handler so that they are able to store the addresses of the caller (in user-space) and the callee (i.e. the trap handler) in the profile data buffer properly. The `gprof` report generator constructs a call-graph across the boundary based on caller-callee information. The call-graph for other services such as page faults needs modification to the `interrupt()`.

5.4 Related tools

Many tools, such as `sar` of System V and various `xxstat` programs of BSD, that report kernel statistics have been used to monitor and tune systems [6]. These tools are also useful to measure the kernel resource consumption of a program if it were possible to control the system environment precisely.

If a target operating system kernel clearly organizes process switching and accesses internal data objects in a consistent manner, pkprof can be implemented in a mechanical way. For example, 4.4BSD-Lite uses *mbuf* structures for network handling in a consistent fashion; similar code patterns appear in many places. In such cases, modification to the kernel to allow pkprof to be implemented also results in embedding the same pattern of codes into many places. ATOM [7] might be useful in such cases.

6. Conclusion

We have proposed a new profiling facility called process-labeled kernel profiling (pkprof). Pkprof measures which resources are used in an operating system kernel for which user program. Seamless profiling, one of pkprof's applications, measures the resource consumption of a program both in user space and kernel space. A programmer is able to understand the impact of a program on kernel CPU usage from the output of seamless profiling. Applying pkprof to a group of processes is useful in understanding system behavior and in improving its overall performance.

Implementation of pkprof heavily depends on the structures of a target operating system. It might be

---

[6] When disabled, PKPROF shows less performance impact than KPROF on 64 byte-data UDP packet receiving. We have no idea of the reason.

Function	No. of instructions		No. of ticks		called
	Original	Added	Original	Added	
cpu_switch()	95	2	373	0	295048
execve()	460	9	179	3	33605
exit1()	266	9	117	5	33571
fork1()	366	5	134	2	33580
idle()	22	2	59459	5	154445
mi_switch()	135	4	83	10	261477
switch_exit()	22	2	3	0	33571
OTHER	222863	–	153975	–	158127999
total	224229	33	214323	25	158973296

(1tick = 15.625msec)

Table 5.2: Results of static analysis and execution time measurements of context switching routines.

Function	No. of instructions		No. of ticks		called
	Original	Added	Original	Added	
interrupt()	120	12	0	0	15119
kn02_intr()	160	4	2	0	15113
lerint()	304	32	14	2	7728
leread()	125	17	1	0	7731
ipintr()	418	98	3	0	7567
udp_input()	467	29	1	0	2577
tcp_input()	1879	13	0	0	14
soreceive()	942	54	18	0	2588
pkprof_a_init()	0	43	0	0	–
pkprof_alloc_buf()	0	80	0	0	–
pkprof_attr_seg()	0	205	0	158	–
pkprof_attribute()	0	211	0	8	33683
pkprof_check_halt()	0	47	0	0	–
pkprof_check_idx()	0	161	0	1	2700
pkprof_free_buf()	0	47	0	0	–
pkprof_hash()	0	122	0	4	18120
pkprof_lookup()	0	65	0	3	31293
pkprof_make_total()	0	209	0	0	–
pkprof_reset()	0	99	0	0	1
OTHER	218299	–	6440	–	1297776
Total	222714	1548	6479	176	1442010

(1tick = 15.625msec)

Table 5.3: Results of static analysis and execution time measurements of kernel routines for UDP packet receiving.

measured under the same conditions as Table 5.1. We modified 8 kernel routines listed in the table to maintain the `svdproc` value. These functions suffer a 2.4% text and 0.04% execution time increase in total. Note that the total execution time for these routines accounts for only a small fraction of the total kernel CPU time, that is, about 0.02%. The process tracking codes, therefore, incur a small overhead on system performance.

Asynchronous event tracking codes need to be embedded to each event handling module. The modification heavily depends on specific kernel implementation. Therefore, the overheads for such codes are difficult to evaluate in a general manner. We selected UDP packet receive processing as an example of event handling in the following discussion.

Table 5.3 shows the results of static analysis and execution time measurements of kernel routines related to Ethernet UDP packet receiving. The execution time for each routine is measured in receiving UDP packets of 4 KByte data. The processing of a UDP packet bigger than 1500 bytes that is received via Ethernet needs to have fragments of the packet collected. The measured CPU time overheads, therefore, includes those for concatenating profile data buffers.

Memory requirements

Both KPROF and PKPROF add codes to the kernel. The text size of the kernel with no profiling for our platform is 775 262 bytes. Sizes with KPROF and PKPROF are 803 022 bytes and 816 142 bytes respectively.

PKPROF allocates a profile data buffer for each process (Figure 4.3) when a process is specified to be profiled. It also allocates the same size for two buffers to store the profile data attributed to the kernel itself and the processes not being profiled. The size required for the buffer depends on the address range of the kernel text. PKPROF allocates 1/2 the size of the address range for the execution count data buffer (caller-callee counts) and about 3/4 for the address sampling data buffer.

The address range size of the sample kernel, for example, is 897 048 bytes. Sample implementation first allocates two 1 112 328 byte buffers for the kernel and not-profiled processes. If three processes are profiled, PKPROF allocates a 3 336 984 byte memory for 3 buffers in addition to the first two buffers.

Recording the profile data for each asynchronous event, on the other hand, needs much less memory. Each asynchronous event is generally processed quickly. That is, fewer kernel routines are called and less sampling is made during event processing than during process life. PKPROF, therefore, allocates a small amount of memory for each event. Since such memory is recycled for process-event attribution, the total amount of memory is limited by the throughput of the system's event processing. The maximum memory theoretically needed for profiling asynchronous events depends on the number of internal data objects that a kernel can allocate simultaneously, the number of functions that participate in each event processing, and the time that each event processing takes.

We again selected UDP packet processing as an example to be able to discuss the PKPROF memory requirements more concretely. PKPROF allocates a statistics array (Figure 4.4) for each *mbuf* chain. Each statistics array is 4096 bytes long and can hold 1020 statistical entries. The actual number of statistical entries used in storing the profile data depends on both the implementation of UDP packet processing and the sampling rate.

Measurements on our platform have shown that, for 4 KByte-data UDP packets received at a maximum processing rate (149 packets/s, See "Overall performance degradation" section), the maximum (average) number of caller-callee entries and current PC entries per *mbuf* list are 533 (94) and 2 (0.2), respectively. The 1020 entries are sufficient to hold these entries. While receiving, on average 127 *mbuf*s and 333 PDBs simultaneously exist in the kernel

```
pkgprof [gprof-options] [options] pid...
            Make profile reports for pid.
pkgprof [gprof-options] [options] path...
            Make profile reports for process of which executable is path.
pkgprof [gprof-options] [options] command-name...
            Make reports for process of which name is command-name.

Options:
      -M [/vmunix]       Specify a kernel file
      -d <dir>           Specify a directory whree profile data files
                         reside.
      -r                 Make profile reports for forked process
                         recursively.
```

Figure 4.6: Command syntax of `pkgprof`.

```
 1: $ pkgmon -x ../bin/cat /x/bigfile >/dev/null
 2:   137 ../bin/cat /x/bigfile
 3: $ pkgprof 137
 4: Created gprof.out.137.cat
 5: $ ls
 6: gmon.out               gmon.out.137.cat       gmon.out.k.total
 7: gmon.out.-k.kernel     gmon.out.137k.cat      gprof.out.137.cat
 8: gmon.out.-k.other      gmon.out.137u.cat
 9: $ rm *
10: $ ps ax | grep nfsiod
11:    63 ??  Ss     0:03.39 nfsiod 4  (nfsiod)
12:    64 ??  I      0:01.47 nfsiod 4  (nfsiod)
13:    65 ??  I      0:00.73 nfsiod 4  (nfsiod)
14:    66 ??  I      0:00.77 nfsiod 4  (nfsiod)
15: $ pkgmon -t 120 -p 63
16: $ pkgprof 63
17: Created gprof.out.63.nfsiod
18: $ ls
19: gmon.out.-k.kernel     gmon.out.63k.nfsiod    gprof.out.63.nfsiod
20: gmon.out.-k.other      gmon.out.63u.nfsiod
21: gmon.out.63.nfsiod     gmon.out.k.total
22: $
```

Figure 4.7: Sample run of profile measurement.

terminates. Reports are generated by running `pkgprof` for the `cat` process (line 3).

Lines 6 through 8 show files created by measurement. `Gmon.out.137k.cat` and `gmon.out.137u.cat` are process 137's profile data files for user-level and kernel-level, respectively. `Gmon.out` is another user-level profile data file; its contents are the same as those for `gmon.137u.cat`. `Gmon.out.137.cat` contains information on measurement such as program name and process identifier; `pkgprof` refers the information to process profile data files. `Gmon.out.-k.kernel` is a profile data file for the kernel's system-wide jobs. `Gmon.out.-k.other` is a kernel-level profile data file for not-profiled processes.

Lines 10 through 21 show another example of measurement. A sever process `nfsiod` is profiled in this example. A process 63 has been specified to be profiled for 120 seconds at line 15. Line 19 through 21 show files created by measurement. Note that file `gmon.out` is not generated by this measurement since the profiled `nfsiod` process is still running after measurement.

5. Discussion

5.1 Overhead

Overheads for pkprof depend on many factors, such as machine architecture, kernel implementation, sampling rate, and event rate. The goal of this section is to discuss the overheads as generally as possible. We will examine KPROF and PKPROF closely and evaluate the overheads for PKPROF using those of KPROF as standards. Since the implementation of KPROF is fairly general and portable, this evaluation should give the reader clues to estimating the overheads of pkprof in her environment.

Module	Kernel	KPROF Added to Kernel	PKPROF Added to KPROF
Monitor (`_mcount()`)			
Machine Instruction	-	145	48
Total tick	-	46411	7611
Sampler (`statclock()`)			
Machine Instruction	128	25	48

`(1tick = 15.625msec)`

Table 5.1: Results of static analysis and execution time measurements of monitor and sampler.

In the following discussion, measurements have been undertaken on DECstation 5000/200 (MIPS R3000, 25MHz clock) with 16MBytes of main memory and a RZ56 600MByte SCSI disk drive.

Execution time overheads

PKPROF adds 4 kinds of codes to KPROF as follows:

- monitor: codes to refer `svdproc` and `svdevent` and to switch the profile data buffer
- sampler: (same as the above)
- process tracking
- event tracking

Table 5.1 show the machine instruction steps of the monitor and the sampler for both KPROF and PKPROF. It also shows the average execution time of the monitor for each call in tick[4]. The time is measured while a test program reads files and writes them to `/dev/null`[5]. Time measurement results reflect the dynamic environments of the platform, such as cache-miss and bus availability as well as the monitor's typical execution traces. The monitor suffers a 33% text and 16% execution time increase in implementing PKPROF. Note that the increase accounts for 3.1% of the total kernel CPU time. The sampler suffers a 31% increase in text.

Table 5.2 shows the results of static analysis and execution time measurements of process tracking codes. The average execution time for each routine is

---

[4] The execution time results are measured by using the sampler. We have collected and classified the sample counts for each PC value.

[5] The test load for measurements is as follows:
Step 1. Read a 256KBytes file and write it to /dev/null.
Step 2. Repeat Step 1 for the same file 15 times.
Step 3. Repeat Step 1 and 2 sixteen times.
Thus the test program reads 16 different 256 KByte-long data files.

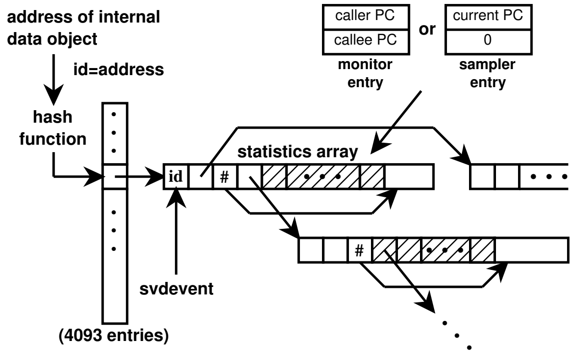

Figure 4.4: Data structures of profile data buffers for asynchronous events.

entries are stored into the array sequentially. The entry can hold a caller-current PC (program counter) pair for the monitor or current PC for the sampler. The sequential structure of the array enables the profiler routines to access contents quickly. Note that `svdevent` actually points out a PDB so that the monitor and sampler can quickly store the profile data in the PDB without looking up the hash table.

The internal data objects are collected and assembled into other objects for some event processing, such as network packet reception. In these cases, we also collect PDBs associated with all the collected objects and aggregate the contents of the PDBs into a PDB associated with the newly created object. The data structures make aggregation efficient since it only needs to link PDBs by pointers.

4.6 Asynchronous event attribution

A process is found to benefit from each asynchronous event when it uses the output of asynchronous event processing. An asynchronous event handler stores internal data objects, e.g. *mbuf* structures, in an appropriate place, e.g. a queue for a socket, and calls `wakeup()` (See Figure 4.2). When a process removes the internal data objects from places, PKPROF reads the contents of PDBs attached to objects and merges them into a profile data buffer for that process. All the resources consumed for internal data objects are consequently attributed to that process.

4.7 Profile utilities

PKPROF provides two utility programs named `pkgmon` and `pkgprof`. `Pkgmon` is used to conduct profiling. `Pkgprof` is a profile report generator.

We modified the `kgmon` program to implement `pkgmon`. The operating system was originally equipped with `kgmon` to control KPROF using the `sysctl()` library routine, offering miscellaneous services to control kernel behavior[3]. The controlled functions include resetting KPROF, starting it, stopping it, and reading the profiled data. We have added new services to the `sysctl()` library routine (and the `__sysctl` system call) and modified the `kgmon` command program so that a user can control PKPROF-specific functionality such as specifying the processes to be tracked and reading process-labeled profile data. Table 4.2 lists the services that have been added to the `sysctl()` library routine. Figure 4.5 summarizes the command syntax for `pkgmon`.

Service Name	Description of parameters
`GPROF_ADDPID`	Add a process to be profiled
`GPROF_PIDMAP`	Array of `PID`s which have been measured
`GPROF_COUNT2 .<PID>`	Array of statistial program counter counts of specified process.
`GPROF_FROMS2 .<PID>`	Array indexed by program counter of call-from points of specified process.
`GPROF_TOS2 .<PID>`	Array of struct tostruct describing destination of calls and their counts of specified process.
`GPROF_WAIT`	Wait unitl: • specified time is elapsed • specified processes exit • all process being profiled exit.

<PID>: Process identifier

Table 4.2: Added `sysctl()` commands.

```
pkgmon -x [options] cmd arg ...
          Execute cmd with arg and profile it.
pkgmon -p [options] pid ...
          Specify processes to be profiled.
pkgmon -a [options]
          Profile all processes.
pkgmon -e [options] path
          Profile all processes specified by path.
pkgmon -D [options]
          Dump contents of profile data buffers in a kernel to files.

Options:
          -r            Profile forked processes in addition to a
                        parent process recursively.
          -h <number>   Profile as much as <number> processes.
          -t <second>   Profile until <second> time elapses.
```

Figure 4.5: Command syntax of `pkgmon`.

`Pkgprof` is a front-end processor of `gprof`. We have made no modification to `gprof`. `Pkgprof` collects profile data files and passes them to `gprof`. It supports seamless profiling by merging the user-level and kernel-level profile data of a process into one profile data file (in `gmon.out` format) and passing it to `gprof`. Figure 4.6 summarizes `pkgprof` command syntax and functionality.

4.8 Example Measurements

Figure 4.7 shows a typical procedure to measure seamless profiles. The first command (line 1) executes a `cat` program and starts profiling it. `Pkgmon` reports the process identifier of the executed program (line 2). When the measured program terminates, `pkgmon` also

---

[3] The library routine actually issues the `__sysctl` system call to control kernel functionality, but the operating system only documents the `sysctl()` specification.

for the device driver. The interrupt handling routine and other related routines eventually finish processing and call `wakeup()` to mark the processes that are waiting for that event as runnable. Internal data objects are often created during processing. These objects are passed from routine to routine, and are eventually stored in a queue or a buffer so that the process can receive the contents of the objects.

We use the address for each internal data object as the identifier of an event. Assigning an identifier to each interrupt does not work properly because one interrupt is often accompanied by multiple events. The reception of multiple packets from a network is a typical example of such cases. Internal data objects are always created for events even in these cases. Thus their addresses can be used to track asynchronous event processing.

We have added a global variable named `svdevent` (stands for "served event") that shows an event identifier. We have embedded codes in all places where new internal data objects are created so that the addresses of objects are recorded and that a profile data buffer is allocated and associated with each object. We have also added codes to places where internal data objects are discarded. These codes maintain a value of `svdevent`. We have also modified the interrupt handlers so that the value of `svdevent` can be properly saved and restored upon interrupts.

4.4 Switching profile data buffer

The monitor and sampler switch the profile data buffer based on the values of `svdproc` and `svdevent`. Table 4.1 summarizes switching of the profile data buffer.

The value of `svdproc` precedes that of `svdevent`. If `svdproc` shows that <process> needs to be profiled, the monitor and sampler store profile data in the profile buffer for that process. `Svdproc` is OTHER when a kernel is serving processes not being profiled; the monitor and sampler aggregate profile data to a special profile buffer for those processes. When both `svdproc` and `svdevent` are NULL, the kernel is serving system-wide jobs such as scheduling; the monitor and sampler store profile data to a profile data buffer for the kernel itself.

svdproc	svdevent	profile buffer
`NULL`	`NULL`	Kernel
`NULL`	<event>	Asynchronus event
<process>	`NULL`	Process
<process>	<event>	Process
`OTHER`	`NULL`	Other processes
`OTHER`	<event>	Other processes

Table 4.1: Switching of profile data buffer.

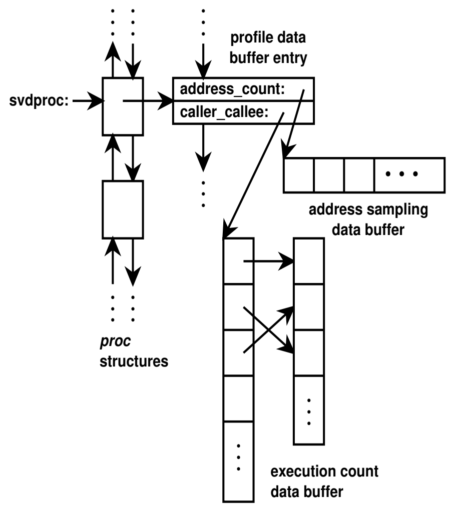

Figure 4.3: Data structures of profile data buffers for processes.

4.5 Structure of profile data buffer

Because a process usually runs over a long period of time, we designed the profile data buffer that is allocated for each process to be space efficient. The data structures for asynchronous events, on the other hand, are designed to be time efficient. This is because each buffer is expected to be short-lived and the buffer contents need to be merged quickly into the profile data buffer for a process.

Figure 4.3 shows how we organized data structures to store the profile data for each process. PKPROF associates each buffer entry with each process through the *proc* structure to provide monitor and sampler access to the buffer entries. `Svdproc` points to a *proc* structure showing the currently served process. Each *proc* structure contains a pointer to the buffer entry for that process. The monitor and the sampler refer to the entry using that particular pointer. The entries themselves construct a singly-linked list so that each entry can be accessed after the termination of the process.

Each buffer entry contains two pointers: a pointer for the address sampling log buffer and a pointer for the execution count log buffer. The structure of both log buffers are identical to those of KPROF and are used to record gprof compatible profile data.

Figure 4.4 shows the data structures storing the profile data for asynchronous event processing. After this, we will refer to each profile data buffer for an asynchronous event as PDB.

Internal data objects, such as the *mbuf* structure, and PDBs are associated in the hash table. Each PDB is composed of a list of the statistics array. Statistical

2. a process accesses an address to which a physical memory page is not assigned at the time (i.e. page fault).
3. a device interrupts the execution of the current process.

When the first event occurs, the kernel starts processing the process's request immediately. If a requisite resource is unavailable at that time, the kernel switches the process to another process. Similar processing also occurs for the second event. The third event, however, starts processing which may be irrelevant to the current process. For example, a process may be interrupted to handle the arrival of a network packet even if this process never accesses the network.

We have added two global variables; `svdproc` indicates what process the kernel is serving and `svdevent` indicates what event the kernel is handling. The kernel updates the value of each variable when process switching and an asynchronous event occur. The monitor and the sampler refer to the variables and store profile data to the proper profile data buffer. Profile data for an asynchronous event handling are merged into the profile data buffer for a process when that process is found to benefit from event handling.

4.2 Served process tracking

The 4.4BSD-Lite kernel maintains a global variable named `curproc`. This points to a *proc* structure that contains the state of the currently running process. Although the name stands for "current process", the value of `curproc` cannot be used to track the "current served process", because its value often remains unchanged while the kernel is serving other processes. For example, the interrupt handler never changes its value. The value of `curproc` is also left untouched while the kernel does system-wide jobs such as process priority calculations.

We have added a new variable named `svdproc` (stands for "served process") that shows a process the kernel is actually serving. Codes that we have embedded into the context switching related kernel routines update the value of `svdproc`.

Figure 4.1 shows the call flow diagram for kernel routines that are responsible for the execution of context switching. Each node shows the kernel routine and each solid arrow indicates the caller-callee relationship between the routines. For example, the figure shows that `exit1()` calls `cpu_exit()`. Each gray arrow shows the transition from the user mode to the kernel mode. The `interrupt()` is invoked by hardware interrupts such as interval timer clock, disk, and network interrupts.

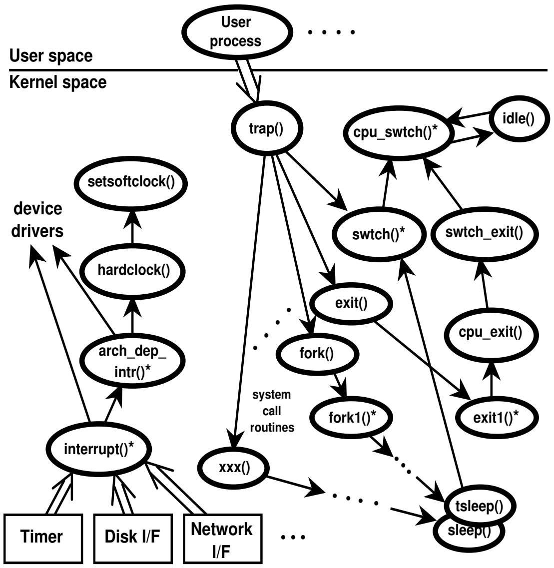

Figure 4.1: Call flow of kernel routines.

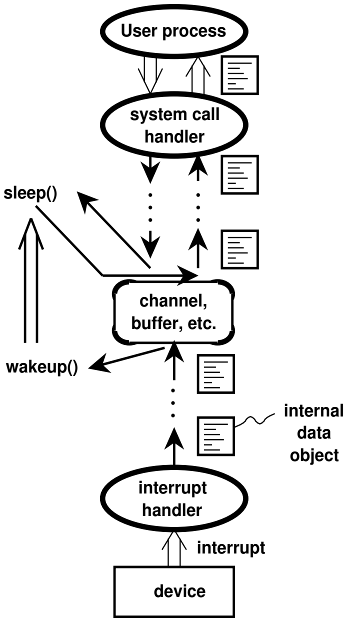

Figure 4.2: Asynchronous event processing.

Only kernel routines marked with '`*`' switch the currently served process. `Switch()` switches the current process to another. `Exit1()` handles process termination. The value of `svdproc` is set to NULL when the process terminates, which means no process is served by the kernel. `Interrupt()` also changes `svdproc` to NULL because it invokes the kernel's interrupt handling activities.

4.3 Asynchronous event tracking

Figure 4.2 depicts a generic view of asynchronous event processing for the 4.4BSD-Lite kernel. Device interrupt is caught by the lowest interrupt handler. It then invokes an appropriate interrupt handling routine asynchronous event is found, the contents of the data buffer for that event can be merged into the profile data buffer for the found process. The buffer for the asynchronous event, then, can be de-allocated and reused for another event. Quick allocation and reuse of the buffers are the keys to making asynchronous event attribution more practical.

3.3 Implementing pkprof

Implementation of a pkprof facility, unlike conventional kernel profiles, heavily depends on the structures of a target operating system. It requires higher-level knowledge of the target such as

- events caused by a process,
- handling of synchronous events,
- location and manner of process switching,
- the way asynchronous events are caught and processed,
- data structures used to process events, and so on.

The part in pkprof concerned with synchronous events can be implemented on top of a conventional kernel profiling facility[4][5]. Since served process tracking and profile log buffer management are functionally independent of a conventional profiler, we can add both functions without modifying the existing profiler. Once the functions are added, we can modify existing profiler routines so that they refer to the currently served process and record resource consumption properly. Of course, we need to adapt existing utilities to the new profile facility.

Implementation of the other part, i.e. the attribution of asynchronous events, needs much more modification of the target operating system. It generally includes:

- enumerating places where asynchronous events are caught and handled,
- embedding codes into places that assign identifiers and profile data buffers to events,
- embedding codes, where appropriate, to track asynchronous event handling, and
- embedding codes to resolve process-event correspondence and to merge event profile data into process profile data.

Note that asynchronous event processing occupies a relatively small fraction of kernel CPU usage in some environments. Implementing only the attribution of synchronous events might prove adequate in such cases.

4. Implementation

This section describes sample implementation of pkprof on a 4.4BSD-Lite UNIX operating system. After this, we will refer to sample implementation as PKPROF. Although PKPROF is experimental and incomplete, it produces the least functionality that is needed to evaluate pkprof. Many techniques used for sample implementation, such as monitor, sampler, and profile data management, are independent of 4.4BSD-Lite kernel architecture. They should be applied to the implementation of pkprof on other operating system kernels. Descriptions of the process switch and asynchronous event tracking, however, depend heavily on the design of the 4.4BSD-Lite kernel.

4.1 Overview

The 4.4BSD-Lite kernel offers a configurable kernel profiling facility called KPROF. KPROF measures:

- how often each kernel routine is called, and
- how much time is spent executing each kernel routine.

The former is implemented by inserting monitoring codes (“monitor”) in the prologue for each kernel routine so that the monitor is executed during every invocation of the routine. The latter is actually measured by sampling the address at which the CPU is executing an instruction. Execution times for kernel routines are inferred from the distribution of the samples within the total execution time of the kernel. This implementation scheme used by KPROF is the same scheme used by gprof.

PKPROF is an extension of KPROF; the monitor and sampler scheme is basically the same as that of gprof. We have added and modified KPROF codes so that the monitor and sampler are able to classify the profile data, and store them in the per-process and per-event profile data buffers properly. PKPROF stores profile data into buffers in the kernel memory space.

Major issues on implementation include:

- where the kernel switches one running process to another,
- how asynchronous events are tagged and tracked,
- how the monitor and sampler keep track of process switching and event processing,
- how profile data buffers are allocated and organized, and
- how measurements are controlled and how profile data is accessed from outside the kernel.

Limiting the impact of performance was one of the key criteria in designing the mechanism and data structures of PKPROF (Section 5.1 discusses overhead).

The 4.4BSD-Lite operating system enters into the kernel mode when one of the following events occur:

1. a process issues a system call.

`index[8]` for example, that the process 137's `read()` function ran for 30.2% of the total CPU time during measurement (`% time` column). It used 0.00 seconds within `137u:read()` itself (`self`), and the descendants of `137u:read()`, i.e. `137k:read()` in this case, needed 2.86 seconds to execute in CPU time. Programmers can grasp the impact on the program of kernel CPU usage from the listing, as well as the CPU time usage of the program itself. They may be able to lower the total CPU time usage by changing the program algorithm, the system calls it issues, or their arguments.

Pkprof is applicable to a group of processes. Programmers can examine the kernel usage and impact on system performance of processes as well as each single process. They may be able to improve the overall performance of the system by carefully examining the listings, finding actual performance bottlenecks, and fixing them.

Applying pkprof to a group of processes is particularly useful in programming based on a client-server scheme. To improve performance in this type of scheme it is not sufficient to profile only the client program since a great deal of work for a job is done by the server. Profiling the server may often be sufficient to improve performance, but applying pkprof to the server and clients may help programmers dramatically improve performance through: changing the scheme itself, changing shared work between the server and clients, changing communication methods, or changing the protocols used for communication.

The pkprof facility offers the following utilities to programmers:

- turns profiler on and off,
- specifies processes the profiler monitors,
- specifies resources to profile,
- reports on measurements.

3. Basic structure

Pkprof monitors the resource consumption in a kernel, attributes consumption to the process benefiting from it, and records consumption with attribution. Pkprof uses the same techniques to monitor resource consumption as conventional profiling tools; it uses embedding codes and sampling techniques. Pkprof switches a profile data buffer when the kernel switches a process or serves asynchronous events (interrupts). It tries to determine what process will later benefit from each asynchronous event.

3.1 Attributing resource consumption

Sources of resource consumption in a modern, multi-tasking operating system kernel can be classified into the following two types of events:

- events caused by a process running, and
- events caused by a device.

The former occur synchronously in respect to the execution of the process. The process asks for kernel services explicitly (e.g. system calls) or implicitly (e.g. page faults) through the events. Therefore, consumption attribution is performed by keeping track of process switching in the kernel. The process running currently is the one being served by the kernel. The latter, however, occurs asynchronously and often has nothing to do with the running process.

The asynchronous event is usually only partially processed. Partial processing results are stored in a temporal buffer, and the remaining processing is performed when the process makes a request that requires response from the processing results. Therefore, resource consumption attribution caused by the event must be deferred until the process that benefits from the event is identified.

Assigning a unique identifier to every asynchronous event is one technique to handle the situation above. The identifier serves as a classification tag and records the resource consumed by processing of the asynchronous event. Once a process benefits from the tagged resource, all the resource consumption recorded with the same tag is attributed to that process.

The address of a data buffer allocated to handle an asynchronous event can be used as such an identifier. The buffer is usually passed from routine to routine during partial processing. The address, therefore, can be used to tag the resource consumed by processing. Note that the asynchronous event handling routines may allocate another buffer and use this during partial processing. The address of a newly allocated buffer should also be treated as another identifier in such cases.

3.2 Recording profile data

The pkprof logically allocates a profile log buffer for every profiled process. It also allocates a buffer for every asynchronous event. Access routines are provided for profiler routines so that they can efficiently record resource consumption in the buffers. It is preferable to construct buffer structures dynamically, and on demand because the number of processes to be profiled is not known in advance. Static allocation of buffers may restrict profile measurements and impose a performance penalty while the profiling facility is turned off.

It should be noted that profile data buffers for asynchronous events are allocated and de-allocated much more frequently than those for profiled processes. When a process that benefits from an analyze data.

Pkprof, in conjunction with a user level profiler, enables a programmer to understand a program's behavior in both the user and kernel space. It reveals hidden system activities that the programmer is unaware of. Such information is extremely beneficial to both the application program and operating system development.

2. Pkprof: Process-labeled kernel profiling

An operating system kernel offers a set of system calls to provide services to user programs. The services are implemented as kernel routines that run in the kernel space. A user program triggers execution of the routines by issuing a system call. Resources consumed by the execution, therefore, can be attributed to that program.

A kernel handles external asynchronous events. Resources consumed to handle such events are also attributed to programs benefiting from consumption. For example, when a network packet arrives, network interface hardware interrupts the normal flow of program execution and invokes the kernel's interrupt handling routines. The interrupt handling routines copy the received packet to a network data buffer and the system then resumes normal execution. The received packet is eventually read by a user program. Consequently, resources consumed by packet reception are attributed to that program when it turns out to have benefited from the reception.

A process-labeled kernel profiler monitors kernel activities and reports where and for what process (program) a kernel consumes resources such as CPU time. In other words, it measures the kernel resources consumed by:

- processes that request kernel services, and
- asynchronous events that are later attributed to processes.

It then reports the extent to which each process benefits from the use of these resources.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 65.7     62.24     62.24      595   104.61   104.61  137k:idle [4]
  7.5     69.35      7.11     5930     1.20     1.20  137k:in_cksum [7]
  4.6     73.68      4.33     6622     0.65     0.65  137k:we_bcopy_in [11]
  4.3     77.74      4.06                             137k:_mcount (1656)
  2.8     80.36      2.62     2923     0.90     0.90  137k:cpcPutc [17]
  2.5     82.76      2.40    30766     0.08     0.08  137k:ovbcopy [22]
  2.2     84.80      2.04      190    10.74    10.74  137k:blkclr [25]
.....
```

(a): Example listing of kernel flat profiling

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 24.2      2.62      2.62     2620     1.00     1.00  137k:in_cksum [9]
 13.7      4.10      1.48    12864     0.12     0.12  137k:ovbcopy [15]
 12.7      5.48      1.38                             137k:_mcount (1656)
 12.5      6.84      1.36     3094     0.44     0.44  137k:we_bcopy_in [16]
  5.0      7.38      0.54    62606     0.01     0.01  137k:_splx [25]
  2.6      8.14      0.28     7877     0.04     0.07  137k:malloc <cycle 1> [24]
  2.4      8.40      0.26     8888     0.03     0.04  137k:free [36]
.....
```

(b): CPU time usage of the kernel for the `cat` program.

Figure 2.1: Example report of pkprof.

```
                               called/total       parents
index  %time    self descendents  called+self    name           index
                               called/total       children

.....

                0.00        2.86      65/65          137u:raw_cat [7]
[8]      30.2   0.00        2.86      65         137u:read [8]
                0.00        2.86      65/65          137k:read [9]

-----------------------------------------------

                0.00        2.86      65/65          137u:read [8]
[9]      30.2   0.00        2.86      65         137k:read [9]
                0.01        2.85      65/65          137k:vn_read [10]

-----------------------------------------------

                0.01        2.85      65/65          137k:read [9]
[10]     30.2   0.01        2.85      65         137k:vn_read [10]
                0.00        2.84      65/76          137k:nfs_read [2]
                0.00        0.02      65/163         137k:lease_check [90]
                0.00        0.00      65/105         137k:nfs_lock [282]
                0.00        0.00      65/101         137k:nfs_unlock [283]
```

Figure 2.2: Example listing of seamless profiling.

Figure 2.1 shows a report on process-labeled kernel profiling. Figure 2.1(a) shows flat profiling results of a kernel running over a period of time. Each line represents how much time is spent by the kernel routine. `we_bcopy_in()`, for example, was called 6622 times (`calls` column) and ran for 4.33 seconds (`self seconds`); that accounts for 4.6% of the total time that elapsed in the kernel (`% time`) during measurement. Each call took 0.65 ms (`self ms/call`) within `we_bcopy_in()` itself and took 0.65 ms (`total ms/call`) for itself and its descendant functions on average. Figure 2.1(b) shows the same information with respect to a process executing a `cat` program over the same period. That is, it shows how much CPU time[2] is consumed by each kernel routine in processing `cat`'s requests. A process identifier and symbol '`k`' are prefixed to the name for each function, e.g. `137k:in_cksum[9]`. The '`137`' denotes the process identifier of the cat process and the '`k`' denotes that the function is the kernel's. We can see that some kernel routines such as `ovbcopy()` are mainly executed for the `cat` process.

A typical pkprof application is *seamless profiling*, which simultaneously measures the resource consumption of a program both in the user and kernel space. A listing of seamless profiling is shown in Figure 2.2. The listing is shown in the same format that `gprof`[2] generates except that the process identifier and symbols '`k`' or '`u`' have been prefixed to the name of each function, e.g. `137k:read[9]`. The '`137`' denotes the process identifier and '`k`' denotes that the function is the kernel's. Symbol '`u`' is shown if the function is the user program's. The listing shows,

---

[2] It can be said that a kernel profiler measures the elapsed time for each kernel routine basically, because the profiler usually measures how much time has been spent in the idle routine as well as other kernel routines. Attributing the idle routine to processes, in general, is a very difficult task. We have decided to attribute the idle routine only to the kernel itself. As kernel profiling results in respect to a process, it can be said to deal with the CPU time usage of kernel routines invoked by that particular process.

Process-labeled kernel profiling: a new facility to profile system activities

Shingo Nishioka, Atsuo Kawaguchi, and Hiroshi Motoda
Advanced Research Laboratory, Hitachi Ltd.

Abstract

Profiling tools that empirically measure the resource usage of a program have been widely used in program development. These tools have traditionally focused on the behavior of the target program. The target program, however, actually performs its job in collaboration with other programs, such as servers and an operating system kernel, in a modern system environment. Process-labeled kernel profiling is a novel facility that measures and attributes the kernel resource consumption of programs benefiting from it. This facility, in conjunction with a conventional profiler, enables a programmer to grasp the resource consumption of programs from an overall system point of view. Using this information, the programmer is better able to reduce overall resource consumption.

1. Introduction

Profiling is one of a number of important techniques used to improve the performance of a complex program. Many tools for profiling, such as prof[1], gprof[2], and cprof[3], have been developed and widely used in application program development. A programmer uses these tools to measure the resource usage (e.g. CPU time) of a program empirically, and to analyze where and why it consumes resources during execution. The measurement results provide clues to finding critical portions of the program which affect its performance, and they also provide evidence that the programmer's code modifications actually improve program performance.

A typical example of complex programs is an operating system kernel. Profiling is useful not only in improving user programs, but also in improving and tuning kernel codes. By empirically measuring resource usage in the kernel over a sufficiently long period of time, a kernel developer is able to locate problems affecting the overall performance of a system. Some problems may need kernel code modification, while others can be eased by changing kernel parameters[4].

User program profiling and kernel profiling are used independently since the kernel and user program are separate. One of the kernel's important roles is offering a well-defined, abstract, and convenient program interface that hides the system's internal operations. Consequently, programmers are able to develop a portable, clearly organized application program without any special knowledge of these internal operations. Moreover, expert programmers understands these operations better than novices, and develop programs that access the program interface in a manner that can be efficiently processed by the kernel. Such implicit optimization often brings about better performance than expected. Providing kernel profiling data as well as user program profiling results should help expert programmers to write more efficient codes from an overall system point of view.

Recent operating systems have relied on many user-level programs (servers) to provide system services. The micro-kernel architecture is a typical example of such an approach. Providing both user programs and kernel profiling data simultaneously is extremely useful in the development operating systems to establish system structures, define server interfaces, write codes, tune prototypes, and so on.

Process-labeled kernel profiling (pkprof) is a novel profiling facility that measures which resources are used in an operating system kernel by which user program (process[1]). It keeps track of which process a kernel is serving, and labels particular resource consumption for a particular process that is to benefit by that consumption. Utility tools are provided to control measurements, to manage profile data, and to

---

[1] We use the word 'process' to represent an execution thread that a kernel identifies as an object of system resource allotment. The UNIX process created by the `fork` system call is such an object.

The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

Process Labeled Kernel Profiling:
A New Facility to Profile System Activities

Shingo Nishioka, Atsuo Kawaguchi, and Hiroshi Motoda
Hitachi Advanced Research Laboratory