A Memory Allocation Profiler for C and Lisp Programs

Benjamin Zorn
Department of Computer Science
University of Colorado at Boulder

Paul Hilfinger
Computer Science Division, Dept. of EECS
University of California at Berkeley

Abstract

This paper describes `mprof`, a tool used to study the dynamic memory allocation behavior of programs. `Mprof` records the amount of memory that a 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 presents the information to a user in several tables. `Mprof` has been implemented for C and Kyoto Common Lisp. Measurements of these implementations are presented.

1 Introduction

Dynamic memory allocation, hereafter referred to simply as memory allocation, is an important part of many programs. By dynamic allocation, we mean the memory allocated from the heap. Unnecessary allocation can decrease program locality and increase execution time for the allocation itself and for possible memory reclamation. 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. It is estimated that Mesa programmers spent 40% of their time solving storage-management related problems before automatic storage reclamation techniques were introduced in Cedar [7]. Even with automatic storage management, in which reclamation occurs transparently, memory allocation has a strong effect on the performance of programs [4]. Although 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 dynamic memory is allocated in a program. It records which functions are directly responsible for memory

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

allocation and also records the dynamic call chain at an allocation to show which functions were indirectly responsible for the allocation. The design of `mprof` was inspired by the tool `gprof`, a dynamic execution profiler [2]. `gprof` is a very useful tool for understanding the execution behavior of programs; `mprof` extends the ideas of `gprof` to give programmers information about the dynamic memory allocation behavior of their programs. `Mprof` is a two-phase tool. A monitor phase is linked into an executing application and collects data as the application executes. A reduction and display phase takes the data collected by the monitor and presents it to the user in concise tables.

A profiling program such as `mprof` should satisfy several criteria. First, a program monitor should not significantly alter the behavior of the program being monitored. In particular, a monitor should not impose so much overhead on the program being monitored that large programs cannot be profiled. Second, a 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, a monitor should provide a programmer with information that he can understand and use to reduce the memory allocation overhead of his programs. We will present an example that illustrates such a use of `mprof`.

In Section 2, we present a simple program and describe the use of `mprof` with respect to the example. In Section 3 we discuss techniques for the effective implementation of `mprof` . 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`

2.1 A Producer/Consumer Example

To illustrate how `mprof` helps a programmer understand the memory allocation in his program, consider the C program in Figure 1. In this program, a simplified producer/consumer simulation, objects are randomly allocated by two producers and freed by the consumer. The function `random_flip`, which is not shown, randomly returns 1 or 0 with equal probability. The function `consume_widget`, which is responsible for freeing the memory allocated, contains a bug and does not free red widgets. If the simulation ran for a long time, memory would eventually be exhausted, and the program would fail.

In the example, `make_widget` is the only function that allocates memory directly. To fully understand the allocation behavior of the program, we must know which functions called `make_widget` and hence were indirectly responsible for memory allocation. `Mprof` provides this information.

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

```c
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;
}
```

```c
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 Program

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 describes each of the tables for the C program in Figure 1.

2.2 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 a programmer which functions allocated the memory associated with memory leaks. The table contains a list of partial call paths that resulted in memory that was allocated and not subsequently freed. The paths are partial because complete path information is not recorded; only the last five callers on the call stack 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 contains the memory leak table for our example.

```
allocs     bytes (%)      path

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

Figure 2: Memory Leak Table for Producer/Consumer Example

In larger 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.

2.3 The Allocation Bin Table

A major part of understanding the memory allocation behavior of a program is knowing what objects were allocated. In C, memory allocation is done by object size; the type of object 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 a programmer recognize which data structures consume the most memory and allows him to concentrate any space optimizations on them.

The allocation bin table breaks down object allocation by the size, in bytes, of allocated objects. 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

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 these were eventually freed, resulting in 1,023,876 bytes of memory lost to the memory leak. The percentages show what fraction of all bins a particular bin contributed. This information is provided to allow a user to rapidly determine which bins are of interest (i.e., contribute a substantial percentage). 99% is the largest percentage possible because we chose to use a 2 character field width.

2.4 The Direct Allocation Table

Another facet of understanding memory allocation is knowing which 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 the functions that allocate 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. The direct allocation table corresponds to the flat profile generated by `gprof`.

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

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

```
     % 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

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`.

2.5 The Allocation Call Graph

Understanding the memory allocation behavior of a programs sometimes requires more information than just knowing the functions that are directly responsible for memory allocation. Sometimes, as happens in Figure 1, the same allocation function is called by several different functions for different purposes. 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 and corresponds to the call graph profile generated by `gprof`.

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

[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).

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

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

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

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

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

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

Figure 5: Allocation Call Graph for Producer/Consumer Example

function is easy to identify in each table entry because a large 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 five callers in the call chain, the *partial call chain*, with the object allocated. `mprof` augments every object allocated with two items: an integer which is the object size as requested by the user (since the allocator may allocate an object of a different size for convenience), and a pointer to a structure that contains the object's partial call chain and a count of allocations and frees of objects with that call chain. A hash table is used to map a partial call chain to the structure containing the counters. When an object is allocated, its partial call chain is used as a hash key to retrieve the structure containing the counters. A pointer to the structure is placed in the allocated object and the allocation counter is incremented. When the object is later freed, the pointer is followed and the counter of frees is incremented. 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 a 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, which is described in the next section.

3.2 Constructing the Allocation Call Graph

To construct the allocation call graph, the monitor must associate the number of bytes allocated with every function on the current dynamic call chain, each time `malloc` is called. Consider the sample call chain in Figure 6, which we abbreviate: `main`->`foo`->`bar`(24).

```
CALL STACK:                       MPROF RECORDS:
main calls foo                         24 bytes over main -> foo
      foo calls bar                    24 bytes over foo -> bar
         bar calls malloc(24)          24 bytes allocated in bar
```

Figure 6: Example of a Dynamic Call Chain

In `mprof`, the monitor traverses the entire call chain by following return addresses. This differs from `gprof`, where only the immediate caller of the current function is recorded. `gprof` makes the assumption that each call takes an equal amount of time and uses this assumption to reconstruct the complete dynamic call graph from information only about the immediate callers. In `mprof`, we actually traverse the entire dynamic call chain and need to make no assumptions.

In choosing to traverse the entire call chain, we have elected to perform an operation that 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 (i.e., in the example we would output [`main`->`foo`->`bar`, 24]). Considering that many programs execute millions of calls to `malloc` and that the depth of a call chain can be hundreds of functions, the amount of information could 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 chain. For the call in the example, we could maintain the pairs [`main`, `foo`] and [`foo`, `bar`], and associate 24 bytes with each pair. Conceptually, the data structure our monitor maintains is an association between caller/callee pairs and the cumulative bytes allocated over the pair, which we denote ([`main`, `foo`], 24). To continue with the example, if the next allocation was: `main`->`foo`->`otherbar`(10), where this is the first call to `otherbar`, we would update the byte count associated with the [`main`, `foo`] pair to 34 from 24. Furthermore, we would create a new association between [`foo`, `otherbar`] and the byte count, 10. A disadvantage with this implementation is that the exact call chains are no longer available. However, from the pairs we can construct the correct dynamic call graph of the program, which is the information that we need for the allocation call graph.

For the overhead imposed by the monitor to be reasonable, we have to make the association between caller/callee pairs and cumulative byte counts fast. 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. In an effort to decrease the number of hash lookups, 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 at which 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 (see section 4.2).

3.3 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 four 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 7. 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.

```
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 7: Problems Caused by Recursive Calls

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 7, 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 and conflicts with our goal to make the monitor as unobtrusive as possible.

The solution that 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 elsewhere [2]. 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.

3.4 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 in which addresses can be quickly mapped to function names, memory profiling in the style of `mprof` is not a difficult problem. In systems in which symbolic names are not available in compiled code, profiling is more difficult. Furthermore, many systems open-code important allocation functions, like `cons`. Because open-coded allocation functions will not necessarily call a central allocation function (like `alloc_object`), such allocations will not be observed by `mprof`. To avoid such a loss of information, `mprof` should be used in conjunction with program declarations that will force allocation functions such as `cons` to be coded out-of-line.

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 FIDIL, a programming language for finite difference computations [3]. The third program, `epoxy`, is an electrical and physical layout optimizer written by Fred Obermeier [5]. The fourth program, `crystal`, is a VLSI timing analysis program [6]. These tests represent a small program (`example`, 100 lines); a medium-sized program (`fidilrt`, 7,100 lines); and two large programs (`epoxy`, 11,000 lines and `crystal`, 10,500 lines). In the remainder of this section, we will look at the resource consumption of `mprof` from two perspectives: execution time overhead and space consumption.

4.1 Execution Time Overhead

There are two sources of execution time overhead associated with `mprof`: additional time spent monitoring an application and the time to reduce and print the data produced by the monitor. The largest source of monitor overhead is the time required to traverse the complete call chain and associate allocations with caller/callee pairs. We implemented a version of `mprof`, called `mprof-`, which does not create the allocation call graph. With this version, we can see the relative cost of the allocation call graph. The ratio of the time spent with profiling to the time spent without profiling is called the *slowdown factor*. Table 1 summarizes the execution time overheads for our four applications. Measurements were gathered running the test programs on a VAX 8800 with 80 megabytes of physical memory.

The slowdown associated with `mprof` varies widely, from 1.5 to 10. `crystal` suffered the worst degradation from profiling because `crystal` uses a depth-first algorithm that results in long call chains. Programs without long call chains appear to slow down by a factor of 2–4. If the allocation call graph is not generated and long call chains are not traversed, the slowdown is significantly less, especially in the extreme cases. Since `mprof` is a prototype and has not been carefully optimized, this overhead seems acceptable. From the table, we see the reduction and display time is typically less than a minute.

Resource Description	Cost			
	`example`	`fidilrt`	`epoxy`	`crystal`
Number of allocations	100000	77376	306295	31158
Execution time with `mprof` (seconds)	62.7	132.7	188.8	134.1
Execution time with `mprof-` (seconds)	44.1	116.0	149.7	25.5
Execution time without `mprof` (seconds)	17.9	107.1	52.1	13.2
Slowdown using `mprof`	3.5	1.2	3.6	10.1
Slowdown using `mprof-`	2.5	1.1	2.9	1.9
Reduction and display time (seconds)	10.3	28.8	28.3	36.8

Table 1: Execution Time Overhead of `mprof`

4.2 Storage Consumption

The storage consumption of `mprof` is divided into the additional memory needed by the monitor as an application executes, and the external storage required by the profile data file. The most significant source of memory used by the monitor is the data stored with each object allocated: an object size and a pointer needed to construct the memory leak table. 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 application is finished. We measured how many bytes of memory and disk space are needed to store this information. Table 2 summarizes the measurements of storage consumption associated with `mprof`.

Resource Description	Cost			
	`example`	`fidilrt`	`epoxy`	`crystal`
Number of allocations	100000	61163	306295	31158
User memory allocated (Kbytes)	20000	2425	6418	21464
Per object memory (Kbytes)	781	477	2393	168
Other monitor memory (Kbytes)	8.7	23.3	52.3	17.5
Total monitor memory (Kbytes)	790	500	2445	186
Monitor fraction (% memory allocated)	4	17	28	1
Data file size (Kbytes)	4.5	8.1	28.6	9.6

Table 2: Storage Consumption of `mprof`

The memory overhead of `mprof` is small, except that an additional 8 bytes of storage are allocated with every object. In programs in which many small objects are allocated, like `epoxy`, `mprof` can contribute significantly to the total memory allocated. Nevertheless, in the worst case, `mprof` increases application size by 1/3, and since `mprof` is a development tool, this overhead seems acceptable. From the table we also see that the data files created by `mprof` are quite small (< 30 Kbytes).

5 Related Work

Mprof is similar to the tool gprof [2], 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: because the information being displayed is quite similar and 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 [1]. Their approach concentrated on finding two specific storage allocation errors: memory leaks and duplicate frees. They modified malloc and free so that every time that 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 data files it generates are of modest size (see the table above).

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 a programmer can detect memory leaks and identify major sources of allocation.

Unlike gprof, mprof records every caller in the call chain every time an object is allocated. The overhead for this recording is large but not impractically so, because we take advantage of the fact that a 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 in which allocation takes place, even in very large programs. We have measured the overhead of mprof, and find that typically it slows applications by a factor of 2–4 times, and adds up to 33% to the memory allocated by the application. Because mprof is intended as a development tool, these costs are acceptable.

Because mprof merges cycles caused by recursive function calls, mprof may be ineffective for programs with large cycles in their call graph. Only with more data will we be able to decide if many programs (especially those written in Lisp) 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]Using mprof, we noted that for a large object-oriented program written in KCL, the system function

As a final note, we have received feedback from C application programmers who have used `mprof`. They report that the memory leak table and the allocation bin table are both extremely useful, while the direct allocation table and the allocation call graph are harder to understand and also less useful. Considering the execution overhead associated with the allocation call graph and the complexity of the table, it is questionable whether the allocation call graph will ever be as helpful C programmers as the memory leak table. On the other hand, with automatic storage reclamation, the memory leak table becomes unnecessary. Yet for memory intensive languages, such as Lisp, the need for effective use of the memory is more important, and tools such as the allocation call graph might prove very useful. Because we have limited feedback from Lisp programmers using `mprof`, we cannot report their response to this tool.

References

[1] 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.

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

[3] Paul N. Hilfinger and Phillip Collela. FIDIL: A language for scientific programming. Technical Report UCRL-PREPRINT 98057, Lawrence Livermore National Laboratory, January 1988.

[4] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[5] Fred Obermeier and Randy Katz. EPOXY: An electrical and physical layout optimizer that considers changes. Technical Report UCB/CSD 87/388, UCBCS, November 1987.

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

[7] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, Palo Alto, California, July 1985.

`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%.