SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **00**:1–99 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Fast, accurate call graph profiling

J. M. Spivey

Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD
mike@comlab.ox.ac.uk

SUMMARY

Existing methods of for call graph profiling, such as that used by *gprof*, deal badly with programs that have shared subroutines, mutual recursion, higher-order functions, or dynamic method binding. This article discusses a way of improving the accuracy of a call graph profile by collecting more information during execution, without significantly increasing the overhead of profiling. The method is based on keeping track of a *context*, consisting of the set of subroutines that are active at a particular moment during execution, together with the calling arcs between these subroutines. The profiler records the time spent in each context during execution of the program, and thus obtains an accurate measurement of the total time during which each subroutine was active. By recording arc information for only the most recent activation of each subroutine, it is possible to arrange that even recursive programs give rise to a finite number of these contexts, and in typical real programs, the number of distinct contexts remains manageably small. The data can be collected efficiently during execution by constructing a finite-state machine whose states correspond to contexts, so that when a context is entered for a second or subsequent time, only a single access of a hash table is needed to update the state of the profiling monitor.

KEY WORDS: call graph profiling

Introduction

Profiling is a family of techniques for gathering information about the behaviour of a program during execution, and especially for recording the amount of time spent in each part of the program. The data can be used to help programmers identify performance bottlenecks, and also to guide optimising compilers. This article focusses on the use of profiling as a programming tool, and on measuring the time taken by each subroutine in the program, rather than on the finer-grained data that is often collected for use by a compiler.

Call-graph profiling involves relating the timing information to the call graph of the program. In one popular way of presenting a call graph profile, used by the *gprof* profiler [1, 2, 3], the time taken by each subroutine is also shared out among its callers, so that programmers can

get a good idea of how total execution time is divided among the major tasks carried out by a program.

In order to minimise the time overhead imposed by profiling, *gprof* adopts a scheme where the only times that are directly measured is the total time spent in executing each subroutine, which *gprof* reports as the *self* time of the routine. The profiler also adds instrumentation code to count the number of times each arc in the call graph is traversed; that is, the number of times each routine calls each of its subroutines. After execution is over, the self time of each routine is shared out among its callers as *child* time. This is done approximately by sharing out the time in proportion to the number of calls on each arc in the call graph. This procedure gives reasonably accurate results if every call of a subroutine takes approximately the same time, or at least if the average time taken by a subroutine is the same for each caller: this is called the *average time assumption*. However, inaccuracies will occur if the running time of some subroutines depends on the values of their arguments, and the arguments are drawn from different distributions in different calls of the same subroutine. Reference [4] discusses this problem at length.

The average time assumption is particularly likely to be misleading in programs written in a functional or object-oriented style. Programs written in a polymorphic higher-order language such as ML are likely to use standard higher-order functions for many unrelated purposes. For example, the function *map f xs* applies the function f to each element of the list xs, gathering the results into a new list. The time taken by this depends on both the time taken by each call of the function f and the length of the list xs, and both these are likely to vary significantly from one caller to another. Serious inaccuracy will result when *gprof* gathers together all the time taken by the various functions f that are passed to *map* and shares this time among the callers of *map* indiscriminately.

Similarly misleading profiles can result from common 'design patterns' in object-oriented programming. For 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 *app* 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 *app*, 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 different buttons are bound to commands with different costs, then the average time assumption leads to seriously inaccurate results.

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, and the dynamic nature of subroutine calls in both styles can lead to dynamic mutual recursion that is not detectable in the program text.

The problem addressed in this article 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 subject program, and these effects are less easy to assess and compensate for. It seems reasonable to take the time overhead of profiling as a rough guide to the likely size of these effects. As will be shown towards the end of the article, initial results indicate that the proposed technique has an overhead comparable with that of *gprof* for typical programs.

The techniques described in this article are the subject of U.K. and U.S. patent applications.

Tracking the execution context

It is clear that the need for the average time assumption and the confusion that arises from mutual recursion could both be eliminated by recording more data during execution of the program. In the extreme, we could record the entire state of the program's subroutine stack at each subroutine call or return; this would then allow us to find which subroutines were active at each point in execution, and where they were called from, and charge the elapsed time to them with perfect accuracy. This proposal, however, would generate huge volumes of profiling data, and the time required to record this data would completely swamp the execution time of the program. Nevertheless, this complete set of profiling data provides an ideal at which we should aim.

Instead of traversing the call stack at each call or return, an alternative is to create a collection of *contexts* during execution. Each context records the identity of the running subroutine, and the *set* of subroutines that are active. It also records some further information about arcs that will be discussed later. The profiling monitor keeps track of the current context, and creates new contexts dynamically as the need arises. Each tick of the profiling clock increments a counter associated with the current context. In interrupt-driven profiling, the counter is incremented directly at each interrupt. Alternatively, a hardware cycle counter could be used by updating the counter for the current context each time the context changes, increasing it by the elapsed time since the previous change of context.

The key idea is that the context does not record the entire layout of the subroutine stack, so that there may be many stack states that correspond to each context, and even recursive programs need only a finite number of contexts to describe their execution. In theory, the number of such contexts, though finite, may be very large; practical experiments indicate, however, that the number remains manageable for real example programs.

At each subroutine call, it is necessary to compute the context that applies when the newly-called subroutine is added to those already active, and this may cause the creation of a new

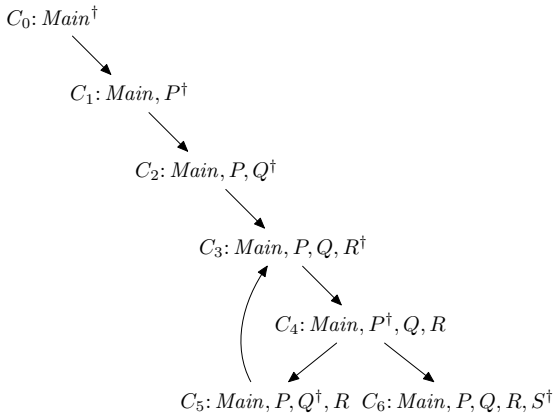

Figure 1. Contexts for $Main \to (P \to Q \to R)^n \to P \to S$

context if it has not occurred before during execution of the program. When a subroutine returns, the context should be reset to the value it had before the call, and for this purpose the profiler maintains a stack of contexts that shadows the subroutine stack of the program. At the end of execution, the list of contexts created during execution is written to a file, saving for each context the set of active subroutines and the time spent in the context. This data can then be analysed to obtain an accurate account of the execution time during which each subroutine was active.

Consider a program that has a main routine *Main*, together with three mutually recursive subroutines P, Q and R that call each other in the pattern $P \to Q \to R \to P$, and another subroutine S that is called by P. During the execution of the program, there may be many copies of P, Q and R on the subroutine stack, but there will only be a small number of distinct contexts, as shown in Figure 1. Each context shows the set of active routines, with the routine that is actually running identified by a dagger sign †. As the first few subroutine calls are made, in the sequence

$$Main \to P \to Q \to R,$$

a new context is added for each call; these contexts are labelled C_1, C_2, C_3 in Figure 1. When R calls P recursively, this leads to another new context C_4; it is different from C_1 even though P is running in both, because in C_1 only *Main* is also active, but in C_4 subroutines Q and R are active. A recursive call of Q from P leads to another context C_5, but a recursive call of R from Q leads back to C_3, because the set of active routines remains the same. Thus the finite number of contexts shown in Figure 1 is sufficient, however many times the cycle of mutual recursion is traversed.

In addition to information about the set of active subroutines, we would also like to record some information about the arcs by which one subroutine called another. To see what information must be gathered, we should consider what is necessary in order to produce a

gprof-like report that attaches time to arcs as well as routines, but (unlike *gprof*) does so in the presence of mutual recursion. In simple, non-recursive programs, it is sufficient simply to record the incoming and outgoing arcs for each active subroutine; thus context C_2 in Figure 1 might become

$$C_2 \colon [Main]P, Main[P]Q, P[Q^\dagger],$$

where the *item* $A[B]C$ denotes that the subroutine B is active with incoming arc $A \to B$ and outgoing arc $B \to C$. Naturally, the main program has no incoming arc, and the running routine has no outgoing arc.

As with *gprof*, direct recursion can be identified and treated specially, so we should next consider the effects of mutual recursion. Suppose a procedure A has been called recursively via B, so that the call stack looks like this:

$$Main \to \ldots \to A \to B \to \ldots \to A \to C \to \ldots \to D.$$

Thus A has called B, and this has lead to a recursive call of A. That invocation of A has in its turn called another routine C, and routine D is running, called directly or indirectly by C. Clearly, the execution time spent in this context should be charged as *self* time to D, and as child time to the other active routines, including *Main*, A, B and C, just once to each. The time must be also be charged to the active arcs in some way, and that means we must choose between charging it to the outgoing arc $A \to B$ from A or the outgoing arc $A \to C$. The choice made in our profiler is to attribute this time to the arc $A \to C$, i.e., to the outgoing arc from the most recent activation of the subroutine. It is necessary to know of this convention in order to make a detailed interpretation of the profiling results that are produced, but in fact the convention helps in answering natural questions. For example, if C is a low-level subroutine that is not mutually recursive with A, we might want to know what effect would be produced by eliminating the call from A to C and replacing it with faster special-case code. With these conventions, an upper bound for the time saved can be obtained by looking at the time associated with the arc $A \to C$, and this is true whether or not A is a routine that is involved in mutual recursion.

The convention is that arc information is recorded only for the *most recent* activation of each subroutine. Returning to the P–Q–R example, and adding arc information to the context, we obtain the set of contexts shown in Figure 2. One more context has been added, in that the context C_3 of Figure 1 has become two contexts C_3 and C_6 here. Those two contexts differ because one contains $Main[P]Q$ and the other contains $R[P]Q$. But again, a finite number of contexts suffices, however deeply nested the mutual recursion may become.

The contexts shown in Figure 2 are represented in a redundant way, since (reading from left to right) an item $A[B]C$ is often followed by another item $B[C]D$ that overlaps with it. As we shall see in the next section, there is a compact representation for contexts that removes this redundancy, representing the whole context by a partial copy of the subroutine stack. Even in that representation, however, a context remains a *set* of active subroutines. When a subroutine is called, a new item is added to the context, but (unlike a stack) that can cause other items to disappear: for example, in the transition from context C_3 of Figure 2 to context C_4, the item $Main[P]Q$ is deleted as the item $R[P^\dagger]$ is added.

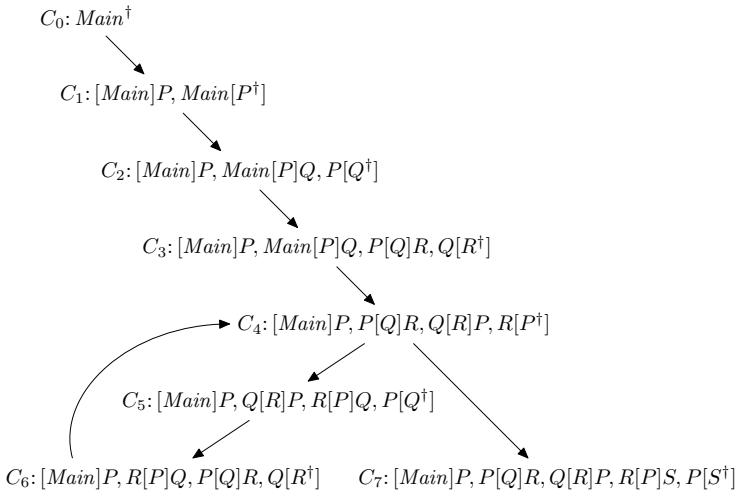

Figure 2. States for $Main \to (P \to Q \to R)^n \to P \to S$ with arcs

Data structures for fast profiling

In implementing the profiling scheme outlined in the previous section, there are two main problems to solve:

1. How to represent contexts compactly, and in such a way that the new context after a subroutine call can be efficiently computed from the context before the call, without having to traverse the subroutine stack of the program.
2. How to organise the contexts that have been created thus far, so as to find quickly at each subroutine call whether an existing context can be used, or whether a new one must be created.

To address the first of these problems, each context is represented by a *history*, a partial copy of the subroutine stack with irrelevant entries eliminated. A history is a sequence $s = \langle s_1, s_2, \ldots, s_n \rangle$ of subroutines, where a subroutine may be represented by the address of its entry point. The idea is that the context contains items of the form $s_{i-1}[s_i]s_{i+1}$. Let us adopt the convention that $s_0 = s_{n+1} = \Lambda$ for some fictitious value Λ, and that $\Lambda[B]C$ denotes the item $[B]C$ and $A[B]\Lambda$ denotes $A[B^\dagger]$. Then the context

$$[Main]P, Main[P]Q, P[Q]R, Q[R^\dagger],$$

(context C_3 of Figure 2) is represented by the sequence

$$\langle Main, P, Q, R \rangle,$$

i.e., by a copy of the subroutine stack.

A difficulty arises when (because of mutual recursion) the entire contents of the stack has not been kept: for example, context C_4 of Figure 2 contains the items

$$[Main]P, P[Q]R, Q[R]P, R[P^\dagger],$$

but does not contain the item $Main[P]Q$. Such contexts may be dealt with by introducing a sequence $m = \langle m_1, \ldots, m_n \rangle$ of *mark bits*, and adopting the convention that an item $s_{i-1}[s_i]s_{i+1}$ is part of the context only if $m_i = T$, and not if $m_i = F$. If we represent the condition $m_i = T$ by underlining s_i, then the context above is represented by the sequence

$$\langle \underline{Main}, P, \underline{Q}, \underline{R}, \underline{P} \rangle.$$

Formally, the context represented by the sequences s and m is

$$\{\, s_{i-1}[s_i]s_{i+1} \mid 1 \le i \le n, m_i = T \,\}.$$

Given this representation of contexts, we must solve the problem of computing the new context after a subroutine call from the context before it. This is done in the following steps:

A1. [Direct recursion.] If $P = s_n$, where P is the procedure being called, then stop: the new context is the same as the old.
A2. [Add P.] Set $s_{n+1} = P$ and $m_{n+1} = T$, and increase n by 1.
A3. [Unmark previous call.] For each i in the range $1 \le i < n$, if $s_i = P$, then set $m_i = F$. [There will be at most one such i where $m_i = F$ does not already hold.]
A4. [Squeeze out redundant entries.] If $m_{i-1} = m_i = m_{i+1} = F$ for any i with $1 < i < n$, then delete s_i and m_i and decrease n by 1. Repeat until no further such deletions are possible.
A5. If $s_i = s_{i+1}$ and $m_i = m_{i+1} = F$ for any i with $1 \le i < n$, then delete s_i and m_i and decrease n by 1. Repeat until no further such deletions are possible.

Step A3 deletes any previous item for P from the context, whilst leaving an unmarked entry that provides outgoing or incoming arcs for the marked entries around it. This is the sole purpose of unmarked entries, and steps A4 and A5 remove unmarked entries that no longer serve this purpose, either because they are surrounded by other unmarked entries, or because they are adjacent to an identical unmarked entry. The effect of steps A3, A4 and A5 can be achieved in a single pass over the sequence s, so the time spent in computing the new context grows only linearly in the length of the sequence. The size of the context is related to the number of distinct subroutines that are active, and does not increase with the nesting of recursion.

In Figure 2, context C_4 gives rise to a call of Q. Following the algorithm above, we form the sequence

$$\langle \underline{Main}, P, Q, \underline{R}, \underline{P}, \underline{Q} \rangle$$

by adding $\underline{Q}$ at the end and unmarking the previous copy of Q. No deletions are possible at this stage, and the new sequence represents the context

$$[Main]P, Q[R]P, R[P]Q, P[Q^\dagger],$$

which is identical with context C_5 in Figure 2. Deletion does take place in the transition from context C_5 to context C_6. Here we add R and unmark a previous copy to obtain

$$\langle \underline{Main}, P, Q, R, \underline{P}, \underline{Q}, \underline{R} \rangle.$$

In this sequence, Q is redundant, and deleting it yields another sequence that also represents the items

$$[Main]P, R[P]Q, P[Q]R, Q[R^\dagger],$$

i.e., those of context C_6 in the figure.

At first sight, the deletion process seems to be an optimisation, aimed at saving a few words of storage in the representation of contexts. But in fact it is essential to the success of the method, because it prevents contexts from growing without bound, and allows us to form cycles in the graph of contexts, representing arbitrarily deep recursion with a finite number of contexts.

We now turn to the second problem, that of organising the existing contexts so as to avoid creating a new context unless it is necessary. Here the theme is one of making common cases as fast as possible. Two methods are used to achieve this: first, the contexts are made into the states of a finite-state machine M, with transitions labelled by subroutines, and second, there is a table of contexts that allows an existing context to be found, given the sequence that it contains.

Consider what happens when the profiler is in context C and subroutine P is called. We carry out the following steps:

B1. Consult the transition function of M to find whether there is a transition $C \xrightarrow{P} C'$ for some state C'. If so, then C' is the new context.

B2. If step B1 does not succeed, compute the history s for the new context using the algorithm presented above. Consult the table of contexts to find if a context C' exists containing s. If so, then add the transition $C \xrightarrow{P} C'$ to M, and return C' as the new context.

B3. If step B2 does not succeed, then create a new context C' containing history s, add the transition $C \xrightarrow{P} C'$ to M, and return C' as the new context.

In practice, these measures are astoundingly effective. Detailed experimental data is given later, but in no case was the number of transitions created more than 0.1% of the number of procedure calls executed. Thus step B1 succeeds in more than 99.9% of cases. Typically, the number of transitions created is only a little larger than the number of contexts, indicating that step B2 is only rarely successful, even when step B1 has failed. This reflects the fact that most programs have rather few cycles of mutual recursion. Nevertheless, step B2 is important because it creates the cycles in the graph of contexts that are made possible by the deletion process in the creation of histories.

It may be simpler to view the technique described in steps B1 to B3 as memoizing the function defined by steps A1 to A5 above. There is an important respect, however, in which it is fruitful to view the contexts as forming a finite-state machine. This is the observation that, though the context is a function of the contents of the subroutine stack, the new context after a subroutine is called can be computed from the old context and the identity of the subroutine, which is crucial to the efficiency of the profiling scheme.

In summary, the data structures used by the profiling monitor are as follows:

- a collection of context records, each containing a history together with counters for how often the context was entered and how many direct recursive calls were made, and a timer that measures how long has been spent in the context.
- a hash table that represents the transition function of the finite-state machine, and also allows us to find all contexts where a given subroutine is running. This hash table is used to find the new context to enter on a subroutine call.
- a stack of contexts that shadows the subroutine stack of the program. When a subroutine is called, the new context is pushed onto this stack, and when the subroutine returns, the profiler returns to its previous context by popping the stack.

The success of the profiling method depends crucially on the number of different contexts that arise during execution. Potentially, each acyclic path in the call graph of the program can give rise to a different context, 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, 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.

Presenting the results

The data collected during execution can be post-processed in a number of ways to produce profiling reports. The simplest and perhaps the most useful of these reports follows 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, by simply taking each context recorded by the profiling monitor and charge the time spent in that context to the active routines and to their incoming and outgoing arcs.

For simple programs where there is no mutual recursion, some simple 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. It is possible to maintain the internal consistency of the profiling data shown for each routine, but to allow the times shown for an arc $P \to Q$ 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 the most meaningful way of dealing with this problem.

To implement this solution, *two* self times, $self_1(P,Q)$ and $self_2(P,Q)$ are associated with each arc $P \to Q$ in the call graph, and also two child times $child_1(P,Q)$ and $child_2(P,Q)$. The times $self_1(P,Q)$ and $child_1(P,Q)$ pertain to the 'outgoing arc' from P and the other times to the 'incoming arc' to Q. A context where P is running is processed by examining each item $A[B]C$, and adding the time spent in the context to $self_1(B,C)$ or $child_1(B,C)$ depending on whether $C = P$, and to $self_2(A,B)$ or $child_2(A,B)$, depending on whether $B = P$. The items $[Main]C$ and $B[P^\dagger]$ are treated as special cases in the obvious way. When there is no mutual recursion, the two sets of times will be the same, because what is added to the first set of times for $A[B]C$ will be added to the second set for the next item $B[C]D$. In the presence of mutual recursion, one or other of these items may not exist, so differences between the two sets of times may occur.

The procedure just outlined allows us to compile a report in the style of *gprof*. However, much more information is contained in the profiling data than is shown in this 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 detail. 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 an unmarked entry earlier in the history. Unlike *gprof*, our profiler can detect those cycles that actually occurred during execution, rather than those that exist only in the call graph.

It would be interesting to experiment with interactive, graphical presentations of the data, but there is space here to 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, and thus examine just one part of a large program.

Implementation

We have made two implementations of the profiling scheme described in this article. The first was part of a bytecode-based implementation of the programming language Oberon–2 [5, 6, 7]. 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, we have made an implementation that can be used as a drop-in replacement for *gprof* under Linux, and (as will be seen later) imposes comparable overheads.

The native Linux implementation uses a feature of the GNU C compiler, whereby the compiler can insert calls to monitoring routines at the entry and exit of each subroutine. The GCC compiler was modified so that it inserted monitoring code better suited to the present purpose, eliminating some of the overhead of profiling. It is possible that further improvements could be made by tuning the instrumentation code at the assembly-language level, but we have not tried this yet.

The profiler represents the histories described earlier as arrays of entry addresses for subroutines. Because each subroutine is aligned at a multiple of 16 bytes, we were able to use a spare low-order bit in each entry address to represent the mark bit. The transition from one context to the next requires a data structure that represents the transition function of the finite-state machine M, and a table that allows a context to be found given its history. In the profiler, both these are represented using a single large hash table. The key of this table is a pair (C, P), where C is the address of a context record, and P is the entry address of a procedure. Each entry in the hash table gives the context C' after the transition $C \xrightarrow{P} C'$. This is the implementation of step B1 of the method given earlier. The same hash table gives access to all contexts in which a given procedure P is running, by looking up (Λ, P) in the hash table, where Λ is the null pointer; all contexts for procedure P are chained together, and a linear search finds whether the desired context already exists in step B2.

At the end of execution, the profiling monitor writes to the file *amon.out* in the current directory information about each state, including the timer, the call count and the history. 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 subject program and translate the entry points recorded in the profiling data into the names of subroutines. The profiler does not at present handle *setjmp*() and *longjmp*(), but there is no reason why it could not be extended do so.

Results and evaluation

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

GCC is the GNU C compiler, profiled 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.

TEX is Donald Knuth's well-known typesetting program, profiled using the command

```
tex texbook.tex,
```

to run TEX on the text of its own manual.

pProlog is a small interpreter for a dialect of Prolog, described in the book [8]. The source code of the interpreter was translated from Pascal to C using a home-grown translator, then the result was compiled with GCC. It was profiled using the command

```
pprolog queens.pp,
```

where *queens.pp* is a Prolog program that finds all solutions to the '9 queens problem'. The *pProlog* program is included here because of its high density of recursive subroutine calls.

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

Table I. Data sizes

Program	Bare	*aprof*			*gprof*			Dummy	
		charged	total	change	charged	total	change	total	change
TEX	5.30 s	6.15 s	10.16 s	(+92%)	5.32 s	12.37 s	(+134%)	6.86 s	(+30%)
GCC	12.48 s	14.17 s	27.61 s	(+121%)	12.71 s	24.81 s	(+99%)	15.64 s	(+25%)
pProlog	4.93 s	5.40 s	9.73 s	(+97%)	4.62 s	14.53 s	(+195%)	6.74 s	(+37%)

Table II. Execution times

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`. Higher levels of optimisation than this start to disturb the structure of the program by inlining, affecting the profiling results.

Table I 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.

Functions gives the number of subroutines listed by *aprof* as having been called at least once.

Calls gives the number of subroutine calls traced by *aprof*.

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 *aprof* and by *gprof* respectively.

Table II 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 our profiler and *gprof* respectively. For each of these, 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.

Dummy refers to the time taken when the benchmark is compiled to call monitoring routines at subroutine entry and exit, but linked with routines that have empty bodies; this allows us to assess the overhead of calling the instrumentation routines.

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 (step B1) cater for all but 57777 of the 61.2 million subroutine calls performed by GCC, i.e. for 99.9% of the calls.

The profiling reports generated by *aprof* allow us to identify subroutines where the average time assumption would give misleading results. There are many such subroutines in *gcc*, but the purpose of most of them means little to an outsider. We can, however, guess at the function of some of them. There is a subroutine *rest_of_compilation* that is always called via one of two other subroutines, *finish_function* and *output_inline_function*. In the trial, the second of these parent routines accounted for 13 of the 76 calls (17%), but only 0.20 seconds of the 7.62 seconds (2.6%) spent in the subroutine and its children.

There are several other similar imbalances associated with specialised routines that each involve around 5% of the total runtime, but particularly interesting is the routine *splay_tree_splay*, where the parent *splay_tree_insert* accounts for 32% of the total runtime of 0.19 seconds, despite making only 18% of the calls. The other parent, *splay_tree_lookup*, thus accounts for disproportionately little of the total runtime, perhaps indicating that splay operations are more commonly needed during insertion than during lookup.

Also, the general-purpose sorting routine *qsort* is called from many places in the compiler, but the call from *global_alloc* consumes 26% of the total runtime of 0.19 seconds, despite making only 0.32% of the calls. Admittedly, the times involved in this example and the preceding one are close to the noise threshold of the profiling data, but nevertheless they illustrate the kind of conclusion about a program that is supported by the more comprehensive profiling information.

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 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, although this would require further changes to the C compiler. It may also be worth considering inlining at least part of the monitoring code for subroutine entry. The majority of subroutine calls use an existing transition in the finite-state machine, and the majority of these transitions will be found by a single probe of the hash table. By inlining the code for this commonest case, it might be possible 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 should interfere with the cache behaviour of the program.

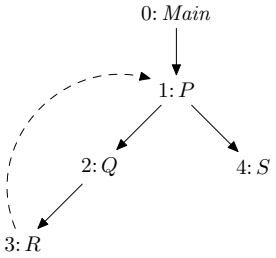

Figure 3. CCT for $Main \rightarrow P \rightarrow Q \rightarrow R \rightarrow P \rightarrow S$

Related work

Reference [4] proposes 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 are much less frequent than calls or returns, or to traverse the stack and increment timers for each routine at each tick (see [9]). 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 considered here, it does not record complete information about numbers of calls that is useful in itself for debugging and performance evaluation.

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.

A close comparison can be made between the method described here and the method of *calling context trees* (CCT's) [10]. 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 node 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. Figure 3 shows a CCT for the same program as Figures 1 and 2. Here,

the call sequence $Main \rightarrow P \rightarrow Q \rightarrow R$ results in the creation of a simple branch in the CCT, leading to the node labelled 3. The subsequent recursive call $R \rightarrow P$ introduces a back-edge to the ancestor node for P (labelled 1). Calling S from P adds another child to this node. Thus time spent executing S can be attributed to P and to *Main*, but the fact that Q and R are also active during this time is lost. However, the 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 be reasonable even when profiling large programs.

A more recent paper [11] 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. Larus' 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 article are similar to those described for lazy functional programming in References [12] and [13]. Building on earlier work [14], this reference 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*.

The cited paper aims to store 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 presented earlier indicate 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 much better cache performance can be expected in scanning an array of routine addresses compared with traversing a chain of pointers.

Reference [15] describes a sophisticated profiling scheme for the higher-order logic programming language Mercury. This 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 subroutines 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 in the paper 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 article also have their place in programming practice.

Conclusions

This article has 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 subject program, 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.

Acknowledgements

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

REFERENCES

1. 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.
2. S. Graham, P. Kessler and M. McKusick, 'An Execution Profiler for Modular Programs', Software – Practice and Experience, **13**, pp. 671-685, 1983.
3. J. Fenlason and R. Stallman, *GNU gprof: the GNU profiler*, Free Software Foundation, 2000.
4. C. Ponder and R. J. Fateman, 'Inaccuracies in program profilers', Software – Practice and Experience, **18**, pp. 459–467, 1988.
5. M. Reiser and N. Wirth, *Programming in Oberon: steps beyond Pascal and Modula*, ACM Press, New York, 1992.
6. H. Mössenböck, *Object-oriented programming in Oberon–2*, Springer-Verlag, Berlin, 1993.
7. J. M. Spivey, 'The Oxford Oberon–2 compiler', `http://spivey.oriel.ox.ac.uk/mike/obc`.
8. J. M. Spivey, *An introduction to logic programming through Prolog*, Prentice-Hall International, 1995.
9. M. Burrows, U. Erlingson, S.-T. Leung, M. Vandevoorde, C. Waldspurger, K. Walker and R. Weihl, 'Efficient and flexible value sampling', Research Report 166, Compaq Systems Research Center, Palo Alto, CA, October 2000.
10. G. Ammons, T. Ball, and J. Larus, 'Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling', in *Proceedings of PLDI'97*, June 1997
11. J. Larus, 'Whole program paths', in *Proceedings of PLDI'99*, May 1999.
12. S. A. Jarvis, *Profiling Large-Scale Lazy Functional Programs*, Ph.D. Thesis, Department of Computer Science, University of Durham, 1996.
13. R. G. Morgan and S. A. Jarvis, 'Profiling Large-Scale Lazy Functional Programs', Journal of Functional Programming **8**(3), May 1998.

14. P. M. Sansom and S. L. Peyton Jones, 'Formally based profiling of higher-order functional languages', ACM Transactions on Programming Languages and Systems, **19**, 1, pp. 334–385, 1997.
15. 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.