Evaluating Models of Memory Allocation

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

CU-CS-603-92 July 1992

University of Colorado at Boulder

Technical Report CU-CS-603-92
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Evaluating Models of Memory Allocation [*]

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

July 1992

Abstract

Because dynamic memory management is an important part of a large class of computer programs, high-performance algorithms for dynamic memory management have been, and will continue to be, of considerable interest. We evaluate and compare models of the memory allocation behavior in actual programs and investigate how these models can be used to explore the performance of memory management algorithms. These models, if accurate enough, provide an attractive alternative to algorithm evaluation based on trace-driven simulation using actual traces. We explore a range of models of increasing complexity including models that have been used by other researchers. Based on our analysis, we draw three important conclusions. First, a very simple model, which generates a uniform distribution around the mean of observed values, is often quite accurate. Second, two new models we propose show greater accuracy than those previously described in the literature. Finally, none of the models investigated appear adequate for generating an operating system workload.

1 Introduction

In this paper, we propose, evaluate and compare a variety of methods for modeling the allocation behavior of actual programs. Some of the models we investigate have been used by other researchers to evaluate dynamic memory management (DMM) algorithms, while other models we investigate are original with this work. The goal of this paper is to measure the effectiveness of different approaches to modeling allocation behavior, to propose alternatives to existing models, and to give researchers in the field an understanding of the accuracy of the models that are available to them.

Although dynamic memory management has always been an important part of a large class of computer programs, there has been a recent surge of interest in this field as evidenced by the number of workshops devoted entirely to the subject[1]. One reason for this increased interest is that object-oriented design encourages programming with large interconnected dynamic structures and broadens the class of programs that use dynamic memory allocation. The increasing use of dynamic memory management brings with it the need to evaluate the performance of new algorithms for memory management and new systems on which these programs will run. For example, operating system evaluation requires workloads

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

[1]Garbage Collection workshops at recent Object-oriented Programming Languages and Systems (OOPSLA) Conferences and the 1992 International Workshop on Memory Management to name a few.

that take into account the increased use of dynamic memory allocation in user programs. Similarly, parallel computer systems require the evaluation of different parallel allocation algorithms.

To evaluate the performance of these algorithms and operating systems, several approaches may be taken including analytic modeling, simulation, and prototype implementation. Of the three, simulation is the most widely used because it provides precise, complete performance information without the implementation cost of building a prototype.

Evaluation based on simulation defines an algorithm or system at a high level; performance is measured by driving the system with a sequence of events representing the external behavior of interest (i.e., requests for resources or other external events, also called the event trace). Because the algorithm description is high-level, it is both easy to write and easy to parameterize. Furthermore, because the simulation is a software model of the system, instrumentation code, such as operation counts and locality measures, is easy to add and modify.

The event trace that drives a simulation can be obtained in two ways: by extracting the events from a program as it is executing (actual traces), or by generating events randomly using some probabilistic model (synthetic traces). Specifically, we are interested in synthetic traces that attempt to reproduce the behavior of an actual program. System evaluators choosing to do simulation based either on synthetic or actual traces must consider the following issues[2]:

- Actual traces are generally more accurate than synthetic traces, which are generated using simplifying assumptions (e.g., exponentially-distributed interarrival times). Accuracy is often considered the most important benefit of using actual traces.

- Actual traces can be very large (often many megabytes of data). Synthetic traces are generated probabilistically and based on data that summarizes that actual program behavior (e.g., the mean object size). The amount of information needed to recreate the trace may vary from a few bytes to hundreds of kilobytes.

- Actual traces represent exactly one behavior of a program, while synthetic traces may be used to represent one or more "similar" executions of the same program. The ability to reproduce an exact trace is beneficial because it allows a completely fair comparison of the systems being simulated using the trace. The ability to generate similar instances of a program trace is valuable when using the trace as part of a system workload in which a number of non-identical instances of a particular program are likely to be executing.

- Actual traces have a finite length, so that they cannot be used to investigate the performance of systems executing beyond the length of the trace. Because synthetic traces are randomly generated, a trace of any length can be synthesized.

[2]Jain presents a nice summary of these issues in "The Art of Computer Systems Performance Analysis" [6].

In summary, actual traces are generally preferred because they are more accurate, but synthetic traces have significant advantages. In particular, if synthetic traces are shown to accurately model the behavior of actual programs, they would be preferred to actual traces. In this paper, we investigate the relative size and accuracy of different synthetic models of program allocation behavior to determine if such models can be used effectively to evaluate the performance of systems.

1.1 Allocation Traces

One of the most successful uses of trace-driven simulation (TDS) has been in the evaluation of memory systems. TDS evaluations have been used to evaluate paging algorithms [2, 13], cache designs [15], and DMM algorithms [12, 14] including garbage collection algorithms [16, 18].

To evaluate and compare the behavior of different DMM algorithms using trace-driven simulation, programs that allocate dynamic data must be instrumented to collect the trace information. The event trace resulting from such instrumentation (which we will call the *allocation trace*) consists of object allocation events (including object size, unique identifier, and time) and object free events (including a unique identifier and time). Such an allocation trace represents all the program behavior that is required to compute the performance costs of sequential dynamic memory management and to compare the relative costs of two algorithms[3].

In this paper, we investigate probabilistic models that attempt to recreate a program's allocation trace from some summary of the actual allocation trace. To recreate an allocation trace, each model must generate samples from probability distributions that model an object's size (or size class), holding time (lifetime), and the interarrival time to the next allocation event. The models differ in how they construct these probability distributions.

The remainder of the paper has the following organization: Section 2 describes other studies of dynamic memory management and the simulation techniques they use. Section 3 describes the different synthetic models we consider in this paper while Section 4 describes the techniques and data we use to compare the models. Finally, Section 5 presents a comparison of the accuracy and size of the different synthetic models, and Section 6 summarizes our results and suggests future work.

2 Related Work

In this section, we summarize past research evaluating different DMM algorithms based on simulation studies. Our purpose is to identify what models have been used to generate synthetic traces and show how they have been used in previous work. The main difference between our work and the related work is that we are investigating the accuracy of the synthetic models used and not the performance of particular DMM algorithms.

[3]Additional information, such as a trace of all object references, could also be included if the algorithm's locality of reference was being considered.

Some DMM algorithm comparisons use synthetic traces that are not based on actual programs at all. Knuth's *Art of Computer Programming*, Volume 1, compares several classic dynamic memory management algorithms using a simple synthetic model [9]. In the comparison, object lifetime was chosen as a uniformly distributed value over an arbitrary interval. Knuth also varied the distribution of object sizes, including two uniform distributions and one weighted distribution. In his experiments, object interarrival time was assumed to be one time unit.

In a later paper, Korn and Vo [10] make similar simplifying assumptions in performing an extensive evaluation of eleven different DMM algorithms. Their model, based on Knuth's, generated sizes and lifetimes uniformly over an arbitrary interval. Their investigation varied the average size and lifetime of objects and observed the dependence of algorithm performance on these parameters.

As early as 1971 Margolin [12] pointed out that observed properties of actual programs should be considered when designing memory management algorithms. His paper describes a thorough performance evaluation study of free-storage algorithms for a time-shared operating system. His evaluation is based on TDS using actual traces of block request patterns collected from operating systems as they were used. The traces were then used to simulate different memory allocation algorithms, and the information obtained was used to guide the design of an effective algorithm.

Bozman et al conducted a follow-up to the Margolin study in 1984 [3]. While the intent was to evaluate algorithms for dynamic memory management in an operating system, the approach taken was to use TDS based on synthetic traces instead of actual traces. Their main reason for using synthetic traces was the high cost of collecting and storing actual traces. Their synthetic model, based on empirical data collected from several operating systems, computes a mean holding time and interarrival time for each distinct object size requested from the system. In an appendix, they present the actual model data that has been used in subsequent performance evaluation studies by other researchers.

Other studies of dynamic memory management have used various TDS approaches to algorithm evaluation. In particular, Oldehoeft and Allan [14] evaluate an allocation algorithm enhancement by employing a number of simulation techniques, including using actual trace data, using a synthetic model based on Margolin's data, and using uniformly distributed random data. Brent [5] also evaluates the performance of a new memory management algorithm using both random uniform distributions and Bozman's empirical data.

This related work suggests two important conclusions: first, that synthetic simulation studies are common, yet the accuracy or validity of a particular approach to synthetic program modeling has never been examined. The second conclusion is that widely published empirical characterizations of actual programs that perform dynamic allocation are lacking. The Margolin and Bozman data have both been used in recent papers, yet there are substantial drawbacks to using this data. The Margolin data is 20 years old, and therefore considerably out-of-date, and both sets of data only present the empirical behavior of a single, albeit important, program (the operating system). This paper examines the issues

```
time 1:  allocate 3 words returns object-1 (its unique id)
time 2:  allocate 3 words returns object-2
time 3:  free object-1
time 4:  allocate 2 words returns object-3
time 5:  allocate 1 word  returns object-4
time 6:  free object-4
time 7:
time 8:  allocate 1 word  returns object-5
time 9:  allocate 2 words returns object-6
time 10:
time 11: free object-5
time 12: free object-2, free object-3, free object-6
```

Figure 1: Sample Allocation Trace

raised by the first conclusion. A companion paper attempts to resolve the problem indicated by the second conclusion [17].

3 Models

In this paper, we are evaluating different models of actual program allocation behavior in terms of the size and accuracy of the model. This section describes the five models we consider and explains why they were chosen.

Models for program behavior can be viewed as a continuum from very simple models that contain little information about the program they are modeling, to complex models that contain much more information and are supposed to be correspondingly more accurate. In this light, the program allocation trace can be viewed as the most accurate model in that it contains information about every allocation and deallocation that occurred in the actual program. Synthetic models abstract this actual behavior in various ways, reducing the amount of information necessary to reproduce the behavior.

The models chosen represent five points in this continuum of complexity. Two of the models were chosen because they have been used in previous studies, while the remaining three models represent our efforts to improve upon these existing models.

As described in the previous section, the goal of any model is to abstract the behavior of the actual program in three parameters: object size class (SC), object holding time (HT), and object interarrival time (IAT). Each model defines statistical characterizations of these parameters that attempt to accurately reconstruct the actual program behavior. To better illustrate these models, we will refer to the allocation trace in Figure 1.

3.1 The Mean-Value Model (MEAN)

This model, which is the simplest, characterizes the behavior of an actual program by computing three means: the mean object size, holding time, and interarrival time. These three values are then used to generate random values with a uniform distribution and range from zero to twice the mean. This model is intentionally very simple, and we originally viewed it as a small but obviously inaccurate model. Experience has shown that our preconceptions were wrong. Another reason this model was chosen is that some variant of it has been used in several performance studies of allocation algorithms [9, 10, 14].

Given the sample allocation trace above, this model computes: mean size = 2 words, mean IAT = 1.6 ticks, and mean HT = 4.5 ticks.

An obvious variant of this model is to record the mean and variance of each observed distribution, and generate samples from three normal distributions[4] based on these observed values.

3.2 The Actual CDF Model (CDF)

While a mean summarizes a distribution very concisely, it does not accurately model the true distribution of values in the data. We considered a much more precise model that constructs the actual cumulative distribution functions (CDF) of the SC, HT, and IAT values from the observed data, and then uses these functions to generate samples. The actual CDF is constructed by maintaining a histogram of the number of occurrences of each distinct size, holding time, and interarrival time in the actual program trace. When the number of distinct values is large, the size of this model may approach the size of the allocation trace. In practice, however, the size is usually considerably smaller.

Given the sample allocation trace above, this model computes:

$$\text{size class CDF(x)} = \begin{cases} 0.0 & \text{if } x < 1 \\ 0.33 & \text{if } 1 \leq x < 2 \\ 0.66 & \text{if } 2 \leq x < 3 \\ 1.0 & \text{if } x \geq 3 \end{cases}$$

The HT CDF and IAT CDF can be computed similarly. A variant of this model approximates the actual CDF by using Jain's piecewise parabolic algorithm [7].

3.3 The Size-class-dependent Model (SIZE)

A natural extension to the mean-value model is to compute average values for holding time and interarrival time as a function of the size class of the objects allocated. For example, we can consider all objects of size 16 bytes as a group, and compute a mean HT and IAT for all the objects in that size class. Thus,

[4]The reader will observe that normal distributions can generate negative values, when in fact size, IAT, and HT are all positive-valued. Our solution to this dilemma is to reject negative samples and resample in that case. In practice, such rejection is not common.

instead of a single mean HT and IAT, this model generates N IAT's and HT's where N is the number of different size classes allocated by the program. In this context, the IAT has a slightly different meaning than it does for the previous models. In particular, while the mean-value model considers the IAT to be the time between *any* two object allocations, this model views the IAT as the time between any two allocations from the same size class. This model can be viewed as running N independent instances of the mean-value model, with each instance allocating objects of only one size. We compute the mean IAT as Bozman et al [3] did: the mean IAT is computed as the total program execution time divided by the number of objects of a particular size class. Unlike Bozman et al, however, we compute the mean HT in each size class by directly measuring the object lifetimes.

Given the sample allocation trace, this model would compute 3 means corresponding to objects of size 1, 2, and 3 words (variances omitted):

size class = 1 word, mean HT = 2, mean IAT = 6 ticks
size class = 2 words, mean HT = 5.5, mean IAT = 6 ticks
size class = 3 words, mean HT = 6, mean IAT = 6 tick

3.4 The Time-dependent Model (TIME)

While the SIZE model divides the space of values by object size, creating distributions for each size class, this model divides the space of values using execution time. Specifically, the total execution time of the actual program is divided into M buckets. For all the objects allocated within a bucket, holding time and size of each object is recorded. As with the SIZE model, the mean IAT in a bucket is determined by dividing the number of allocations in the bucket by the size of the bucket. This model can be varied by increasing or decreasing M as desired. For example, if each program cycle is considered a different bucket, an exact record of all allocations results, reproducing the information in the allocation trace. On the other hand, if a single bucket is chosen, this model degenerates into a mean-value model. Our studies assume a value of M of 10,000, which was chosen for two reasons: the space required to store this number of values was not excessive, and yet the granularity was fine enough to capture some of the interesting time-dependent behavior of the programs.

Given the sample allocation trace and a value of M = 3, we compute 3 sets of means based on allocations performed in ticks 1–4, 5–8, and 9–12, respectively (variances omitted):

ticks = 1–4, mean HT = 6.7 ticks, mean IAT = 1.3 ticks, mean size = 2.6 words
ticks = 5–8, mean HT = 2 ticks, mean IAT = 2 ticks, mean size = 1 word
ticks = 9–12, mean HT = 3 ticks, mean IAT = 4 ticks, mean size = 2 words

In one variant of this model, both mean and variance were recorded, and the generated values were sampled from a normal distribution. In another variant, the variance was ignored and the mean value was generated without variation.

3.5 The Size Transition Model (Trans)

This model, the most complex considered, is a refinement of the Size model described above. In that model, the allocation behavior of objects of different sizes is assumed to be independent. This model assumes there is a dependence between successive allocations and captures that dependence in a size transition matrix. The size transition matrix contains, for each size class X, the number of times size class Y was allocated immediately after size class X. For example, consider the sample allocation trace above. The size transition matrix for that example is shown in Table 1.

Current	Next Size		
Size	1 word	2 words	3 words
1 word	1	1	0
2 words	1	0	0
3 words	0	1	1

Table 1: The size transition matrix for the example allocation trace.

This table represents the frequency with which allocations of size X are followed by allocations of size Y (and therefore represents a conditional probability density of an allocation of size Y following an allocation of size X). A prime motivation for this model is the empirical observation that certain transitions in this matrix are very frequent while others are very infrequent, indicating that in actual programs there is a dependence relation between allocations of different-sized objects (invalidating the hypothesis of the Size model). As with the example above, in the actual size transition matrices generated, we find that a large percentage of the elements are zero, indicating no such transitions occurred.

The size transition matrix can be generalized to include information both about the holding time and interarrival time of an object of size Y, given that an object of size X has just been allocated. By observing each size transition in the actual program, we can compute the mean holding time of objects of size X given that size Y has just been allocated and likewise the conditional mean value of the interarrival time.

The size transition table has size $O(N^2)$, where N is the number of size classes allocated by the program. In the programs we have measured, N ranges from 22 to 328. In practice, we have not found the size of the size transition table to be unacceptable because the matrix is often sparse, and can be represented using sparse matrix techniques quite efficiently.

Given the sample allocation trace above, the fully defined size transition matrix would contain the information in Table 2 (variances excluded).

Current Size	Next Size		
	1 word	2 words	3 words
1 word	1, 3, 3	1, 3, 1	0
2 words	1, 1, 1	0	0
3 words	0	1, 8, 2	1, 10, 1

Table 2: The complete size transition matrix for the example allocation trace. Each entry in the table contains the number of such transitions, the mean holding time of Y given a transition from X, and the mean interarrival time between the allocation of X and Y.

3.6 Summary

Having described the models investigated in this paper, we now show a taxonomy of possible approaches to modeling and indicate how our five models fit into that taxonomy. Our initial categorization breaks the models into those that treat all input data uniformly (homogeneous models) and those that separate the input data into groups (e.g., by size or time—heterogeneous models). Further subcategories are illustrated in Figure 2. The figure shows that the five basic models described cover a significant part of the taxonomy, although there are notable models we do not consider. Another aspect of the taxonomy not clearly illustrated is that each of the size, IAT, and HT distributions can be selected independently of the others. For example, the IAT could be selected with a time-dependent heterogeneous model while the HT could be selected using a homogeneous model or a heterogeneous size-class-dependent model. In any event, part of the exploration of the space of models is the attempt to understand what is possible and to provide some assurance that a significant part of the space is being considered. We leave further exploration of the space to later research.

4 Comparison Methods

This section describes the methods used to compare the five models presented in the previous section.

4.1 Sample Programs

We used six allocation intensive programs, listed in Table 3 and Table 4, to compare the different program models. At least two input sets for each program were measured, but the different input sets contributed little to the final results of our study, and we discuss a single input set for expository simplicity. The programs were all written in the C programming language.

The execution of each program was traced using AE [11] on a Sun SPARC processor. AE is an efficient program tracing tool that captures all instruction and data references, as well as special indicators for

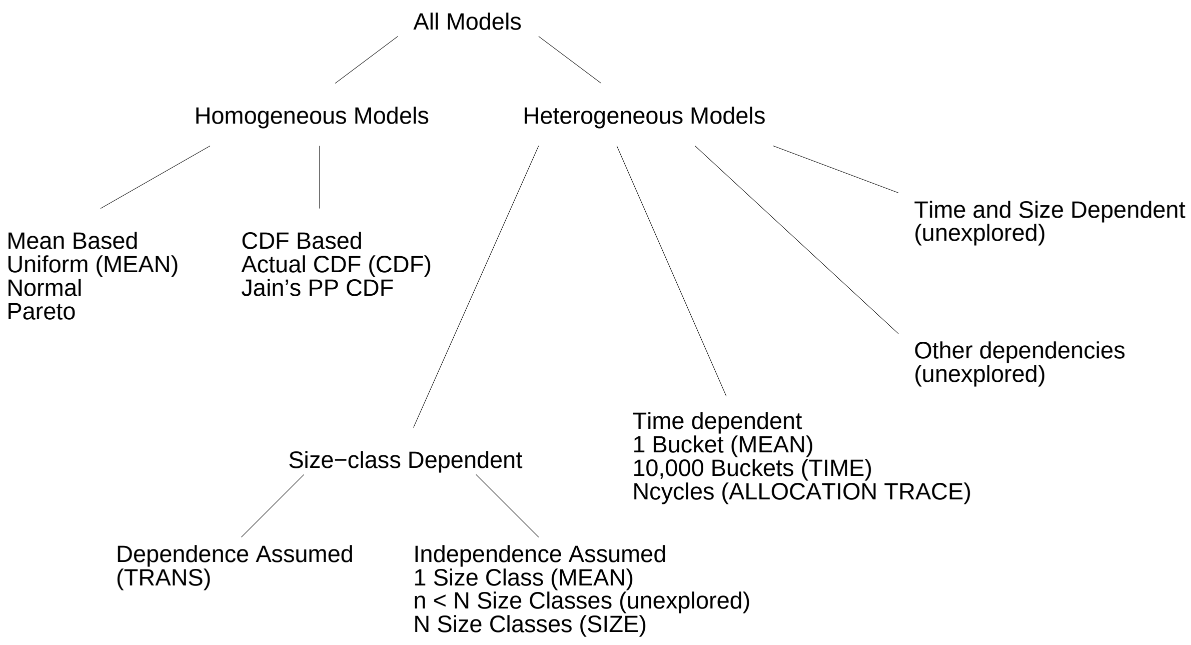

Figure 2: Taxonomy of Possible Methods Used to Model Allocation Behavior. The figure illustrates ways in which the allocation data can be decomposed and indicates how the models we are considering fit into this taxonomy (indicated by names MEAN, SIZE, TIME, TRANS, CDF, and ALLOCATION TRACE). Other models we explored include the mean/normal, mean/Pareto, and Jain's PP CDF (a piecewise parabolic approximation of the actual CDF). The figure also illustrates what models in the taxonomy we have yet to explore.

CFRAC	Cfrac is a program to factor large integers using the continued fraction method. The input is a 22-digit number that is the product of two primes.
GS	GhostScript, version 2.1, is a publicly available interpreter for the PostScript page-description language. The input used is the Users Guide to the GNU C++ Libraries (126 pages). This execution of GhostScript did not run as an 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. The input used was a perl script that reorganizes the internet domain names located in the file /etc/hosts.
GAWK	Gnu Awk, version 2.11, is a publicly available interpreter for the AWK report and extraction language. The input script processes a large file containing numeric data, computing statistics from that file.
CHAM	Chameleon is an N-level channel router for multi-level printed circuit boards. The input file was one of the example inputs provided with the release code (ex4). We also measured another channel router (YACR), but the results obtained were not significantly different that those from CHAM.
ESPRESSO	Espresso, version 2.3, is a logic optimization program. The input file was one of the larger example inputs provided with the release code (cps).

Table 3: General Information about the Test Programs

Program	CFRAC	CHAM	ESPRESSO	GAWK	GS	PERL
Lines of Code	6,000	7,500	15,500	8,500	29,500	34,500
Objects Allocated	227,091	103,548	186,636	32,165	108,550	26,390
Max Objects Allocated	1,231	103,413	2,959	2,447	6,195	483
Bytes Allocated	3,339,166	2,927,254	14,641,338	722,970	18,767,795	790,801
Max Bytes Allocated	17,395	2,711,158	136,966	63,834	467,739	24,452
Size Classes (SC)	22	22	328	48	177	79
Interarrival Time Classes (ITC)	911	4,316	13,425	856	3,502	587
Holding Time Classes (HTC)	12,748	13,169	76,299	5,638	15,339	5,053
Execution Time (Millions of Instructions)	66.9	87.1	611.1	17.6	159.2	33.5
Allocation Trace Size (Millions of Bytes)	4.5	2.1	3.7	0.6	2.2	0.5

Table 4: Test Program Performance Information. The SC, ITC, and HTC values indicate the number of distinct size, interarrival times, and holding times respectively in each of the sample programs.

calls to the `malloc` and `free` procedures used for memory allocation. These large, complex traces were distilled to a time-ordered memory allocation trace including only calls to `malloc` and `free`. Each memory allocation trace event was time-stamped using the number of instructions since the beginning of the program.

The version of CHAM that we measured does not release much of its allocated memory by calling `free`. For this program, we monitored the data references of the traced program, and artificially deallocated memory when it was no longer referenced. The `free` events were inserted in the memory allocation trace, essentially modeling perfect memory deallocation.

4.2 Comparing The Models

Allocation traces from the test programs were used to construct the information needed for the synthetic models described in Section 3. The allocation traces are also used as the baseline in our comparison in Section 5. In that context, we refer to these traces as the "actual execution" of a particular program.

To determine the accuracy of a model, we computed various metrics of the model and compared the result with value produced by the actual execution. The metrics used for comparison are divided into two categories. Intrinsic metrics represent absolute measures of a model's performance when compared with the actual allocation trace. The total number of bytes allocated by the program is an example of an intrinsic metric. These metrics indicate how close a model comes to reproducing the allocation behavior embodied by the allocation trace. While intrinsic metrics indicate whether or not the gross properties of the models remain close to those of the allocation trace, they do not indicate if the models are useful for comparisons of storage allocation algorithms, as the models have been used in related work.

Bytes	The total amount of memory allocated, expressed in bytes.
Objects	The total number of allocated objects. Each call to `malloc` returned a new object.
MaxBytes	The maximum or peak amount of memory allocated, expressed in bytes.
MaxObjects	The maximum number of allocated objects.

Table 5: Intrinsic Metrics of Interest

Extrinsic metrics are measures of how the use of a particular model influences the outcome of a simulation based on the model. For example, if we are interested in measuring the performance of a "best-fit" DMM algorithm, we can compute its performance using trace-driven simulation based either on an actual allocation trace or on a synthetic trace. The CPU overhead of the best-fit algorithm is an example of an extrinsic metric. If the extrinsic metrics of synthetic models are sufficiently accurate, the model is appropriate for use in a comparison of storage allocation algorithms.

The experimental procedures and the data collected are described below. The data presented in Section 5 includes 90% confidence intervals from running ten trials for each experiment for each model. Each trial used a new random number seed for the stochastic models. In most cases, the small number of trials provided very tight bounds on the mean. We felt that measuring over a small number of trials was important, because, by its nature, trace-driven simulation is costly to perform and as a result less amenable to repeated sampling. Thus, a method showing considerable variance may be unsuitable in many studies.

4.3 Intrinsic Metrics

Table 5 describes the information we collected from each program using the different program models. Bytes and MaxBytes represent the load a program places on a system. Bytes represents the workload placed on the DMM algorithm being used for allocation. Many DMM algorithms (such as first-fit and buddy algorithms) are sensitive to the total amount of allocation that is performed. MaxBytes represents the memory load that a particular program places on a system in which it is executing. If the models are being used to generate a heap-intensive program workload for operating system measurement, an inaccurate measure of MaxBytes will result in an inaccurate workload. The Objects and MaxObjects metrics are important for list-based memory allocator algorithms, which are sensitive to the number of objects in the allocation freelist.

4.4 Extrinsic Metrics

The metrics described in the previous section provide a wealth of statistical detail about the artificial allocation traces and how they differ from the actual trace. However, they do not indicate if the differences produce significantly different behavior when the synthetic traces are used to evaluate memory management algorithms.

BSD	Buddy-based algorithm developed by Chris Kingsley, and commonly distributed with BSD UNIX systems [8].
FIRST	An implementation of Knuth's first-fit algorithm (Algorithm C with enhancements suggested by Knuth), written by Mark Moraes.
BEST	The FIRST algorithm modified for best-fit.
CACHE	The FIRST algorithm with an adaptive cache. A five element list (S_i, F_i) is used to cache storage items of frequently occurring sizes. When an item of size B_i bytes is returned via `free`, we compute $S' = \lfloor(B_i + R - 1)/R\rfloor$, using a rounding factor $R = 32$. If S' matches any of S_i, it is chained to the free list F_i; otherwise, the least recently used cache entry is flushed, and the freed cache entry is used for items of size S'. Each call to `malloc` scans the list, preferentially allocating an item from the free list if applicable. The cache list is always maintained in order of the most recent access. This allocator is based on ideas suggested in [4] and in [14].

Table 6: Memory Allocators Used for Comparison

CPU	The average amount of CPU time spent allocating *and* deallocating an object.
%MemEff	The *memory efficiency* of the allocator. This is the mean amount of memory requested by the program model divided by the mean amount of memory requested by the *memory allocator* (e.g. by using the `sbrk` command in Unix). Values close to one indicate that a memory allocator uses requested memory very efficiently.

Table 7: Extrinsic Metrics of Interest

Thus, we performed a second series of experiments, in which we used each model to drive four different memory allocation algorithms, listed in Table 6. We chose a set of memory allocators we felt should expose any program dependent behavior. The BSD allocator takes roughly constant time, but consumes considerable space because it does not coalesce items. The FIRST allocator is sensitive to allocation order because it coalesces items; coalescing not only reduces memory fragmentation, but also reduces the size of the freelist, affecting the speed of memory allocation. The BEST allocator is more sensitive to the size of the freelist because it scans the entire list. The CACHE allocator is sensitive to the temporal distribution of allocation sizes. The cumulative effect of searching the small cache, without repeated references to items of similar size, can be large. Likewise, a program model with artificially high temporal locality may skew the behavior of the CACHE model.

We compare the different models using the metrics listed in Table 7. CPU is a metric that is commonly reported by other comparisons of memory allocators. We recorded the CPU time by counting the number of machine instructions used in the `malloc` and `free` subroutines using the QP profiling tool [1]. %MemEff represents how efficiently a particular allocator uses the memory it has requested from the operating system. In practice, allocators like BEST will show higher memory efficiency than BSD, and it is important that evaluations based on synthetic data produce results that agree with the actual execution.

Model	MEAN	CDF	SIZE	TIME	TRANS
Worst Case Size (Words)	3	SC + ITC + HTC	$3 \times$ SC	3×10000	SC + $3 \times$ SC^2
Example Size (Words)	3	19,018	531	30,000	4002 (using sparse array)

Table 8: Comparison of Model Sizes for the GS Program

5 Results

In this section, we compare the models described in Section 3 using the metrics described in Section 4. We begin by examining the size of the different models. Next, we compare the results of the intrinsic metrics and follow by looking at the extrinsic metrics. We conclude with a discussion of the differences between model variants.

5.1 Model Size

Table 8 shows the model sizes for the GS program. The table illustrates the relative size of the different models considered. The allocation trace size for GS was 550,000 words. The table illustrates that the models are smaller than the allocation trace by one to five orders of magnitude. The table also shows that the CDF model, while being relatively large, is not overwhelmingly so. The size of the TIME model is based on our choice of dividing the program execution time into 10,000 buckets. The table also illustrates that a sparse representation of the TRANS model required far less data than the worst case storage requirements (only 4.2% of worst case space).

In this table and in the subsequent comparison, we present the most accurate variant of each model. The MEAN variant uses a uniform distribution about the mean; the CDF uses the actual measured CDF's; the SIZE model uses the mean HT and IAT in each size class and assumes the data is exponentially distributed (as the Bozman study did); the TIME model uses the mean size, IAT, and HT in each time bucket and assumes no variance; and the TRANS model uses the size transition matrix to determine IAT and HT and assumes the data is normally distributed around the mean.

5.2 Model Accuracy/Intrinsic Metrics

The intrinsic metrics allow us to compare a synthetic allocation trace generated by a model with the actual allocation trace. Figure 3 and Figure 4 present a comparison of the total bytes and total objects allocated by each model with the actual number of bytes and objects allocated, respectively. For each model, ten trials were recorded and a 90% confidence interval was plotted around the mean of the trials. For each model, the amount of allocation relative to the actual amount is plotted (i.e., a value of 1.0 represents the most accurate model). A value of zero indicates we were unable to run the model because it consumed too much memory.

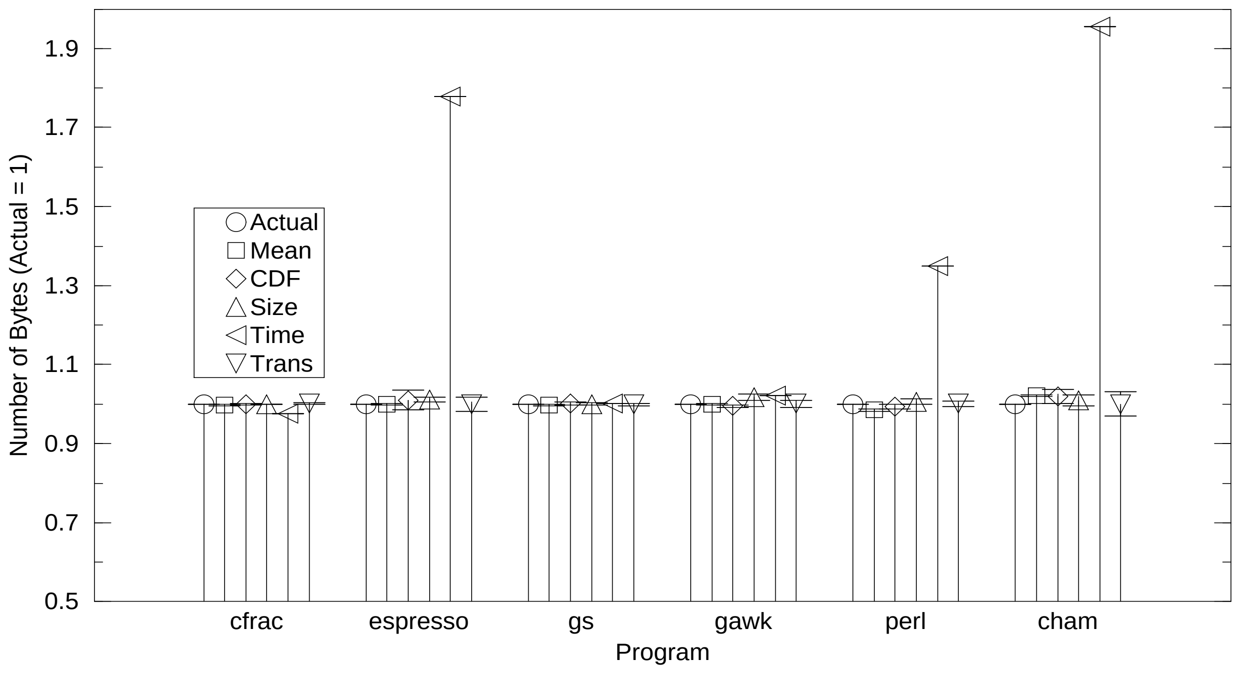

Figure 3: Total Bytes Allocated By Model Per Program

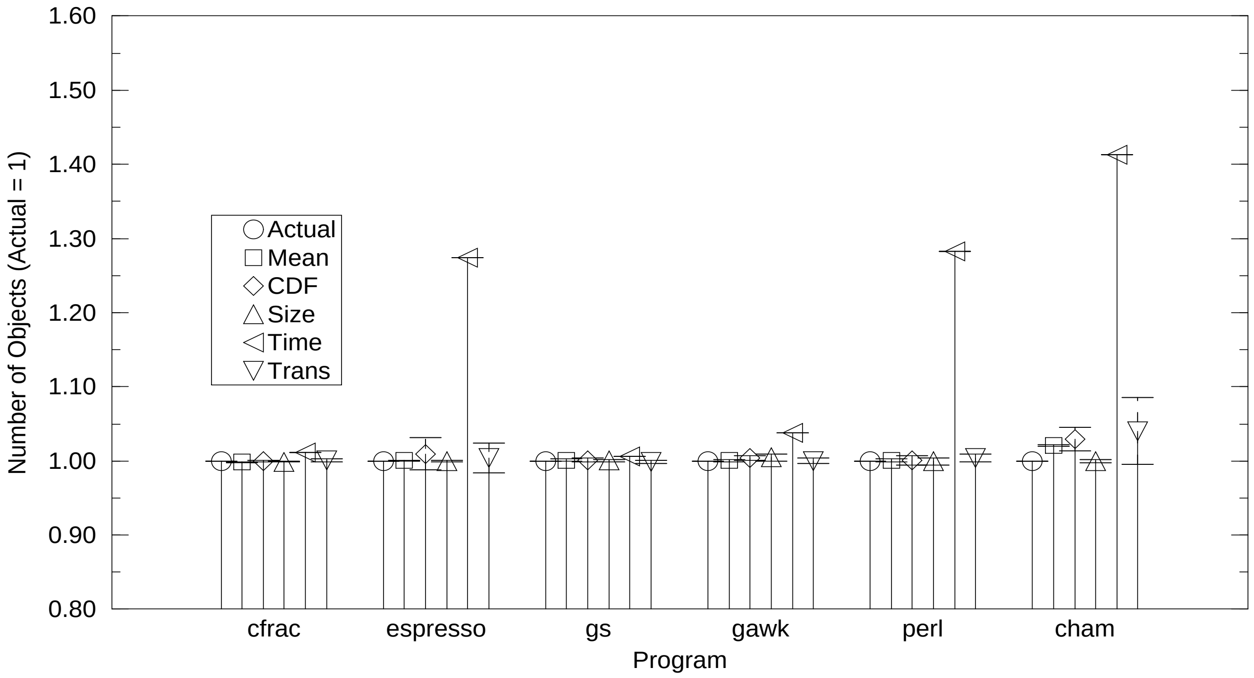

Figure 4: Total Objects Allocated By Model Per Program

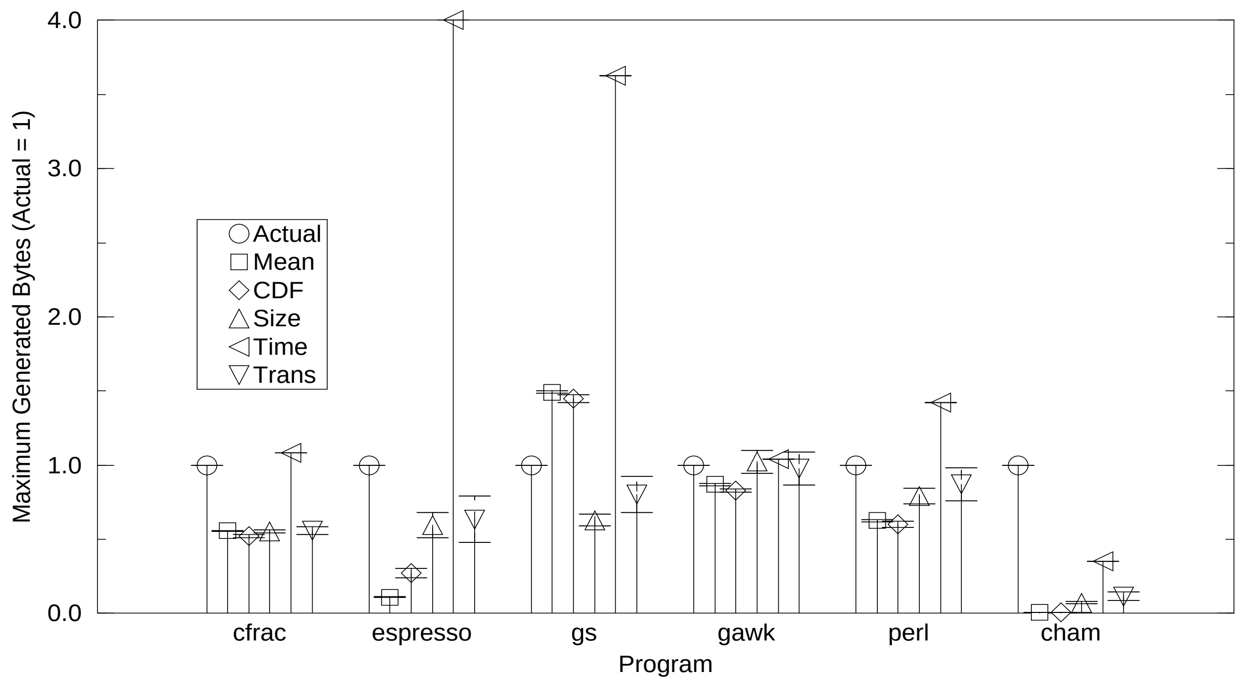

Figure 5: Maximum Generated Bytes By Model Per Program

The figures illustrate that most of the models accurately reproduce the correct total amount of allocation in the six test programs. In all cases, the observed variance of ten trials of the model is small relative to the mean. The only model that generates a significantly different amount of allocation is the Time model, which performs substantially more allocation than the actual program in several of the test programs.

This aberrant behavior of Time can be explained in the following way. Because the Time model divides the space of objects into many time buckets, some of the time buckets may contain a small number of samples (or none at all). Consider, for example, if a time bucket contains many small frequently allocated objects and few very large infrequently allocated objects. The Time model averages this behavior, and as a result, allocates many medium sized objects. The net result is that the Time model does not track the total allocation characteristics of the actual program particularly well.

The second set of intrinsic metrics, the maximum bytes and objects allocated by the program models, are presented in Figure 5 and Figure 6.

Unlike the total metrics, the maximum metrics are less accurately mirrored by the models. While no model consistently matches the maximum bytes allocated by the actual program, some models are more consistent than others. As with the total allocation metrics, the Time model shows the worst accuracy in matching the maximum program allocations. Otherwise, the Mean and Cdf models appear to behave similarly, as do the Size and Trans models. Of all the models, the Trans model appears to be the most accurate (e.g., in gs and perl), although it fails miserably in the cham program. These figures also illustrate that the heterogeneous models, Size and Trans, tend to exhibit more variance

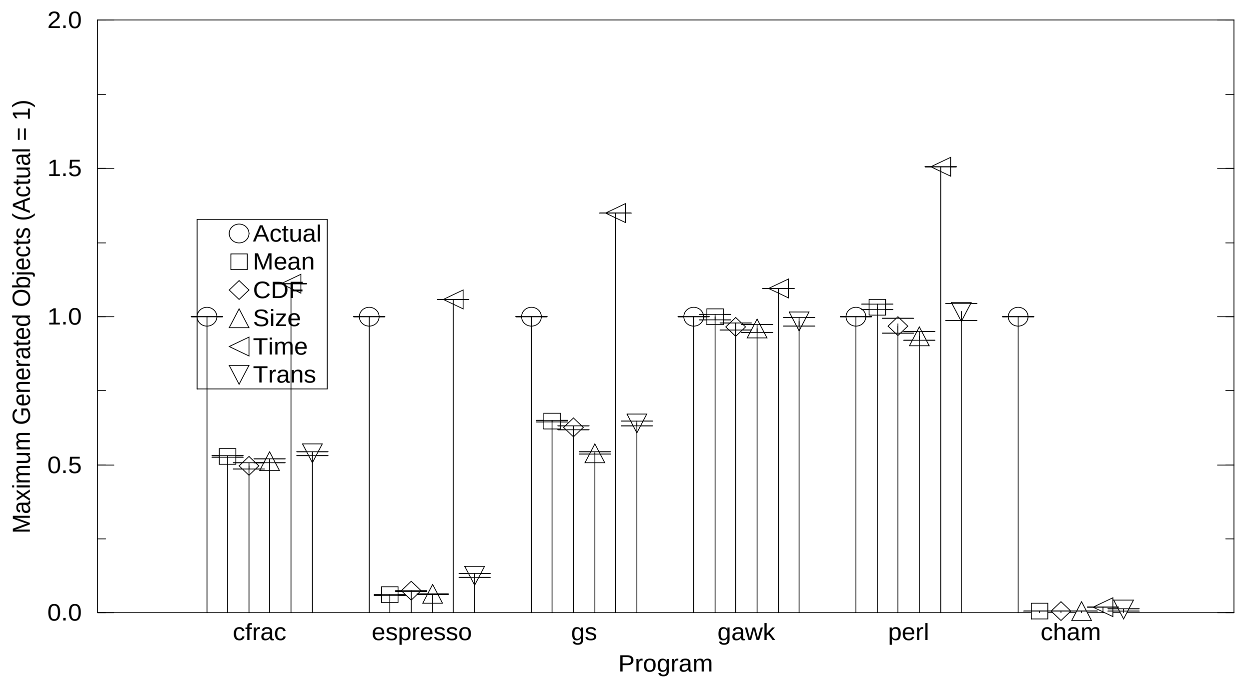

Figure 6: Maximum Generated Objects By Model Per Program

than the homogeneous models. In particular, the TRANS model displays a significant amount of variance in behavior in the ESPRESSO program.

The figures also illustrate some of the characteristics of the test programs. GAWK and PERL are relatively well-behaved programs with respect to maximum bytes and objects allocated. In particular, these programs maintain a relatively constant level of allocation throughout their execution. A more difficult time-varying behavior to model is indicated by the CFRAC and GS applications, in which the number of allocated objects increased monotonically as the program ran. Finally, the ESPRESSO and CHAM programs showed a widely varying allocation of objects and bytes, and all models had trouble accurately estimating the correct maxima for these programs.

In summary, most of the models very effectively emulate the total bytes and objects allocated by the test programs, while none of the models accurately emulate the maximum allocations for all the test programs. Because the models have trouble emulating maximum program allocations, using these models to generate synthetic system loads would not result in accurate loads. The heart of the problem lies with the inability of the models investigated to accurately track rapid time-varying changes in the amount of data allocated. The TIME model was our attempt to address this problem, but as the figures in this section show, that model is not particularly accurate in any case.

Actual Allocation Time Per Object (Instructions)				
	BSD	CACHE	FIRST	BEST
CFRAC	68.2	84.2	127.7	152.8
CHAM	87.1	151.1	152.4	175.5
ESPRESSO	74.0	89.2	121.4	432.8
GS	87.5	111.6	1957.6	2125.1
GAWK	70.6	87.8	119.9	158.1
PERL	75.0	85.1	119.3	186.3

Table 9: Actual Allocation Times for Four Allocators in Six Programs. The high number of instructions for the FIRST and BEST allocators in the GS program show that the worst-case performance of these allocators, which scan a potentially long freelist, can be quite poor.

5.3 Model Accuracy/Extrinsic Metrics

While we have established that none of the models are particularly good for generating operating system loads, they may still be valuable in evaluating the relative performance of DMM algorithms. In this section, we investigate the accuracy of the models in predicting the CPU performance and memory efficiency of four DMM algorithms (described in Section 4).

The CPU performance is well understood for many common DMM algorithms. The four allocators we use to measure the model accuracy are BSD, CACHE, FIRST, and BEST (listed in order of increasing CPU overhead). To better understand the actual CPU performance of these allocators, we present the average CPU cycles per `malloc/free` operation for the six test programs in Table 9.

Table 10 indicates the relative errors of the different models in predicting the CPU performance of the four allocators[5].

For each program, Table 10 summarizes the mean error over all allocators for each model, and the mean error over all models for each allocator. Table 11 summarizes the average of the relative errors over all programs, both by model and by allocator. From the summary, we see that the average relative error of the CDF ranks the lowest at slightly less than 18%, while the relative error of the TIME model is over 200%. Surprisingly, the MEAN model, which contains by far the least amount of information, resulted in a small relative error of only 25%. The relative errors of the SIZE and TRANS models are quite similar (as were the values of intrinsic metrics), and from this we conclude that the added complexity of the TRANS model does not result in a substantial increase in accuracy over the simple SIZE model.

In looking at the relative errors per allocator, averaged over all models and programs, we see that the BSD allocator had a far smaller average relative error than the others. This is explained by the fact that the BSD algorithm does not search a free list to find a free object and so the execution time is relatively constant and therefore depends less on the allocation trace. The BEST allocator, which is very

[5]The relative error is computed as: $RelErr = \frac{|Actual - Model|}{Actual}$. For example, if the actual CPU overhead of an `malloc/free` operation using the BSD allocator was 70 cycles, and the MEAN model predicted the operation would take 75 cycles, the relative error would be $0.071 = \frac{|70-75|}{70}$

CFRAC					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.022	0.000	0.002	0.657	0.170
Cdf	0.000	0.002	0.185	0.302	0.122
Size	0.000	0.002	0.014	0.615	0.158
Time	0.017	0.001	0.044	0.404	0.116
Trans	0.000	0.002	0.010	0.713	0.181
Mean Over Models	0.008	0.001	0.051	0.538	

ESPRESSO					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.072	0.070	0.040	0.190	0.093
Cdf	0.005	0.031	0.008	0.265	0.077
Size	0.212	0.213	0.100	0.299	0.206
Time	0.127	0.205	0.031	5.943	1.577
Trans	0.004	0.032	0.004	0.160	0.050
Mean Over Models	0.084	0.110	0.037	1.371	

GS					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.023	0.053	0.814	0.070	0.240
Cdf	0.018	0.033	0.499	0.214	0.191
Size	0.010	0.081	0.349	0.155	0.149
Time	0.114	0.870	0.859	1.349	0.798
Trans	0.004	0.047	0.450	0.272	0.193
Mean Over Models	0.034	0.217	0.594	0.412	

GAWK					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.014	0.003	0.491	1.137	0.411
Cdf	0.001	0.003	1.361	0.029	0.348
Size	0.001	0.004	0.571	0.138	0.179
Time	0.016	0.008	0.028	0.695	0.186
Trans	0.000	0.000	0.302	0.041	0.086
Mean Over Models	0.006	0.004	0.551	0.408	

PERL					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.038	0.000	0.120	0.651	0.202
Cdf	0.000	0.000	0.430	0.037	0.117
Size	0.001	0.001	0.091	0.211	0.076
Time	0.015	0.003	0.012	0.642	0.168
Trans	0.000	0.001	0.124	0.191	0.079
Mean Over Models	0.011	0.001	0.155	0.346	

CHAM					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.180	0.441	0.126	0.866	0.403
Cdf	0.201	0.438	0.114	0.058	0.203
Size	0.192	0.432	0.183	0.765	0.393
Time	0.168	0.327	0.167	7.086	1.937
Trans	0.184	0.424	0.199	1.646	0.613
Mean Over Models	0.185	0.413	0.158	2.084	

Table 10: Relative Errors in Predicting CPU Performance for Five Synthetic Allocation Models

Average Relative Error By Model				
MEAN	CDF	SIZE	TIME	TRANS
0.253	0.177	0.193	0.797	0.200

Average Relative Error By Allocator			
BSD	CACHE	FIRST	BEST
0.055	0.124	0.258	0.860

Table 11: Summary of Relative Errors in Predicting CPU Performance for Five Synthetic Allocation Models

sensitive to the specific sizes of requests made, shows a much higher overall relative error. The table also illustrates that some applications are more difficult to model than others. In particular, CHAM shows higher relative errors for all models and allocators than the other programs. The combination of the GS program and the FIRST allocator also presents problems for models, generating an average 59% error over all models.

Looking more closely at the tables indicates that no model has a consistently small relative error, however all four of the CDF, MEAN, SIZE, and TRANS models show a relative error less than 20% more than two-thirds of the cases. Furthermore, if the BEST allocator is not considered, the average relative error of the four best models drops to 10–20%. In addition, all five of the models accurately predict the relative CPU overhead ranking of the four allocators (i.e., BSD being fastest, CACHE second fastest, etc). In summary, any of the these three models appear to provide relatively good accuracy in predicting the CPU performance of a range of different DMM allocators.

The other extrinsic metric we measured was the memory efficiency of the four allocators. Again, the memory efficiency of the allocators under consideration is well understood. That is, the efficiency is likely to be inversely proportional to the execution time, with the exception of the CACHE allocator, which offers both high efficiency and low CPU overhead. The actual efficiencies of the four allocators for the six test programs are presented in Table 12. This table shows that the BSD algorithm consistently underutilizes the memory, but that the best-fit enhancement increases the memory utilization of the FIRST allocator only marginally. The CACHE allocator, however, offers both low CPU overhead and memory efficiency that sometimes outperforms the BEST algorithm.

Table 13 shows the models' relative errors in predicting the memory efficiency of the different allocators and Table 14 summarizes the average of the relative errors over all programs, both by model and by allocator. Here again we see that the TIME model is not accurately predicting the memory efficiencies of the allocators. In this case, however, the TRANS model also fails to provide an accurate result relative to the homogeneous MEAN, CDF, and SIZE models. The best model is again the CDF model, with an average relative error of 22%.

Actual Memory Efficiency (%)				
	Bsd	Cache	First	Best
cfrac	0.218	0.514	0.480	0.497
cham	0.674	0.708	0.708	0.708
espresso	0.170	0.263	0.301	0.314
gs	0.594	0.820	0.852	0.866
gawk	0.547	0.731	0.702	0.702
perl	0.278	0.706	0.626	0.626

Table 12: Actual Memory Efficiency for Four Allocators in Six Programs. The low efficiency of espresso in all allocators can be attributed to its episodic use of memory, in which large quantities of memory are allocated and then immediately released.

The memory efficiency metric also shows that some programs are harder to model than others. In this case, attempting to model espresso and cham results in higher relative errors than the other applications. This result is understandable, as the memory efficiency of an algorithm is closely related to the time-varying memory usage of the program. Because the memory usage behavior of these applications is highly variable, the memory efficiency of the storage allocators is also difficult to correctly model. If the espresso and cham programs are not included in the average, the average relative error of all of the models drops significantly. In this case, the Cdf model is still the most accurate with a relative error of 7.7%.

In this table, we fail to see a strong correlation between the relative error of the models and the allocator used, as we saw in the CPU summary table. Memory efficiency is not as strongly influenced by choice of allocator as CPU performance is, and therefore the relative errors of the models also tend not to be as strongly correlated with the allocator used.

In summary, the extrinsic metrics suggest the following conclusions. The most accurate model studied was the Cdf, which, on average, reproduced the CPU performance and memory efficiency of four very different allocators with 20% relative error. Three other models, Mean, Size, and Trans, were almost as accurate as the Cdf model, with the Mean model being particularly attractive because it is relatively accurate using very little data. The Time model appears to be quite inaccurate both in predicting CPU performance and memory efficiency.

5.4 Variants of the Models

We have presented data for the most accurate variant of the models studied. In this section, we discuss the performance of the other variants and suggest why their performance was not as good as the variant chosen.

CFRAC					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.246	0.047	0.132	0.161	0.147
Cdf	0.067	0.014	0.090	0.026	0.049
Size	0.099	0.049	0.227	0.021	0.099
Time	0.168	0.462	0.427	0.446	0.376
Trans	0.157	0.234	0.317	0.171	0.220
Mean Over Models	0.147	0.161	0.239	0.165	

ESPRESSO					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.338	1.077	0.662	0.740	0.704
Cdf	0.545	0.745	0.834	0.406	0.632
Size	0.745	0.737	0.562	0.694	0.685
Time	0.754	0.876	0.808	0.813	0.813
Trans	0.090	0.412	0.561	0.203	0.316
Mean Over Models	0.494	0.769	0.685	0.571	

GS					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.001	0.067	0.133	0.002	0.051
Cdf	0.079	0.094	0.164	0.032	0.092
Size	0.086	0.348	0.403	0.070	0.227
Time	0.465	0.402	0.373	0.381	0.406
Trans	0.416	0.463	0.554	0.418	0.463
Mean Over Models	0.210	0.275	0.326	0.181	

GAWK					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.135	0.123	0.246	0.131	0.159
Cdf	0.039	0.063	0.045	0.009	0.039
Size	0.049	0.093	0.074	0.011	0.057
Time	0.518	0.516	0.495	0.495	0.506
Trans	0.010	0.041	0.032	0.005	0.022
Mean Over Models	0.150	0.167	0.178	0.130	

PERL					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.190	0.080	0.221	0.120	0.153
Cdf	0.246	0.209	0.022	0.032	0.127
Size	0.184	0.268	0.200	0.077	0.182
Time	0.008	0.275	0.140	0.140	0.141
Trans	0.372	0.414	0.416	0.330	0.383
Mean Over Models	0.200	0.249	0.200	0.140	

CHAM					
	Bsd	Cache	First	Best	Mean Over Allocators
Mean	0.416	0.119	0.259	0.166	0.240
Cdf	0.559	0.352	0.360	0.246	0.379
Size	0.038	0.136	0.097	0.184	0.114
Time	0.144	0.089	0.195	0.195	0.156
Trans	0.505	0.449	0.371	0.352	0.419
Mean Over Models	0.333	0.229	0.256	0.229	

Table 13: Relative Errors in Predicting Memory Efficiency for Five Synthetic Allocation Models

Average Relative Error By Model				
MEAN	CDF	SIZE	TIME	TRANS
0.242	0.220	0.227	0.399	0.304

Average Relative Error By Allocator			
BSD	CACHE	FIRST	BEST
0.256	0.308	0.314	0.236

Table 14: Summary Relative Errors in Predicting Memory Efficiency for Five Synthetic Allocation Models

5.4.1 Variants Using Variance

In considering the MEAN model, other distributions about the mean could be chosen. We measured both a normal distribution and Pareto distribution, and found that they both result in greater inaccuracy when considering intrinsic and extrinsic metrics. This decreased accuracy can be explained by the introduction of variance into the synthetic model. Some of the values sampled, such as holding times, range over many orders of magnitude. This large range results in a very high variance, which in turn tends to make the synthetic model less accurate.

The variant of SIZE used in the comparisons is the same variant used by Bozman et al (although their computation of the HT was different). We also used a normal distribution to compute the HT in this model and found that the introduction of variance caused the models to perform less accurately, just as the normal MEAN model was less accurate. Likewise, the use of variance data for the TIME model decreased the accuracy of that model. We conclude that the introduction of the variance in these models results in less accurate models.

5.4.2 Approximations of the CDF

While we considered the actual CDF and found it to be our most accurate model, we also considered using Jain's piecewise parabolic algorithm [7] to approximate the actual CDF. We found, however, that Jain's algorithm is not particularly appropriate for approximating the CDF's of the size, HT, and IAT distributions because it assumes the underlying probability density function is continuous, while the actual density functions are very discontinuous (i.e., there may be 100,000 objects of size 28 but only 2 objects of size 29). As a result, a model based on an approximate CDF is not particularly accurate.

5.4.3 Other Measures of Time

In the studies presented, we measure time as the number of instructions executed since the start of the program. We also investigated measuring time not in terms of instructions, but in terms of events. With this metric, a virtual clock is kept by counting allocate and free events. Holding times and interarrival times are measured in terms of this virtual clock. We found that using this measure of time resulted in no significant differences in the accuracy of the models.

6 Summary

The goal of this research is to explore the size and accuracy of synthetic models of program allocation behavior. In the past, DMM algorithms have often been evaluated using trace-driven simulation, sometimes using synthetic traces. However, the accuracy of synthetic allocation traces has never been carefully investigated.

Allocation traces have two primary uses: first, they can and have been used to test the relative performance of different dynamic memory management algorithms. Second, they can be used to generate an operating system workload for OS performance evaluation.

We have proposed and measured the size and accuracy of five synthetic models (with variants). The CDF model, which uses the actual observed cumulative distributions of size, HT, and IAT, is the most accurate model. Surprisingly, the MEAN model, using the mean observed values of size, IAT, and HT, and assuming a uniform distribution, was only slightly less accurate than the CDF model, and required much less storage.

One of the new models we investigated, TRANS is a more complex version of the previously used SIZE model. Our measurements indicate that both models are sufficiently accurate, but the added complexity of the TRANS model does not bring with it greatly increased accuracy.

We also observed that the synthetic models are more accurate for predicting CPU performance in high-performance algorithms such as BSD and CACHE than they are for more costly algorithms such as BEST. Some of the test programs were more difficult to model, in particular programs in which the number of bytes allocated varies greatly and rapidly as a function of time (ESPRESSO and CHAM).

Unfortunately, while four of the models were relatively accurate for use in evaluating memory management algorithms, none of the models was consistently accurate in reproducing the maximum bytes or objects allocated in the actual programs. Thus, none of the models would be particularly accurate for generating operating system workloads. We designed the TIME model specifically for this purpose, but it was sufficiently inaccurate for other reasons that it is altogether inadequate.

We feel that the CDF model provides an excellent basis for comparison of dynamic memory management algorithms and that the MEAN model is also acceptable, especially if a small model size is particularly important. On the other hand, none of the models examined is consistently accurate as a basis for workload generation, and for that purpose, the allocation trace is the only acceptable alternative.

6.1 Future Work

A question of great importance in this research is whether a model showing an average 20% relative error is accurate enough. That is, are allocation traces the only acceptable means available to evaluate DMM algorithms. We feel strongly that the best models we investigated are sufficient for the following reasons. First, the models were generally much more accurate (10% and less) for well-behaved programs (such as PERL) and allocators (such as BSD and CACHE). Second, the best models always correctly predicted the rank ordering of the allocators with respect to the CPU performance. The best models also correctly predicted the memory efficiency rank ordering unless the efficiency of the top allocators was very close (within a few percent). Thus, the best models would correctly predict which allocator provided the best CPU performance and memory efficiency, which is exactly what they would be used for. Finally, synthetic models of allocation have already been used in performance studies. This research quantifies the accuracy of the models used, and leaves it to the system evaluator to decide if a 20% relative error is unacceptable.

Clearly, it would be ideal to have a model whose relative error was smaller than 20% over the six programs and four allocators measured. Currently, the greatest source of inaccuracy lies in modeling highly irregular programs such as ESPRESSO and CHAM. A more accurate model must capture the irregularity of these programs to be successful.

This research was originally motivated by our interest in developing scalable, robust and efficient dynamic memory management algorithms for *parallel* programs. Observing that there is currently a lack of large, memory-intensive parallel programs, we sought to understand the limitations of synthetic models of program allocation. While the research presented here measures sequential programs, in the future we hope to extend our results into the domain of parallel programs.

References

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

[2] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.

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

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

[5] R. P. Brent. Efficient implementation of a first-fit strategy for dynamic storage allocation. *ACM Transactions on Programming Languages and Systems*, 11(3):388–403, July 1989.

[6] Raj Jain. *The Art of Computer Systems Performance Evaluation.* Wiley Professional Computing. John Wiley and Sons, Inc., New York, 1991.

[7] Raj Jain and Imrich Chlamtac. The P^2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Communications of the ACM*, 28(10):1076–1085, October 1985.

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

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

[10] David G. Korn and Kiem-Phong Vo. In search of a better malloc. In *Proceedings of the Summer 1985 USENIX Conference*, pages 489–506, 1985.

[11] James R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software—Practice and Experience*, 20(12):1241–1258, December 1990.

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

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

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

[15] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[16] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generation garbage collection: a case for large and set associative caches. Technical Report UIC-EECS-90-5, Software Systems Lab, University of Illinois at Chicago, Chicago, IL, December 90.

[17] Benjamin Zorn and Dirk Grunwald. 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.

[18] Benjamin G. Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, May 1991.

A Raw Values of Extrinsic Metrics

CFRAC					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	68.246	84.299	127.756	152.886	108.297
Mean	66.747	84.284	127.545	253.289	132.967
Cdf	68.227	84.145	151.449	199.107	125.732
Size	68.236	84.153	126.008	246.909	131.326
Time	69.397	84.373	122.147	214.679	122.649
Trans	68.237	84.160	129.006	261.874	135.819

ESPRESSO					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	74.020	89.221	121.439	432.840	179.380
Mean	79.339	95.446	126.243	350.699	162.932
Cdf	73.620	92.005	122.386	318.021	151.508
Size	89.745	108.264	133.543	303.407	158.740
Time	83.434	107.552	125.260	3005.345	830.398
Trans	73.735	92.110	120.929	363.648	162.605

GS					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	87.559	111.676	1957.650	2125.159	1070.511
Mean	89.590	117.623	363.957	2274.490	711.415
Cdf	89.156	107.935	980.652	1670.653	712.099
Size	86.678	102.684	1273.517	1796.410	814.822
Time	97.518	208.813	275.275	4992.232	1393.459
Trans	87.169	106.432	1077.374	2702.649	993.406

GAWK					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	70.620	87.813	119.987	158.160	109.145
Mean	71.626	88.104	178.924	337.949	169.151
Cdf	70.563	88.041	283.293	153.574	148.868
Size	70.717	88.160	188.526	179.982	131.847
Time	71.716	87.149	116.619	268.043	135.882
Trans	70.622	87.832	156.177	164.632	119.816

PERL					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	75.078	85.134	119.308	186.391	116.478
Mean	72.251	85.122	133.600	307.729	149.676
Cdf	75.045	85.122	170.620	179.482	127.567
Size	75.140	85.244	130.117	225.683	129.046
Time	73.924	85.350	120.723	306.075	146.518
Trans	75.071	85.210	134.075	222.078	129.108

CHAM					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	87.139	151.159	152.441	175.548	141.572
Mean	71.472	84.468	133.237	327.605	154.196
Cdf	69.640	84.927	135.006	185.726	118.825
Size	70.398	85.813	124.599	309.922	147.683
Time	72.539	101.745	127.018	1419.524	430.206
Trans	71.108	87.052	122.154	464.507	186.205

Table 15: CPU Overhead Predicted by Five Synthetic Allocation Models. All values are in instructions.

cfrac					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	0.218	0.514	0.480	0.497	0.427
Mean	0.272	0.489	0.417	0.417	0.399
Cdf	0.204	0.521	0.437	0.509	0.418
Size	0.197	0.488	0.371	0.486	0.386
Time	0.182	0.276	0.275	0.275	0.252
Trans	0.184	0.393	0.328	0.412	0.329

espresso					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	0.170	0.263	0.301	0.314	0.262
Mean	0.228	0.546	0.500	0.546	0.455
Cdf	0.077	0.067	0.050	0.186	0.095
Size	0.297	0.456	0.470	0.531	0.439
Time	0.042	0.033	0.058	0.059	0.048
Trans	0.155	0.154	0.132	0.250	0.173

gs					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	0.594	0.820	0.852	0.866	0.783
Mean	0.593	0.875	0.738	0.868	0.769
Cdf	0.547	0.897	0.712	0.895	0.763
Size	0.543	0.535	0.508	0.806	0.598
Time	0.317	0.490	0.534	0.536	0.469
Trans	0.346	0.440	0.380	0.504	0.418

gawk					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	0.547	0.731	0.702	0.702	0.671
Mean	0.473	0.641	0.529	0.610	0.563
Cdf	0.526	0.685	0.671	0.696	0.645
Size	0.520	0.664	0.650	0.695	0.632
Time	0.264	0.354	0.355	0.355	0.332
Trans	0.542	0.701	0.679	0.706	0.657

perl					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	0.278	0.706	0.626	0.626	0.559
Mean	0.331	0.650	0.488	0.551	0.505
Cdf	0.210	0.559	0.612	0.646	0.507
Size	0.227	0.517	0.501	0.578	0.456
Time	0.276	0.512	0.538	0.538	0.466
Trans	0.175	0.414	0.366	0.419	0.343

cham					
	Bsd	Cache	First	Best	Mean Over Allocators
Actual	0.674	0.708	0.708	0.708	0.699
Mean	0.394	0.624	0.525	0.590	0.533
Cdf	0.297	0.459	0.453	0.534	0.436
Size	0.648	0.804	0.639	0.838	0.732
Time	0.577	0.771	0.846	0.846	0.760
Trans	0.334	0.390	0.445	0.459	0.407

Table 16: Memory Efficiency Predicted by Five Synthetic Allocation Models