Fast, accurate call graph profiling

Michael Spivey
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Abstract

Profiling is a technique for identifying performance bottlenecks in programs by measuring the time spent in each subroutine as the program runs. In call graph profiling, time used by each subroutine is also charged to its callers in order to give a better idea of how the time is divided among major tasks in the program. In order to do this with acceptable performance impact, a common implementation technique (used by GNU *gprof* among others) is to count the number of calls of each subroutine, and charge its time to the callers in proportion to the number of calls they make. Whilst this gives acceptably accurate results for simple programs, it is inaccurate for subroutines whose running time depends on the values of their arguments, and becomes almost useless for programs that exploit higher-order functions or dynamic method binding. Programs containing mutually recursive subroutines cause additional problems with this approach.

In this article, we discuss a way of improving on *gprof* by collecting more information during execution, without significantly increasing the overhead of profiling. The method is based on the idea of keeping track of the set of subroutines that are active at each moment during the execution of the program being analysed and the calling arcs between these subroutines. By considering only the most recent activation of each subroutine, we arrange that even recursive programs give rise to a finite number of these contexts that is usually fairly small. The information can be collected efficiently by dynamically constructing a finite state machine whose states correspond to execution contexts in the program.[1]

1 Introduction

Whilst the data produced by *gprof* is very useful, it is important to recognise that it is an approximation based on an assumption that may be inaccurate. The only times that are directly measured are the total times spent running each subroutine: the *self* times. The profiler also counts the number of times each arc in the program's call graph is traversed during execution; that is, the number of times each routine calls each of its subroutines. After execution is over, the time taken by each subroutine is charged also to its parents, in proportion to the number of times each parent called the routine. This procedure gives reasonably accurate results if each call of a subroutine takes approximately the same time, or if the average time it takes is independent of the place in the program from which it is called: we call this the *average time assumption*. However, serious inaccuracies will result if the time taken by a subroutine depends on its arguments, and the arguments are drawn from different distributions in different calls of the same subroutine. Ponder and Fateman [11] discuss this problem at length.

The average time assumption is particularly likely to be violated in programs written in a functional or object-oriented style. For example, a compiler written in a polymorphic higher-order language such as ML might represent both the declarations in a subroutine heading and the subroutines in a program by instances of the polymorphic list type, and different parts of the compiler might contain the expressions,

$$map\ alloc\ decls,$$

to allocate storage for each declaration, and

$$map\ translate\ subrs,$$

to translate each subroutine into machine code. Both these expressions use the same polymorphic higher-order function *map* to apply a single function to each element of a list, but we expect the function *alloc* that allocates storage for a declaration to be much cheaper than the function *translate* that translates a subroutine. Nevertheless, the average time assumption charges all these costs as child time to *map*, and recharges them to the calling routines that contain these two expressions. The result would be misleading enough if there were only one evaluation of each of these expressions, but there are likely to be many lists of declarations but only one list of subroutines is a typical program; the result is that almost all the time, both for processing declarations and for translating subroutines, will be charged to the routine for processing declarations.

Some concrete numbers may help to make the nature of the problem clear. Suppose that the program being compiled contains 10 subroutines, each with 10 variables, that allocating storage for a variable takes 1 tick and that translating a subroutine takes 100 ticks. The profile for this program that would be shown by *gprof* is shown in Table 1. In this profile, the 100 ticks that are spent running *alloc* and the 1000 ticks spent running *translate* add up to a total of

[1] The work presented in this paper is the subject of United Kingdom patent application number 0110302.7

Submitted to PLDI 2002

total	self	child	calls	
	20	1000	10/11	do_decls
	2	100	1/11	do_trans
1122	22	1100	11	map
	100	0	100/100	alloc
	1000	0	10/10	translate
1030	10	1020	10	do_decls
	20	1000	10/11	map
	1000	0	10/10	map
1000	1000	0	10	translate
103	1	102	1	do_trans
	2	100	1/11	map
	100	0	100/100	map
100	100	0	100	alloc

Table 1: Profile generated by *gprof*

total	self	child	calls	
	20	100	10/11	do_decls
	2	1000	1/11	do_trans
1122	22	1100	11	map
	100	0	100/100	alloc
	1000	0	10/10	translate
1003	1	1002	1	do_trans
	2	1000	1/11	map
	1000	0	10/10	map
1000	1000	0	10	translate
130	10	120	10	do_decls
	20	100	10/11	map
	100	0	100/100	map
100	100	0	100	alloc

Table 2: An accurate profile

1100 ticks spent running children of *map*, and this is shared out in a ratio of 10:1 between the callers of *map*, namely *do_decls* and *do_trans*. Substantially all the time in this program is taken up with translating subroutines, an activity that is controlled by routine *do_trans*, but this routine is shown with a small time in the profile.

An accurate profile for the compiler is shown in Table 2. Note that in this profile, the time taken by routines *alloc* and *translate* are correctly attributed to their grandparents *do_decls* and *do_trans* respectively, even though the routine *map* acts as intermediary in both cases.

Similarly misleading profiles can result from common 'design patterns' in object-oriented programming. For one example, suppose that buttons in the user interface of a word processor are bound to *Command* objects, and when activated, they send these commands to the *execute* method of another object that represents the application itself. The application object is responsible for saving the file once every 100 commands, maintaining an 'undo' list, etc., but principally calls the *perform* method on each *Command* object it receives. The result will be that all the time spent carrying out commands will be charged as child time to the *execute* method of the application, and will be recharged to the buttons in proportion to the number of times each button was clicked, regardless of the relative costs of processing the clicks. If one button puts the current word in bold type, and takes one tick to execute, while another spell-checks the entire document and takes 100 ticks, then the same kind of situation is created as we saw in the previous example, and the average time assumption will give a seriously misleading impression.

If *gprof* behaves badly for higher-order and object-oriented programs, it becomes even worse with programs where there is mutual recursion; that is, where two or more routines are linked in a ring where each calls the next, so that the call graph contains a non-trivial cycle. Since *gprof* only counts the number of traversals of each arc in the call graph, but records no information about longer chains of calls, the best it can do is to lump together all the routines in each strongly-connected component of the call graph (which it calls a 'cycle' in the report), aggregating the time taken by the routines themselves and the time taken by all routines they call, and re-charging these times according to the number of calls along arcs that lead into the component. Functional and object-oriented programming styles naturally lead to programs with mutual recursion, since both encourage the development of recursive data structures and recursive subroutines to traverse them.

Note that it is not necessary for mutual recursion to occur dynamically for *gprof* to suffer this problem; that is, in the case of two routines A and B, it is not necessary that a situation ever arises where A calls B and then B makes a recursive call to A. The problem occurs whenever there is a cycle in the dynamic call graph. If sometimes A calls B, and at other times B calls A, then a cycle exists in the call graph, and *gprof* will lump A and B together in its report.

The problem we address in this paper is how to improve the accuracy of the data produced by a tool like *gprof* without increasing too much the runtime overhead of profiling. This overhead consists largely of the time spent running instrumentation code that is added to the program in order to collect timing and call-count data during execution. To a certain extent, this time can be excluded from the final statistics and thus ignored, unless the overhead is so high that it makes the runtime of the profiled program impractically long. However, instrumentation code may also have an effect on the memory consumption and cache performance of the program being analysed, and these effects are less easy to compensate for.

2 Tracking the call stack

The need for the average time assumption and the confusion that results from mutual recursion could both be eliminated by recording more data during execution. In the extreme, we could record the entire state of the call stack at each subroutine call or return; this would then allow us to find which subroutines are active at each point and charge execution time to them. This proposal, however, would generate huge quantities of profiling data, and the time required to record this data would swamp the real execution time of the program. Nevertheless, this complete set of profiling data provides the ideal at which we should aim.

In [11], Ponder and Fateman propose a scheme with a global clock and a timer associated with each routine. When a routine is called, the timer is decremented by the current clock time; when it exits, the timer is incremented by the clock time, so that the net effect is to increase the timer by the overall time spent in the routine. By keeping track of the number of activations of the routine, it is possible to deal with recursion by increasing the timer only for the outermost activation of each routine. This technique deals in a satisfactory way with both direct and mutual recursion, but it does not collect any information that relates timing to the call graph of the program. It is difficult to determine what methods are used by commercial profiling tools, but it seems likely that this technique is used by several of them.

Where time is measured by PC sampling, it is possible to record the state of the call stack only at timer ticks, which we expect to be much less frequent than calls or returns, or to traverse the stack and increment timers for each routine at each tick. This reduces the profiling overhead in a very useful way, but it cannot be used with a hardware cycle timer to get more fine-grained timing information. Also, unlike the other techniques we consider, it does not record complete information about numbers of calls that is useful in itself for debugging and performance evaluation. We do not consider this approach any further in this paper.

Instead of traversing the stack at each call or return, our proposal is to create a collection of *states* during execution. Initially, each state records the running routine and the set of all routines on the stack, although we shall add more information to the states later. The profiling code keeps track of which state describes the current layout of the subroutine stack, creating states dynamically as the need arises. At each tick of the profiling clock, we will increment a counter associated with the current state. Alternatively, we may use a hardware cycle counter by updating the counter for the current state whenever the state changes, adding to it the elapsed time since the previous change of state.

Because the set of routines active after a subroutine call can be obtained by adding the called routine (if necessary) to the set of routines active before the call, we can update the current state efficiently at a subroutine call by organizing the states into a finite-state machine, also adding transitions dynamically the first time they occur during execution. When a subroutine returns, we should reset the current state to the value it had before the subroutine was called, and this can be achieved by maintaining a stack of states in the profiler that is parallel to the ordinary subroutine stack.

At the end of execution, we can write to a file the list of states created during execution, saving for each state the set of active subroutines and the time spent in the state. This data can then be analysed to obtain an accurate account of the execution time during which each subroutine was active.

The success of this profiling method depends crucially on the number of different states that arise during execution. Potentially, each acyclic path in the call graph of the program can give rise to a different state, and the number of such paths may be very large for non-trivial programs. Thus a theoretical bound on the number of states grows very rapidly with the number of subroutines in the program. In practice, however, we have found that the number of states created during execution of real-life programs stays fairly small. After an initial period of rapid growth as the pattern of computation in the program emerges, the number of states typically becomes nearly stable, with subsequent periods of growth when the program enters a new phase of operation, such as when a compiler turns from analysis of the source program to optimization of the object code. Measurements of the number of states created for some typical programs are given in Section 6 of this paper.

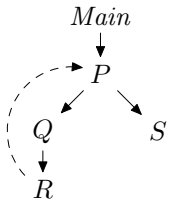

Figure 1: CCT for $Main \to P \to Q \to R \to P \to S$

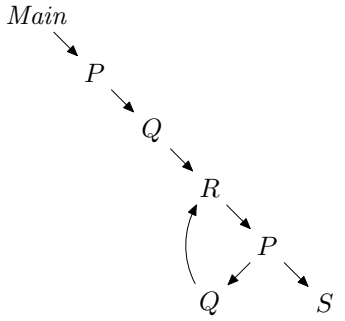

Figure 2: States for $Main \to (P \to Q \to R)^n \to P \to S$

We give in Section 7 an extended comparison of the proposed method of profiling with other techniques, but it is worthwhile to pause here and underline the difference between what we propose and the method of *calling context trees* (CCTs) presented by Larus and others in their PLDI'97 paper [1]. In both methods, direct recursion is handled very simply and does not lead to a change of profiler state. In building a CCT, mutual recursion is detected by finding an ancestor of the current state in the tree that refers to the same subroutine as is now being called. The call is handled by adding a back-edge to the tree and returning to the state previously created. Thus after a call sequence $Main \to P \to Q \to R \to P$, there will be a path $Main \to P \to Q \to R$ in the CCT, and the profiler returns to the entry for P in this path. Another subroutine S that is called from the recursive invocation of P will add a child labelled S to this state for P (see Figure 1). Time spent in executing S can be attributed also to P and to $Main$, but the fact that Q and R are active has been lost.

Our approach is more extravagant, because after the recursive call $R \to P$, the set of active routines, $\{Main, P, Q, R\}$, is different from the set $\{Main, P\}$ that was active at the previous invocation of P. Therefore a new state is created for the recursive call, and another new state is created when P calls S. In this program, each subroutine call creates a new state; but if the cycle of recursion $P \to Q \to R \to P$ is traversed multiple times, only one additional set of states is created, because the second time around the cycle, the set of active routines is the same as the first time (see Figure 2). Thus while our method results in more states than Larus', there are no more than twice as many for this pattern of recursion.

3 Recording arcs

So far, we have concentrated on recording the set of active routines, so that time can be charged to them accurately;

but it is also useful to record the caller of each active routine and the routine it calls, so as to compile a profile that identifies the important arcs in the call graph.

To achieve this, we make two extensions to the information recorded in each state. First, the active routines are recorded as an ordered list with no duplicates, by taking the subroutine stack and eliminating all but the most recent activation of each routine. Second, we record for each routine both the incoming and the outgoing arc in the call graph.

A compact way to represent this information is a list of subroutine addresses, each with a one-bit mark. The list of active subroutines is represented by unmarked entries in this list, and each active subroutine is surrounded if necessary by marked entries, so that its incoming and outgoing arcs are recorded. In many execution environments we can find a spare bit in each subroutine address to hold the mark bit, so that each history entry is a single word.

When a subroutine is called, we can compute the new history by appending the subroutine to the end of the current history and marking any occurrence of the subroutine that is already in the history. Following this, any marked entries in the history that are surrounded by other marked entries may be deleted, as they are not needed for the purpose of recording incoming or outgoing arcs of surrounding routines.

It is not necessary to perform this computation of the new history on every subroutine call, because as before we can construct a finite-state machine whose states are labelled with histories, adding states and transitions to the machine as needed during execution. The current state of the machine may be represented by a pointer to a state record, and its transition function may be implemented as a hash table, with a key consisting of the current state and the address of the subroutine being called.

In the *P*–*Q*–*R*–*S* example of Figure 2, the histories that are created are those that lead up to $Main \to P \to Q \to R$, then further histories as follows, where square brackets denote marked entries:

1. $Main \to [P] \to Q \to R \to P$.
2. $Main \to [P] \to [Q] \to R \to P \to Q$.
3. $Main \to [P] \to [R] \to P \to Q \to R$. (Here a marked entry for Q is elided.)
4. $Main \to [P] \to Q \to R \to P$. (Here the sequence $[P] \to [R] \to [P]$ is truncated to $[P]$.)
5. $Main \to [P] \to Q \to R \to P \to S$.

Observe the history 4 is the same as history 1, so that a cycle of transitions is created in the finite state machine. The marked entries allow us to identify the outgoing and incoming arcs of each active routine: for example, in history 4 we can see that work is being done on behalf of *Main* because it called P, and that Q was invoked by P – although these two invocations of P may not be the same.

The decision to keep the most recent activation of each routine means that the time spent by routine S in state 5 will be charged to the arc $P \to S$ rather than the arc $P \to Q$, as it would be if the earlier activation of P had been kept instead. This decision means that, when a subroutine P calls itself by indirect recursion, each activation of P is treated independently for profiling. Any time spent in other routines called directly or indirectly by an activation of P, but not through another activation of P, is charged to that activation and to outgoing arcs from it. This seems to us the most meaningful way to make sense of arc times in the presence of mutual recursion.

The actions of the profiler at a subroutine call are as follows. Suppose that the profiler is in state s when routine P calls routine Q. First, determine a new state s' for the profiler according to these rules:

1. If P and Q are the same, then this is a direct recursive call, and the new state s' is the same as the current state s.
2. Otherwise, use the hash table to discover whether a transition exists for calling Q from state s. If so, the transition specifies the new state s' that should be entered. This case covers nearly every call encountered during execution of typical programs.
3. If no such transition exists, compute the history list of the desired state as described above. Use a table of states to find whether a state s' with this history already exists. If so, create a new transition from s to s' for calling Q and add it to the hash table.
4. If the desired state does not exist, create it and create also a new transition, adding them to the tables.

In each case, the new state s' is pushed onto a stack of states, and subsequent timer ticks increment the time counter in s'.

On subroutine return, the stack of states is popped to reveal the state that was current before the subroutine was called. Non-local returns such as C's `setjmp/longjmp` mechanism can be accommodated by popping multiple states from the stack.

In our implementations of the profiler, the table of existing states is organized by linking together all state records for the same subroutine, and using the same hash table as is used for transitions to find the first state in the chain, given the subroutine Q that is being called.

4 Presenting the results

The data collected during execution can be post-processed in a number of ways to produce profiling reports. The simplest and most useful of these reports follow the format used by *gprof*: a flat profile showing the time spent in each routine, and a profile based on the dynamic call graph of the program. Both of these reports are easy to compute: we simply take each state recorded in the profiling data and charge the time spent in that state to the active routines in the history and to their incoming and outgoing arcs.

When a routine P calls itself indirectly, there will be states where P appears at the top of the history and a marked occurrence of P appears lower down. By slightly modifying the code for computing histories, we can arrange that an entry (marked if necessary) appears for the routine Q that was called by this marked occurrence of P, so that we know which of P's children led to the mutually recursive call of P. Then we can flag in the profiling display for P those children that lead to mutual recursion: we do this by adding a star to the name of these routines when they are shown as children of P.

For simple programs where there is no mutual recursion, some algebraic relationships exist between the different times shown in the call graph profile: the child time shown for a routine is the sum of the self and child times shown for outgoing arcs, whilst the self and child times for incoming arcs sum to the self and child times shown for the routine itself. Finally, the times shown for the arc $P \to Q$ are the

same when Q is shown as a child of P as they are when P is shown as a parent of Q.

These relationships cannot continue to hold when a program has mutual recursion, because the child times shown for outgoing arcs from a subroutine P must exclude any time spent running recursive calls of P as a subroutine of the routines it calls; otherwise, that time would be accounted for twice. Our preference is to maintain the internal consistency of the profiling data shown for each routine, but to allow the times shown for an arc $P \rightarrow Q$ to differ when it is shown as an outgoing arc of P and as an incoming arc of Q, so that the times for the incoming arc include time spent running P as a direct or indirect subroutine of Q, but the times for the outgoing arc do not. This seems to us the most meaningful way of dealing with this problem.

To implement this solution, we associate *two* sets of self and child times with each arc $P \rightarrow Q$ in the call graph: one set pertains to the 'outgoing arc' from P and the other to the 'incoming arc' to Q. In processing a profiling state, we add the time spent in that state to the first set of times for each arc $P \rightarrow Q$ if P is unmarked in the state, to the second set if Q is unmarked, and to both sets if both P and Q are unmarked. The time in the state is added to one or both self times if the active routine in the state is Q, and to one or both child times otherwise. If the program has no mutual recursion, then the two sets of times will be identical.

Much more information is contained in the profiling data than is shown in the *gprof*-style output format. As always, the problem is to present the data in a form that is accessible to programmers without overwhelming them with a mass of details. One useful form of display shows all the cycles in the call graph that were traversed during execution; these cycles are represented by states where the running routine P also appears as a marked entry lower down the history. Unlike *gprof*, our profiler can detect those cycles that actually occurred during execution, rather than those that simply exist in the dynamic call graph.

It would be interesting to experiment with interactive, graphical presentations of the data, but here we mention only one possibility that is very easy to implement. By allowing a single routine or a group of routines to be specified, and accumulating data from only those profiling states where one of the specified routines is active, it is possible to generate a profile just for those routines and their descendants.

5 Implementation

We have made two implementations of the profiling scheme we have described. The first was part of a bytecode-based implementation of Niklaus Wirth's programming language Oberon–2 [10, 12, 14]. It is easy to add a counter to the bytecode interpreter that is incremented for each instruction executed, thus providing a cycle counter in firmware. More recently, Andreas Sorensen and the author have made an implementation that can be used as a drop-in replacement for *gprof* under Linux. It uses a feature introduced in version 2.95 of the GNU C compiler where the compiler inserts calls to monitoring routines at the entry and exit of each C function. These monitoring routines receive as arguments the addresses of the call site and the called routine.

Linux does not directly provide the `profil()` system call of Unix, but implements it as a library routine using an interval timer that sends a Unix signal periodically. This implementation is inferior to one that provides `profil()` directly, since each timer tick requires additional context switches to deliver the signal, whereas in a direct implementation of `profil()`, the kernel can increment a counter in the user address space without the need for a context switch.

In our profiler, we do not need the complication of histogram-based profiling, but just a counter that is incremented on each tick. We therefore used the Linux `setitimer()` system call directly, installing a signal handler that just increments a counter in the current state. On a Unix system that provides `profil()` as a system call, a better alternative would be to set up a trivial histogram that treats the entire program as a single bin, thereby getting the effect of a single counter.

At the end of execution, our profiling code writes to the file `amon.out` in the current directory information about each state, including the timer, the call count and the history list. A separate analysis program reads this file and produces a report, using the same format as *gprof*. We used the GNU Binary File Descriptor library `libbfd` to read the symbol table of the program and translate the entry points recorded in the profiling data into the names of subroutines.

6 Results and evaluation

In order to assess the overhead caused by our profiling technique, we compared the timings of our profiler with *gprof* on three applications:

- *TEX* is Donald Knuth's well-known typesetting program. In the trials, we used the command

  ```
  tex texbook.tex
  ```

  to run TEX on the text of its own manual. The trials used pre-computed format and font files.

- *GCC* is the GNU C compiler; we profiled it using the command

  ```
  cc1 combine.i -O3 -o combine.s,
  ```

  where `cc1` is the main part of the C compiler (without the C preprocessor and the driver program that is usually invoked as `gcc`), and `combine.i` is the file obtained by pre-processing `combine.c`, the largest source file of GCC itself.

- *pProlog* is a small interpreter for a dialect of Prolog, described in the book [13]. We translated the source of the interpreter from Pascal to C using a home-grown translator, then compiled the result with GCC. For the trials, we used the command

  ```
  pprolog queens.pp,
  ```

  where `queens.pp` is a Prolog program that finds all solutions to the '9 queens problem'. We included this program because of its high density of recursive subroutine calls.

All these programs were statically linked with the GNU C library. In each case, the library code was compiled for profiling in the same way as the application source. The compiler throughout was GCC with optimization level `-O2`.

Table 3 shows various statistics related to the size of the programs and the data files generated in profiling. The columns of the table give the following information:

- *Text size* gives the size of the text segment for the program, without instrumentation code.

Program	Text size	Functions	Calls (millions)	States	Transitions	Depth (ave)	Depth (max)	Profiling memory	Size of `amon.out`	Size of `gmon.out`	*gprof* cycles
TEX	0.57 MB	456	32.6	8286	8355	9.30	23	0.66 MB	0.44 MB	0.27 MB	5
GCC	1.97 MB	1703	61.2	55437	57777	13.13	29	5.24 MB	3.79 MB	0.94 MB	15
pProlog	0.32 MB	115	47.6	435	443	10.31	19	0.07 MB	0.02 MB	0.14 MB	2

Table 3: Data sizes

Program	Bare	*aprof*			*gprof*			Dummy	
		charged	total	change	charged	total	change	total	change
TEX	4.20 s	6.09 s	8.20 s	(+95%)	4.34 s	9.89 s	(+135%)	5.74 s	(+37%)
GCC	10.18 s	13.68 s	22.24 s	(+118%)	10.20 s	19.87 s	(+95%)	12.94 s	(+27%)
pProlog	4.00 s	5.77 s	7.83 s	(+96%)	3.77 s	11.57 s	(+190%)	5.53 s	(+38%)

Table 4: Execution times

- *Functions* gives the number of functions listed by our profiler as having been called at least once.
- *Calls* gives the number of function calls traced by our profiler.
- *States* and *Transitions* give the number of profiling states and transitions created.
- *Depth* gives the average and the maximum number of entries (both marked and unmarked) in the history lists of profiling states.
- *Profiling memory* gives the total amount of memory allocated for storage of profiling states and transitions. This figure does not include the hash table for transitions (0.5 MB) and the profiling stack (64 KB).
- *Size of* `amon.out` and *Size of* `gmon.out` give the sizes of the files of profiling data written by our profiler and by *gprof* respectively.
- *gprof cycles* gives the number of 'cycles' in the call graph identified by *gprof*.

Table 4 shows the execution times for the benchmarks under various conditions:

- *Bare* refers to the time taken by the benchmark without profiling.
- *aprof* and *gprof* refer to the time taken by the benchmark using *gprof* and our profiler respectively.
- *Dummy* refers to the time taken when the benchmark is compiled to call monitoring routines at function entry and exit, but linked with routines that have empty bodies; this allows us to assess the overhead of calling the instrumentation routines.

For *gprof* and *aprof*, the column labelled *charged* gives the total time charged by the profiler to the running program, and the column labelled *total* gives the total runtime. The percentage increase of the total time with respect to the time for *bare* is also shown. All these timings were obtained by running the programs 10 times on a Pentium-III machine with 256 MB of memory and a clock speed of 500 MHz and taking the mean user time as reported by the Linux `time` command. In all cases, the execution times were extremely consistent between different runs of the programs.

These data allow us to draw some encouraging conclusions about our profiling method in comparison with *gprof*. The number of states created is much higher for GCC than for the other two programs, and this comes about because there are complex patterns of mutual recursion in parts of GCC that perform various sorts of analysis of the program being compiled. Nevertheless, the average depth of a history remains moderate, and both the amount of memory needed to store the states and transitions and the size of the resulting data file are reasonable by modern standards. In each case, the profiling overhead in execution time is comparable with that for *gprof*, and in two cases is rather smaller. This reflects the speed gained by using a hash table to find transitions that have already been created. The figures reveal that existing transitions cater for all but 57777 of the 61.2 million function calls performed by GCC, i.e. for 99.9% of the calls.

It is disappointing to see that the time charged by *aprof* is so much greater than the running time of the bare program. The increase is approximately the same as the extra time needed to run the program with dummy monitoring routines, which suggests that the problem is caused by profiling ticks that occur during the calling sequence of these routines. We are careful to set a flag on entry to the monitoring routines that prevents ticks from being charged during the search for an existing transition and the creation of new states, but this does not disable ticks that occur while the calls are being prepared.

Compared with *gprof*, our approach suffers more overhead because a call to a monitoring routine must be inserted at both the entry and the exit of each profiled routine, whereas *gprof* must monitor only entry to a routine in order to accumulate arc counts. In addition, the profiling interface provided by GCC passes more arguments to the monitoring routines than under *gprof*, so that the code for each call takes more time. It seems likely that the effect of this problem could be lessened by setting the flag to disable profiling inline before the call sequence. We could also adopt a simpler profiling interface in order to reduce the cost of the calls. Both these improvements could be made by building a special version of the compiler, or by editing the object code, but we have yet to try them.

In fact, the code for monitoring subroutine exit is very simple: it just pops the stack of states. This code could be inlined in order to reduce the profiling overhead. It may also be worth considering inlining at least part of the monitoring code for subroutine entry. We expect that the majority of subroutine calls will use an existing transition in the finite-state machine, and that the majority of these transitions will be found by a single probe of the hash table. By inlining the

code for this commonest case, we might be able to reduce the profiling overhead further, at the expense of a moderate increase in code size. Care is needed, however, lest the increase in code size interfere with the cache behaviour of the program.

7 Related work

James Larus and his collaborators have published a series of papers on different forms of profiling. Much of this work concerns measurement at a finer granularity than that considered here, so that the path of control flow within subroutines is recorded. The emphasis is more on collecting data that could be used for branch prediction in a compiler than on producing a report for programmers studying the behaviour of the program. We discuss here only the work that concerns inter-procedural profiling. We have already mentioned (in Section 2) the methods presented in the PLDI'97 paper [1]. This paper introduces the idea of building a data structure to record context information dynamically during execution, and encourages the belief that the size of such data structures may reasonable even when profiling large programs.

In a more recent PLDI paper [8], Larus proposes a profiling scheme that tracks the execution path of the whole program and uses a compression algorithm on program paths to reduce the volume of data collected. Our histories can also be viewed as a kind of lossy compression scheme for program paths, in contrast to the lossless compression used by Larus, in which a simple context-free grammar for the paths is generated during compression. This compression scheme, though asymptotically linear in the size of the paths, is nevertheless much more expensive in time than the method based on histories.

The profiling techniques described in this paper are similar to those investigated for lazy functional programming by Stephen Jarvis in his Ph.D. thesis [7, 9]. Jarvis discusses the use of *cost-centre stacks* to record the execution context of a program, and these are similar to the histories used in this paper. Our approach differs in the representation chosen for histories, in the use of a finite-state machine with a fast transition function, and in the recording of information about incoming and outgoing arcs that enables the profiling report to mimic that produced by *gprof*.

Jarvis aims to represent histories more compactly by using a pointer-linked scheme where each history is represented by a node that contains one routine address and a pointer to the preceding history. In most cases, new histories can be created by adding a single new node. However, detection of mutual recursion requires a complete traversal of the chain of ancestor links, and this traversal may be expensive in cache misses. When mutual recursion causes the elision of ancestors in the history, it may be necessary to create many new nodes all at once. The data we have presented indicates that the depth of histories remains moderate, so that representing the entire history explicitly in each state does not have an unacceptable cost in memory, and we can expect much better cache performance in scanning an array of routine addresses compared with traversing a chain of pointers.

Conway and Somogyi [3] describe a sophisticated profiling scheme that they have implemented for the higher-order logic programming language Mercury. Their scheme depends on an analysis by the Mercury compiler of the call graph of the program, which discovers among other things the cycles of mutually recursive functions in the program, and produces efficient instrumentation code. Because Mercury has higher-order features, this analysis is necessarily partial, since the value of procedure-valued variables cannot be known at compile time. The profiling scheme can also handle mutual recursion that arises through calls to such procedures, but generates instrumentation code that is very efficient for the common case of calling a known procedure with a known pattern of recursion.

Techniques such as those proposed by Conway and Somogyi have the potential to reduce dramatically the cost of profiling. Nevertheless, they rely on compiler support, and we believe that the simpler techniques proposed in this paper also have their place in programming practice.

8 Conclusions

We have presented a profiling technique that can be used in place of that used in profilers like *gprof*. At comparable cost in execution time, the new technique gathers accurate data about the relationship between execution time and paths in the call graph. Although the data collected provides much more information about the behaviour of the program being profiled, the volume of data in terms of the amount of memory used for profiling and the size of the resulting file of profiling data remains moderate.

The new technique can be used to produce an accurate profile in the same format as that output by *gprof*, and can also be used to produce reports that focus on a particular routine and its children, or as the basis for an interactive exploration of the distribution of execution time with respect to the call graph of the program.

The new technique deals with mutual recursion without lumping together families of mutually recursive routines, and can give an accurate report of which paths of mutual recursion were actually taken during execution. This makes it especially attractive for assessing the performance of higher-order and object-oriented programs, where families of mutually recursive routines are common.

Acknowledgments

The author is grateful to Andreas Sorensen for his careful and thorough implementation of the ideas described in this paper.

References

[1] G. Ammons, T. Ball, and J. Larus, 'Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling', in *Proceedings of PLDI'97*, June 1997

[2] T. Ball and J. Larus, 'Efficient path profiling', MICRO–29, December 1996.

[3] T. C. Conway and Z. Somogyi, 'Deep profiling: engineering a profiler for a declarative programming language', Technical report, Department of Computer Science and Software Engineering, University of Melbourne, Australia.

[4] J. Fenlason and R. Stallman, *GNU gprof: the GNU profiler*, Free Software Foundation, 2000.

[5] S. Graham, P. Kessler and M. McKusick, 'An Execution Profiler for Modular Programs', Software – Practice and Experience, **13**, pp. 671-685, 1983.

[6] S. Graham, P. Kessler and M. McKusick, 'gprof: A Call Graph Execution Profiler', in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, SIGPLAN Notices, **17**(6), pp. 120-126, June 1982.

[7] S. A. Jarvis, *Profiling Large-Scale Lazy Functional Programs*, Ph.D. Thesis, Department of Computer Science, University of Durham, 1996.

[8] J. Larus, 'Whole program paths', in *Proceedings of PLDI'99*, May 1999.

[9] R. G. Morgan and S. A. Jarvis, 'Profiling Large-Scale Lazy Functional Programs', Journal of Functional Programming **8**(3), May 1998.

[10] H. Mössenböck, *Object-oriented programming in Oberon–2*, Springer-Verlag, Berlin, 1993.

[11] C. Ponder and R. J. Fateman, 'Inaccuracies in program profilers', Software – Practice and Experience, **18**, pp. 459–467, 1988.

[12] M. Reiser and N. Wirth, *Programming in Oberon: steps beyond Pascal and Modula*, ACM Press, New York, 1992.

[13] J. M. Spivey, *An introduction to logic programming through Prolog*, Prentice-Hall International, 1995.

[14] J. M. Spivey, 'The Oxford Oberon–2 compiler', `http://spivey.oriel.ox.ac.uk/mike/obc`.