-

Profiling in the Presence of Optimization and Garbage Collection

*Andrew W. Appel**
Bruce F. Duba†
David B. MacQueen†

November 1988

ABSTRACT

Profiling the execution of programs can be a great help in tuning their performance, and programs written in functional languages are no exception. The standard techniques of call-counting and statistical (interrupt-driven) execution time measurement work well, but with some modification. In particular, the program counter is not the best indicator of ‘‘current function.’’ Our profiler inserts explicit increment and assignment statements into the intermediate representation, and is therefore very simple to implement and completely independent of the code-generator.

* Supported in part by NSF Grant CCR-8806121 and by a Digital Equipment Corp. Faculty Incentive Grant.

† AT&T Bell Laboratories, Murray Hill, NJ.

1. Execution profiling

A large program usually consists of many small functions. When such a program is to be tuned for efficiency, it is necessary to identify which of those functions are taking the bulk of the execution time. Then the commonly-used functions can be made more efficient, or called less often, or both. By using a theoretical analysis of the algorithms used in a program, such functions can be identified; but a complete theoretical analysis is complex and impractical for large programs.

An execution profiler provides an empirical measurement of the time spent in each function. A widely-used Unix tool, **prof** [Unix], provides a count of how many times each function is called, and how many seconds were spent in each function. This information is very useful in identifying which functions are in need of improvement, after which a theoretical analysis of just those functions might be carried out, a much less forbidding endeavor than analyzing the whole program.

Prof gathers call-count information by having the compiler insert at the beginning of each function an instruction that increments a call-count variable associated with the function. An approximation to the amount of total time spent in the function is gathered by the use of a timer interrupt: every 1/60th of a second, the operating system notes in a ‘‘histogram’’ array the value of the program counter. (This is an ancient technique [Johnston70].) Then, at the end of program execution, **prof** estimates the amount of time spent in each function by summing the values in the histogram array corresponding to program counter samples between the beginning and end of the machine code for that function. The interrupt-driven sampling method has a much lower overhead than querying a clock on each entry to and exit from a function.

A more elaborate profiling tool, **gprof** [Graham82], provides even more information. When one primitive function (e.g. a table-lookup routine) is used in many places, it is useful to know, not only the total time for the execution of the primitive, but also, how much time to ‘‘charge’’ the calling functions. With **gprof** this is approximated by keeping a count of the number of times each call-site is used, and on that basis apportioning the average execution time of the called function to the functions that call it.

These approaches to execution-time measurement and apportionment pose certain problems for optimizing compilers and for functional languages:

1. The machine-code for a function is not necessarily all contiguous. A function may be turned into several pieces of code, with portions of the code for other functions interspersed. This problem could certainly be solved by elaborate bookkeeping in the optimizer and code generator, but we wanted to avoid that complexity.
2. An optimizer can expand functions in-line in other functions. The program-counter method will charge the calling function instead of the called function, even though it might be desirable for in-line expansion to be made semantically invisible.
3. The histogram array in **prof** must be proportional in size to the address range spanned by pieces of code for executable functions. Our runtime system intersperses code and data throughout memory; even worse, it periodically garbage collects, moving code and data from place to place. This problem

could have been solved by elaborate bookkeeping in the runtime system, which we also wanted to avoid.

We had to deal with these problems in the course of implementing a profiler for an optimizing compiler for the functional language Standard ML [Appel87]. The approach we used is described in the next section.

2. Intermediate Representation of call-counting and current-function

For execution-time estimation we use a timer interrupt, as does **prof**, to increment a histogram entry. However, we don't use the program counter to calculate which histogram entry to increment. Instead, we maintain a ''pointer-to-current-function-entry'' in a global variable called **current** that is accessible to the timer-interrupt handler. Each function has associated with it two auxiliary variables: a call-count and an interrupt-count. On entry to a function, it increments the call-count and assigns the address of the interrupt-count variable into the global **current** variable. Then, when a timer interrupt occurs, the interrupt handler just increments the variable that **current** points to.

When a function returns — either normally or via an exception — **current** must be set back to the interrupt-count variable of the function that it is returning to. This resetting could be done either by the calling function (after the called function has returned), or by the called function before exit. For several reasons, it is better done by the calling function. If the called function does the reset, a stack of current-function pointers is required; this is expensive to maintain. A stack of current-pointers would also greatly complicate the treatment of exception-handlers; with the caller-reset method, the exception-handler justs sets the **current** to point at the appropriate counter variable on entry. On recursive calls, **current** need not be reset (as the calling and called function are the same), but only the calling function knows which calls are recursive. And finally, tail-calls can be optimized if the caller resets **current**.

A tail-call is one that is not followed by any executable code before the function returns. After a tail-call, the function will immediately return and therefore **current** will immediately be reset. Therefore, it is not necessary for the calling function to reset **current** after a tail-call. This is a useful optimization, and it is particularly important when used with a compiler that optimizes tail-calls into jumps; if the current pointer had to be reset after the tail-call, it would no longer be a tail-call and performance would suffer dramatically. Fortunately, it is easy to identify tail-calls statically as the profiling instructions are being inserted.

We insert the profiling instructions as ordinary assignment statements in the intermediate representation. In almost any compiler's intermediate representation it is easy to represent the operations of fetching, adding one, and storing, for the call-count increment operation; and storing, for the assignment to the **current** variable.

Functional programming languages introduce another problem for the design of profilers: what to do with anonymous, first-class functions. The simplest choice is to do nothing; collect no call-counts and let the time be charged to the caller of the unnamed function. The main disadvantage of this solution, besides not having the call-counts, is that there is no convenient way to find the code that contributes to the cost of a profiled function that calls anonymous functions.

Probably the most general solution is to make up names for the unnamed functions (for example, an unnamed function statically enclosed in function *f* might be called *f.anon*). If anonymous functions are given names they can be treated just as any other function; call counts and execution time will be reported. Of course, the user will need to associate the new names with the correct function, but in practice this is rarely a problem.

3. An example

To illustrate the technique, we present a simple example (figure 1). The ML function `subset` takes a predicate function as an argument, and returns a function that maps lists to lists; the output list will be that sublist of the input list containing just those elements that satisfy the predicate. The user's program is displayed in typewriter font; the compiler puts some scaffolding around it (indicated in italics) to make a record containing all the functions declared by the user.

```
let fun subset pred =
        let fun f nil = nil
              | f (a::r) = if pred a then a::f(r) else f(r)
         in f
        end

    fun isPrime x =
        let fun test i = i>=x orelse (x mod i <> 0 andalso test(i+1))
         in test 2
        end

    val primes = subset isPrime
 in (subset, isPrime, primes)
end
```

Figure 1.

If this code is compiled with profiling enabled, the compiler inserts the call-counting and current-function instructions into the intermediate representation. Here, we display the effects as if written in the source language (figure 2).

For each function, two variables are introduced: a call-count and an interrupt-count. On entry to a function, the call-count is incremented, and the global variable **current** is set to point to the interrupt-count. On re-entry to a function after a subroutine call, **current** is reset to the function's interrupt-count variable. However, this is not necessary after recursive calls and tail calls, e.g. the calls to `f`.

The initial *let*-bindings create all the count variables, and the last four lines produce, instead of just a record containing the user's declared objects, a pair of records: the user's declared objects, and a list of records containing profiling variables, each with an identifying string constant. These string constants will be

```
let val subset.CC = ref 0 and subset.IC = ref 0
   and subset.f.CC = ref 0 and subset.f.IC = ref 0
   and isPrime.CC = ref 0 and isPrime.IC = ref 0
   and isPrime.test.CC = ref 0 and isPrime.test.IC = ref 0

    fun subset pred =
        (subset.CC := !subset.CC + 1;
        current := subset.IC;
        let fun f x =
            (subset.f.CC := !subset.f.CC + 1;
            current := subset.f.IC;
            case x of
              nil => nil
            | a::r => let val pa = pred a
                       in current := subset.f.IC;
                          if pa then a :: f(r) else f(r)
                      end
         in f
        end

    fun isPrime x = (isPrime.CC := !isPrime.CC + 1;
                     current := isPrime.IC;
                     . . . )

    val primes = subset isPrime

 in ((subset, isPrime, primes),
    ((subset.CC, subset.IC, "subset"),
     (subset.f.CC, subset.f.IC, "subset.f"),
     (isPrime.test.CC, isPrime.test.IC, "isPrime.test"),
     (isPrime.CC, isPrime.IC, "isPrime")))
end
```

Figure 2.

embedded in the executable code for this module, and will enable the call-count variables to be self-identifying. Our runtime system maintains a global list of these 3-element records; when it is time to print an execution profile, they are sorted in decreasing order of interrupt-count.

Our output looks like the output of **prof**:

```
%time  cumsecs   #call ms/call  name
 90.4     3.52   78189    .045  isPrime.test
  8.4     3.85    1000    .330  isPrime
   .7     3.88       0          (unprofiled)
   .2     3.89    1001    .009  subset.f
   .0     3.89    1001    .000  natlist
   .0     3.89       1    .000  subset
```

Now, armed with this information, a programmer might decide that it is worthwhile re-writing the `isPrime` function to make it as efficient as possible. But at a certain point the programmer will want to know what functions are calling `isPrime` so he can make them call it less often. By re-compiling with `isPrime` unprofiled, any time spent in `isPrime` will now be charged to the function that called it. This is because `isPrime` will not change the **current** variable, so that the timer-interrupt will increment the count for the function that last set **current** — and this will be the one that called **isPrime**. The profiling system won't do this automatically, but by comparing two different execution profiles, one with `isPrime` compiled with profile instructions and one with `isPrime` unprofiled, an accurate estimate can be made of who is calling it.

4. Advantages of our current-function method

Since we use ordinary intermediate-representation operators for profiling, the optimizer and code-generator ''believe'' that profiling operations are part of the program. Since an optimizer must not modify the semantics of the program, the semantics of profiling will not be modified either. Therefore, if one function is copied and inserted in-line into another, the call-count and current-function instructions will be copied and inserted at the right place. Other optimizations that break functions into several disjoint pieces of code will leave the profiling instructions in the appropriate places.

Furthermore, the result is that the implementation of the profiler is completely independent of the code generator. We have four different code generators for our compiler (two different algorithms each for the Vax and the Motorola 68020), and not a line of any of them was modified for the installation of the profiler.

By compiling some functions unprofiled, as described in the previous section, we can find out what callers are responsible for most of their execution time. This kind of trick serves much the same purpose that the more elaborate program **gprof** does; and it's a trick that wouldn't work with a program-counter histogram. Furthermore, our method is more accurate than **gprof**. Suppose functions *f* and *g* both call a function *isPrime*, but *f* consistently makes expensive calls (that take a long time) while *g* makes cheap ones. **Gprof** allocates the total time spent in *isPrime* on the basis of call counts from *f* and *g*; this will miss the fact that *f* is responsible for most of the cost. In this example, when profiling for *isPrime* is turned off, *f* and *g* will be charged for the actual time spent in *isPrime* on their behalf. (On the other hand, **Gprof** will give an accurate breakdown of call-site counts that our method does not provide.)

If a profiled function calls an unprofiled function, then during the execution of the called function, all timer interrupts will be charged to the caller (since **current** still points to the caller's variable). This is often desirable, as described above. But if an unprofiled function calls a profiled function, then upon return to the unprofiled function the **current** pointer won't be reset, and interrupts will continue to be charged to the called function after it has returned. This is undesirable, and should be prevented by the compiler. In a language with first-class functions, it is difficult to prevent profiled functions from being passed as arguments to unprofiled functions that might then call them. In practice, this has not proved to be a problem, probably because unprofiled functionals are typically simple primitives like *app* and *map*, which do little intrinsic computation.

5. Overhead measurements

We ran the same program several times with various of our profiling features enabled; this gives a reasonably accurate measurement of profiling overhead:

	Time	%Overhead
User code	2801 sec	
Call counts	568	20.3%
Setting current function	286	10.2
Interrupts	47	1.7
Total Overhead	901	32.2%

The total overhead of 32% is not prohibitively expensive. Our code generator takes three instructions to increment a call-count (fetch, add, store); a better instruction-selector could probably reduce this overhead to 8%, and the total overhead to 20%.

There is also an implementation overhead; it turned out to be fairly simple to get this profiler running.

Insertion of profiling instructions	49 lines
Interrupt handling	32
Global database	16
Report generation	72
Total	169 lines

In contrast, this paper is about 500 lines long.

6. Conclusion

Traditional approaches to profiling run into problems when we attempt to apply them to functional languages where code may be moved around by garbage collection, and the task is further complicated when an optimizing compiler freely rearranges the code. The basic difficulty is that the mapping between the current pc and the currently executing function is difficult to maintain.

We have found a simple way around this difficulty, which consists of maintaining a global variable that always points to the interrupt count for the current function, and which is to be charged whenever there is a timer interrupt. Because we manipulate this variable in the intermediate representation of the compiler, our method is very easy to implement and has no nasty interactions with code generation or garbage collection algorithms (which already preserve semantics of intermediate-representation operations).

This method has acceptable overhead and accuracy. Furthermore, by judiciously mixing profiled and unprofiled functions, one can extract information on inherited costs as well as the direct costs of calling particular functions. This information is similar to that provided by sophisticated profilers like gprof, but is more accurate.

References

[Appel87] Appel, Andrew W. and MacQueen, David B. ‘‘A Standard ML compiler,’’ in *Functional Programming Languages and Computer Architecture*, LNCS 274, G. Kahn, ed., pp 301-324, 1987

[Graham82] Graham, Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. ‘‘gprof: a call graph execution profiler’’" in *Proc. SIGPLAN ’82 Symp. on Compiler Construction, SIGPLAN Notices* 17(4), pp. 120-126, 1982.

[Johnston70] Johnston, T. Y., and Johnson, R. H., *Program Performance Measurement*, SLAC User Note 33, Rev. 1, Stanford University, California, 1970.

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