Reprinted from
Proceedings of the ACM SIGPLAN ’82 Symposium on Compiler Construction
SIGPLAN Notices 17, 6 (June 1982), pp 120-126†

gprof: a Call Graph Execution Profiler[1]

by
Susan L. Graham
Peter B. Kessler
Marshall K. McKusick

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

Abstract

Large complex programs are composed of many small routines that implement abstractions for the routines that call them. To be useful, an execution profiler must attribute execution time in a way that is significant for the logical structure of a program as well as for its textual decomposition. This data must then be displayed to the user in a convenient and informative way. The **gprof** profiler accounts for the running time of called routines in the running time of the routines that call them. The design and use of this profiler is described.

1. Programs to be Profiled

Software research environments normally include many large programs both for production use and for experimental investigation. These programs are typically modular, in accordance with generally accepted principles of good program design. Often they consist of numerous small routines that implement various abstractions. Sometimes such large programs are written by one programmer who has understood the requirements for these abstractions, and has programmed them appropriately. More frequently the program has had multiple authors and has evolved over time, changing the demands placed on the implementation of the abstractions without changing the implementation itself. Finally, the program may be assembled from a library of abstraction implementations unexamined by the programmer.

Once a large program is executable, it is often desirable to increase its speed, especially if small portions of the program are found to dominate its execution time. The purpose of the **gprof** profiling tool is to help the user evaluate alternative implementations of abstractions. We developed this tool in response to our efforts to improve a code generator we were writing [Graham82].

The **gprof** design takes advantage of the fact that the programs to be measured are large, structured and hierarchical. We provide a profile in which the execution time for a set of routines that implement an

[1]This work was supported by grant MCS80-05144 from the National Science Foundation.

abstraction is collected and charged to that abstraction. The profile can be used to compare and assess the costs of various implementations.

The profiler can be linked into a program without special planning by the programmer. The overhead for using **gprof** is low; both in terms of added execution time and in the volume of profiling information recorded.

2. Types of Profiling

There are several different uses for program profiles, and each may require different information from the profiles, or different presentation of the information. We distinguish two broad categories of profiles: those that present counts of statement or routine invocations, and those that display timing information about statements or routines. Counts are typically presented in tabular form, often in parallel with a listing of the source code. Timing information could be similarly presented; but more than one measure of time might be associated with each statement or routine. For example, in the framework used by **gprof** each profiled segment would display two times: one for the time used by the segment itself, and another for the time inherited from code segments it invokes.

Execution counts are used in many different contexts. The exact number of times a routine or statement is activated can be used to determine if an algorithm is performing as expected. Cursory inspection of such counters may show algorithms whose complexity is unsuited to the task at hand. Careful interpretation of counters can often suggest improvements to acceptable algorithms. Precise examination can uncover subtle errors in an algorithm. At this level, profiling counters are similar to debugging statements whose purpose is to show the number of times a piece of code is executed. Another view of such counters is as boolean values. One may be interested that a portion of code has executed at all, for exhaustive testing, or to check that one implementation of an abstraction completely replaces a previous one.

Execution counts are not necessarily proportional to the amount of time required to execute the routine or statement. Further, the execution time of a routine will not be the same for all calls on the routine. The criteria for establishing execution time must be decided. If a routine implements an abstraction by invoking other abstractions, the time spent in the routine will not accurately reflect the time required by the abstraction it implements. Similarly, if an abstraction is implemented by several routines the time required by the abstraction will be distributed across those routines.

Given the execution time of individual routines, **gprof** accounts to each routine the time spent for it by the routines it invokes. This accounting is done by assembling a *call graph* with nodes that are the routines of the program and directed arcs that represent calls from call sites to routines. We distinguish among three different call graphs for a program. The *complete call graph* incorporates all routines and all potential arcs, including arcs that represent calls to functional parameters or functional variables. This graph contains the other two graphs as subgraphs. The *static call graph* includes all routines and all possible arcs that are not calls to functional parameters or variables. The *dynamic call graph* includes only those routines and arcs traversed by the profiled execution of the program. This graph need not include all routines, nor need it include all potential arcs between the routines it covers. It may, however, include arcs to functional parameters or variables that the static call graph may omit. The static call graph can be determined from the (static) program text. The dynamic call graph is determined only by profiling an execution of the program. The complete call graph for a monolithic program could be determined by data flow analysis techniques. The complete call graph for programs that change during execution, by modifying themselves or dynamically loading or overlaying code, may never be determinable. Both the static call graph and the dynamic call graph are used by **gprof**, but it does not search for the complete call graph.

3. Gathering Profile Data

Routine calls or statement executions can be measured by having a compiler augment the code at strategic points. The additions can be inline increments to counters [Knuth71] [Satterthwaite72] [Joy79] or calls to monitoring routines [Unix]. The counter increment overhead is low, and is suitable for profiling statements. A call of the monitoring routine has an overhead comparable with a call of a regular routine, and is therefore only suited to profiling on a routine by routine basis. However, the monitoring routine solution has certain advantages. Whatever counters are needed by the monitoring routine can be managed by the monitoring routine itself, rather than being distributed around the code. In particular, a monitoring routine can easily be called from separately compiled programs. In addition, different monitoring routines can be linked into the program being measured to assemble different profiling data without having to change the compiler or recompile the program. We have exploited this approach; our compilers for C, Fortran77, and Pascal can insert calls to a monitoring routine in the prologue for each routine. Use of the monitoring routine requires no planning on part of a programmer other than to request that augmented routine prologues be produced during compilation.

We are interested in gathering three pieces of information during program execution: call counts and execution times for each profiled routine, and the arcs of the dynamic call graph traversed by this execution of the program. By post-processing of this data we can build the dynamic call graph for this execution of the program and propagate times along the edges of this graph to attribute times for routines to the routines that invoke them.

Gathering of the profiling information should not greatly interfere with the running of the program. Thus, the monitoring routine must not produce trace output each time it is invoked. The volume of data thus produced would be unmanageably large, and the time required to record it would overwhelm the running time of most programs. Similarly, the monitoring routine can not do the analysis of the profiling data (e.g. assembling the call graph, propagating times around it, discovering cycles, etc.) during program execution. Our solution is to gather profiling data in memory during program execution and to condense it to a file as the profiled program exits. This file is then processed by a separate program to produce the listing of the profile data. An advantage of this approach is that the profile data for several executions of a program can be combined by the post-processing to provide a profile of many executions.

The execution time monitoring consists of three parts. The first part allocates and initializes the runtime monitoring data structures before the program begins execution. The second part is the monitoring routine invoked from the prologue of each profiled routine. The third part condenses the data structures and writes them to a file as the program terminates. The monitoring routine is discussed in detail in the following sections.

3.1. Execution Counts

The **gprof** monitoring routine counts the number of times each profiled routine is called. The monitoring routine also records the arc in the call graph that activated the profiled routine. The count is associated with the arc in the call graph rather than with the routine. Call counts for routines can then be determined by summing the counts on arcs directed into that routine. In a machine-dependent fashion, the monitoring routine notes its own return address. This address is in the prologue of some profiled routine that is the destination of an arc in the dynamic call graph. The monitoring routine also discovers the return address for that routine, thus identifying the call site, or source of the arc. The source of the arc is in the *caller*, and the destination is in the *callee*. For example, if a routine A calls a routine B, A is the caller, and B is the callee. The prologue of B will include a call to the monitoring routine that will note the arc from A to B and either initialize or increment a counter for that arc.

One can not afford to have the monitoring routine output tracing information as each arc is identified. Therefore, the monitoring routine maintains a table of all the arcs discovered, with counts of the numbers of times each is traversed during execution. This table is accessed once per routine call. Access to it must be as fast as possible so as not to overwhelm the time required to execute the program.

Our solution is to access the table through a hash table. We use the call site as the primary key with the callee address being the secondary key. Since each call site typically calls only one callee, we can reduce (usually to one) the number of minor lookups based on the callee. Another alternative would use the callee as the primary key and the call site as the secondary key. Such an organization has the advantage of associating callers with callees, at the expense of longer lookups in the monitoring routine. We are fortunate to be running in a virtual memory environment, and (for the sake of speed) were able to allocate enough space for the primary hash table to allow a one-to-one mapping from call site addresses to the primary hash table. Thus our hash function is trivial to calculate and collisions occur only for call sites that call multiple destinations (e.g. functional parameters and functional variables). A one level hash function using both call site and callee would result in an unreasonably large hash table. Further, the number of dynamic call sites and callees is not known during execution of the profiled program.

Not all callers and callees can be identified by the monitoring routine. Routines that were compiled without the profiling augmentations will not call the monitoring routine as part of their prologue, and thus no arcs will be recorded whose destinations are in these routines. One need not profile all the routines in a program. Routines that are not profiled run at full speed. Certain routines, notably exception handlers, are invoked by non-standard calling sequences. Thus the monitoring routine may know the destination of an arc (the callee), but find it difficult or impossible to determine the source of the arc (the caller). Often in these cases the apparent source of the arc is not a call site at all. Such anomalous invocations are declared ‘‘spontaneous’’.

3.2. Execution Times

The execution times for routines can be gathered in at least two ways. One method measures the execution time of a routine by measuring the elapsed time from routine entry to routine exit. Unfortunately, time measurement is complicated on time-sharing systems by the time-slicing of the program. A second method samples the value of the program counter at some interval, and infers execution time from the distribution of the samples within the program. This technique is particularly suited to time-sharing systems, where the time-slicing can serve as the basis for sampling the program counter. Notice that, whereas the first method could provide exact timings, the second is inherently a statistical approximation.

The sampling method need not require support from the operating system: all that is needed is the ability to set and respond to ‘‘alarm clock’’ interrupts that run relative to program time. It is imperative that the intervals be uniform since the sampling of the program counter rather than the duration of the interval is the basis of the distribution. If sampling is done too often, the interruptions to sample the program counter will overwhelm the running of the profiled program. On the other hand, the program must run for enough sampled intervals that the distribution of the samples accurately represents the distribution of time for the execution of the program. As with routine call tracing, the monitoring routine can not afford to output information for each program counter sample. In our computing environment, the operating system can provide a histogram of the location of the program counter at the end of each clock tick (1/60th of a second) in which a program runs. The histogram is assembled in memory as the program runs. This facility is enabled by our monitoring routine. We have adjusted the granularity of the histogram so that program counter values map one-to-one onto the histogram. We make the simplifying assumption that all calls to a specific routine require the same amount of time to execute. This assumption may disguise that some calls (or worse, some call sites) always invoke a routine such that its execution is faster (or slower) than the average time for that routine.

When the profiled program terminates, the arc table and the histogram of program counter samples is written to a file. The arc table is condensed to consist of the source and destination addresses of the arc and the count of the number of times the arc was traversed by this execution of the program. The recorded histogram consists of counters of the number of times the program counter was found to be in each of the ranges covered by the histogram. The ranges themselves are summarized as a lower and upper bound and a step size.

4. Post Processing

Having gathered the arcs of the call graph and timing information for an execution of the program, we are interested in attributing the time for each routine to the routines that call it. We build a dynamic call graph with arcs from caller to callee, and propagate time from descendants to ancestors by topologically sorting the call graph. Time propagation is performed from the leaves of the call graph toward the roots, according to the order assigned by a topological numbering algorithm. The topological numbering ensures that all edges in the graph go from higher numbered nodes to lower numbered nodes. An example is given in Figure 1. If we propagate time from nodes in the order assigned by the algorithm, execution time can be propagated from descendants to ancestors after a single traversal of each arc in the call graph. Each parent receives some fraction of a child’s time. Thus time is charged to the caller in addition to being charged to the callee.

Let C_e be the number of calls to some routine, e, and C_e^r be the number of calls from a caller r to a callee e. Since we are assuming each call to a routine takes the average amount of time for all calls to that routine, the caller is accountable for C_e^r / C_e of the time spent by the callee. Let the S_e be the *selftime* of a

8 9

3 7

2 5 6

1 4

Topological ordering
Figure 1.

routine, e. The selftime of a routine can be determined from the timing information gathered during profiled program execution. The total time, T_r, we wish to account to a routine r, is then given by the recurrence equation:

$$T_r = S_r + \sum_{r \text{ CALLS } e} T_e \times \frac{C_e^r}{C_e}$$

where r CALLS e is a relation showing all routines e called by a routine r. This relation is easily available from the call graph.

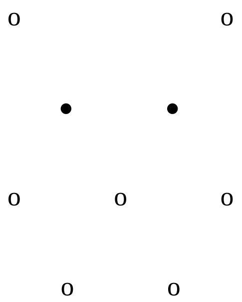
However, if the execution contains recursive calls, the call graph has cycles that cannot be topologically sorted. In these cases, we discover strongly-connected components in the call graph, treat each such component as a single node, and then sort the resulting graph. We use a variation of Tarjan's strongly-connected components algorithm that discovers strongly-connected components as it is assigning topological order numbers [Tarjan72].

Time propagation within strongly connected components is a problem. For example, a self-recursive routine (a trivial cycle in the call graph) is accountable for all the time it uses in all its recursive instantiations. In our scheme, this time should be shared among its call graph parents. The arcs from a routine to itself are of interest, but do not participate in time propagation. Thus the simple equation for time propagation does not work within strongly connected components. Time is not propagated from one member of a cycle to another, since, by definition, this involves propagating time from a routine to itself. In addition, children of one member of a cycle must be considered children of all members of the cycle. Similarly, parents of one member of the cycle must inherit all members of the cycle as descendants. It is for these reasons that we collapse connected components. Our solution collects all members of a cycle together, summing the time and call counts for all members. All calls into the cycle are made to share the total time of the cycle, and all descendants of the cycle propagate time into the cycle as a whole. Calls among the members of the cycle do not propagate any time, though they are listed in the call graph profile.

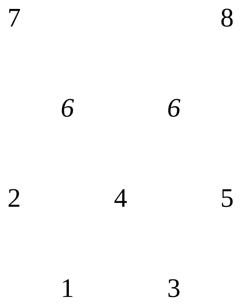
Figure 2 shows a modified version of the call graph of Figure 1, in which the nodes labelled 3 and 7 in Figure 1 are mutually recursive. The topologically sorted graph after the cycle is collapsed is given in Figure 3.

Since the technique described above only collects the dynamic call graph, and the program typically does not call every routine on each execution, different executions can introduce different cycles in the dynamic call graph. Since cycles often have a significant effect on time propagation, it is desirable to incorporate the static call graph so that cycles will have the same members regardless of how the program runs.

The static call graph can be constructed from the source text of the program. However, discovering the static call graph from the source text would require two moderately difficult steps: finding the source



Cycle to be collapsed.
Figure 2.



Topological numbering after cycle collapsing.
Figure 3.

text for the program (which may not be available), and scanning and parsing that text, which may be in any one of several languages.

In our programming system, the static calling information is also contained in the executable version of the program, which we already have available, and which is in language-independent form. One can examine the instructions in the object program, looking for calls to routines, and note which routines can be called. This technique allows us to add arcs to those already in the dynamic call graph. If a statically discovered arc already exists in the dynamic call graph, no action is required. Statically discovered arcs that do not exist in the dynamic call graph are added to the graph with a traversal count of zero. Thus they are never responsible for any time propagation. However, they may affect the structure of the graph. Since they may complete strongly connected components, the static call graph construction is done before topological ordering.

5. 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 program.

5.1. The Flat Profile

The flat profile consists of a list of all the routines that are called during execution of the program, 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 program is also available to verify that nothing important is omitted by this execution. 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 program. Notice that for this profile, the individual times sum to the total execution time.

5.2. The Call Graph Profile

Ideally, we would like to print the call graph of the program, 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.

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 following example is a typical fragment of a call graph.

CALLER1 CALLER2

EXAMPLE

SUB1 SUB2 SUB3

The entry in the call graph profile listing for this example is shown in Figure 4.

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]

Profile entry for EXAMPLE.
Figure 4.

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.

6. Using the Profiles

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 program.

The easiest optimization that can be performed is a small change to a control construct or data structure that improves the running time of the program. An obvious starting point is a routine that is called many times. For example, suppose an output routine is the only parent of a routine that formats the data. If this format routine is expanded inline in the output routine, the overhead of a function call and return can be saved for each datum that needs to be formatted.

The drawback to inline expansion is that the data abstractions in the program 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. For example, if the symbol table functions ‘‘lookup’’, ‘‘insert’’, and ‘‘delete’’ are all merged into a single parameterized routine, it will be impossible to determine the costs of any one of these individual functions from the profile.

Further potential for optimization lies in routines that implement data abstractions whose total execution time is long. For example, a lookup routine might be called only a few times, but use an inefficient linear search algorithm, that might be replaced with a binary search. Alternately, the discovery that a rehashing function is being called excessively, can lead to a different hash function or a larger hash table. 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 program, eliminating one bottleneck, then finding some other part of the program that begins to dominate execution time. For instance, we have used **gprof** on itself; eliminating, rewriting, and inline expanding routines, until reading data files (hardly a target for optimization!) represents the dominating factor in its execution time.

Certain types of programs are not easily analyzed by **gprof**. They are typified by programs that exhibit a large degree of recursion, such as recursive descent compilers. The problem is that most of the major routines are grouped into a single monolithic cycle. As in the symbol table abstraction that is placed in one routine, it is impossible to distinguish which members of the cycle are responsible for the execution time. Unfortunately there are no easy modifications to these programs that make them amenable to analysis.

A completely different use of the profiler is to analyze the control flow of an unfamiliar program. If you receive a program from another user that you need to modify in some small way, it is often unclear where the changes need to be made. By running the program on an example and then using **gprof**, you can get a view of the structure of the program.

Consider an example in which you need to change the output format of the program. For purposes of this example suppose that the call graph of the output portion of the program has the following structure:

CALC1 CALC2 CALC3

FORMAT1 FORMAT2

‘‘WRITE’’

Initially you look through the **gprof** output for the system call ‘‘WRITE’’. The format routine you will need to change is probably among the parents of the ‘‘WRITE’’ procedure. The next step is to look at the profile entry for each of parents of ‘‘WRITE’’, in this example either ‘‘FORMAT1’’ or ‘‘FORMAT2’’, to determine which one to change. Each format routine will have one or more parents, in this example ‘‘CALC1’’, ‘‘CALC2’’, and ‘‘CALC3’’. By inspecting the source code for each of these routines you can determine which format routine generates the output that you wish to modify. Since the **gprof** entry shows all the potential calls to the format routine you intend to change, you can determine if your modifications will affect output that should be left alone. If you desire to change the output of ‘‘CALC2’’, but not ‘‘CALC3’’, then formatting routine ‘‘FORMAT2’’ needs to be split into two separate routines, one of which implements the new format. You can then retarget just the call by ‘‘CALC2’’ that needs the new format. It should be noted that the static call information is particularly useful here since the test case you run probably will not exercise the entire program.

7. Conclusions

We have created a profiler that aids in the evaluation of modular programs. For each routine in the program, the profile shows the extent to which that routine helps support various abstractions, and how that routine uses other abstractions. The profile accurately assesses the cost of routines at all levels of the program decomposition. The profiler is easily used, and can be compiled into the program without any prior planning by the programmer. It adds only five to thirty percent execution overhead to the program being profiled, produces no additional output until after the program finishes, and allows the program to be measured in its actual environment. Finally, the profiler runs on a time-sharing system using only the normal services provided by the operating system and compilers.

8. 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. L., Henry, R. R., Schulman, R. A., ‘‘An Experiment in Table Driven Code Generation’’, SIGPLAN ’82 Symposium on Compiler Construction, June, 1982.

[Joy79]
Joy, W. N., Graham, S. L., Haley, C. B. ‘‘Berkeley Pascal User’s Manual’’, Version 1.1, Computer Science Division University of California, Berkeley, CA. April 1979.

[Knuth71]
Knuth, D. E. ‘‘An empirical study of FORTRAN programs’’, Software - Practice and Experience, 1, 105-133. 1971

[Satterthwaite72]
Satterthwaite, E. ‘‘Debugging Tools for High Level Languages’’, Software - Practice and Experience, 2, 197-217, 1972

[Tarjan72]
Tarjan, R. E., ‘‘Depth first search and linear graph algorithm,’’ *SIAM J. Computing* **1**:2, 146-160, 1972.

[Unix]
Unix Programmer’s Manual, ‘‘**prof** command’’, section 1, Bell Laboratories, Murray Hill, NJ. January 1979.