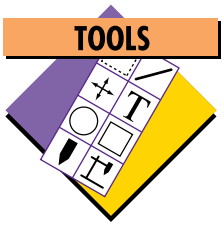
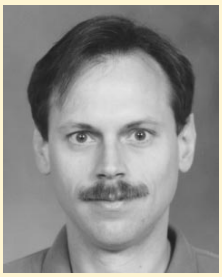


utld— A Trace-based Performance Tool

By Bret R. Olszewski and Jim Van Fleet

Trace-based analysis provides an excellent means of analyzing the behavior of an AIX system. The utld tool provides a convenient way to analyze traces. It is particularly effective for identifying problems associated with high kernel CPU usage. This article describes the utld performance tool used in AIX development, now available via the Internet.


Bret R. Olszewski


Jim Van Fleet

System performance analysis is dependent on tools—tools that do time-based profiling (`tprof`), tools that monitor system resources over time (`iostat`, `vmstat`, `sar`), tools that evaluate instantaneous resource consumption (`svmon`), and event-based tools. The new `utld` performance tool provides valuable information for analyzing traces in AIX.

Trace is a particularly useful built-in instrumentation capability in AIX. Trace contains two parts: hook entries and buffer management. Hook entries are macros included in source code that selectively insert events into a trace buffer.[1] Trace events, which are usually time-stamped, have fixed information fields. When trace is on, the selected trace events are placed into a trace buffer in the order they are generated.

The trace buffer either collects events until the buffer is filled or double buffers the events. When the buffer fills, trace writes them to disk or to a trace reader application such as `tprof`.[1] Because trace adds significant system overhead in path length, disk space, and memory consumption, it does not run continuously; only the root user can enable it. Since trace is invasive, it should be used selectively to address problems. It has proven useful for identifying functional problems, such as application problems and performance analysis.

Using AIX Trace

This section contains recommendations on using AIX trace, but for definitive documentation on trace, consult the standard AIX documentation. To invoke trace, use a set of parameters such as the following:

```
trace -a -d -f -T 8000000 -L 8000000 -o
  ./trace.out
```

The classic time-versus-resource trade-off exists with trace. On a fast, busy system with all trace hooks collected (the default), trace will produce megabytes of trace output for each second of real time. This requires focusing on the phenomenon that will be studied. In the following example, the trace facility is started, detached (`-a`), and deferred (`-d`); trace runs in the background, but it is not yet collecting trace hooks.

Trace will collect eight million bytes of trace data (`-T`) and stop collecting trace data when the buffer fills (`-f`). The system must have sufficient memory for the eight million bytes to be dedicated to trace; otherwise, the trace command will fail or the workload will be perturbed (perhaps by excessive paging). The maximum length of the trace output file is eight million bytes (`-L`); the output is written to the file `./trace.out` (`-o`). When deferred trace collection is used, the `trcon` command starts trace and the `trcoff` command ends trace collection.

[1] Waters, Frank. *AIX Performance Tuning.* Englewood Cliffs, N.J.: Prentice Hall 1996. p. 214.

The following shows how to trace the program `dummy1`:

```
trace -a -d -f -T 8000000 -L 8000000 -o
   ./trace.out
trcon; ./dummy1; trcoff
```

The `-f` flag indicates that the buffer is filled and trace will end. If this happens, the `trcoff` command may fail, and an error message will appear stating that trace is not currently running.

If the workload is more complex or running in the background (for example, a database engine), the same basic technique can be used. When the workload is exhibiting interesting behavior, the trace can be started as follows:

```
trace -a -d -f -T 8000000 -L 8000000 -o
   ./trace.out
trcon; sleep 60; trcoff
```

If the trace was started with the `-f` flag, it is unlikely that 60 seconds of trace will be collected (`sleep 60`); instead, we will get one full trace buffer that will generally contain events corresponding to less than the requested time. Since trace adds considerable path length in system time, using trace will distort the user/system time mix of the system.

The trace data is collected in a binary file that the `trcrpt` program can translate into a readable file. The example in Figure 1 collects a trace of the `find` command and generates a readable view of the trace via `trcrpt`.

The `trcrpt` output includes ID, the unique identifier of the trace hook, the elapsed time in the trace, the delta milliseconds between adjacent trace hooks, and some hook-specific text. The `trcrpt` program formats trace hooks via rules found in the file `/etc/trcfmt`.

How to Get utld

The `utld` tool can be obtained via the Internet with normal browsers by opening http://www.software.ibm.com/download. From there, select Free Software Packages. This directory contains `utld` (an AIX performance tool) as an `installp` image.

The `utld` tool is currently provided free of charge; however, it also comes without the warranty or support provided for products.

State-full Analysis

Trace events are time-stamped, which enables the trace to paint a complete picture of system behavior. The most interesting events are bounded by hooks indicating the beginning and end of activities. For example, when a thread is dispatched, a trace hook appears; then trace hooks appear when system calls enter and exit the system.

From this we can deduce that the running process is responsible for the system calls while it is dispatched. Similarly, we can see that a running thread is preempted by an interrupt, so the CPU consumption of the interrupt handler can be accounted separately from the thread that started the time slice. Although not technically difficult, it would be tedious to write analysis programs to parse the trace. For that reason, we provide `utld` to summarize system behavior with special emphasis on CPU consumption.

```
trace -a -d -T 2000000 -L 6000000 -f -o ./trace.out
trcon; find . -type d; trcoff
trcrpt ./trace.out | pg

ID    ELAPSED_SEC    DELTA_MSEC    APPL      SYSCALL  KERNEL  INTERRUPT

001   0.000000000    0.000000      TRACE     ON       channel 0
                                                      Fri Apr  5 08:54:29 1996
101   0.000165888    0.165888      close LR = 10003F0C
12E   0.000178432    0.012544      close fd=6
104   0.000280320    0.101888      return from close [114 usec]
100   0.000331776    0.051456      DATA ACCESS PAGE FAULT
1B2   0.000357888    0.026112      VMM pagefault:    V.S=01F5.10C6
                                                working_storage
1B0   0.000546048    0.188160      VMM page assign:          V.S=01F5.10C6 ppage=05B5
```

Figure 1. Output of trcrpt

Using utld

Three steps are required to use `utld`. First, collect a trace. Second, use the `trcnm` tool to collect the names of the system. This tool builds a load map of the contents of the AIX kernel so the trace can identify items such as device drivers and kernel extensions. Typically, a command such as `trcnm > names.out` can collect names. Another useful command is `lsdev -C adapter`, which provides an explanation of each adapter driver in your system. This can be helpful in identifying device driver names that map to adapters.

The last step is to "unwrap" the trace. Since the trace is double-buffer collected, the log file must be put in the correct order. Since this step requires two copies of the trace file, be sure that sufficient disk space is available. The `trcrpt` command shown below unwraps the trace:

```
trcrpt -r trace.original > trace.unwrapped
```

Now, the `utld` command can be invoked as follows:

```
utld -I trace.unwrapped -n names.out >
   utld.out
```

After `utld` is complete, do not be surprised if the `utld.out` file is quite large.

CPU Utilization

The first section of `utld` output (see Figure 2) summarizes CPU consumption during the trace period. The CPU consumption has six categories.

Processing Total (msecs)	Percent Total Time	Percent Busy Time	Processing Category
43337.935	49.529	49.531	Application
34317.186	39.220	39.221	Kernel
5203.835	5.947	5.947	FLIH
3103.539	3.547	3.547	SLIH
1534.118	1.753	1.753	Dispatch
87496.612	99.997	100.000	CPU(s) Busy Time
2.911	0.003		WAIT
87499.523	100.000		Total

Total number of process dispatches = 34734
Average time between same process dispatch = 54.738713 msec
Average process to processor affinity = 0.454131

Figure 2. Processor utilization system summary

User: Total time spent running user programs and commands.

Kernel: Time spent in the operating system while executing system calls. For example, when a program reads data from disk, the disk I/O is handled via a system call.

First-Level Interrupt Handler (FLIH): Total time spent by the system in first-level interrupt handlers. This is time spent in the AIX kernel dealing with interrupts, both external (such as I/O) and internal (such as page faults). To maintain a high level of responsiveness, I/O interrupt FLIHs execute quickly. Although page fault FLIHs take much longer to execute, they reduce their interrupt priority levels so they can be preempted by higher-priority interrupts.

Second-Level Interrupt Handler (SLIH): Total amount of time spent in second-level interrupt handlers. Again, this is time spent in the AIX kernel, also dealing with interrupts. Second-level interrupt handlers can execute considerably longer than first-level interrupt handlers, so they occasionally represent a significant amount of CPU utilization.

Dispatch: Time that the system spends dispatching processes. Although the measures of this section are not as precise as the others, they still provide good insight into dispatching overhead.

Idle: Time that the system was idle.

Note that `utld` accounts for CPU utilization in six categories; four (Kernel, FLIH, SLIH, and Dispatch) represent system time. More commonly used tools, such as `sar`, `iostat`, and `vmstat` report total time spent in the system, but `utld` enables further analysis of the sources of system time CPU consumption by category.

CPU Utilization on SMP

For a trace collected on an SMP system, the CPU breakdown includes a weighted average of all processors, as well as information for each processor. On SMP, Figure 3 also details the number of dispatches and a measure of the processor affinity of threads, the number of times a thread was redispatched on the same processor on which it was last dispatched.

Misinterpreting Trace Data

There are several ways `utld` can be fooled into misinterpreting trace data. A frequent case is the accounting of kernel time by tracking system call entry and exit. If a thread has entered the kernel before the trace begins, its time may be improperly accounted to user time. This happens with

Network File System (NFS) daemons because they execute in the AIX kernel and never return to user code; therefore, `utld` does not account them correctly.

Another example is system calls that do not return, such as `sigreturn`, which is special cased in `utld` to account a small amount of time to kernel. Compare `utld` output to other tools such as `sar`, `iostat`, and `vmstat` if `utld` output is suspect.

Wait Summary

The next section summarizes idle time per processor during the trace. It includes statistics about the number of times each processor was idle as well as the minimum, average, and maximum durations of idle time. This section is typically not interesting for performance analysis.

Application and Kernel Summary (Per Thread/Process)

This section categorizes CPU consumption—user and kernel—by individual threads. There is an entry for each thread that was dispatched during the traces, as well as the thread's process name. The `ps` command produces similar information.

Figure 4 shows individual thread data from the application and kernel summary output.

Application and Kernel Summary

This section, although similar to the previous one, has thread CPU consumption rolled up by process name—all processes with the same name are summarized in one entry. For example, if 100 instances of the program `awk` are dispatched in the trace, the total cost of running all instances of `awk` is represented. This information is useful if several workloads, identified by unique names, are present on the same system. For example, if DB2® and Lotus Notes® were running on the same system, the percentage of CPU resource consumed by each in the trace can be identified.

Figure 5 shows example application and kernel summary output.

Processing Total (msecs)	Percent Total Time	Percent Busy Time	Processing Category
5439.218	49.730	49.730	Application
4258.144	38.932	38.932	Kernel
1051.193	9.611	9.611	Interrupt
188.885	1.727	1.727	Dispatch
10937.440	100.000	100.000	CPU Busy Time
0.000	0.000		WAIT
10937.440	100.000		Total

Total number of process dispatches = 4287

Figure 3. Processor #2 summary for an SMP

Kernel (System Call)

The Kernel (system call) section summary is generally the most useful if the workload is CPU bound and contains a high percentage of time (>40%). It summarizes total kernel time CPU consumption by system call.

Since interrupt handlers (FLIH and SLIH) and Dispatch do not often contribute significantly to CPU consumption, system calls are usually the villains for high system time workloads. If the workload contains a high percentage of time in the operating system, this report usually indicates the activity that causes the system time.

Figure 6 shows example kernel (system call) output.

Processing Total (msecs)			Percent of Total Processing Time			Process Name
Combined	Application	Kernel	Combined	Application	Kernel	(process id/thread id)
1333.238	1333.238	0.000	1.524	1.524	0.000	kproc (3096 3355)
1269.453	1269.453	0.000	1.451	1.451	0.000	kproc (3096 3613)
1244.047	1244.047	0.000	1.422	1.422	0.000	kproc (3096 4129)
906.621	0.033	906.588	1.036	0.000	1.036	syncd (4896 5673)
	* * *					
207.807	108.782	99.025	0.237	0.124	0.113	ora7_srv6.1 (96548 367917)
207.657	207.657	0.000	0.237	0.237	0.000	swapper (0 3)
207.204	75.739	131.465	0.237	0.087	0.150	ora7_srv6.1 (342492 342501)
205.972	95.343	110.629	0.235	0.109	0.126	oracle (100388 100397)

Figure 4. Individual thread data excerpt

Processing Total (msecs) Combined	Application	Kernel	Percent of Total Processing Time Combined	Application	Kernel	Process Name (process id/thread id)
4586.477	4586.477	0.000	5.242	5.242	0.000	kproc (7)
906.621	0.033	906.588	1.036	0.000	1.036	syncd (1)
40091.177	21763.985	18327.192	45.819	24.873	20.945	oracle (221)
30617.885	16143.846	14474.040	34.992	18.450	16.542	ora7_srv6.1 (198)
207.657	207.657	0.000	0.237	0.237	0.000	swapper (1)
1224.767	624.459	600.308	1.400	0.714	0.686	bisam_srv6.1 (75)
12.802	10.412	2.390	0.015	0.012	0.003	cron (1)
7.086	0.884	6.202	0.008	0.001	0.007	shmtimer6.1 (1)
0.650	0.184	0.466	0.001	0.000	0.001	trace (1)

Total number of processes = 506

Figure 5. Application and kernel summary data

Processing Total (msecs)	Percent Process Time	Count	Path in msecs Minimum	Average	Maximum	System Call
15461.009	17.670	000025225	0.042	0.613	7.556	kreadv
10177.104	11.631	000025135	0.058	0.405	5.577	kwritev
2180.326	2.492	000036739	0.014	0.059	0.309	kioctl
	* * *					
0.000	0.000	000000001	0.000	0.000	0.000	sigreturn
34317.186	39.220					

Figure 6. Kernel (system call) excerpt

FLIH

FLIH shows CPU time used by first-level interrupt handlers, including instruction access page fault, data access page fault, I/O interrupt, decrementer, level 50, and floating-point unavailable.

Instruction access page faults are caused by attempting to fetch an instruction to execute when the page containing the instruction is not resident in memory. Instruction page faults are often resolved by a disk I/O to bring the page into memory. Similarly, a data access page fault is caused by a load or store to data where the data page is not resident in memory. Data faults can be resolved by creating a new page (demand zero) or by reading static data from disk (vmapped fault).

Another way to view page fault activity is through `vmstat`, particularly `vmstat -s` that details fault activity since the system was booted. Interrupts caused by external devices, such as disks and network adapters, cause I/O interrupts. This interrupt handler identifies the device causing the interrupt and calls the appropriate device driver.

The decrementer interrupt is the system heartbeat. On a uniprocessor system, the processor is interrupted 100 times per second for clock management and process dispatching. On an SMP system, each processor is interrupted 100 times per second. Normally, 1% to 2% of total system time is consumed by this interrupt because it does considerable housekeeping work.

The level 50 software interrupt typically serializes communications adapter drivers. The floating-point unavailable interrupt results from AIX's lazy save of floating-point registers. On context switch, the floating-point registers are not saved. When another process attempts to do a floating-point operation, an interrupt occurs to save the floating-point registers for the last thread and restore the registers for the current thread.

The lazy register save is typically good for floating-point applications, because frequently only one job or a few large jobs are executing. Therefore, the floating-point registers often do not need to be saved and restored across several context switches.

Processing Total (msecs)	Percent Process Time	Count	Path in msecs Minimum	Average	Maximum	SLIH Type
2991.944	3.419	000006486	0.027	0.461	6.276	entdd
76.205	0.087	000000256	0.113	0.298	1.407	sdpin
35.389	0.040	000000129	0.133	0.274	0.513	ascsiddpin
3103.539	3.547	000006871				

Figure 7. SLIH summary

SLIH

SLIH shows CPU time used by second-level interrupt handlers, such as device drivers for disks, communications adapters, terminal equipment, as well as others. Some drivers, particularly communications adapters, may spend considerable time in SLIHs. Figure 7 shows sample SLIH output.

Detailed Process Reports

This final section summarizes CPU consumption for each process in the system. It includes system call use; on SMP traces, it includes detailed dispatch information. This section can help determine which processes heavily use system calls. If many threads were run during the trace period, this section would be very large.

Lock Reports

As an option, `utld` will include a detailed report on lock utilization from a trace that has been collected with instrumentation enabled.[2]

To get the `utld` lock report, invoke the `utld` command with the `-l` option as follows:

```
utld -I trace.unwrapped -l lock.out -z -n
   names.out > utld.out
```

The lock output file will contain multiple sections. The Processing Time Summary is the same as the System Summary under CPU Utilization. A Lock Summary report shows cumulative times per processor of lock-held time, processor time spent processing a miss on a blocking lock, and processor time spent spinning. The "unknown" processor represents those lock events in which `utld` processes the unlock trace entry, but not the dispatcher entry (which tells `utld` which processor is executing) and/or the lock trace entry.

Figure 8 shows the lock summary by lock type.

Cumulative Lock CPU Times Per Processor (in msec) Processor	Lock Type	Held	Block Miss	Block Spin
0	swrit	4952.667	12.606	1041.029
0	cwrit	78.564	0.000	0.000
0	lockl	0.068	0.000	0.000
1	swrit	4559.844	14.514	1163.660
1	cwrit	57.259	0.000	0.000
2	swrit	4622.448	13.909	1075.810
2	cwrit	77.352	0.000	0.000
3	swrit	4743.197	13.244	1104.009
3	cwrit	152.326	0.000	0.000
4	swrit	4646.445	13.203	1206.241
4	cwrit	183.835	0.000	0.000
5	swrit	4450.559	16.550	1149.127
5	cwrit	58.848	0.000	0.000
6	swrit	4657.279	15.180	1180.408
6	cwrit	178.170	0.000	0.000
7	swrit	4765.941	13.275	1159.256
7	cwrit	120.007	0.000	0.000
unknown	swrit	5.517	0.000	0.123

Figure 8. Lock summary by processor by lock type

The `utld` individual lock summary provides extensive data on each lock referenced during the trace being examined. The miss probability is the number of misses (spin or block) divided by the total count of attempts. In this case, the miss probability is 53.77%.

The collision cross section is the held, blocked, and spin time divided by the total time. In the SMP case, the total time is the time for all processors. In Figure 9, the Held Elapsed cross section is 0.070—7% of the total time; or, cumulatively, the equivalent of 56% of one processor was spent under the lock in question and 6.9% of the total time was spent spinning, waiting for the lock.

For busy locks, if the miss probability is greater than 5% and/or the collision cross

[2] See documentation on `lockstat` for a description of how to enable lock instrumentation. Information is available on InfoExplorer™ under the `bosboot` command.

Lock: proc_base_lock				0015b7f8 (00000000)			miss prob. = 0.5377			
Total	msec while Held		Blocked	Blocked Miss	msec		Blocked Spin	msec		
Count	Cpu	Elapsed	Trys	Count	Cpu	Elapsed	Count	Cpu	Elapsed	
113147	5489.215	6136.699	0	0	0.000	0.000	60840	6038.430	6038.430	total
113147	0.049	0.054	0	0	0.000	0.000	60840	0.099	0.099	average
	0.063	0.070			0.000	0.000		0.069	0.069	collision xsection

Figure 9. Individual lock summary

```
112  0.001360256 0.004352  lock:   lock lock addr=15B7F8 lock status=1C1F5
        requested_mode=LOCK_SWRITE return addr=18560 name=00A5.0002
```

Figure 10. Trace report lock entry

```
25732  0.076  0.076  0  0  0.000  0.000  13363  0.092  0.092  swrit  .kwakeup          000174c4
(00000064)
25546  0.019  0.019  0  0  0.000  0.000  15564  0.107  0.107  swrit  .e_block_thread   00018560
(000001bc)
25738  0.015  0.015  0  0  0.000  0.000  14775  0.097  0.097  swrit  .e_assert_wait    00018938
(000000a0)
```

Figure 11. Detailed individual lock data excerpts

section is greater than 20% divided by the number of processors (or 2.5% for an 8-way processor), then the possibility that a locking problem exists is high. This example has a very stressed `proc_base_lock`.

The `utld` individual lock summary names the lock that provides data. Since the data used by `utld` when making the report is often not sufficient to provide the name, it simply gives the lock address. The identity of the lock, in terms of class and instance in class, can be determined by using `utld` output, `trcrpt`, and `/usr/include/sys/lockname.h`. The `trcrpt` output for a simple lock provides the lock address (which correlates with the third field of the first line of the individual lock summary) and the lock name. Figure 10 shows the trace report lock entry.

The lock name consists of the class number (00A5) of the lock given at `lock_alloc` and the instance in the lock class (0002). Correlating the class number 0x00A5= decimal 165 in the `lockname.h` file shows that the lock in question is the second instance in the `PROC_INT_CLASS`.

The voluminous data after the individual lock summary (see Figure 11) can be helpful in understanding why a particular lock is busy. It provides insight into the locks held when this lock was blocked, as well as individual function statistics on attempts, blocks, and so on. In this case, much of the effort is spent waiting and waking up. In the database example, the kernel is used primarily to initiate I/O and to wait for its completion.

Conclusion

Trace-based analysis provides an excellent means of analyzing the behavior of an AIX system. The `utld` tool provides a convenient means of analyzing traces and identifying problems associated with high kernel CPU usage.

Bret R. Olszewski, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Olszewski is a senior programmer working on MP performance. He has a BS in Computer Science from the University of Minnesota.

Jim Van Fleet, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Van Fleet is a senior programmer in AIX Performance in Austin. He has held several technical and managerial positions at IBM, with a specialty in Symmetric Multiprocessing systems. Mr. Van Fleet has a BS in Mathematics from Michigan State University and an MS in Computer Science from Union College.