Using gprof to Tune the 4.2BSD Kernel

Marshall Kirk McKusick

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

This paper describes how the *gprof* profiler accounts for the running time of called routines in the running time of the routines that call them. It then explains how to configure a profiling kernel on the 4.2 Berkeley Software Distribution of UNIX® for the VAX‡ and discusses tradeoffs in techniques for collecting profile data. *Gprof* identifies problems that severely affects the overall performance of the kernel. Once a potential problem areas is identified benchmark programs are devised to highlight the bottleneck. These benchmarks verify that the problem exist and provide a metric against which to validate proposed solutions. Two caches are added to the kernel to alleviate the bottleneck and *gprof* is used to validates their effectiveness.

---

‡ VAX is a trademark of Digital Equipment Corporation.

TABLE OF CONTENTS

1. Introduction

The purpose of this paper is to describe the tools and techniques that are available for improving the performance of the kernel. The primary tool used to measure the kernel is the hierarchical profiler *gprof*. The profiler enables the user to measure the cost of the abstractions that the kernel provides to the user. Once the expensive abstractions are identified, optimizations are postulated to help improve their performance. These optimizations are each individually verified to insure that they are producing a measurable improvement.

2. The *gprof* Profiler

The purpose of the *gprof* profiling tool is to help the user evaluate alternative implementations of abstractions. The *gprof* design takes advantage of the fact that the kernel though large, is structured and hierarchical. We provide a profile in which the execution time for a set of routines that implement an abstraction is collected and charged to that abstraction. The profile can be used to compare and assess the costs of various implementations [Graham82] [Graham83].

2.1. Data presentation

The data is presented to the user in two different formats. The first presentation simply lists the routines without regard to the amount of time their descendants use. The second presentation incorporates the call graph of the kernel.

2.1.1. The Flat Profile

The flat profile consists of a list of all the routines that are called during execution of the kernel, with the count of the number of times they are called and the number of seconds of execution time for which they are themselves accountable. The routines are listed in decreasing order of execution time. A list of the routines that are never called during execution of the kernel is also available to verify that nothing important is omitted by this profiling run. The flat profile gives a quick overview of the routines that are used, and shows the routines that are themselves responsible for large fractions of the execution time. In practice, this profile usually shows that no single function is overwhelmingly responsible for the total time of the kernel. Notice that for this profile, the individual times sum to the total execution time.

2.1.2. The Call Graph Profile

Ideally, we would like to print the call graph of the kernel, but we are limited by the two-dimensional nature of our output devices. We cannot assume that a call graph is planar, and even if it is, that we can print a planar version of it. Instead, we choose to list each routine, together with information about the routines that are its direct parents and children. This listing presents a window into the call graph. Based on our experience, both parent information and child information is important, and should be available without searching through the output. Figure 1 shows a sample *gprof* entry.

index	%time	self	descendants	called/total called+self called/total	parents name children	index
		0.20	1.20	4/10	CALLER1	[7]
		0.30	1.80	6/10	CALLER2	[1]
[2]	41.5	0.50	3.00	10+4	EXAMPLE	[2]
		1.50	1.00	20/40	SUB1 <cycle1>	[4]
		0.00	0.50	1/5	SUB2	[9]
		0.00	0.00	0/5	SUB3	[11]

Figure 1. Profile entry for EXAMPLE.

The major entries of the call graph profile are the entries from the flat profile, augmented by the time propagated to each routine from its descendants. This profile is sorted by the sum of the time for the routine itself plus the time inherited from its descendants. The profile shows which of the higher level routines spend large portions of the total execution time in the routines that they call. For each routine, we show the amount of time passed by each child to the routine, which includes time for the child itself and for the descendants of the child (and thus the descendants of the routine). We also show the percentage these times represent of the total time accounted to the child. Similarly, the parents of each routine are listed, along with time, and percentage of total routine time, propagated to each one.

Cycles are handled as single entities. The cycle as a whole is shown as though it were a single routine, except that members of the cycle are listed in place of the children. Although the number of calls of each member from within the cycle are shown, they do not affect time propagation. When a child is a member of a cycle, the time shown is the appropriate fraction of the time for the whole cycle. Self-recursive routines have their calls broken down into calls from the outside and self-recursive calls. Only the outside calls affect the propagation of time.

The example shown in Figure 2 is the fragment of a call graph corresponding to the entry in the call graph profile listing shown in Figure 1.

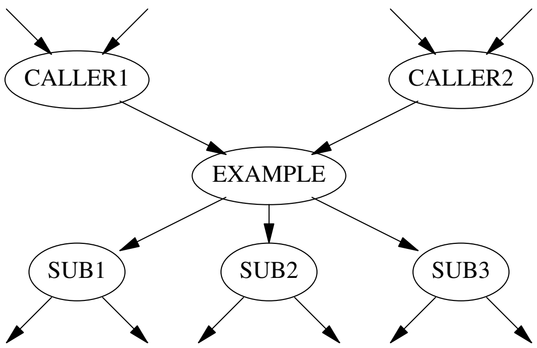

Figure 2. Example call graph fragment.

The entry is for routine EXAMPLE, which has the Caller routines as its parents, and the Sub routines as its children. The reader should keep in mind that all information is given *with respect to EXAMPLE*. The index in the first column shows that EXAMPLE is the second entry in the profile listing. The EXAMPLE routine is called ten times, four times by CALLER1, and six times by CALLER2. Consequently 40% of EXAMPLE's time is propagated to CALLER1, and 60% of EXAMPLE's time is propagated to CALLER2. The self and descendant fields of the parents show the amount of self and descendant time EXAMPLE propagates to them (but not the time used by the parents directly). Note that EXAMPLE calls itself recursively four times. The routine EXAMPLE calls routine SUB1 twenty times, SUB2 once, and never calls SUB3. Since SUB2 is called a total of five times, 20% of its self and descendant time is propagated to EXAMPLE's descendant time field. Because SUB1 is a member of *cycle 1*, the self and descendant times and call count fraction are those for the cycle as a whole. Since cycle 1 is called a total of forty times (not counting calls among members of the cycle), it propagates 50% of the cycle's self and descendant time to EXAMPLE's descendant time field. Finally each name is followed by an index that shows where on the listing to find the entry for that routine.

2.2. Profiling the Kernel

It is simple to build a 4.2BSD kernel that will automatically collect profiling information as it operates simply by specifying the **−p** option to *config*(8) when configuring a kernel. The program counter sampling can be driven by the system clock, or by an alternate real time clock. The latter is highly recommended as use of the system clock results in statistical anomalies in accounting for the time spent in the kernel clock routine.

Once a profiling system has been booted statistic gathering is handled by *kgmon*(8). *Kgmon* allows profiling to be started and stopped and the internal state of the profiling buffers to be dumped. *Kgmon* can also be used to reset the state of the internal buffers to allow multiple experiments to be run without rebooting the machine. The profiling data can then be processed with *gprof*(1) to obtain information regarding the system's operation.

A profiled system is about 5-10% larger in its text space because of the calls to count the subroutine invocations. When the system executes, the profiling data is stored in a buffer that is 1.2 times the size of the text space. All the information is summarized in memory, it is not necessary to have a trace file being continuously dumped to disk. The overhead for running a profiled system varies; under normal load we see anywhere from 5-25% of the system time spent in the profiling code. Thus the system is noticeably slower than an unprofiled system, yet is not so bad that it cannot be used in a production environment. This is important since it allows us to gather data in a real environment rather than trying to devise synthetic work loads.

3. Techniques for Improving Performance

This section gives several hints on general optimization techniques. It then proceeds with an example of how they can be applied to the 4.2BSD kernel to improve its performance.

3.1. Using the Profiler

The profiler is a useful tool for improving a set of routines that implement an abstraction. It can be helpful in identifying poorly coded routines, and in evaluating the new algorithms and code that replace them. Taking full advantage of the profiler requires a careful examination of the call graph profile, and a thorough knowledge of the abstractions underlying the kernel.

The easiest optimization that can be performed is a small change to a control construct or data structure. An obvious starting point is to expand a small frequently called routine inline. The drawback to inline expansion is that the data abstractions in the kernel may become less parameterized, hence less clearly defined. The profiling will also become less useful since the loss of routines will make its output more granular.

Further potential for optimization lies in routines that implement data abstractions whose total execution time is long. If the data abstraction function cannot easily be speeded up, it may be advantageous to cache its results, and eliminate the need to rerun it for identical inputs. These and other ideas for program improvement are discussed in [Bentley81].

This tool is best used in an iterative approach: profiling the kernel, eliminating one bottleneck, then finding some other part of the kernel that begins to dominate execution time.

A completely different use of the profiler is to analyze the control flow of an unfamiliar section of the kernel. By running an example that exercises the unfamiliar section of the kernel, and then using *gprof*, you can get a view of the control structure of the unfamiliar section.

3.2. An Example of Tuning

The first step is to come up with a method for generating profile data. We prefer to run a profiling system for about a one day period on one of our general timesharing machines. While this is not as reproducible as a synthetic workload, it certainly represents a realistic test. We have run one day profiles on several occasions over a three month period. Despite the long period of time that elapsed between the test runs the shape of the profiles, as measured by the number of times each system call entry point was called, were remarkably similar.

A second alternative is to write a small benchmark program to repeated exercise a suspected bottleneck. While these benchmarks are not useful as a long term profile they can give quick feedback on whether a hypothesized improvement is really having an effect. It is important to realize that the only real assurance that a change has a beneficial effect is through long term measurements of general timesharing. We have numerous examples where a benchmark program suggests vast improvements while the change in the long term system performance is negligible, and conversely examples in which the benchmark program run more slowly, but the long term system performance improves significantly.

An investigation of our long term profiling showed that the single most expensive function performed by the kernel is path name translation. We find that our general time sharing systems do about 500,000 name translations per day. The cost of doing name translation in the original 4.2BSD is 24.2 milliseconds, representing 40% of the time processing system calls, which is 19% of the total cycles in the kernel, or 11% of all cycles executed on the machine. The times are shown in Figure 3.

part	time	% of kernel
self	14.3 ms/call	11.3%
child	9.9 ms/call	7.9%
total	24.2 ms/call	19.2%

Figure 3. Call times for *namei*.

The system measurements collected showed the pathname translation routine, *namei*, was clearly worth optimizing. An inspection of *namei* shows that it consists of two nested loops. The outer loop is traversed once per pathname component. The inner loop performs a linear search through a directory looking for a particular pathname component.

Our first idea was to observe that many programs step through a directory performing an operation on each entry in turn. This caused us to modify *namei* to cache the directory offset of the last pathname component looked up by a process. The cached offset is then used as the point at which a search in the same directory begins. Changing directories invalidates the cache, as does modifying the directory. For programs that step sequentially through a directory with N files, search time decreases from $O(N^2)$ to $O(N)$.

The cost of the cache is about 20 lines of code (about 0.2 kilobytes) and 16 bytes per process, with the cached data stored in a process's *user* vector.

As a quick benchmark to verify the effectiveness of the cache we ran ''ls −l'' on a directory containing 600 files. Before the per-process cache this command used 22.3 seconds of system time. After adding the cache the program used the same amount of user time, but the system time dropped to 3.3 seconds.

This change prompted our rerunning a profiled system on a machine containing the new *namei*. The results showed that the time in *namei* dropped by only 2.6 ms/call and still accounted for 36% of the system call time, 18% of the kernel, or about 10% of all the machine cycles. This amounted to a drop in system time from 57% to about 55%. The results are shown in Figure 4.

part	time	% of kernel
self	11.0 ms/call	9.2%
child	10.6 ms/call	8.9%
total	21.6 ms/call	18.1%

Figure 4. Call times for *namei* with per-process cache.

The small performance improvement was caused by a low cache hit ratio. Although the cache was 90% effective when hit, it was only usable on about 25% of the names being translated. An additional reason for the small improvement was that although the amount of time spent in *namei* itself decreased substantially, more time was spent in the routines that it called since each directory had to be accessed twice; once to search from the middle to the end, and once to search from the beginning to the middle.

Most missed names were caused by path name components other than the last. Thus Robert Elz introduced a system wide cache of most recent name translations. The cache is keyed on a name and the inode and device number of the directory that contains it. Associated with each entry is a pointer to the corresponding entry in the inode table. This has the effect of short circuiting the outer loop of *namei*. For each path name component, *namei* first looks in its cache of recent translations for the needed name. If it exists, the directory search can be completely eliminated. If the name is not recognized, then the per-process cache may still be useful in reducing the directory search time. The two cacheing schemes complement each other well.

The cost of the name cache is about 200 lines of code (about 1.2 kilobytes) and 44 bytes per cache entry. Depending on the size of the system, about 200 to 1000 entries will normally be configured, using 10-44 kilobytes of physical memory. The name cache is resident in memory at all times.

After adding the system wide name cache we reran ‘‘ls −l’’ on the same directory. The user time remained the same, however the system time rose slightly to 3.7 seconds. This was not surprising as *namei* now had to maintain the cache, but was never able to make any use of it.

Another profiled system was created and measurements were collected over a one day period. These measurements showed a 6 ms/call decrease in *namei*, with *namei* accounting for only 31% of the system call time, 16% of the time in the kernel, or about 7% of all the machine cycles. System time dropped from 55% to about 49%. The results are shown in Figure 5.

part	time	% of kernel
self	9.5 ms/call	9.6%
child	6.1 ms/call	6.1%
total	15.6 ms/call	15.7%

Figure 5. Call times for *namei* with both caches.

Statistics on the performance of both caches show the large performance improvement is caused by the high hit ratio. On the profiled system a 60% hit rate was observed in the system wide cache. This, coupled with the 25% hit rate in the per-process offset cache yielded an effective cache hit rate of 85%. While the system wide cache reduces both the amount of time in the routines that *namei* calls as well as *namei* itself (since fewer directories need to be accessed or searched), it is interesting to note that the actual percentage of system time spent in *namei* itself increases even though the actual time per call decreases. This is because less total time is being spent in the kernel, hence a smaller absolute time becomes a larger total percentage.

4. Conclusions

We have created a profiler that aids in the evaluation of the kernel. For each routine in the kernel, the profile shows the extent to which that routine helps support various abstractions, and how that routine uses other abstractions. The profile assesses the cost of routines at all levels of the kernel decomposition. The profiler is easily used, and can be compiled into the kernel. It adds only five to thirty percent execution overhead to the kernel being profiled, produces no additional output while the kernel is running and allows the kernel to be measured in its real environment. Kernel profiles can be used to identify bottlenecks in performance. We have shown how to improve performance by caching recently calculated name translations. The combined caches added to the name translation process reduce the average cost of translating a pathname to an inode by 35%. These changes reduce the percentage of time spent running in the system by nearly 9%.

5. Acknowledgements

I would like to thank Robert Elz for sharing his ideas and his code for cacheing system wide names. Thanks also to all the users at Berkeley who provided all the input to generate the kernel profiles. This work was supported by the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

6. References

[Bentley81] Bentley, J. L., ‘‘Writing Efficient Code’’, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, CMU-CS-81-116, 1981.

[Graham82] Graham, S., Kessler, P., McKusick, M., ‘‘gprof: A Call Graph Execution Profiler’’, Proceedings of the SIGPLAN ’82 Symposium on Compiler Construction, Volume 17, Number 6, June 1982. pp 120-126

[Graham83] Graham, S., Kessler, P., McKusick, M., ‘‘An Execution Profiler for Modular Programs’’ Software - Practice and Experience, Volume 13, 1983. pp 671-685

[Ritchie74] Ritchie, D. M. and Thompson, K., ''The UNIX Time-Sharing System'', CACM 17, 7. July 1974. pp 365-375