CustoMalloc: Efficient Synthesized Memory Allocators

Dirk Grunwald Benjamin Zorn
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

CU-CS-602-92 July 1992

University of Colorado at Boulder

Technical Report CU-CS-602-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

CustoMalloc: Efficient Synthesized Memory Allocators [*]

Dirk Grunwald Benjamin Zorn
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

July 1992

Abstract

The allocation and disposal of memory is a ubiquitous operation in most programs. Rarely do programmers concern themselves with details of memory allocators; most assume that memory allocators provided by the system perform well. Yet, in some applications, programmers use domain-specific knowledge in an attempt to improve the speed or memory utilization of memory allocators.

In this paper, we describe a program (CustoMalloc) that synthesizes a memory allocator customized for a specific application. Our experiments show that the synthesized allocators are uniformly faster than the common binary-buddy (Bsd) allocator, and are more space efficient. Constructing a custom allocator requires little programmer effort. The process can usually be accomplished in a few minutes, and yields results superior even to domain-specific allocators designed by programmers. Our measurements show the synthesized allocators are from two to ten times faster than widely used allocators.

1 Introduction

The allocation and disposal of memory is a ubiquitous operation in most programs, yet one largely ignored by most programmers. Some programmers use domain-specific knowledge in an attempt to improve the speed or memory utilization of memory allocators; however, the majority of programmers use the memory allocator provided in a given programming environment, believing it to be efficient in time or space. In many virtual memory systems, space efficiency is usually a secondary concern, although important in some applications.

In this paper, we describe a program (CustoMalloc) that synthesizes a memory allocator customized for a specific application. Our experiments show that the synthesized allocators are uniformly faster than the common binary-buddy (Bsd) allocator, and are more space efficient.

[*]This material is based upon work supported by the National Science Foundation under Grants No. CCR-9010624, CCR-9121269 and CDA-8922510

Constructing a custom allocator requires little programmer effort, with the process taking only a few minutes. The results are typically superior to domain-specific allocators designed by programmers with detailed knowledge of the application. Our measurements show the synthesized allocators are from two to ten times faster than widely used allocators.

The central idea of CustoMalloc is simple: the allocation pattern of a program is measured across multiple input sets, and a customized memory allocator is generated using the information collected in the measurement phase. A similar strategy has been used to produce the QuickFit memory allocator [Weinstock & Wulf 88, Standish 80]; however, the QuickFit algorithms depends on the observation that many programs allocate a large number of small objects and generalizes that common condition, treating all programs similarly. By comparison, CustoMalloc tailors the allocator design to each application, and can accomodate applications that do not fit this pattern.

During a study to determine appropriate models of program allocation [Zorn & Grunwald 92b, Zorn & Grunwald 92a], we noticed a common pattern in many of the programs we examined. We recorded the frequency of different *size classes*, or amount of memory requested for each allocation request. In most applications, we found that a small number of size classes dominanted the range of object allocation sizes. This observation is not unique; indeed, similar observations motivated the QuickFit algorithm[Bozman et al. 84, DeTreville 90, Batson et al. 70, Margolin et al. 71]. However, by examining a broader spectrum of programs, we noted that the occurence of size classes was variable; in most applications, very small object sizes were very frequent, while in other applications, other sizes dominated.

We also noted that the *transition probability* between sizes classes was much smaller than that within a class; this meant that once a program allocated a particular object size, it tended to allocate that object size frequently. In effect, there was a great degree of temporal locality in the sizes of memory requests. Numerous memory allocation mechanisms have also made use of this observation; for example, Oldehoft [Oldehoeft & Allan 85] measured the performance of adaptive caching strategies for memory allocation. Likewise, numerous programs use this property for *ad hoc* allocation strategies, that is, static caching strategies based on the (commonly misjudged) relative allocation frequency of objects of different sizes. The CustoMalloc memory allocator also uses such information using direct measurements of the application across a spectrum of program inputs.

In the next section, we describe the design and structure of CustoMalloc. In §3, we show the accuracy of measuring allocation frequency from multiple program executions. The design of memory allocators is often considered to be a prosaic topic; reports of increasing memory efficiency by a few percent appear common, and the total benefit to program execution time is often difficult to discern. In contrast, in §4 we measure the performance of CustoMalloc in absolute and relative terms with a number of actual applications. We have found that differences between memory allocation strategies can be significant; our CustoMalloc implementation shows significant improvement over widely used algorithms that are considered to be extremely fast. We show how a commercial memory allocator can consume over 20% of *total program execution time*; by comparison, the allocator produced by CustoMalloc consumes a tenth that amount. We conclude in §5 with observations and future work.

2 The Design Of CustoMalloc

The CustoMalloc allocator first measures the allocator behavior for an application and then generates a customized memory allocator using that information.

2.1 Measurement

Prior to measurement, the user executes a command that produces a source file containing the measurement-based memory allocator. This file is compiled and linked with the application, replacing the memory allocator normally provided. The measurement allocator records the frequency of memory allocations and deallocations (e.g., via the `malloc` and `free` functions on UNIX). Information for each memory request is stored in a list of *cached freelists.* The measurement allocation pool is structured as a list of lists; each list maintains statistics for objects of a specific size and a pointer to a freelist of objects at least that large.

Although our previous study [Zorn & Grunwald 92b] showed that a plurality of requests were for a specific number of bytes, it also showed that a greater majority could be satisfied by rounding the request size to a small R-byte size boundary. A request for B bytes then becomes a request for $R\lfloor(B + R - 1)/R\rfloor$ bytes. In any case, the architectures used in our experimentation required that objects be stored on quad-byte boundaries, necessitate rounding of some sort. Thus, we round all measured memory requests to a number of *allocation units.* In CustoMalloc, the allocation unit size can be specified by the user, with the default allocation unit being 32 bytes. Allocation requests fall into a small number of distinct *size classes*, based on the number of allocation units requested. These size class units are used to partition the freelists in each algorithm examined; for example, all requests for objects from 96 to 127 bytes fall into the single size class of objects that have at least 96 bytes allocated.

Using the instrumented application, we count the number of allocation and disposal requests for each size class; moreover, we measure the maximum and mean number of objects allocated and on the freelist. This provides a coarse summary of the allocation history of the program. We also measure the freelist length each time an item is added to or removed from the freelist to compute the mean freelist length. In effect, the number of allocations and releases defines a time order for mean usage. Our previous study[Zorn & Grunwald 92b] showed this is an accurate approximation to the true mean freelist length.

```
MallocPtrType malloc(MallocArgType bytes)
{
  unsigned int size = size_external_to_internal(bytes);

  if ( size == 32 ) {
    if (__customalloc_FreeList[0]) return(__customalloc_unlink(0));
    else return( __fast_malloc(32) );
  }
  else
    if ( size == 64 ) {
      if (__customalloc_FreeList[1]) return(__customalloc_unlink(1));
      else return( __general_malloc(64) );
    }
                        ....

  return __general_malloc(size);
};
```

Figure 1: Sample Memory Allocator Code Fragment

The data collected by the measurement allocator is stored in a file when the program finishes execution. Successive runs of the instrumented application update the contents of the file. To capture the typical behavior of the application, the instrumented application should be run with multiple input sets. Our experience has shown that a small number of executions are sufficient to provide a "typical" allocation profile for a given program. Most object sizes that are allocated are related to particular data structures in the application and do not vary between executions. Others, typically strings or bitmaps, are usually small or infrequently allocated. If they are infrequently allocated, the time to allocate those objects contributes little to the total execution time of the program, and performance does not suffer by missing or ignoring information about those objects.

2.2 Synthesized Allocator

Once the measurement has been completed, a customized allocator is created using the CustoMalloc program. The synthesized allocator is composed of freelists and two internal memory allocators. Objects for specific size classes are stored on separate freelists.

For each memory request, the request size is rounded to a number of allocation units, after which, we locate the appropriate freelist for that object size. Freelists are examined in order of frequency of occurrence, reducing the mean number of freelists considered when searching for a particular size class. Figure 1 shows a sample of the synthesized allocator using the C language. We found that explicitly ordering the freelist search based on allocation frequency was faster than

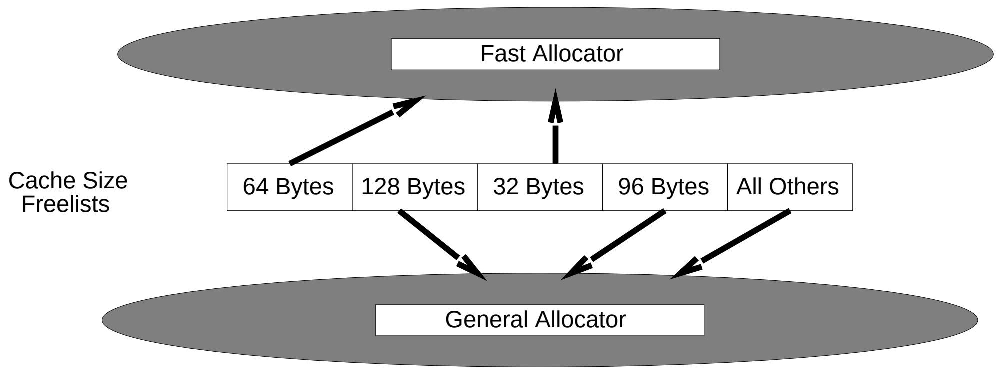


Figure 2: Structure of synthesized memory allocator

using an equivalent `switch` or `case` statement. Switch statements typically consume more cycles computing a general index function and require more active registers; by comparison, the code sequence in Figure 1 can take advantage of measured allocation frequency, reducing the mean number of instructions executed.[1]

When preallocated objects are not found in the appropriate freelist, one of two internal allocators are used to create new objects. All objects within a particular size class are allocated by the same internal allocator. As Figure 2 suggests, we use both a "fast allocator" and a more general internal allocator. If the recorded statistics suggest that there is little advantage to reclaiming the store of a particular size class, we use the fast allocator. The fast allocator has no mechanism for coalescing or recovering storage. Objects allocated by the fast allocator must always be returned to the freelist structure or their storage will be lost. The fast algorithm allocates from contiguous extents; allocation involves updating pointers and possible allocating new contiguous extents.

The synthesized allocator must also use a general allocator internally because some object sizes may not be represented in the freelist. Likewise, some size classes may have their storage reclaimed from the freelists, and thus require a more general allocator. We use an efficient first-fit coalescing allocator for the general allocator[Lea]. In §4, we compare the performance of CustoMalloc to that allocator as well as a number of other allocation algorithms.

The decision to use the fast or general allocator is based on the ratio of the mean freelist length to the mean number of allocated objects in use. If the mean freelist length for a size class is small, there is little advantage in reclaiming storage from that freelist – there is very little to reclaim. Likewise, if the mean freelist length is large with respect to the mean number of objects in use, this indicates that the size class undergoes episodic or periodic use. For example, in one application we measured, a large number of objects for a particular size class are allocated in a sparse-matrix

[1]As a side note, our implementation assumed the majority of function calls shown in Figure 1 can be inlined or macro-expanded.

subroutine. The objects are then returned and, although that size class is still the most frequently referenced class, the remainder of the program does not use the large number of objects allocated by the sparse matrix algorithm. Thus, if storage efficiency is important, storage from this size class can be reclaimed. The total number of calls to the allocation and free routines also guide this strategy. Some program we measured have size classes where storage is *never* returned; we always use the fast allocator for those size classes.

The selection criteria for choosing the fast or general allocator can be set by the user. The only advantage of the general allocator is that storage may be returned and later used to allocate other objects. When recovering storage, we want to return as little storage as needed to satisfy the current request to the general allocator. For each size class, we compute $P_m(s)$ and $P_f(s)$, the probability that a particular call to **`malloc`** or **`free`** will concern an object of size s. Likewise, we have already recorded $\hat{F}(s)$, the mean size of the freelist during the measurement phase. We examine each freelist that uses the general allocator, ordered by the prioritization function $P_f(s)(1-P_m(s)) \times s\hat{F}(s)$. The intuition behind this prioritization is that, in a steady state, $s\hat{F}(s)$ represents the mean number of bytes that can be recovered from the free list for objects of size s. The term $P_f(s)$ favors size classes that frequently return objects to the freelist – there is little point in examining a freelist if storage is rarely returned for that size class. Similarly, the term $1 - P_m(s)$ favors size classes that tend to *not* allocate storage, because they will simply need to allocate storage again. This prioritization is computed when the allocator is synthesized, not during execution time. Storage is reclaimed from the freelists until twice the amount requested by the general allocator is returned. If this storage can not be coalesced into a sufficiently large block, more storage is requested from the operating system.

In §4, all synthesized allocators use the standard options to produce an allocate that (usually) combines the fast and general allocator. If allocator speed is particularly important, the user can indicate that all size classes use the fast allocator; however, we have found that the allocator chosen by the heuristics has excellent performance.

The structure of the memory deallocation routine, shown in Figure 3, is similar to that for allocation. Each storage object is tagged with it's allocation size and the freelists are examined in the order of deallocation frequency rather than allocation frequency. We encountered some programs where the orderings by allocation and deallocation frequencies differed. In cases when the UNIX **`realloc`** function is used, objects allocated by the the fast allocator may not exactly match a specific size class. In this infrequent case, the objects are returned to the freelist with the closest suitable size.

In general, we do not dedicate a freelist for each size class in the application. In the next section, we see that over 95% of all allocation requests can usually be satisfied by four freelists for most programs we examined. We cull the number of freelists to reduce allocator complexity – the

```
FreeRetType free(FreePtrType p)
{
  if ( p ) {
    MallocChunk *chunk = external_to_malloc(p);
    int size = size_malloc_to_internal(malloc_size(chunk) - (2 * SIZE_SZ));
    if ( size == 96 ) {
      __customalloc_link(2, p); return;
    }
    else
      if ( size == 64 ) {
        __customalloc_link(1, p); return;
      }
      else
              .....

    if (is_fast(p)){ __fast_free(p, size); return;};
    __general_free(p);
  };
};
```

Figure 3: Sample Memory Deallocator Code Fragment

small number of allocations of the remaining infrequent size classes contribute little to the overall program execution time.

We also found that frequently allocated sizes classes were usually related to structures or records defined in the application. A common idiom in the C language is

```
        FooPtr *foo = (FooPtr) malloc( sizeof(struct Foo) );
```

where the size of the allocation request is known at compile time. To take advantage of this, we also generated a version of the `malloc` and `free` routines that could be "inlined", or compiled without procedure calls. Not only does this remove function call overhead (approximately 20% of the average CustoMalloc allocator cost), but the computation of the rounded allocation request size can be computed by the compiler (approximately 10% of the average call in CustoMalloc).

To reduce the chance of potential code explosion from aggressive inlining, we include only the most frequent size classes in the inlined function; the remainder are handled by another routine. Consider an example using the SPARC architecture. If the allocation request is similar to the idiomatic usage shown above, the combination of dead-code elimination and constant propagation in the compiler remove the examination of extraneous freelists, leaving only the following eight instructions to be executed if items are available on the freelist:

```
sethi %hi(___customalloc_FreeList),%o0          ;; load free list
ld [%o0+%lo(___customalloc_FreeList)],%i0
cmp %i0,0                                       ;; check if empty
be L32
or %o0,%lo(___customalloc_FreeList),%o1         ;; if not, remove
ld [%i0],%o0
b L29
st %o0,[%o1+0]                                  ;; and link next item
```

If the freelist is empty, the fast allocator takes an additional twenty one instructions. These results are similar to those noted by others[Weinstock & Wulf 88].

We stress that this is not a contrived or atypical example; the previous code fragment was taken from one of the applications we instrumented. The program modifications to include the inlined allocators are fairly simple, again taking a few minutes. The inlined example is illustrative because it defines an easily achievable lower-bound for memory allocation – basically, any allocation scheme that can reuse storage must take at least this many instructions. As we see in §4, the non-inlined allocator produced by CustoMalloc takes approximately twice as long as this lower bound. Although the inlined version is easy to use, we do not use it comparisons in this paper.

3 Inter-run Allocation Frequency Accuracy

The success of CustoMalloc is predicated on being able to accurately measure the allocation frequency of different size classes. We also tacitly assume that the size class orders, determined by allocation frequency, is similar between different runs of the same program, even with different input data. In this section, we show these assumptions are generally valid.

3.1 Sample Programs

We used seven allocation intensive programs, listed in Table 1, to compare the similarity of allocation frequency between different inputs and to compare the performance of CustoMalloc to other allocators. Previously, we examined numerous synthetic models for comparing memory allocators[Zorn & Grunwald 92a, Zorn & Grunwald 92b], and considered using one of those models. However, to measure the sensitivity of allocation distribution to the input set, we needed to instrument actual applications; once this was done, it was as simple to use those applications to compare performance. The programs were all written in the C programming language. The version of yacr that we measured did not release much allocated memory by calling `free`. The empirical behavior of these problems is presented in [Zorn & Grunwald 92b].

We had limited detailed knowledge about the sample programs; we did not have to concern ourselves with the internal structure of the programs because measurement was performed by

CFRAC	A program to factor large integers using the continued fraction method. The inputs are products of two large primes.
GS	GhostScript, version 2.1, is a publicly available interpreter for the PostScript page-description language. The input files were a variety of small and large files, including an 126 page user manual. This execution of GhostScript did not run as in interactive application as it is often used, but instead was executed with the NODISPLAY option that simply forces the interpretation of the Postscript without displaying the results.
PERL	Perl 4.10, is a publicly available report extraction and printing language, commonly used on UNIX systems. A number of input scripts were used.
YACR	YACR (Yet Another Channel Router), version 2.1, is a channel router for printed circuit boards. The input files are provided with the YACR release.
GAWK	Gnu Awk, version 2.11, is a publicly available interpreter for the AWK report and extraction language. A number of scripts were used.
MAKE	Gnu-make, version 3.62 is a version of the common 'make' utility used on UNIX. Different input sets were provided by using the instrumented MAKE to build other sample programs.
ESPRESSO	Espresso, version 2.3, is a logic optimization program. The input files were provided with the release code.

Table 1: General Information about the Test Programs

the CustoMalloc program. Size classes were rounded to 32-byte boundaries. Table 2 shows the relative ordering of the four dominant size classes, ordered by the allocation frequencies that are shown in parentheses. We selected two sample inputs from the suite of available inputs and measured each application. We recorded the actual distribution for the first ("Input 1") and second ("Input 2") runs, and the distribution from both runs ("Input 1&2").

The most notable feature of Table 2 is that a single size class usually dominates all allocation requests (from 85% to 100% of all allocations, depending on the application) and that 95% of all allocations were matched by no more than four size classes for all input sets in all applications.

Secondly, for all applications other than GS, the relative allocation frequency order does not change significantly across the input files. We compute allocation frequency relative to the total number of allocations, giving equal weight to allocations occurring in different execution runs. For GS, the first input set had significantly more allocations than the second. Thus, although there was a substantial difference in allocation frequency between the two input sets, the larger input ("Input 1") dominated the combined allocation frequency.

Even in the GS application, only the *relative order* of the four most frequent object sizes changes; no new objects sizes appear. Each conditional statement in the allocator adds $\approx$ 3 instructions the execution time; thus, even if the allocation order is not completely accurate, the additional

CFRAC	Input 1	32 (100.0%)	288 (0.0%)	160 (0.0%)	64 (0.0%)
	Input 2	32 (99.9%)	64 (0.1%)	288 (0.0%)	352 (0.0%)
	Input 1&2	32 (99.9%)	64 (0.1%)	288 (0.0%)	160 (0.0%)
GS	Input 1	64 (77.9%)	288 (5.9%)	32 (5.9%)	128 (4.5%)
	Input 2	128 (37.7%)	288 (37.2%)	64 (12.5%)	32 (7.9%)
	Input 1&2	64 (71.0%)	288 (9.2%)	128 (8.0%)	32 (6.1%)
PERL	Input 1	32 (90.8%)	64 (3.7%)	96 (2.4%)	160 (1.6%)
	Input 2	32 (94.7%)	96 (4.4%)	64 (0.9%)	1216 (0.0%)
	Input 1&2	32 (90.8%)	64 (3.7%)	96 (2.4%)	128 (1.5%)
YACR	Input 1	32 (99.0%)	160 (0.4%)	3488 (0.4%)	1280 (0.3%)
	Input 2	32 (98.1%)	1696 (1.2%)	928 (0.5%)	96 (0.1%)
	Input 1&2	32 (98.9%)	160 (0.4%)	3488 (0.4%)	1280 (0.3%)
GAWK	Input 1	32 (86.0%)	256 (7.2%)	64 (6.1%)	96 (0.8%)
	Input 2	32 (86.4%)	256 (6.9%)	64 (5.9%)	96 (0.8%)
	Input 1&2	32 (86.3%)	256 (7.0%)	64 (5.9%)	96 (0.8%)
MAKE	Input 1	32 (96.8%)	64 (2.0%)	224 (0.7%)	128 (0.2%)
	Input 2	32 (90.8%)	64 (4.8%)	224 (3.7%)	96 (0.2%)
	Input 1&2	32 (94.5%)	64 (3.1%)	224 (1.8%)	128 (0.2%)
ESPRESSO	Input 1	32 (85.5%)	64 (7.0%)	96 (2.0%)	128 (1.1%)
	Input 2	32 (84.3%)	64 (6.5%)	96 (3.0%)	544 (2.0%)
	Input 1&2	32 (85.5%)	64 (7.0%)	96 (2.0%)	128 (1.1%)

Table 2: Allocation Frequency With Different Runs
(Each entry shows the object size and its allocator frequency)

overhead is minimal. If the application is made to use the inlined functions and the idiomatic usage shown at the end of §2 is common, the relative allocation frequency is even less important, because the compiler selects the appropriate freelist directly. The GS application is interesting because the dominant size classes are not small; allocation algorithms using allocation frequencies based on heuristics or anecdotal observation, such as QUICKFIT, may perform poorly for this application.

We have examined a large number of applications in addition to those mentioned here, using a number of input sets, and our experience has shown that the pattern evinced by the data in this section is representative of most programs:

- Programs typically allocate a small number of size classes.
- A small subset of the size classes dominate allocation frequency.
- Although the relative ordering of the dominant size classes can change between runs of a program, the changes are usually minor.

4 Performance Comparison

In this section, we compare the CUSTOMALLOC allocator to implementations of a number of other algorithms. In the algorithms we implemented (QUICKFIT, ADAPTIVECACHE, CUSTOMALLOC) considerable effort was taken to optimize the code. For other algorithms (FIRSTFIT, BSD), efficient, commonly used implementations were used. Source code was not available for the last algorithm (CARTESIAN), but it is provided with a widely used operating system, and we assume it has been extensively optimized.

FIRSTFIT This algorithm, described by Knuth, is a straightforward implementation of a first-fit strategy with several optimizations [Knuth 73]. We measured a publicly available implementation of the classic Knuth algorithm written by Doug Lea. This variant uses an array of freelists. In each freelist, free blocks are connected together in a double-linked list. An appropriate freelist is selected based on the log of the allocation request; this is done to increase the probability of a better fit. During allocation the selected freelist is scanned for the first free block that is large enough. The block found is split into two blocks, one of the appropriate size, and returned. As an optimization, if the extra piece is too small (in this case less than 16 bytes), the block is not split.

This implementation is used as the "general allocator" in the CUSTOMALLOC allocator. Comparison to other "first-fit" implementations indicates this is very a efficient implementation.

BSD As an alternative to a more conventional first-fit algorithm, Chris Kingsley implemented a very fast buddy algorithm that was distributed with the 4.2 BSD Unix release [Kingsley 82]. Kingsley's algorithm allocates objects in a limited number of different size classes, namely powers of two minus a constant. Allocation requests are rounded up to the nearest size class and a freelist of objects of each size class is maintained. If no objects of a particular size class are available, more storage is allocated. No attempt is made to coalesce objects.

Because this algorithm is so simple, it is also easy to provide a fast implementation. On the other hand, it also wastes considerable space, especially if the size requests are often slightly larger than the size classes provided. This algorithm illustrates one extreme of the time/space tradeoffs possible in dynamic storage management. Interestingly, its widespread use would suggest that users often consider CPU performance more important than memory usage in these systems (or, perhaps, users are not aware of the penalty).

CARTESIAN This algorithm, sometimes called "better-fit", is provided by the Sun Operating System library routines `malloc` and `free`[Stephenson 83, Sun 90]. Rather than place the free blocks in a linear list, they are placed in a Cartesian tree[Vuillemin 80]. Descendents in the tree are ordered both by address (left descendents have lower addresses than right descendents), and by size (descendents on the left are smaller than descendents on the right). This algorithm is attractive because the worst-case cost of all operations on the tree (allocation, deallocation, and moving blocks around) is $O(d)$, where d is the depth of the tree.

QUICKFIT This is the quick-fit algorithm described by [Weinstock & Wulf 88, Standish 80], and is somewhat similar to CUSTOMALLOC. Allocation requests less than 32 bytes are grouped into eight size classes rounded to four byte sizes. Allocations in those size classes use the same fast allocator used in CUSTOMALLOC. All other allocations use the same general allocator used in CUSTOMALLOC.

This algorithm provides a comparison to CUSTOMALLOC that applies the observation that *only* small objects tend to be allocated frequently. The performance of this algorithm is extremely sensitive to the size range selected for coverage by the freelists. In the literature, from four to 16 freelists are used. Our implementation of the QUICKFIT allocator manages objects from one to 32 bytes and allocates objects using more precise allocation units, rounding to four rather than 32 bytes.

ADAPTIVECACHE This algorithm is similar to CUSTOMALLOC in that it uses freelists for objects in different size classes. Allocation requests are rounded to 32 bytes. When an allocation is requested, the list of freelists is searched for an appropriate object size. If none is found, or the appropriate freelist is empty, the memory is allocated using the fast allocator used in CUSTOMALLOC. When an object is deallocated, the list of of freelists entries is again searched. If a freelist does not exist, a new freelist entry is created. In both allocation and deallocation, the most recently accessed freelist entry is moved to the front of the list of freelists. Variants on this algorithm have been suggested [Bozman 84, Oldehoeft & Allan 85, Leverett & Hibbard 82], although we were unable to find a previous implementation of our exact algorithm.

This allocator takes advantage of temporal locality in object size references; the freelist for the dominant object size will always be near the beginning of the list of freelists. This algorithm provides a comparison to CUSTOMALLOC, using dynamic rather than pre-computed allocation frequencies. Although the number of size classes considered by this allocator per allocation request may be lower than that for CUSTOMALLOC, the cost of moving entries to the front of the list can increase the mean time for allocation.

CustoMalloc This is the CustoMalloc described in this paper, using profiles from two of the available input sets. As mentioned, we did not use the inlined version of CustoMalloc in our measurements.

4.1 Experimental Design

Each program was compiled on a system using the SPARC architecture[2]. We compiled the programs using version 2.1 of the the Gnu C compiler with normal (-O) optimization levels enabled.

The data in this section is derived using "Input 1" from Table 2. We measured the memory efficiency of each allocator by recording the amount of dynamic memory requested from the operating system via the Unix **sbrk** function. This is a coarse metric, but does illustrate the peak usage requested by the program.

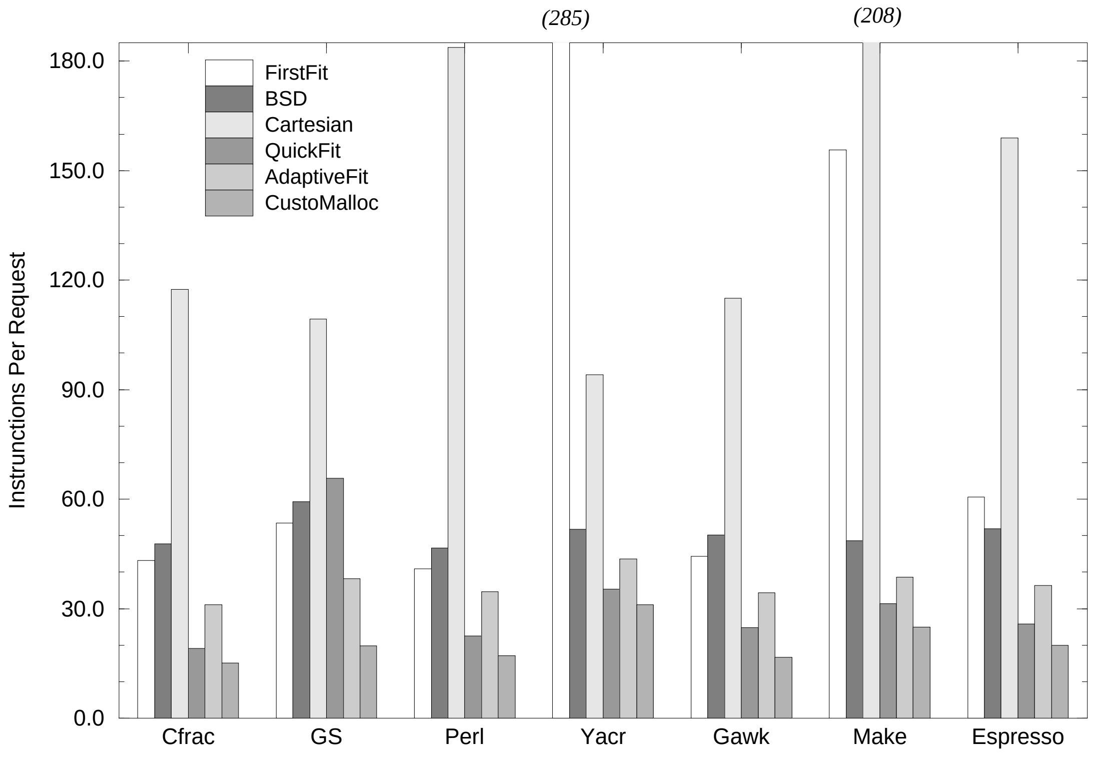
We measured execution time using the QP utility[Ball & Larus 92]. This tool provides a dynamic execution count for each subroutine in terms of instructions. This removes any variability in measuring execution time, greatly simplifying the experimental design. It also provides a more accurate and intuitive measure of the allocation time; as we shall see, allocation can take as little as sixteen cycles on average – accurately measuring this in a real application is very difficult. Moreover, the QP tool instruments the same binary used to generate the memory efficiency comparison. Unfortunately, QP does not account for secondary effects such as cache misses caused by poor data locality; these factors may be considered in a future study.

4.2 Performance: CPU Time

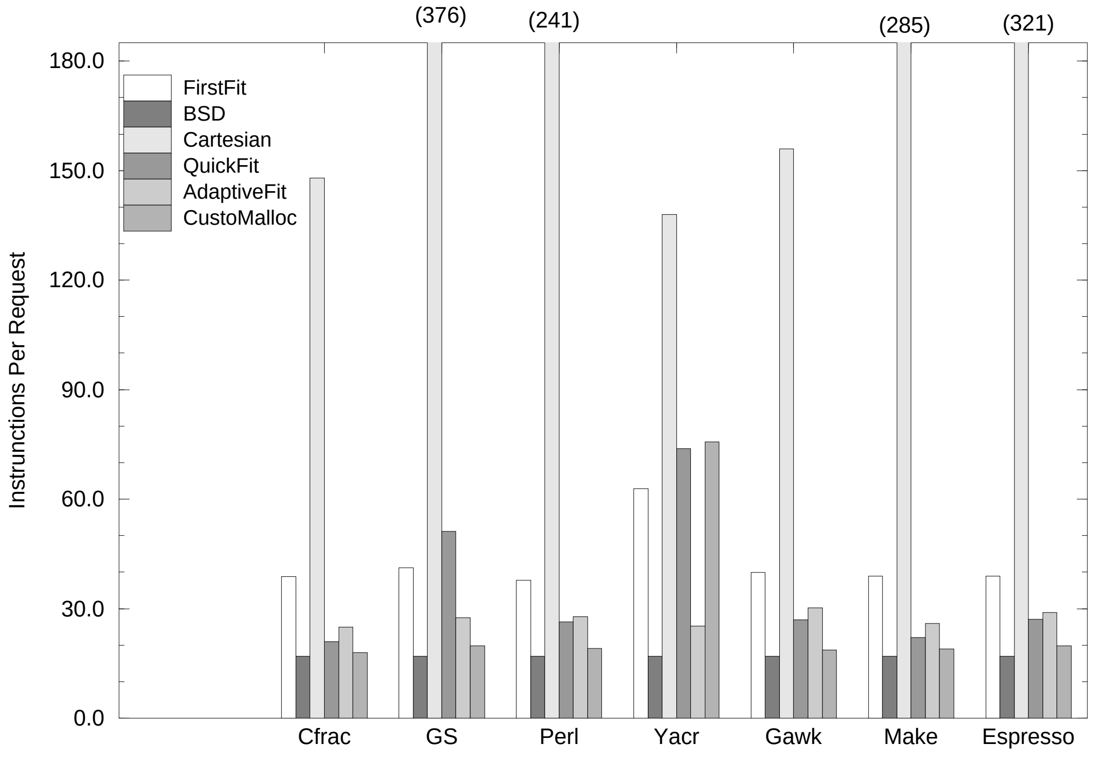
Figure 4 shows the number of machine instructions used during the average call to **malloc** (Figure 4(a)) and **free** (Figure 4(b)) for each allocation algorithm across the sample programs. Some values in Figure 4 are significantly larger than others, and were truncated to avoid obscuring the remaining data. Figure 5 shows the total percentage of executed instructions spent in the **malloc** and **free** subroutines for each application.

The most notable feature of Figure 4 is the range in the number of instructions needed for the different allocators. The Cartesian allocator is particular slow; this is surprising because it is distributed with a widely used operating system. Surprisingly, allocation using the Bsd algorithm is often *slower* than using FirstFit – however, the total allocation time (**malloc** *and* **free**) is faster when using Bsd (calls to **free** for Bsd take an average of $\approx$ 17 cycles across all applications). Figure 5 shows that the CustoMalloc allocator is consistently faster than Bsd; this is encouraging, because Bsd is widely considered to be a very fast algorithm.

[2]Due to our measurement strategy, we did not have to concern ourselves with the actual machine model.



(a) Allocation Time Per Request Across Sample Programs



(b)Deallocation Time Per Request Across Sample Programs

Figure 4: Allocation and Deallocation Times

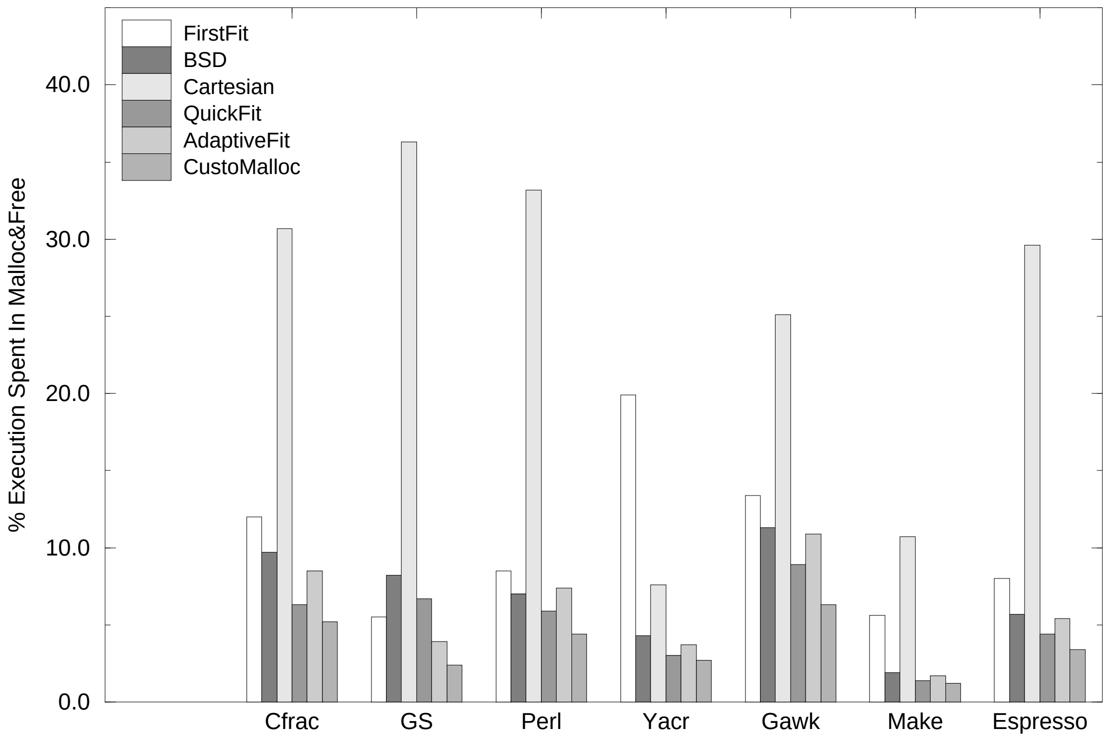


Figure 5: Total Percent of Execution Spent in **`malloc`** & **`free`**.

The three allocators with consistently good performance are AdaptiveCache, QuickFit and CustoMalloc. The CustoMalloc allocator is uniformly the fastest allocator across all the applications. This is particularly evident in the gs application. This application allocates very few objects less than 32 bytes ($\approx 7\%$); thus, the QuickFit algorithm incurs the overhead of QuickFit with little commensurate advantage. Indeed, even the AdaptiveCache algorithm, with its higher overhead, is faster for this application. The yacr application does not return much storage via **`free`**. Thus the time to deallocate memory for yacr, shown in Figure 4, is based on a small number of samples and is not very meaningful. Since every allocation in yacr requires new storage, the FirstFit algorithm has particularly poor performance, because it attempts to recover storage before requesting more storage from the system.

We found the QuickFit allocator to be extremely sensitive to the pre-selected size range managed by the freelists. Table 3 shows the total percentage of execution time spent in the **`malloc`** and **`free`** routines for a QuickFit implementation that handles objects of 32 bytes (eight freelists) or less and another that handles objects of 16 bytes or less (four freelists). The total allocation time is doubled when the smaller number of freelists is used. The number of freelists recommended in the literature ranges from two to sixteen freelists, with the general exhortation that the span "should cover the most common sizes" allocated by an application. This is clearly a possible, albeit

	Percentage Execution Spent in `malloc` and `free`						
	CFRAC	GS	PERL	YACR	GAWK	MAKE	ESPRESSO
QUICKFIT($\leq$ 32 byte)	6.0%	6.0%	5.7%	2.8%	8.6%	1.3%	4.2%
QUICKFIT($\leq$ 16 bytes)	9.3%	11.5%	8.0%	5.3%	14.4%	1.7%	7.7%

Table 3: Sensitivity of the QUICKFIT Allocator

infrequently exercised, option. In contrast, CUSTOMALLOC always determines the appropriate size range. Moreover, CUSTOMALLOC can use additional information to improve on the QUICKFIT allocator, even in cases where the underlying QUICKFIT algorithm is a very good choice. By design, CUSTOMALLOC can also handle cases where QUICKFIT shows little advantage (such as in GS). This latter point is particularly important in certain application domains; the CUSTOMALLOC allocator is being ported to a Cray system, where we hope to measure improvements on large scientific applications such as the Community Climate Weather Model. These applications may have allocation distributions that differ from the applications (compilers and the like) that originally provided the the anecdotal evidence to guide the design of QUICKFIT.

4.3 Performance: Memory Efficiency

The allocator synthesized by CUSTOMALLOC is uniformly the fastest allocator; however, experience has shown that fast allocators (e.g., BSD) typically waste considerable memory. Figure 6 shows the maximum amount of memory needed by each applications when using a particular allocator; the values are normalized to the space needed for the FIRSTFIT allocator.

As noted, Figure 6 shows that the BSD allocator takes consistently more memory than other allocators, and, that the CUSTOMALLOC allocator has good memory efficiency despite being faster than BSD.

The maximum memory request for CFRAC and YACR shown in Figure 6 shows one flaw in CUSTOMALLOC – rounding to 32 bytes. The ADAPTIVECACHE allocator, which also rounded to 32 byte allocation units has similar problems. Both CFRAC and YACR allocate a large number of very small (8 byte) objects. Furthermore, the YACR application never returns any items via `free`. In these cases, the 32 byte rounding used by CUSTOMALLOC (and ADAPTIVECACHE) causes significant memory overhead. However, we note that this is a rare occurrence in the applications we measured; generally, the memory demands of CUSTOMALLOC are close to the most space-efficient allocators. The scaled values used in Figure 6 can also obscure the impact of the memory efficiency; each allocator in the CFRAC application takes less than 82,000 bytes of storage, while the allocators for YACR take over 20,000,000 bytes. Due to the way storage is requested from the system (in 8192-bytes units), small differences appears significant in CFRAC.

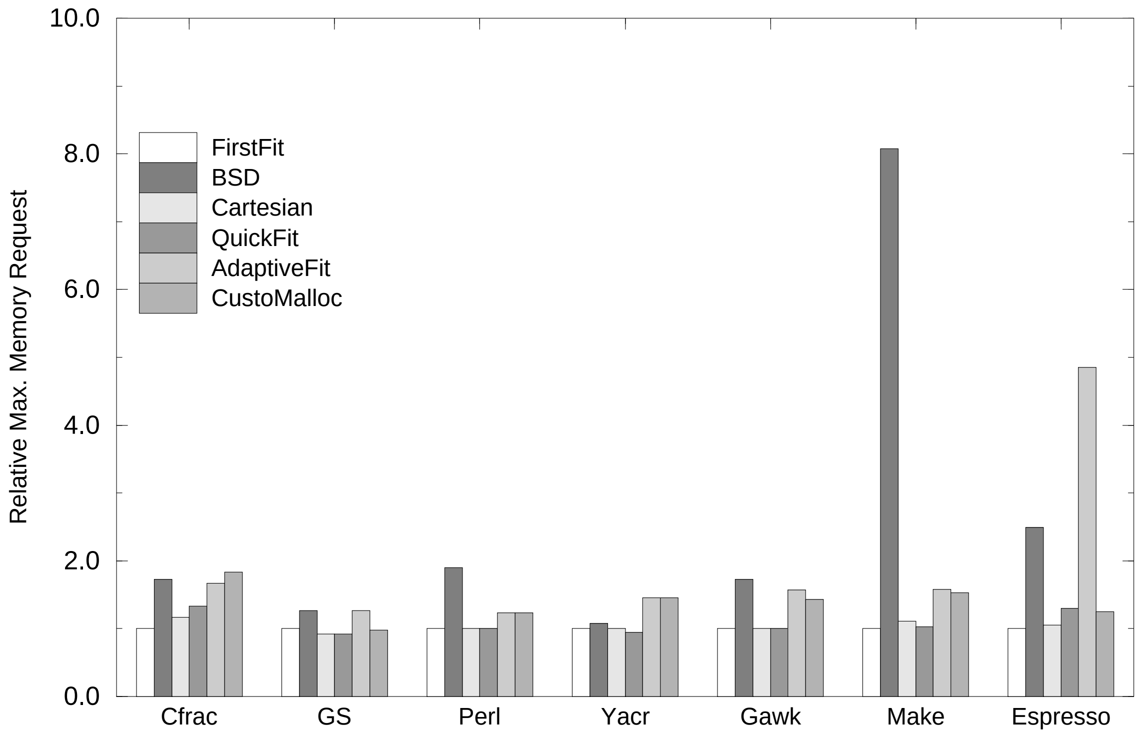


Figure 6: Maximum Memory Requested From Operating System
(Normalized to maximum memory needed by FirstFit allocator.)

Application	Size	Allocator	Allocations	Mean Allocated Objects	Deallocations	Mean Freelist Length
MAKE	32	Fast	17,873	6,652	9,045	102
ESPRESSO	32	General	1,438,916	572	1,438,916	4,471

Table 4: Measured Statistics for Dominant Size Classes in Two Applications

	Average Instruction Cycles Spent in `malloc`						
	CFRAC	GS	PERL	YACR	GAWK	MAKE	ESPRESSO
CustoMalloc (heuristics)	15.0	19.4	17.3	31.3	16.7	24.6	19.8
CustoMalloc (always fast)	15.0	19.4	17.0	31.3	16.7	24.6	18.2
CustoMalloc (always general)	15.5	22.4	27.7	122.9	18.7	78.0	19.8

Table 5: Advantages of Using the Fast Internal Allocator

In ESPRESSO, the ADAPTIVECACHE allocator consumes significantly more space than other allocators. This application has a large number of distinct size class, and the ADAPTIVECACHE allocator allocates a freelist for each size class. The ADAPTIVECACHE allocator also returned all items to freelists, meaning none are reclaimed or coalesced. Our decision to cull the number of freelists appears to be well-chosen; objects not cached in the freelist are available for coalescing, and the overhead for those freelists is obviated. Since these objects have sizes that are outliers on the distribution of allocations, using the general allocator in these cases does not appreciably increase the execution time.

An important advantage of CUSTOMALLOC is that it measures aspects of an application and adapts an allocator to those characteristics. In particular, we use heuristics to determine if a particular freelist should use a general or fast internal allocator. The former allows storage to be recovered and used for other object sizes, while the later is significantly faster than the general allocator.

For example, consider the information collected for ESPRESSO and MAKE, shown in Table 4. We reasoned that a program with an average freelist length larger than the average number of allocated objects, such as ESPRESSO, must undergo episodic allocation – that is, a large number of objects are allocated, briefly used and the deallocated. This indicates that this particular size class will eventually have copious amounts of extra storage on a freelist, and we should be able to reclaim it. Thus, this particular size class for ESPRESSO uses the general allocator; although this internal allocator is slower, it allows us to reclaim and coalesce storage.

By comparison, the dominant freelist in MAKE will rarely contain enough objects to compensate for the increased overhead of the general allocator. Table 5 shows the difference in the number of instructions executed per allocation for variants of CUSTOMALLOC. We synthesized allocators using our standard heuristics and other allocators using just the fast allocator or just the general allocator. The choice of fast or general allocator only affects the allocation time. In most applications, the measured data was similar to that for MAKE, and the fast internal allocator was chosen. The ESPRESSO application demonstrates the episodic allocation pattern the heuristics attempt to detect. In ESPRESSO, using the general allocator and the recovery heuristics described in §2, a maximum of 327,680 bytes of storage were needed, compared to 425,984 bytes when using just the fast allocator. The CUSTOMALLOC and FIRSTFIT allocators used the same amount of storage, the least storage of all the allocators, but the CUSTOMALLOC allocator was more than twice as fast.

These examples show that the measured data is useful when synthesizing the customized allocator. Experience with the first version of CUSTOMALLOC has shown that we can increase the storage efficiency of CUSTOMALLOC, with little or no increase in allocation time. We are modifying CUSTOMALLOC to measure allocation behavior using a finer storage granularity (eight bytes). Using this finer allocation information, we can automatically select an appropriate aggregation size (e.g.,

	Performance Relative to CARTESIAN(optimized)			
Allocator	CFRAC	GAWK	GS	PERL
CARTESIAN (unoptimized)	1.62	1.22	1.15	1.24
CARTESIAN (optimized)	1.00	1.00	1.00	1.00
BSD (unoptimized)	1.10	0.75	0.85	0.82
BSD (optimized)	1.03	0.73	1.28	0.79

Table 6: Comparison of "unoptimized" and "optimized" Memory Allocators

16 or 32 bytes) for an application. Larger aggregate sizes speed program execution, because more allocations fall into the most common case. At times, large aggregate sizes may increase memory efficiency because previously allocated objects can be reused for other sizes; however, in many cases, larger aggregate sizes decrease memory efficiency, e.g., in YACR and CFRAC. We are developing an heuristic to minimize the total memory wasted, as indicated by the measured allocation frequencies, while maximizing the aggregation size to increase speed.

We can also use the measured data to consider other allocation policies. For example, we currently use ordered freelists of aggregated sizes; clearly, in some situations, an indexed freelist similar to QUICKFIT may be more appropriate. The choice between these methods depends on the number of size classes and the frequency of allocation and deallocation. If allocation of several size classes is common (e.g., the application is equally likely to allocate 48, 52, 56 or 60 bytes), a QUICKFIT mechanism may be most appropriate. However, if a single size class dominates, or measurements indicate that rounding to a particular size aggregate will not waste considerable space, then the ordered freelist mechanism is more appropriate.

4.4 Comparison to "Optimized" Memory Allocation

Obviously the optimizations performed by CUSTOMALLOC can be performed manually by programmers, and, to a large part, *are* performed in many allocation intensive programs. Programmers use domain-specific knowledge about the (perceived) allocation frequency of various objects, typically constructing a freelist structure similar to that of CUSTOMALLOC.

How well do these *ad hoc* solutions perform compared to CUSTOMALLOC? Surprisingly, such optimizations are occasionally "de-optimizations" – they actually slow the program execution. For example, in a previous comparison of different memory allocators and a conservative garbage collection algorithm, one of us collected the data shown in Table 6[Zorn 92]. The values in the table are the execution time for each application normalized to that of the optimized CARTESIAN allocator. Larger values indicate that the program took longer to execute. Three of the applications uniformly benefit from the added *ad hoc* optimizations, although not to a great extent. The last application, GS, benefits from the optimization when a slow allocator (CARTESIAN) is used, but is penalized when a faster allocator (BSD) is used. Nonetheless, the largest improvement we see using the optimized BSD allocator is $\approx 7\%$, while the allocator generated by CUSTOMALLOC is generally $\approx 30\%$ faster than BSD and uses less storage.

5 Conclusions

We feel the optimization of memory allocators, like the optimization of register allocation, common subexpression elimination and the like, is a task best left to automated tools. CUSTOMALLOC is a good first step towards a tool for such optimizations. When allocating memory, a large number of "special cases" exist that advocate construction of customized memory allocators, particularly if the construction can be easily automated, as in CUSTOMALLOC. Our experiments show that the synthesized allocators are uniformly faster than the common binary-buddy (BSD) allocator, and are more space efficient. We feel that a general purpose memory allocator will not usually be competative with synthesized allocators; however, their study and use is important, as they must be used within sythensized allocators.

As mentioned, we feel that slight improvements on the code generation strategy for the synthesized allocator can yield both time and space performance unsurpassable by an algorithm that attempts to address general allocation profiles. We also note that these admittedly simple optimizations can yield dramatic performance improvements for some applications.

Our original interest was in developing a scalable, robust and efficient memory allocation algorithm for *parallel* programs. Experience has shown that sound parallel algorithms are usually based on the best available sequential algorithm. In the near future, we hope to extend the measurement-directed code synthesis of CUSTOMALLOC to parallel memory allocation.

6 Acknowledgements

We would like to thank James Larus for development of the QP utility, which greatly simplified our experimentation. Likewise, we thank Doug Lea and Chris Kingsley for use of their efficient memory allocators. The CUSTOMALLOC software is available via anonymous FTP from `cs.colorado.edu` in `pub/cs/misc/customalloc.tar.Z`; contact `grunwald@cs.colorado.edu` for more information.

References

[Ball & Larus 92] Ball, T. and Larus, J. R. Optimally profiling and tracing programs. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[Batson et al. 70] Batson, A. P., Ju, S. M., and Wood, D. C. Measurements of segment size. *Communications of the ACM*, 13(3):155–159, March 1970.

[Bozman 84] Bozman, G. The software lookasize buffer reduces search overhead with linked lists. *Communications of the ACM*, 27(3):222–227, March 1984.

[Bozman et al. 84] Bozman, G., Buco, W., Daly, T. P., and Tetzlaff, W. H. Analysis of free-storage algorithms—revisited. *IBM Systems Journal*, 23(1):44–64, 1984.

[DeTreville 90] DeTreville, J. Heap usage in the Topaz environment. Technical Report 63, Digital Equipment Corporation System Research Center, Palo Alto, CA, August 1990.

[Kingsley 82] Kingsley, C. Description of a very fast storage allocator. Documentation of 4.2 BSD Unix malloc implementation, February 1982.

[Knuth 73] Knuth, D. E. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, chapter 2, pages 435–451. Addison Wesley, Reading, MA, 2nd edition, 1973.

[Lea] Lea, D. An efficient first-fit memory allocator. (From comments in source and personal communication).

[Leverett & Hibbard 82] Leverett, B. W. and Hibbard, P. G. An adaptive system for dyanmic storage allocation. *Software – Practice and Experience*, 12:543–555, 1982.

[Margolin et al. 71] Margolin, B. H., Parmelee, R. P., and Schatzoff, M. Analysis of free-storage algorithms. *IBM Systems Journal*, 10(4):283–304, 1971.

[Oldehoeft & Allan 85] Oldehoeft, R. R. and Allan, S. J. Adaptive exact-fit storage management. *Communications of the ACM*, 28(5):506–511, May 1985.

[Standish 80] Standish, T. *Data Structures Techniques*. Addison-Wesley Publishing Company, 1980.

[Stephenson 83] Stephenson, C. J. Fast fits: New methods for dynamic storage allocation. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 30–32, Bretton Woods, NH, October 1983.

[Sun 90] Sun Microsystems, Mountain View, CA. *Unix Manual Page for malloc*, SunOS 4.1 edition, 1990.

[Vuillemin 80] Vuillemin, J. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, April 1980.

[Weinstock & Wulf 88] Weinstock, C. B. and Wulf, W. A. Quickfit: An efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–144, October 1988.

[Zorn & Grunwald 92a] Zorn, B. and Grunwald, D. Empirical measurements of six allocation-intensive c programs. Technical Report CS-CS-604-92, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, July 1992.

[Zorn & Grunwald 92b] Zorn, B. and Grunwald, D. Evaluating models of memory allocation. Technical Report CS-CS-603-92, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, July 1992. In preparation.

[Zorn 92] Zorn, B. The measured cost of conservative garbage collection. Technical Report CU-CS-573-92, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, February 1992.