Improving UNIX Kernel Performance using Profile Based Optimization

Steven E. Speer (Hewlett-Packard)
Rajiv Kumar (Hewlett-Packard)
and
Craig Partridge (Bolt Beranek and Newman/Stanford University)

Abstract

Several studies have shown that operating system performance has lagged behind improvements in application performance. In this paper we show how operating systems can be improved to make better use of RISC architectures, particularly in some of the networking code, using a compiling technique known as Profile Based Optimization (PBO). PBO uses profiles from the execution of a program to determine how to best organize the binary code to reduce the number of dynamically taken branches and reduce instruction cache misses. In the case of an operating system, PBO can use profiles produced by instrumented kernels to optimize a kernel image to reflect patterns of use on a particular system. Tests applying PBO to an HP-UX kernel running on an HP9000/720 show that certain parts of the system code (most notably the networking code) achieve substantial performance improvements of up to 35% on micro benchmarks. Overall system performance typically improves by about 5%.

1. Introduction

Achieving good operating system performance on a given processor remains a challenge. Operating systems have not experienced the same improvement in transitioning from CISC to RISC processors that has been experienced by applications. It is becoming apparent that the differences between RISC and CISC processors have greater significance for operating systems than for applications. (This discovery should, in retrospect, probably not be very surprising. An operating system is far more closely linked to the hardware it runs on than is the average application).

The operating system community is still working to fully understand the important differences between RISC and CISC systems. Ousterhout did a study in 1990 that showed that the failure of memory systems to keep pace with improvements in RISC processor performance has had a disproportionately large effect on operating system performance [1]. Mogul and Borg [2] showed that context switches in RISC processors take a surprisingly long time and suggested cases where busy waiting was actually more efficient than taking a context switch. Implementation work by Van Jacobson at Lawrence Berkeley Labs [3] (supported by work by Partridge and Pink [4]) has shown that techniques such as fitting the instructions to do an IP checksum into the otherwise unused instruction cycles between load and store instructions in a memory copy can dramatically improve networking performance. These studies are probably only the beginning of a series of improvements that will come about as operating systems designers better understand RISC platforms.

(We should note that newer CISC processors are increasingly using features such as pipelines and superscalar architectures pioneered by RISC systems. As a result, many of the optimizations used on RISC processors now can be applied to kernels on CISC processors as well. However, to simplify discussion the rest of this paper will continue to talk of ‘‘RISC’’ systems).

This paper looks at another performance question in operating system performance on RISC systems. By their nature, operating systems spend a lot of time doing tests such as confirming that arguments to system calls are valid, that packet headers received from a network are valid, or that a device is idle (or busy). Ultimately, these logical tests are encoded as conditional branch instructions on the target processor.

The frequent tests have at least two causes. First, operating systems typically have multiple concurrent operations in progress at once, and must regularly check that the operations do not interfere. Second, an operating system spends much of its time handling small requests from possibly defective (and thus untrustworthy) applications and therefore must confirm that any information it receives is correct. One can contrast this approach with programs, which rarely support concurrency and often do large amounts of computation without having to worry about external sources of errors.

Because operating systems have to do so much testing, we hypothesized that one reason operating systems do less well than applications on RISC systems is that they suffer more from branch penalties and cache misses. In other words, because operating systems have a lot of branches and because mispredicting which way a branch will go often causes a processor pipeline stall and even a cache miss, operating systems lose a lot of performance due to mispredicted branches. (For instance, in the HP-UX operating system [5] on the PA-RISC processor [6], nearly one instruction in five is a branch instruction, and almost half of those are conditional branches). In our work, we expected branch-related performance problems would be particularly notable in the networking code, because networking code must deal with both input from applications and input from the network. To test our hypothesis, we examined ways to tune the kernel to minimize mispredicted branches, with particular attention paid to performance improvements in the networking code.

2. RISC Processors and Profile Based Optimization

The technique we used to experiment with kernel performance is called profile based optimization (PBO). This section describes RISC architectures in brief to illustrate the problems PBO tries to solve and then explains PBO.

2.1. RISC Processors

RISC processors differ from other processor architectures primarily in the extreme uniformity enforced on the instruction set in order to create a simple instruction pipeline. Because our work was done on a PA-RISC processor (the PA-RISC 1.0, known as Mustang), this section will use that processor as the example.

The PA-RISC Mustang has the five stage CPU-pipeline as shown below:

PA-RISC Mustang CPU Pipeline				
Stage 1	**Stage 2**	**Stage 3**	**Stage 4**	**Stage 5**
Fetch	**Inst Decode**	**Branch**	**Arithmetic**	**Results**

The pipeline can be viewed as an assembly line where one new instruction is fed into the pipeline at the start of each clock cycle and moves one stage further through the pipeline with each clock tick. Five instructions can be processed concurrently in the pipeline. Instruction fetch takes place in the first stage, followed by instruction decoding. After the instruction is decoded, branch targets are calculated, logic operations are done, and conditions codes are set in the third and fourth stages. Finally, the results of the operations are written in the fifth stage.

If an instruction to be fetched in the first stage is not in the processor’s instruction cache, the processor stalls. Execution resumes when the missing instruction is retrieved from main memory. Instructions are loaded from main memory in groups of 8 instructions (a 32-byte cache line). The processor can start executing the missing instruction as soon as it comes in from main memory (even if the cache load is not yet complete). The instruction cache on this machine is a direct mapped cache and distinct from the data cache.

One problem with pipelining is that for conditional branch instructions, the instruction must move through the first two stages of the pipeline before the processor can determine the value of the condition. The value of the condition determines which of two instructions (the instruction after the branch, or the target of the branch) will next be executed. Because the outcome is not known until the branch instruction is at the end of the third stage, if the processor is to keep the pipeline full, it must make a guess about the outcome of the condition. If the processor guesses wrong, it must nullify the instructions that have been incorrectly started down the pipeline and then start over, loading the correct instruction in the first stage of the pipeline. The cycles wasted due to mispredicted branches (those occupied by instructions that were fetched incorrectly and later nullified) are called control hazards or branch stalls. Their effect on execution time is roughly the same as inserting several NO-OP instructions into the executing programs instruction stream.

Another general problem with branches (conditional or not) is that when a branch is taken, its target instruction may not be in the processor's instruction cache, causing a processor stall (even more NO-OPs). To reduce the chance of a cache miss, it is usually desirable to have the most likely target of the branch be near the branch instruction so that the branch and its target are more likely to be cache resident at the same time.

A less obvious penalty for mispredicting a branch is that the mispredicted instruction may also cause a cache miss. This cache miss can be harmful in two ways. First, the code containing the mispredicted instruction may overwrite a section of processor cache that contains active code (which may have to be reloaded later). Second, if the correct instruction is also not in cache, loading the wrong 8 instructions may delay loading the correct instruction because the processor cache line is still busy loading the wrong instructions when the processor issues the request for the correct instruction. Thus the net effect of a mispredicted branch is often that processor resources are used very inefficiently [7].

Overall, therefore, it is reasonable to expect that minimizing the number of mispredicted branches has the potential to reduce the execution time of a program or operating system, both by reducing branch stalls and increasing overall instruction cache efficiency. At a minimum, each branch stall eliminated will reduce execution time by at least one clock cycle.

2.2. Profile Based Optimization

The work described in this paper employs a technique for improving branch prediction and instruction cache usage using profile information gathered from a running system and analyzed by the compiler. The technique is known as Profile Based Optimization or PBO [8]. Support for some PBO has been available in the HP-UX compilers since version 8.02. We used the HP-UX 9.0 compiler which is more aggressive in its use of profile data to do code optimization.

Although PBO has already been applied to several user level applications such as SPEC92, database servers, compilers and Xserver, it has not previously been applied to an operating system. We had to implement additional profiling support in the HP-UX kernel to make these experiments possible. (PBO requires kernel modifications for the same reason `grof` [9] does. Unlike applications, the kernel does not load initialization routines from a library and does not normally terminate. So routines to cause the kernel to initialize PBO data structures and to retrieve PBO data from a running kernel were needed).

2.3. What PBO Does

The Profile Based Optimization available in version 9.0 of the HP-UX language products currently performs two optimizations. These optimizations are:

1. To reorder the basic blocks of a procedure such that the most heavily used basic blocks (a code sequence without a branch) are placed contiguously in an order which favors the hardwares branch prediction logic. This optimization reduces the overall number of mispredicted branches during execution.
2. To reorder the procedures of a program such that procedures on most frequent call-chains are laid out contiguously. By repositioning the procedures in this order, paging and to a lesser extent, cache misses, are reduced [8]. Also by placing most caller-callee pairs closer together in the code layout, fewer long branch sequences are needed to reach the callee (a long branch on PA-RISC takes two instructions).

Observe that both optimizations are code optimizations and nothing is done to reorder the data segment of the program. Furthermore, the two optimizations are independent of one another.

2.4. How PBO Works

PBO produces and uses an execution profile from a program to perform its optimizations in a three step process.

In the first step, the program code is instrumented to collect an execution profile. On the HP-UX C compiler, the `+I` option causes the code to be instrumented. Every arc connecting the nodes of a basic blocks control graph for each procedure is instrumented during the code generation phase. After code generation, the linker then adds instrumentation for every arc of the procedure call graph.

The second step is to run the instrumented executable with some typical input data. For regular programs that run to completion, a profile database is generated when the program completes. In the kernel's case, a separate application must extract profile information from the kernel when profiling has been completed.

The last and the final step is to recompile the program using the profile data as an input to the compiler. Under HP-UX, the `+P` option tells the compiler to examine the profile data. The compiler reads the profile database and does basic block repositioning according to the profile. After the object files are built, linking is performed. The linker reads the same database and performs procedure repositioning as dictated by the profile.

3. Benchmark Results

To test the effects of PBO on the HP-UX kernel, we selected four benchmarks that spent a significant amount of their total execution times in *system* (as opposed to *user*) mode. The benchmarks chosen were:

1. *McKusick Kernel Performance Benchmark.*
   This benchmark was developed to assist in the performance improvement of 4.2BSD UNIX at U.C. Berkeley in the 80's [10]. The benchmark was designed to be relatively insensitive to hardware configuration changes as long as its minimum needs are met. It performs a number of common system services with several different variations in system call parameters. In this case, the benchmark works heavily on exec, pipes, fork and other system calls. The overall time to execute the applications (rather than the kernel performance) is measured.
2. *OSBENCH Operating System Performance Benchmark.*
   The OSBENCH benchmark was developed by John Ousterhout at U.C. Berkeley and used in the 1990 study [1]. It is designed to measure the performance of some basic operating system services. The benchmark performs memory copies with various size buffers, performs some rudimentary file manipulation using various sizes of files, tests pipe performance, checks simple system call response time using getpid(), measures read and write performance with a variety of parameters. It reports results for the specific micro benchmarks.
3. *The KBX Kernel Benchmark.*
   This benchmark was developed by Doug Baskins at Hewlett-Packard to aid in the performance tuning of the HP-UX operating system on HP workstations. It is similar to the other kernel benchmarks in that it measures the time it takes to perform several iterations of various system requests.
4. `Netperf` *Network Performance Benchmark.*
   `Netperf` was also developed at Hewlett-Packard by the Information Networks Division. It is a proprietary tool designed primarily to measure bulk data transfer rates and request/response performance using TCP and UDP. In our tests, we used it to measure the time it took to move data through the protocol stack in the kernel.

The first three benchmarks were chosen because they are generally well-known and understood kernel benchmarks. The `netperf` benchmark was chosen because it stressed the networking part of the kernel, which we thought might give particularly promising results.

The instrumented kernel was run with a single benchmark to obtain a profile database. The kernel was then recompiled with the profile database to produce an optimized version. Note that the kernel was optimized separately for each of the four benchmarks and the only optimization applied was PBO.

Once an optimized kernel was generated, it was run with the same benchmark and benchmark inputs used to generate the PBO database. The results of the optimized kernel were then compared to those obtained by running the base kernel that was built without any optimization. In the sections that follow, performance improvements are always calculated as:

$$(time\_base - time\_opt) / time\_opt$$

where *time_base* is the time it took to execute the benchmark (or portion thereof) on the base kernel and *time_opt* is the time it took to execute the benchmark on the optimized kernel. We also report the standard deviation for the optimized results measured.

The results reported were obtained on an HP 9000/720 workstation with 64MB of RAM. The machine was configured with 2 SCSI disks, each with approximately 100 MB of swap space, and connected to a 10 Mb/Sec Ethernet. The workstation was running Release 8.07 of the HP-UX Operating System.

3.1. McKusick Results

The McKusick test measures aggregate performance and was run for five iterations on each system (because the time for each iteration was small). The results were:

McKusick Benchmark Performance Results

Mean Time Base System (sec)	Mean Time Optimized System (sec)	Std Dev	Improvement
117.40	112.00	0.89	4.8%

Note that because the McKusick benchmark measures application performance, the kernel improvement is diluted by application time spent in user space (which was not optimized). System time accounted for approximately 85% of execution time which hints that the OS performance improvement measured by this benchmark is around 6%.

3.2. OSBENCH Results

OSBENCH was run three times each on both the base system and the optimized systems. The results are summarized in the table below. (Cases where the standard deviation was larger than the measured improvement are marked with a ‘*’).

OSBENCH Optimization Measurements

Time Optimized	Time Base	Standard	Perf. Impr.	units measured
32.8	32.4	0.66	*	cwcor_1K milliseconds
32.2	32.6	0.12	1.1%	cwcor_4K milliseconds
32.2	32.3	0.24	*	cwcor_8K milliseconds
56.3	65.8	0.53	17.0%	cwcor_16K milliseconds
99.2	115.3	0.59	16.2%	cwcor_64K milliseconds
149.7	166.2	0.19	11.0%	cwcor_128K milliseconds
0.42	0.42	0.01	*	cswitch milliseconds
0.17	1.18	0.00	*	getpid microseconds per iteration
0.16	0.17	0.01	*	open_close milliseconds per open/close pair
30.095	31.697	0.443	5.3%	read megabytes/sec.
53.25	60.04	5.72	12.8%	select1/0 microseconds per iteration
56.09	58.33	0.13	4.0%	select1/1 microseconds per iteration
86.26	94.27	5.65	9.3%	select10/0 microseconds per iteration
87.14	92.07	0.31	5.7%	select10/5 microseconds per iteration
83.14	86.52	3.87	*	select10/10 microseconds per iteration
107.81	109.62	0.51	1.7%	select15/0 microseconds per iteration
109.4	110.8	5.90	*	select15/5 microseconds per iteration
102.6	102.8	2.41	*	select15/15 microseconds per iteration

The results were somewhat disappointing because any improvement in most of the micro benchmarks was obscured by the variation in the measured performance values.

3.3. KBX Kernel Benchmark

The KBX benchmarks proved to give far more useful results. Its micro benchmark results are shown below and are the results of five runs of the benchmark.

KBX Optimization Measurements

Time Optimized	Time Base	Std Dev	Perf. Impr.	Microbench Measured
1.6	1.6	0	0%	getpid()
32.3	39.7	0.48	23%	ioctl(0, TCGETA, &termio)
19.8	19.2	0	-3%	gettimeofday(&tp, NULL)
1.0	1.0	0	0%	sbrk(0);
1.5	1.6	0.04	7%	j = umask(0);
402.6	416.7	1.25	4%	switch -- 2 × (if (read/write(fdpipe, BUF, 20) != 20))
139.0	141.2	0.32	2%	if (write/read(fdpipe, BUF, 20) != 20)
175.1	179.1	0.40	2%	if (write/read(fdpipe, BUF, 1024) != 1024)
425.2	441.9	1.39	4%	if (write/read(fdpipe, BUF, 8192) != 8192)
393.5	448.8	5.39	14%	j = creat("/tmp/temp", 0777) & close(j))
273.3	295.5	2.41	8%	pipe(pd); close(pd[0]); close(pd[1])
6327.6	6441.9	12.69	2%	if (fork() == 0) exit(0);
4321.2	4406.0	6.43	2%	if (vfork() == 0) _exit(0);
15067.3	15907.1	15.04	6%	if (fork() == 0) execve("kbx", NULL, NULL);
16604.2	17409.9	68.15	5%	if (fork() == 0) execve("kbx", args(1000), NULL);
15087.5	15901.6	31.00	5%	if (vfork() == 0) execve("kbx", NULL, NULL);
16590.6	17411.8	36.72	5%	if (vfork() == 0) execve("kbx", args(1000), NULL);
66.9	66.1	0.57	-1%	if (read(fd_file, BUF, 20) != 20) from cache
88.5	88.1	0.10	-.5%	if (read(fd_file, BUF, 1024) != 1024) from cache
230.4	231.2	0.59	.3%	if (read(fd_file, BUF, 8192) != 8192) from cache
99.4	106.8	0.45	7%	if (write/read(inet_socket, BUF, 20) != 20)
316.3	360.5	2.58	14%	if (write/read(inet_socket, BUF, 1024) != 1024)
1504.3	1622.8	53.75	8%	if (write/read(inet_socket, BUF, 8192) != 8192)
210.1	230.0	17.67	9%	if (write/read(unix_socket, BUF, 20) != 20)
228.9	241.3	3.76	5%	if (write/read(unix_socket, BUF, 1024) != 1024)
614.0	653.8	4.24	6%	if (write/read(unix_socket, BUF, 8192) != 8192)
22.78s	23.68s	0.27	4%	real time
3.28s	3.46s	0.15	5%	user time
19.17s	19.92s	0.11	4%	system time

With the exception of the huge improvement in *ioctl* performance (which was rather startling), the benchmarks generally suggest that major performance improvements came in tests that did context switching. Most of the read commands from files achieved little or no improvement, but activities that created a new context (e.g., *fork*) or sent data between processes (the socket and pipe tests) generally show a 5% to 10% performance improvement.

Some of the tests that make up this benchmark actually took slightly longer to run after being optimized than they did in the base case. This is because the optimization took data from the entire benchmark (not individual micro benchmarks) to generate a new operating system and the best performance for the entire benchmark sometimes forced blocks that did not get used as much into less advantageous positions relative to the arbitrary locations that they occupy in the base case. This observation is a strong reminder of the importance of selecting profiling inputs that mimic the softwares most likely uses. In extreme cases, failing to do so could actually produce a system that would run slower overall after being optimized.

3.4. `Netperf` Network Performance Benchmark

`Netperf` can run a variety of benchmarks. The benchmark we used measures the performance single byte transactions (sending one byte messages which get one byte replies) using UDP and TCP over the loopback interface in the kernel. This benchmark was designed to measure the impact of PBO on the protocol processing (e.g., code to handle the protocol headers) portion of the networking code by minimizing the data in each packet. To avoid performance effects due to one-behind caches in the operating system, this benchmark was run several times, with increasing numbers of parallel senders and receivers. These results are the averages of 3 runs sending over the loopback interface. Performance is measured in bytes sent per second, but since each byte was sent in a single packet and received a single byte ack, the transmission rate is equivalent to one half the packets per second processed.

`netperf` Optimization Measurements
Loopback Interface

Number of Processes	TCP/UDP	Transmission (b/s) Base	Transmission (b/s) Optimized	Std Dev	Performance Increase
1	TCP	1568.13	2115.69	26.68	35%
2	TCP	1592.89	2124.56	39.87	33%
4	TCP	1556.98	2100.17	12.13	35%
8	TCP	1492.07	2001.53	9.65	34%
16	TCP	1459.67	1879.47	8.00	29%
1	UDP	1801.68	2164.94	10.03	20%
2	UDP	1800.52	2158.79	45.76	20%
4	UDP	1761.23	2164.87	25.44	23%
8	UDP	1685.63	2055.01	14.47	22%
16	UDP	1609.51	1951.92	8.27	21%

While the results do show that TCP, which has a number of internal caches, improves less as the number of processes increases, the transaction rate still improves sharply for both protocols, even when the number of parallel processes is large.

As an experiment, we then ran the test to a remote (unoptimized) host over an otherwise quiet Ethernet. This test is less representative of the effects of PBO, because the performance of both the remote host and the Ethernet affect the results, but it gives a feel for how much an optimized kernel might benefit from dealing with unoptimized kernel. Keep in mind as well, that while the kernel loopback is generally reliable and does not lose packets, real networks do lose packets so the transmission rates may be influenced by the need to retransmit a few packets.

Isolated Network Link to Remote Host

Number of Processes	TCP/UDP	Transmission (b/s) Base	Transmission (b/s) Optimized	Std Dev	Performance Increase
1	TCP	1070.51	1204.96	4.86	13%
2	TCP	1821.14	2079.58	2.05	14%
4	TCP	1831.10	2081.56	19.97	14%
8	TCP	1838.91	2072.57	11.79	13%
16	TCP	1866.70	2109.90	12.31	13%
1	UDP	1163.50	1261.34	4.72	8%
2	UDP	1966.01	2064.20	4.81	5%
4	UDP	2011.84	2091.83	16.57	4%
8	UDP	2023.67	2094.12	18.34	3%
16	UDP	2052.65	2141.47	14.63	4%

The performance improvements are clearly far less than those achieved over the loopback interface in an optimized kernel, but a 14% improvement in TCP performance is still quite useful.

4. Conclusions

The results of the different benchmarks vary from a modest few percent to dramatic (over 30% in parts of the networking code), but the overall results are certainly encouraging.

The results imply that, using PBO, users can at tune their UNIX kernels to better serve the particular load patterns they experience without changing a line of source code. Obviously it is still important to replace slow kernel code with code that runs faster. The point is that PBO adds a final, finishing, level of optimization that can be very useful. Indeed, in some situations, such as the TCP and UDP transactions tested with `netperf`, the PBO optimizations can yield exceptional performance improvements.

Biographies

Steven Speer received his BS in Computer Science from Oregon State University and MS in Computer Science from Stanford University. He works for Hewlett-Packard in Ft. Collins, CO., in the division responsible for the operating systems on PA-RISC platforms.

Rajiv Kumar received his BS in Electrical Engineering from Indian Institute of Technology, Kharagpur and MS in Computer Science from University of Oregon, He works for the Hewlett-Packard Company with the group responsible for building optimizers for the PA-RISC architecture.

Craig Partridge received his A.B. and Ph.D. from Harvard University. He works for Bolt Beranek and Newman, where he is the lead scientist of BBN’s gigabit networking project. He is an consulting assistant professor at Stanford University, Editor-in-Chief of *IEEE Network Magazine*, and the author of *Gigabit Networking*.

References

1. J.K. Ousterhout, “Why Aren’t Operating Systems Getting Faster as Fast as Hardware?,” *Proc. 1990 Summer USENIX Conf.*, Anaheim, June 11-15, 1990.
2. J.C. Mogul and A. Borg, “The Effects of Context Switches on Cache Performance,” *Proc. 4th Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Santa Clara, 8-11 April 1991.
3. V. Jacobson, *Tutorial Notes from SIGCOMM ’90*, Philadelphia, September 1990.
4. C. Partridge and S. Pink, “A Faster UDP,” *IEEE/ACM Trans. on Networking*, vol. 1, no. 4, August 1993.
5. F.W. Clegg, et al, “The HP-UX Operating System on HP Precision Architecture Comuters,” *Hewlett-Packard Journal*, vol. 37, no. 12, December 1986.
6. J.S. Birnbaum and W. S. Worley, “Beyond RISC: High-Precision Architecture",” *Hewlett-Packard Journal*, vol. 35, no. 8, August 1985.
7. S. McFarling, “Program Analysis And Optimization For Machines With Instruction Cache,” *Tech Report, Computer Systems Laboratory, Stanford University, Stanford CA*, September 1991.
8. K. Pettis and R.C. Hansen, “Profile Guided Code Positioning,” *Proc. SIGPLAN on Programming Language Design and Implementation (SIGPLAN Notices)*, vol. 25, no. 6, June 1990.
9. S.L. Graham, P.B. Kessler, and M.K. McKusick, “gprof: A Call Graph Execution Profiler,” *Proceedings of the SIGPLAN ’82 Symposium on Compiler Construction*, pp. 120-126, June 1982.
10. S. Leffler, M. Karels, M.K.McKusick, “Measuring and Improving the Performance of 4.2BSD,” *Proc. 1984 Summer USENIX Conf.*, Salt Lake City, June 12-15, 1984.