A Memory Allocation Profiler for C and Lisp Programs

Benjamin Zorn Paul Hilfinger

February 16, 1988

Abstract

This paper describes **mprof**, a tool used to study the memory allocation behavior of programs. **mprof** records the amount of memory each function allocates, breaks down allocation information by type and size, and displays a program's dynamic call graph so that functions indirectly responsible for memory allocation are easy to identify. **mprof** is a two-phase tool. The monitor phase is linked into executing programs and records information each time memory is allocated. The display phase reduces the data generated by the monitor and displays the information to the user in several tables. **mprof** has been implemented for C and Kyoto Common Lisp. Measurements of these implementations are presented.

1 Introduction

Memory allocation is an important part of most programs. Unnecessary allocation can result in decreased program locality, increased execution time for the allocation itself, and additional overhead to reclaim memory. If reclamation is not performed, or if some objects are accidently not reclaimed (a "memory leak"), programs can fail when they reach the memory size limit. Programmers often write their own versions of memory allocation routines to measure and reduce allocation overhead. In Lisp, allocation and reclamation occur transparently, yet they have a strong effect on the performance of programs. Even though memory allocation is important, few software tools exist to help programmers understand the memory allocation behavior of their programs.

mprof is a tool that allows programmers to identify where and why memory is being allocated in a program. It records which functions are directly responsible for memory allocation and also records the dynamic call chain at each allocation to show which functions were indirectly responsible for allocation.

For example, consider the C program in Figure 1, a simplified producer/consumer simulation. Objects of different sizes are randomly allocated by producers and eventually consumed by the consumer. The consumer is responsible for freeing the objects passed to it, but there is a bug in this program so that the consumer does not free red widgets. If the simulation ran for a long time, memory would eventually be exhausted, and the program would fail.

This research was funded by DARPA contract number N00039-85-C-0269 as part of the SPUR research project.

```
typedef struct {
    enum color c;
    int data[50];
} widget;

#define WSIZE sizeof(widget)

widget
*make_widget()
{
    widget      *w;

    w = (widget *) malloc(WSIZE);
    return w;
}

widget
*make_blue_widget()
{
    widget      *w;

    w = make_widget();
    w->c = BLUE;
    return w;
}

widget
*make_red_widget()
{
    widget      *w;

    w = make_widget();
    w->c = RED;
    return w;
}
```

```
void
consume_widget(w)
widget  *w;
{
    if (w->c == BLUE) {
        /* record blue widget */
        free(w);
    } else {
        /* record red widget */
    }
}


#define NUM_WIDGETS     10000

int
main()
{
    int         i;
    widget      *wqueue[NUM_WIDGETS];

    for (i = 0; i < NUM_WIDGETS; i++)
      if (random_flip())
        wqueue[i] = make_blue_widget();
      else
        wqueue[i] = make_red_widget();

    for (i = 0; i < NUM_WIDGETS; i++)
      consume_widget(wqueue[i]);

    return 0;
}
```

Figure 1: A Simple Producer/Consumer Simulation Example. The producer allocates memory that is eventually passed to the consumer and freed. This program has a bug so that some of the memory allocated is never freed. The function `random_flip`, which is not shown, randomly returns 1 or 0 with equal probability.

We use this example to make several points. First, the function make_widget is the only function that allocates memory directly. Knowing what functions call malloc directly would not help us discover where objects are leaking because, in this example, all objects are allocated by the same function. Our profiling tool must record that make_red_widget and make_blue_widget were indirectly responsible for calls to malloc and take advantage of that information. In this example, **mprof** tells us how many bytes make_blue_widget and make_red_widget requested from make_widget, how many bytes were never deallocated, and most specifically, which call chain ending with make_widget allocated memory that was never subsequently released.

A monitoring program such as **mprof** should satisfy several criteria. First, the monitor should not significantly alter the behavior of the program being monitored. In particular, the monitor should not impose so much overhead on the program being monitored that large programs cannot be profiled. Second, the monitor should be easy to integrate into existing applications. To use **mprof**, programmers simply have to relink their applications with a special version of the system library. No source code modifications are required. Finally, the monitor should provide the programmer with information he can understand and use to reduce the memory allocation overhead of his programs. Using the example above, this paper will illustrate such a use of **mprof**.

In Section 2, we describe the use of **mprof** in more detail and in Section 3 we discuss techniques for its effective implementation. Section 4 presents some measurements of **mprof**. Section 5 describes other memory profiling tools and previous work on which **mprof** is based, while Section 6 contains our conclusions.

2 Using mprof

To use **mprof**, programmers link in special versions of the system functions malloc and free, which are called each time memory is allocated and freed, respectively. The application is then run normally. The **mprof** monitor function, linked in with malloc, gathers statistics as the program runs and writes this information to a file when the application exits. The programmer then runs a display program over the data file, and four tables are printed: a list of memory leaks, an allocation bin table, a direct allocation table, and a dynamic call graph. Each table presents the allocation behavior of the program from a different perspective. The rest of this section presents the output tables for the C program in Figure 1. Fields in the tables are described in detail in the appendix.

2.1 The Memory Leak Table

C programmers must explicitly free memory objects when they are done using them. Memory leaks arise when programmers accidently forget to release memory. Because Lisp reclaims memory automatically, the memory leak table is not necessary in the Lisp version of **mprof**.

The memory leak table tells the programmer what functions allocated memory associated with memory leaks. The table contains a list of partial call paths that resulted in memory being allocated and never subsequently freed. The paths are partial because complete path information is not recorded. Only the last five callers on the callstack are listed in the memory leak table. In our simple example, there is only one such path, and it tells us immediately what objects are not freed. Figure 2 shows what the memory leak table looks like for our example.

```
allocs      bytes (%)       path

  5019    1023876 (99)    || > main > make_red_widget > make_widget
```

Figure 2: Memory Leak Table for Producer/Consumer Example. Only one partial allocation path is listed because only one path resulted in objects being allocated and never freed. Because the callstack depth was less than five, the allocation path is complete in this example.

In large examples, more than one path through a particular function is possible. We provide an option that distinguishes individual call sites within the same function in the memory leak table if such a distinction is needed.

2.2 The Allocation Bin Table

A major part of understanding the memory allocation of a program is knowing what objects were allocated. In C, memory allocation is done by object size; the object type being allocated is not known at allocation time. The allocation bin table provides information about what sizes of objects were allocated and what program types correspond to the sizes listed. This knowledge helps the programmer recognize what data structures he is using that consume the most memory and allow him to concentrate any space optimizations on these types.

The allocation bin table breaks down object allocation by the size in bytes of objects allocated. Figure 3 shows the allocation bin table for the program in Figure 1 .

```
   size:     allocs     bytes (%)       frees      kept (%)      types

    204      10000    2040000 (99)      4981    1023876 (99)    widget
 > 1024          0          0              0          0

<TOTAL>      10000    2040000           4981    1023876
```

Figure 3: Allocation Bin Table for Producer/Consumer Example. Only one object size is listed because only one kind of object was allocated by the program. The type of that object (`widget`) is listed at the right.

The allocation bin table contains information about objects of each byte size from 0 to 1024 bytes and groups objects larger than 1024 bytes into a single bin. For each byte size in which memory was allocated, the allocation bin table shows the number of allocations of that size (`allocs`), the total number of bytes allocated to objects of that size (`bytes`), the number of frees of objects of that size (`frees`), the number of bytes not freed that were allocated to objects of that size (`kept`[1]), and user types whose size is the same as the bin size (`types`). From the example, we can see that 10,000 widgets were allocated by the program, but only 4,981 of the widgets allocated were eventually freed.

2.3 The Direct Allocation Table

Another facet of understanding memory allocation is knowing what functions allocated memory and how much they allocated. In C, memory allocation is performed explicitly by calling `malloc`, and so programmers are often aware of which functions allocated memory. Even in C, however, knowing how much memory was allocated can point out functions that do unnecessary allocation and guide the programmer when he attempts to optimize the space consumption of his program. In Lisp, memory allocation happens implicitly in many primitive routines such as `mapcar`, , `*`, and `intern`. The direct allocation table can reveal unsuspected sources of allocation to Lisp programmers. Figure 4 contains the direct allocation table for our example.

```
 % mem        bytes | % mem(size) |   bytes kept |  % all kept |        calls   name

--------------------------s--m--l--x------------------s--m--l--x--------------

 -----      2040000 |   99        |      1023876 |   99        |        10000   <TOTAL>

 100.0      2040000 |   99        |      1023876 |   99        |        10000   make_widget
```

Figure 4: Direct Allocation Table for Producer/Consumer Example. Only one function is listed because only one function called `malloc` in Figure 1.

The first line of the direct allocation table contains the totals for all functions allocating memory. In this example, only one function, `make_widget`, allocates memory. The direct allocation table prints percent of total allocation that took place in each function (`% mem`), the number of bytes allocated by each function (`bytes`), the number of bytes allocated by the function and never freed (`bytes kept`), and the number of calls made to the function that resulted in allocation (`calls`). The `% mem(size)` fields contain a size breakdown[2] of the memory allocated by each function as a fraction of the memory allocated by all functions. In this example, 99% of the memory allocated by the program was allocated in `make_widget` for medium-sized objects. Blank columns indicate values less than 1%. The other size breakdown given in the direct allocation table is for the memory that was allocated and never freed. The `% all kept` field contains a size breakdown of the memory not freed by a particular function as a fraction of all the memory not freed. In the example, 99% of the unfreed memory was of medium-sized objects allocated by `make_widget`.

[1]The label **kept** is used throughout the paper to refer to objects that were never freed.

[2]Both the direct allocation table and the dynamic call graph break down object allocation into four categories of object size: small (s), from 0–32 bytes; medium (m), from 33–256 bytes; large (l), from 257–2048 bytes; and extra large (x), larger than 2048 bytes. For Lisp, categorization is by type rather than size: cons cell (c), floating point number (f), structure or vector (s), and other (o).

2.4 The Allocation Call Graph

Understanding the memory allocation behavior of a programs sometimes requires more information than just knowing the functions directly responsible for memory allocation. Sometimes, as happens in Figure 1, the same allocation function is called by several different functions for different purposes. Functions that are indirectly responsible for allocation are also of interest to the programmer. The allocation call graph shows all the functions that were indirect callers of functions that allocated memory.

Because the dynamic caller/callee relations of a program are numerous, the dynamic call graph is a complex table with many entries. Often, the information provided by the first three tables is enough to allow programmers to understand the memory allocation of their program. Nevertheless, for a full understanding of the allocation behavior of programs the allocation call graph is useful. Figure 5 contains the allocation call graph for the producer/consumer example.

```
        self                     |     /ances  |      /ances  |  called/total      ancestors
index    +         self (%)      | size-func   | frac         |  called/recur   name [index]
        desc                     |     \desc   |      \desc   |  called/total      descendents

------------------------------s--m--l--r----s--m--l--r---------

[0]    100.0           0 (0)     |             | ----------- |       0         main [0]
             1023876 (50)        |    99       |     50      |   5019/5019        make_red_widget [2]
             1016124 (49)        |    99       |     49      |   4981/4981        make_blue_widget [3]
          . all  2040000         |    99       |             |

------------------------------s--m--l--r----s--m--l--r---------

           all  2040000          |    99       |             |
             1023876 (50)        |    99       |     50      |   5019/5019        make_red_widget [2]
             1016124 (49)        |    99       |     49      |   4981/4981        make_blue_widget [3]
[1]    100.0  2040000 (100)      |    99       | ----------- |  10000         make_widget [1]

------------------------------s--m--l--r----s--m--l--r---------

             1023876 (100)       |    99       |     99      |   5019/10000       main [0]
[2]     50.2           0 (0)     |             | ----------- |   5019         make_red_widget [2]
             1023876 (100)       |    99       |     99      |   5019/10000       make_widget [1]

------------------------------s--m--l--r----s--m--l--r---------

             1016124 (100)       |    99       |     99      |   4981/10000       main [0]
[3]     49.8           0 (0)     |             | ----------- |   4981         make_blue_widget [3]
             1016124 (100)       |    99       |     99      |   4981/10000       make_widget [1]

------------------------------s--m--l--r----s--m--l--r---------
```

Figure 5: Allocation Call Graph for Producer/Consumer Example. This table presents the subset of the example program's dynamic call graph in which allocation occurred.

The allocation call graph is a large table with an entry for each function that was on a call chain when memory was allocated. Each table entry is divided into three parts. The line for the function itself (called the *entry function*); lines above that line, each of which represents a caller of the entry function (the ancestors), and lines below that line, each of which represents a function called by the entry function (the descendents). The entry function is easy to identify in each table entry because a large horizontal rule appears in the `frac` column on that row. In the first entry of Figure 5, `main` is the entry function; there are no ancestors and two descendents.

The entry function line of the allocation call graph contains information about the function itself. The `index` field provides a unique index to help users navigate through the call graph. The `self + desc` field contains the percent of total memory allocated that was allocated in this function and its descendents. The call graph is sorted by decreasing values in this field. The `self` field contains the number of bytes that were allocated directly in the entry function. The `size-func` fields contain a size breakdown of the memory allocated in the function itself. Some functions, like `main` (index 0) allocated no memory directly, so the `size-func` fields are all blank. The `called` field shows the number of times this function was called during a memory allocation, with the number of recursive calls recorded in the adjacent field.

Each caller of the entry function is listed on a separate line above it. A summary of all callers is given on the top line of the entry if there is more than one ancestor. The `self` field of ancestors lists the number of bytes that the entry function and its descendents allocated on behalf of the ancestor. The `size-ances` field breaks down those bytes into size categories, while the `frac-ances` field shows the size breakdown of the bytes requested by this ancestor as a fraction of bytes allocated at the request of all ancestors. For example, in the entry for function `make_widget` (index 1), the ancestor `make_red_widget` can be seen to have requested 1,023,876 bytes of data from `make_widget`, 99% of which was of medium-sized objects. Furthermore, calls from `make_red_widget` accounted for 50% of the total memory allocated by `make_widget` and its descendents. Other fields show how many calls the ancestor made to the entry function and how many calls the ancestor made in total. In a similar fashion, information about the function's descendents appears below the entry function.

Had the memory leak table not already told us what objects were not being freed, we could use the allocation call graph for the same purpose. The direct allocation table told us that `make_widget` allocated 1,023,876 bytes of unfreed memory, all for medium-sized objects. From the allocation call graph, we can see that the function `make_red_widget` was the function calling `make_widget` that requested 1,023,876 bytes of medium-sized objects.

Cycles in the call graph are not illustrated in Figure 5. As described in the next section, cycles obscure allocation information among functions that are members of a cycle. When the parent/child relationships that appear in the graph are between members of the same cycle, most of the fields in the graph must be omitted.

3 Implementation

We have implemented **mprof** for use with C and Common Lisp programs. Since the implementations are quite similar, the C implementation will be described in detail, and the minor differences in the Lisp implementation will be noted at the end of the section.

3.1 The Monitor

The first phase of **mprof** is a monitor that is linked into the executing application. The monitor includes modified versions of `malloc` and `free` that record information each time they are invoked. Along with `malloc` and `free`, **mprof** provides its own `exit` function, so that when the application program exits, the data collected by the monitor is written to a file. The monitor maintains several data structures needed to construct the tables.

To construct the leak table, the monitor associates a list of the last 5 callers in the call chain, the *partial call chain*, with the object allocated. When objects are allocated, the partial call chain is used as a key to retrieve an allocation counter associated with the partial call chain that is then incremented. When the object is later freed, the partial call chain associated with the object is used as a key to retrieve and increment a counter of frees. Any partial call chain in which the number of allocations does not match the number of frees indicates a memory leak and is printed in the leak table.

To construct the allocation bin table, the monitor has an 1026 element array of integers to count allocations and another 1026 element array to count frees. When objects of a particular size from 0–1024 bytes are allocated or freed, the appropriated bin is incremented. Objects larger than 1024 bytes are grouped into the same bin.

The construction of the direct allocation table falls out directly from maintaining the allocation call graph information. In order to build the allocation call graph, each time `malloc` is called, the monitor must associate the number of bytes allocated with the current dynamic call chain. This operation is potentially expensive both in time and space. One implementation would simply record every function in every chain and write the information to a file. Considering that many programs execute millions of calls to `malloc` and that the depth of the call chain can be hundreds of functions, the amount of information would be prohibitive.

An alternative to recording the entire chain of callers is to break the call chain into a set of caller/callee pairs, and associate the bytes allocated with each pair in the set. The disadvantage with this implementation is that the exact call chains are no longer available. However, by associating how much memory was allocated with each caller/callee pair in the chain, the correct dynamic call graph of the program can be recreated.

Associating caller/callee pairs with bytes allocated requires a data structure that allows caller/callee pairs to be mapped to a byte count quickly. We use a hash table in which the hash function is a simple byte-swap XOR of the callee address. Each callee has a list of its callers and the number of allocated bytes associated with each pair. We noted that from allocation to allocation, most of the call chain remains the same. Our measurements show that on the average, 60–75% of the call chain remains the same between allocations. This observation allows us to cache the pairs associated with the current caller chain and to use most of these pairs the next time a caller chain is recorded. Thus, on any particular allocation, only a few addresses need to be hashed. Here are the events that take place when a call to `malloc` is monitored:

1. The chain of return addresses is stored in a vector.

2. The new chain is compared with the previous chain, and the point where they differ is noted.

3. For the addresses in the chain that have not changed, the caller/callee byte count for each pair is already available and is incremented.

4. For new addresses in the chain, each caller/callee byte count is looked up and updated.

5. For the tail of the chain (i.e., the function that called `malloc` directly), the direct allocation information is recorded.

Maintaining allocation call graph information requires a byte count for every distinct caller/callee pair in every call chain that allocates memory. Our experience is that there are a limited number of such pairs, even in very large C programs, so that the memory requirements of the **mprof** monitor are not large (for a range of programs, measurements show the memory required by the **mprof** monitor is usually less than 50 kilobytes).

3.2 Reduction and Display

The second phase of **mprof** reads the output of the monitor, reduces the data to create a dynamic call graph, and displays the data in three tables. The first part of the data reduction is to map the caller/callee address pairs to actual function names. A program `mpfilt` reads the executable file that created the monitor trace (compiled so that symbol table information is retained), and outputs a new set of function caller/callee relations. These relations are then used to construct the subset of the program's dynamic call graph that involved memory allocation.

The call graph initially can contain cycles due to recursion in the program's execution. Cycles in the call graph introduce spurious allocation relations, as is illustrated in Figure 6.

```
CALL STACK:                  MPROF RECORDS:
main calls F                 (10 bytes over main -> F)
     F calls G               (10 bytes over F -> G)
       G calls F             (10 bytes over G -> F)
         F calls G           (10 MORE bytes over F -> G)
           G calls malloc(10)  (10 bytes allocated in G)
```

Figure 6: Problems Caused by Recursive Calls. In this example, `main` is credited as being indirectly responsible for 10 bytes, but because we only keep track of caller/callee pairs, `F` appears to have requested 20 bytes from `G`, even though only 10 bytes were allocated.

We considered several solutions to the problems caused by cycles and adopted the most conservative solution. One way to avoid recording spurious allocation caused by recursion is for the monitor to identify the cycles before recording the allocation. For example, in Figure 6, the monitor could realize that it had already credited `F` with the 10 bytes when it encountered `F` calling `G` the second time. This solution adds overhead to the monitor, however, and our goal was to make the monitor as unobtrusive as possible.

The solution we adopted was to merge functions that are in a cycle into a single node in the reduction phase. Thus, each strongly connected component in the dynamic call graph is merged into a single node. The result is a call graph with no cycles. This process is also used by **gprof**, and described carefully in [GKM83]. Such an approach works well in **gprof** because C programs, for which **gprof** was primarily intended, tend to have limited amounts of recursion. Lisp programs, for which **mprof** is also intended, intuitively contain much more recursion. We have experience profiling a number of large Common Lisp programs. We observe several recursive cycles in most programs, but the cycles generally contain a small percentage of the total functions and **mprof** is quite effective. Only with more data will we be able to decide if many Lisp programs contain so many recursive calls that cycle merging makes **mprof** ineffective. Nevertheless, **mprof** has already been effective in detecting KCL system functions that allocate memory extraneously.[3]

3.3 Lisp Implementation

So far, we have described the implementation of **mprof** for C. The Lisp implementation is quite similar, and here we describe the major differences. C has a single function, `malloc`, that is called to allocate memory explicitly. Lisp has a large number of primitives that allocate memory implicitly (i.e., `cons`, `*`, `intern`, etc.). To make **mprof** work, these primitives must be modified so that every allocation is recorded. Fortunately, at the Lisp implementation level, all memory allocations may be channeled through a single routine. We worked with KCL (Kyoto Common Lisp), which is implemented in C. In KCL, all Lisp memory allocations are handled by a single function, `alloc_object`. Just as we had modified `malloc` in C, we were able to simply patch `alloc_object` to monitor memory allocation in KCL.

The other major difference in monitoring Lisp is that the addresses recorded by the monitor must be translated into Lisp function names. Again, KCL makes this quite easy because Lisp functions are defined in a central place in KCL and the names of the functions are known when they are defined. Many other Lisp systems are designed to allow return addresses to be mapped to symbolic function names so that the call stack can be printed at a breakpoint. In this case, the monitor can use the same mechanism to map return addresses to function names. Therefore, in Lisp systems where addresses can be quickly mapped to function names, memory profiling in the style of **mprof** is not a difficult problem. In systems where symbolic names are not available in compiled code, profiling is more difficult. Furthermore, many systems open-code important allocation functions, like `cons`. In this case, the open-coded functions will not appear in the profile directly; instead the functions containing the open-coded functions will be credited with the allocation.

4 Measurements

We have measured the C implementation of **mprof** by instrumenting four programs using **mprof**. The first program, `example`, is our example program with the number of widgets allocated increased to 100,000 to increase program execution time. The second program, `fidilrt`, is the runtime library of a programming language for finite difference computa-

[3]Using **mprof**, we noted that for a large object-oriented program written in KCL, the system function `every` accounted for 13% of the memory allocated. We rewrote `every` so it would not allocate any memory, and decreased the memory consumption of the program by 13%.

tions. The third program, `epoxy`, is an electrical and physical layout optimizer written by Fred Obermeier [OK87]. The fourth program, `crystal`, is a VLSI timing analysis program [Ous85]. These tests represent a small program (`example`, 100 lines); a medium-sized program (`fidilrt`, 2,700 lines); and two large programs (`epoxy`, 11,000 lines and `crystal`, 10,500 lines).

mprof consumes four resources when it is used. The monitor adds execution time to the program being profiled. We measured the programs executing with and without profiling. The ratio of the time with profiling to the time without profiling is called the *slowdown* factor. The most significant source of memory used by the monitor is the partial call chain of five addresses allocated with every object and used to construct the leak table. Table 1 reports this overhead as a fraction of the total user memory. The monitor also uses memory to record the memory bins and caller/callee byte counts and must write this information to a file when the profile is finished. We measured how many bytes of memory and disk space are needed to store this information. A final resource is the time required to reduce the monitor data and print the tables. Table 1 summarizes the measurements.

	Cost			
Resource Description	`example`	`fidilrt`	`epoxy`	`crystal`
User memory allocated (Kbytes)	20000	3644	6418	21464
Number of allocations	100000	77376	306295	31158
Execution time with **mprof** (seconds)	47.7	128.4	188.8	134.1
Execution time without **mprof** (seconds)	15.4	93.7	52.1	13.2
Overhead per allocation (milliseconds)	0.3	0.5	0.4	3.9
Slowdown factor	3.1	1.4	3.6	10.1
Leak Table memory usage (Kbytes)	1953	1510	5982	517
Leak Table fraction (% memory allocated)	9	29	48	2
Other Monitor memory usage (Kbytes)	8.7	15.9	52.3	17.5
Intermediate data file size (Kbytes)	4.5	8.1	28.6	9.6
Reduction and display time (seconds)	10.3	16.9	28.3	36.8

Table 1: Resource Usage of **mprof**. Measurements were gathered running the test programs on a VAX 8800 with 80 megabytes of physical memory.

Profiling adds 0.4–4 milliseconds to each object allocation. The `crystal` test suffered the worst degradation from profiling (slowing down an order of magnitude) because `crystal` uses a depth-first algorithm that results in long call chains. Our measurements show that typically **mprof** slows program execution by a factor of two or three. Since **mprof** is a prototype and has not been carefully optimized, this overhead seems acceptable. If less overhead is required, we will consider only recording partial call chains for the allocation call graph or eliminating the allocation call graph altogether.

The memory overhead of **mprof** is small except that storing the partial call chains for the leak table with each object allocated can account for half or more of the total memory consumption of programs that allocate many small objects. We are considering an implementation that would reduce the additional per object memory needed to maintain the partial call chains from five words to one, thus reducing the leak table memory overhead by a factor of five.

5 Related Work

mprof is similar to the tool **gprof** [GKM83], a dynamic execution profiler. Because some of the problems of interpreting the dynamic call graph are the same, we have borrowed these ideas from **gprof**. Also, we have used ideas from the user interface of **gprof** for two reasons: first, because the information being displayed is quite similar, and second, because users familiar with **gprof** would probably also be interested in **mprof** and would benefit from a similar presentation.

Barach, Taenzer, and Wells developed a tool for finding storage allocation errors in C programs [BTW82]. Their approach concentrated on finding two specific storage allocation errors: memory leaks and duplicate frees. They modified `malloc` and `free` so that every time these functions were called, information about the memory block being manipulated was recorded in a file. A program that examines this file, **prleak**, prints out which memory blocks were never freed or were freed twice. This approach differs from **mprof** in two ways. First, **mprof** provides more information about the memory allocation of programs than **prleak**, which just reports on storage errors. Second, **prleak** generates extremely large intermediate files that are comparable in size to the total amount of memory allocated by the program (often megabytes of data). Although **mprof** records more useful information, the intermediate files it generates are modest in size (typically smaller than 30 kilobytes).

6 Conclusions

We have implemented a memory allocation profiling program for both C and Common Lisp. Our example has shown that **mprof** can be effective in elucidating the allocation behavior of a program so that programmers can detect memory leaks and identify major sources of allocation.

mprof records every caller in the call chain every time an object is allocated. The overhead for this recording is large but not impractically large, because we take advantage of the fact that the call chain changes little between allocations. Moreover, recording this information does not require large amounts of memory because there are relatively few unique caller/callee address pairs on call chains where allocation takes place, even in very large programs.

Because some Lisp programs contain many recursive functions, large cycles in the call graph may make **mprof** ineffective for this class of programs. Future work may include a different treatment of recursive cycles so that less information about functions within the cycle is lost. We also plan to optimize the monitor to decrease the memory and performance overhead associated with using **mprof**.

References

[BTW82] David R. Barach, David H. Taenzer, and Robert E. Wells. A technique for finding storage allocation errors in C-language programs. *ACM SIGPLAN Notices*, 17(5):16–23, May 1982.

[GKM83] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software Practice & Experience*, 13:671–685, 1983.

[OK87] Fred Obermeier and Randy Katz. *EPOXY: An Electrical and Physical Layout Optimizer that Considers Changes.* Technical Report UCB/CSD 87/388, UCBCS, November 1987.

[Ous85] John Ousterhout. A switch-level timing verifier for digital MOS VLSI. *IEEE Transactions on CAD*, CAD-4(3), July 1985.

NAME

mprof – display dynamic memory allocation data

SYNTAX

mprof [options] [a.out [mprof.data]]

void set_mprof_autosave(count)
int count;

void mprof_stop()

void mprof_restart(filename)
char *filename;

DESCRIPTION

The **mprof** command produces four tables that summarize the memory allocation behavior of C programs, similar in style to the **gprof** command. The arguments to **mprof** are the executable image *(a.out* default) and the profile data file *(mprof.data* default). The *mprof.data* file is generated by linking a special version of *malloc* into the executing image. This new version, called *mprof_malloc.o* must be linked in at the end of the command that creates the executable image. For example:

```
cc -g -o test main.o sub1.o sub2.o mprof_malloc.o
```

Users' programs can contain additional calls to customize the user interface to **mprof.** The function *set_mprof_autosave* allows users to save the profile data periodically. The *count* parameter specifies to save after that number of allocations. A value of 10,000 or 100,000 is typical for the *count* parameter for long running programs. A value of 0 (the default) causes the the profile data to be written only when the program exits. The function *mprof_stop* causes memory profiling to be discontinued and the profile data to be written to the output file. The function *mprof_restart* restarts profiling. The *filename* parameter to *mprof_restart* specifies the name of the file to write the profile data to.

The output of **mprof** consists of four tables, the fields of which are described in detail below. The first table breaks down the memory allocation of the program by the number of bytes requested. For each byte size the number of allocations and frees is listed along with the program structure types that correspond to that byte size.

The second table lists partial call chains over which memory was allocated and never freed (call chains resulting in memory leaks). The table shows how much memory was allocated by each chain and how much each chain contributed to the total memory leakage.

The third table lists the functions in which allocation occurred directly (i.e., called *malloc),* indicates how much memory was allocated, shows how much of that was not later freed, breaks down allocation roughly by size, and shows how many times each function was called.

The fourth table contains the subgraph of the program's dynamic call graph in which allocation occurred. This table allows programmers to identify what functions were indirectly responsible for memory allocation.

The following options are available:

–verbose
Every bin in which memory was allocated is printed; the call chain for every memory leak is shown.

–normal
Only bins that contributed a reasonable fraction to the total allocation are printed; call chains for leaks contributing more than 0.5% to the total are shown. This is the default verbosity setting.

–terse Only bins that contributed a significant fraction to the total allocation are printed. Call chains contributing more than 1% to the total leakage are shown.

–leaktable
Print out the memory leak table without printing out call site offsets. This is the default.

–noleaktable
Do not print out the memory leak table.

–offsets Print out the memory leak table and distinguish different call sites within a function by indicating the offset in the function as part of the path. This is useful to identify a particular call site in a function with many call sites that allocate memory.

FIELDS IN THE OUTPUT

When data is broken down by size categories, the categories mean the following:

s = small	x <= 32 bytes
m = medium	32 < x <= 256 bytes
l = large	256 < x <= 2048 bytes
x = extra large	x > 2048 bytes

where x is the exact size of the object being allocated by a call to malloc. When data is broken into categories, percentages are always given. If no number appears in such a listing, then the number is less than 1%. Throughout this document, we refer to such a listing as a ''breakdown''.

TABLE 1: ALLOCATION BINS

The memory allocation is broken down by the sizes of objects requested and freed.

size	The size in bytes of the object allocated or freed.
allocs	The number of calls to malloc requesting allocation of this size.
bytes (%)	The total number of bytes allocated to objects of this size. The percent indicates the percent of the total bytes allocated. A blank percent indicates less than 1%.
frees	The number of times objects of this size were freed.
kept (%)	The number of bytes of objects of this size that were never freed. The percent indicates what fraction of unfreed bytes were allocated to objects of this size. Blank indicated less than 1%.
types	A list of the program names of structure types or typedefs that define objects of this size.

TABLE 2: MEMORY LEAKS

The memory leak table lists the partial call chains which allocated memory that was never freed. At most five functions in the call chain are listed.

allocs	The number of allocations that occurred on this partial call chain.
bytes (%)	The number of bytes allocated on this partial call chain. The percent indicates the percent of the total bytes allocated and never freed. A blank percent indicates less than 1%.
frees	The number of frees that occurred on this partial call chain. If no objects were freed this and the following field are ommitted.
bytes (%)	The number of bytes freed on this partial call chain. This field is omitted if no bytes were freed.
path	The partial call chain. Call chains starting with "..." indicate that more callers were present, but were ommitted from the listing. Call chains consist of function names (and possible call site offsets) separated by ">". Call site offsets are indicated by a +n following the function name, where n is the distance in bytes of the call site from the start of the function. Call site offsets are printed using the -offset option.

TABLE 3: DIRECT ALLOCATION

The <TOTAL> row of the direct allocation listing contains a summary of all the functions where such a summary makes sense.

% mem	Percentage of the total memory allocated that was allocated by this function.
bytes	The total number of bytes allocated by this function.
% mem(size)	Size breakdown of the memory allocated by this function as a percentage of the total memory allocated by the program. For example, if the values for function MAIN are s=5, m=20, l=4, x=0, then direct calls to MALLOC from MAIN account for 5+20+4+0 = 29% of the total memory allocated by the program. Moreover, 20% of the total memory allocated by the program was of medium sized objects (between 33 and 256 bytes) by the function MAIN. The <TOTAL> row represents the breakdown by size of all the memory allocated by the program.
bytes kept	The number of bytes allocated by this function that were never freed (by calls to FREE).
% all kept	The size breakdown of objects never freed by this function as a percentage of all objects never freed. For example, if <% all kept> values for function MAIN are s=2, m=10, l=<blank>, x=<blank>, then 10% of the total bytes not freed were allocated by MAIN and were allocated in medium-sized chunks. The <TOTAL> row represents the size breakdown of all the memory allocated but never freed.
calls	The number of times this function was called to allocate an object.
name	The name of the function.

TABLE 4: ALLOCATION CALL GRAPH

A star (*) indicates that this field is omitted for ancestors or descendents in the same cycle as the function.

Cycles are listed twice. The first appearance shows all the functions that are members of the cycle and the amount of memory allocated locally in each function, including the breakdown of the local allocation by size and the breakdown by size as a fraction of the total cycle. The second appearance shows what the call graph would look like if all the functions in the cycle were merged into a single function.

index	A unique index used to aid searching for functions in the call graph listing.
self + desc	The percent of the total allocated memory that was allocated by this function and its descendents.
self (%)	The number of bytes allocated by the function itself. The percentage indicates the fraction of the bytes allocated by the function and its descendents that were allocated in the function itself.
size-func	The size breakdown of objects allocated in the function itself (not including its descendents.)
called	The number of times this function was called while allocating memory.
recur	The number of recursive function calls while allocating memory.
name	The function name including possible cycle membership and index.

ANCESTOR LISTINGS

If the word ''all'' appears in the <self + desc> column, then this row represents a summary of all the ancestors and presents the total number of bytes requested by all ancestors in the <bytes> column, and the breakdown of these bytes by size in the <self-ances> breakdown columns. If there is only one ancestor, then this summary is omitted.

*self (%)	The number of bytes allocated by the function and its descendents that were allocated on behalf of this parent. The percentage indicates what fraction of the total bytes allocated by the function and its descendents were allocated on behalf of this parent.
*size-ances	The size breakdown of the bytes allocated by the function and its descendents on behalf of this parent.
*frac-ances	The size breakdown of the objects allocated in the function and its descendents on behalf of this parent as a percentage of all objects allocated by the function and its descendents.

For example if parent P1 of function F has <frac-ances> values s=<blank>, m=<blank>, l=30, x=<blank>, then 30% of all objects allocated by F and its descendents are of large objects allocated on behalf of parent P1.

called	The number of times this parent called this function while requesting memory.
*total	The number of calls this parent made requesting memory from any function.
ancestors	The name of the parent including possible cycle membership and index.

DESCENDENT LISTINGS

If the word "all" appears in the <self + desc> column, then this row represents a summary of all the descendents and presents the total number of bytes allocated by all descendents in the <bytes> column, and the breakdown of these bytes by size in the <self-desc> breakdown columns. If there is only one descendent, then this summary is omitted.

*self (%)	The number of bytes allocated in this descendent that were allocated at the request of the function. The percentage indicates what fraction of the total bytes allocated in descendents of the function were allocated in this descendent.
*size-ances	The size breakdown of the bytes allocated by this descendent on behalf of the function.
*frac-desc	The size breakdown of the objects allocated in this descendent on behalf of the function as a percentage of all objects allocated by all descendents on behalf of this function. For example if descendent C1 of function F has <frac-desc> values s=35, m=<blank>, l=<blank>, x=<blank>, then 35% of all objects allocated by children of F on its behalf were allocated in child C1 and were small objects.
called	The number of times the function called this descendent while requesting memory.
*total	The number of times this descendent was called during a memory request.
descendents	The name of the child including possible cycle membership and index.

FILES

a.out	contains symbol table information.
mprof.data	memory allocation call graph information.
mprof_malloc.o	special version of malloc which profiles allocation.

SEE ALSO

cc(1), gprof(1)

A Memory Allocation Profiler for C and Lisp Programs, Benjamin Zorn and Paul Hilfinger, draft of a U. C. Berkeley Tech Report.

AUTHOR

Written by Benjamin Zorn, zorn@ernie.berkeley.edu, as part of Ph.D. research sponsored by the SPUR research project.

BUGS

The code that determines the names and sizes of user types is poorly written and depends on the program being compiled with the -g option. In some cases (mostly very simple cases) the user type names are not correctly determined.