Using Lifetime Predictors to Improve Memory Allocation Performance

David A. Barrett and Benjamin G. Zorn
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

Abstract

Dynamic storage allocation is used heavily in many application areas including interpreters, simulators, optimizers, and translators. We describe research that can improve all aspects of the performance of dynamic storage allocation by predicting the lifetimes of short-lived objects when they are allocated. Using five significant, allocation-intensive C programs, we show that a great fraction of all bytes allocated are short-lived ($> 90\%$ in all cases). Furthermore, we describe an algorithm for lifetime prediction that accurately predicts the lifetimes of 42–99% of all objects allocated. We describe and simulate a storage allocator that takes advantage of lifetime prediction of short-lived objects and show that it can significantly improve a program's memory overhead and reference locality, and even, at times, improve CPU performance as well.

1 Introduction

.

Dynamic storage allocation (DSA) is used heavily in many application areas including interpreters, simulators, optimizers, and translators. Furthermore, object-oriented programming languages, such as C++, encourage a programming style that uses more dynamic memory allocation than their predecessors. But dynamic storage allocation has been criticized because it is not as efficient as static allocation [17, p. 465]. Indeed, programmers frequently write their own domain-specific allocation routines to attempt to increase the performance of their programs [22]. In this paper we describe and investigate a method for improving the performance of dynamic storage allocation that automates the process that programmers currently use to increase allocation performance: tune the allocation strategy to the behavior of the program.

Programmers often use runtime profiles to analyze the behavior of a typical execution of their program. This analysis reveals performance bottlenecks that include calls to a dynamic storage allocator (e.g., the *malloc* function in C, or the *new* operator in C++). The programmer then analyzes which calls to the memory allocator may be customized by a special purpose routine and writes one that is specifically tuned for these allocation events.

[0]This paper will appear in the SIGPLAN'93 Conference on Programming Language Design and Implementation, June 1993, Albuquerque, New Mexico

In short, our research is an exercise in automating this process using the general technique of *profile-based optimization*. With profile-based optimization, programs are executed with training sets of test data and behavioral aspects of the programs are recorded. After these training sessions, a DSA implementation is generated that optimizes the DSA algorithm for the observed program behavior.

In this paper we describe a system of profile-based optimization that accurately predicts the objects that will be short-lived at the time they are allocated and segregates these objects from the longer-lived ones. In related work, Hanson describes the advantages of segregating short-lived objects, in his case by having the programmer explicitly specify what is short-lived [10].

Our work automates Hanson's algorithm by using the allocation site (an abstraction of the call-stack) and object size to identify and segregate short-lived objects. Using five significant, allocation-intensive C programs, we show that a great fraction of all bytes allocated are short-lived. Furthermore, we describe an algorithm for lifetime prediction that accurately predicts the lifetimes of 42–99% of all objects allocated.

We also describe an algorithm that uses this information to increase the performance of dynamic storage allocation. In our algorithm, objects that are predicted to be short-lived are rapidly allocated in small spaces by incrementing a pointer and deallocated together when they are all dead. With this algorithm, CPU performance may be improved because allocation is cheap and the deallocations are done in batches. Memory fragmentation is reduced because the many small short-lived objects are not polluting the address space occupied by long-lived objects. Finally, program reference locality is increased because the short-lived objects (a large fraction of the total objects allocated) are allocated in a small part of the heap, less than 100 kilobytes in all the programs we measured.

1.1 Related Work

Our research combines two active fields of research in programming languages: dynamic storage allocation and profile-based optimization. Other researchers have designed memory allocation systems based upon various characteristics of object allocation behavior. For example, generational garbage collectors exploit the general observation that objects tend to be short-lived [15, 18, 16]. Our approach can improve the performance of generational collectors by predicting object lifetimes when they are born.

Object size information has also been used to improve allocator performance. The conservative garbage collector described by Boehm and Weiser [2] uses size to segregate objects but does not attempt to predict object lifetimes. They mention, however, that

memory overhead would be improved if living objects were segregated from dead objects. In later related work, Demers *et al* [4] use an object's allocation site (based on the current stack pointer) to predict object lifetimes. Our work expands upon theirs and presents detailed measurements of the effectiveness of the lifetime prediction approach.

In work that is closely related to ours, Hanson [10] describes how segregating objects by lifetime improves the performance of DSA algorithms. His results show that segregation can dramatically reduce the cost of allocation and deallocation of short-lived objects. His work differs from ours because he requires the programmer to indicate how long objects will live, while we predict object lifetimes automatically. We enhance Hanson's algorithm by using the behavior of previous program executions to automatically generate an allocator customized for short-lived objects.

Our work uses the state of the dynamic call-stack to predict object lifetimes. Other researchers have used the call-stack as a predictor as well. In particular, Hayes [11] and Wilson [21] use stack deallocation events to detect clustering of object deaths and trigger garbage collections. Their work uses the heuristic that a small call-stack predicts there will be more garbage available for collection and hence collection algorithms should be invoked if possible when the stack is low. Like Hayes and Wilson, we use program measurements to derive useful heuristics. However, we use the contents of the stack and not just the size.

There has also been a recent flurry of research in the area of profile-based optimization. Wall has investigated the problem of determining how well execution profiles predict actual behavior [20]. He concludes that profile-based optimization is more accurate than static attempts to determine where programs spend time and what global variables are used most often. Most recently, Fisher and Freudenberger report considerable success in using previous runs of a program to perform branch-prediction optimizations in large C and Fortran programs [7].

Grunwald and Zorn [9] describe a tool called CUSTOMALLOC that performs profile-based optimization of DSA algorithms. In their work, information collected from previous program executions is used to automatically generate high-speed explicit allocators. However, no optimization based upon predicted lifetimes is performed in their work.

This paper has the following organization. First, we describe the goals of our approach, define the important terms, and discuss the methods used to perform our measurements. Next we present measurements that show lifetime prediction can be used to accurately predict the lifetimes of short-lived objects. Finally, we present the results of simulating DSA algorithms both with and without lifetime prediction.

2 Predictors

Our goal is to implement high-performance dynamic storage allocation algorithms by successfully predicting an object's lifetime when it is allocated. The success of our approach depends on the following hypotheses:

1. One execution of a program will provide useful information about other runs of that program.
2. Information available at the time of object birth can predict that object's lifetime.

While the first hypothesis is generally accepted in the domain of of execution profiles, it has yet to be thoroughly investigated with respect to other aspects of program behavior. In this paper, we present evidence that supports the first hypothesis with respect to program allocation behavior and we illustrate how such information can be used to optimize allocator implementations.

Specifically, we use the *allocation site* and the *object size* to predict object lifetimes. The allocation site specifies where the object was created in the control sequence of the program and the size specifies how much memory is required by that object. When available, the object's type or class can also be used [5, 6], but we chose to only use the allocation site and size because the type of an object is not directly available at C birth events. Extensions of lifetime prediction algorithms that use type information, which is available in languages such as C++, Modula-2, and Modula-3, are the subject of future research.

3 Methods

3.1 Sample Programs

Since our system makes use of a heuristic and a hypothesis concerning empirical program behavior, our evaluation is based on measurements of actual programs. We selected a representative sample of programs, observed their memory allocation behavior, and designed a system to exploit that behavior. To show that the technique is widely applicable, we selected programs from a variety of application areas that require intensive dynamic storage allocation.

Our measurements are based on five application programs: CFRAC, which factors large integers; ESPRESSO, a PLA logic optimizer; GAWK, the GNU implementation of the AWK programming language; GHOST or GhostScript, a publically available PostScript interpreter; and PERL, a report extraction language commonly used in UNIX systems. We measured the allocation behavior of each of these programs by instrumenting them with Larus' AE trace generation tool [14]. Table 1 describes the programs and their input sets and Table 2 summarizes the execution behavior of these programs. Multiple input datasets were measured; the performance results presented apply to the largest of the input sets in all cases.

3.2 Lifetimes and the Allocation Site

We seek to find a correlation between the *lifetime* of an object and the *allocation site* and/or *size* of that object. To understand the measurement results, the following terms must be defined in more detail.

We define *object lifetime* to be the total number of bytes allocated between the time the object is allocated and when it is deallocated. While bytes allocated may seem an odd measure of time, it is completely appropriate. Time can be measured in many ways but one of the most natural measures, CPU instructions, is not particularly suitable for our purposes. To understand why this is true, consider the point of view of the memory allocation system. To it, the only interesting events are object allocations and deallocations. The demands placed upon the allocator are directly proportional to the rate of these events rather than elapsed time. Two time measurements based on allocations are easily available: the number of allocation events or the number of bytes allocated [11]. We chose the latter because it more closely reflects the demands placed upon the memory system.

The *call-chain* of an event corresponds to an abstraction of the program's call-stack at the time that the event occurred. In particular, we define the call-chain to be the ordered list of functions

CFRAC	Cfrac is a program that factors large integers using the continued fraction method. The inputs included 20–40 digit numbers that were the product of two primes.
ESPRESSO	Espresso, version 2.3, is a logic optimization program. The inputs used were examples provided with the release code.
GHOST	GhostScript, version 2.1, is a publicly-available interpreter for the PostScript page-description language. The inputs used include a large reference manual and a masters thesis. These executions of GhostScript did not run as interactive applications as it is often used, but instead were executed with the NODISPLAY option that simply forces the interpretation of the PostScript program without displaying the results.
GAWK	GNU AWK, version 2.11, is a publicly-available interpreter for the AWK report and extraction language. The inputs used an AWK script to format the words of several dictionaries into filled paragraphs.
PERL	Perl 4.10, is a publicly-available report extraction and printing language commonly used on UNIX systems. The input scripts sorted the contents of a file and formated the words in a dictionary into filled paragraphs.

Table 1: General information about the test programs.

Program	Source Lines of C	Instructions Executed ($\times 10^6$)	Function Calls ($\times 10^6$)	Total Bytes ($\times 10^6$)	Total Objects ($\times 10^6$)	Maximum Bytes ($\times 10^3$)	Maximum Objects	Heap Refs (%)
CFRAC	6000	1490	18.4	65.0	3.8	83	5236	79
ESPRESSO	15500	2419	9.55	105	1.7	254	4387	80
GAWK	8500	2072	28.7	167	4.3	35	1384	47
GHOST	29500	1035	1.21	89.7	0.9	2113	26467	69
PERL	34500	894	23.4	33.5	1.5	62	1826	48

Table 2: Performance information about the memory allocation behavior for each of the test programs. Total Bytes and Total Objects refer to the total bytes and objects allocated by each program. Maximum Bytes and Maximum Objects show the maximum number of bytes and objects, respectively, that were allocated by each program at any one time. Heap Refs shows the percentage of all memory references that were made to objects in the heap (on a SPARC architecture).

present on the runtime stack at any particular program event with cycles of recursive function invocations removed.

We chose the entire call-chain because it contains a significant amount of state information about the dynamic context of an event. As an alternative, we could have simply used the last element of the call-chain (e.g., the function calling *malloc*) but we felt that a single caller does not provide enough information. Because recursive calls add no additional information, we chose to prune out all recursive loops in the definition of the call-chain. Users of the *gprof* execution profiler will be familiar with this approach of collapsing cycles in the dynamic call graph [8].

In our studies, we consider the entire call-chain to see how much information would be available in the ideal case, and sub-chains of the complete call-chain as well. In practice, the sequence of calls in the call-chain may be very deep. Thus, we define the *length-N sub-chain* to be the last N callers in the complete call-chain. For example, the length-1 sub-chain of a particular complete call-chain is another way to describe the function that directly calls *malloc*.

We distinguish between two types of call-chains: one consisting of the chain of functions and the other consisting of the chain of return addresses. Our tools made it easy to use the former, so we don't distinguish between two calls within the same function, although we will in future work.

We define the *allocation site* (or simply site) to be the call-chain to the allocation routine at each object birth event. Because the size is always available at the time of allocation, we assume that the object size is part of the allocation site unless mentioned otherwise. Thus, the same call-chain allocating 8 bytes at one time and 16 bytes another corresponds to 2 distinct allocation sites.

4 The Effectiveness of Lifetime Prediction

In this section, we describe the methods used to perform lifetime prediction and present measures of the effectiveness of this technique.

4.1 Lifetime Quantile Histograms

Our goal is to determine how well the allocation site and size predict the lifetime of the objects created. A given allocation site creates many objects of varying lifetimes. We measure the associated lifetime distribution of each allocation site using Jain's quantile histogram algorithm [12], which generates approximations of a set of quantiles for a distribution. We use Jain's algorithm because it allows us to compute the quantiles with minimal storage requirements.

To collect the quantile histograms at each allocation site, we instrument each program with AE, which allows us to maintain the current call-chain and associate each object with its allocation site. When an object is allocated, we enter the object and its site into a database. When the object is later freed, we add its observed lifetime to the quantile histogram associated with its allocation site. When the instrumented program finishes, we are left with a mapping of allocation sites to quantile histograms for each allocation site in the program.

Table 3 illustrates the quartiles of the object lifetime distributions in each of the five applications. The table shows that most objects in all of the applications are very short-lived, with over half of the objects in each program living less than 10,000 bytes. We also see in the table that the oldest objects in a program live orders of magnitude longer than the median age. Table 3 illustrates the kind of data we collect for each allocation site in the programs we measured.

For the purpose of improving a memory allocation system, we would like to group objects with similar lifetimes together. One method is to find especially short-lived objects and place them into a segregated area of memory or *arena* (after Hanson [10]). To predict that a particular allocation site will allocate short-lived objects, we look at the quantile histogram for that site. If a large percentage of the objects allocated at that site are short-lived, we consider that site to be an excellent predictor of short-lived objects. If a collection of such sites predicts the lifetimes of a large fraction of all objects allocated, we say that the predictor is successful.

How short is short-lived? The answer to this question depends on several factors. First, we must realize that the longer we consider "short-lived" to be, the more objects we will be able to predict as short-lived (consider the degenerate case of the maximum object lifetime). On the other hand, the shorter we predict short-lived objects to live, the smaller the region of memory (arena) we need to set aside to store them. Making this region as small as possible has two advantages. First, it decreases the total heap size, reducing the memory overhead of storage allocation. Second, it localizes the references to short-lived objects, reducing the cache and page miss rates. In addition to balancing these concerns, we must also consider that each program will have different lifetime distributions.

For the rest of this paper, we choose "short-lived" to mean less than 32 kilobytes. We felt this size does not adversely increase the heap size and at the same time (as we will see) it allows us to effectively predict a large fraction of objects as being short-lived. We have investigated other values for this parameter, and the correct choice of value is clearly application dependent. In general, this value would be determined automatically by the tool that analyses the program behavior. Our arbitrary choice of 32 kilobytes simplifies the descriptions in this paper and also shows good performance in our particular applications.

Another point relating to lifetime quantile histograms must also be considered. Earlier, we said that we chose allocation sites that allocate a "large percentage" of short-lived objects. How large should this percentage be? The answer to this question is related to the cost of incorrect prediction: the cheaper incorrect prediction is, the more sites that allocate non-short-lived objects we can allow. In our case, the algorithm that takes advantage of lifetime prediction will not work well if incorrect prediction is frequent. Therefore, we only consider allocation sites in which *all* of the objects allocated lived less than 32 kilobytes. That is, our algorithm predicts that an allocation site will allocate short-lived objects if all objects allocated at that site in the training inputs were short-lived objects.

Table 4 shows the fraction of total bytes allocated in the five test programs that were predicted as short-lived based on their allocation sites. This table shows what is possible based entirely on examining allocation sites and does not present a measurement of an actual simulation. In a later section, we present simulation results. Throughout this paper we distinguish between *self* and *true* prediction. By *self prediction*, we mean prediction in which the same input is used for both the training and test executions of a program. *True prediction* implies that different inputs are used in the training and test executions.

To perform true prediction, we must map the allocation sites from the training inputs into allocation sites in the test execution. We found that requiring an exact mapping of allocation sites sometimes prevented corresponding sites from correctly mapping between runs. By rounding the object size to a multiple of four bytes, we found the corresponding sites were more likely to map cor-

Program	Object Lifetime Quantiles				
	0% (min)	25%	50% (median)	75%	100% (max)
CFRAC	10	32	48	849	64,994,593
ESPRESSO	4	196	2,379	25,530	104,881,499
GAWK	2	29	257	1192	167,322,377
GHOST	16	4330	8,052	393,531	89,669,104
PERL	1	64	887	1306	33,528,692

Table 3: Quantile histogram of object lifetimes in five programs. The table presents quartiles of the distributions of object lifetimes (in bytes) in each program, illustrating how object lifetimes are distributed. To read the table, each column gives the lifetime for which that percentage of bytes is alive. For example, 75% of the bytes allocated in GAWK live less than 1192 bytes. Note that the quantile histograms presented here are approximations of the true quantiles. For example, actual measurements show that the true 75% quantile for GHOST should be less than 32,000, but the quantile histogram approximates this value as 393,531.

Program	Total Sites	Actual Short-lived Bytes (%)	Self Prediction			True Prediction		
			Sites Used	Predicted Short-lived Bytes (%)	Error Bytes (%)	Sites Used	Predicted Short-lived Bytes (%)	Error Bytes (%)
CFRAC	134	100	110	79.0	0.00	77	47.3	3.65
ESPRESSO	2854	91	2291	41.8	0.00	855	18.1	0.06
GAWK	171	98	93	99.3	0.00	91	99.3	0.00
GHOST	634	97	256	80.9	0.00	211	71.8	0.00
PERL	305	99	74	91.4	0.00	29	20.4	1.11

Table 4: Fraction of total bytes predicted as being short-lived based upon allocation site and size in each of five programs. Total Sites indicates the total allocation sites in the program while the Actual column shows the true percentage of short-lived bytes. The Sites Used columns indicate how many distinct allocation sites were used to identify the short-lived objects. The Predicted Bytes columns show what percentage of total bytes allocated are correctly predicted as short-lived based upon their allocation site and size. Objects are considered "short-lived" if they are deallocated (die) before 32 kilobytes of new data are allocated. The Error Bytes columns indicate what fraction of total bytes were predicted as short-lived by the predictor but were actually long-lived.

rectly. Rounding to a larger multiple of two reduced the mapping effectiveness because too much size information was eliminated.

In understanding the results using true prediction, it is also important to know how the inputs used for training and measurement differed. Table 4 illustrates some of the important aspects of how the input sets differed in the programs measured. For example, the two PERL inputs represent two distinct PERL programs operating on distinct inputs. As such, we expect to see less effective true prediction in PERL. By contrast, the two GAWK inputs use the same gawk program and only differ in what data the gawk program is fed. In this case, we expect to see very effective true prediction.

We see in Table 4 that the generational hypothesis (most objects die young) used in garbage collection was valid for all our programs. Short-lived objects accounted for more than 90% of all bytes allocated in every program. Furthermore, in most cases, the allocation sites identified many of the short-lived objects. This result suggests that lifetime prediction based on allocation site and size is possible and potentially very effective.

Table 4 also shows that the training sets do not always exhibit the same behavior as the test inputs. That is, sometimes predictors will incorrectly identify long-lived objects as being short-lived. In the case of self prediction, errors do not occur because sites that only allocate short-lived objects are selected. With true prediction, those same sites may occasionally allocate long-lived objects. The table shows that incorrect prediction is usually a very small percentage and only potentially a problem with the CFRAC and PERL applications.

After showing that prediction is possible, we sought to determine if the size and complete allocation site are both needed for effective prediction. We first examined the effect of using only the size to predict the lifetime as shown in Table 5. The table shows that, while in some cases size can be slightly effective, in every case the effectiveness is much less than that of using both the allocation site and size. This result confirms a similar observation made by Ungar and Jackson about the correlation of size and lifetime in Smalltalk programs [19].

Another way to reduce the cost of lifetime prediction is to use less than the complete call-chain to predict lifetimes. To determine the effect of reduced information, we examined what happened to the predictions if the complete call-chain in the allocation site length was restricted to a length-N sub-chain. Table 6 presents the relationship between call-chain length and the effectiveness of lifetime prediction. The table shows that as site length increases, a larger percentage of short-lived objects is identified. Furthermore, for each application there was a length at which this percentage abruptly increases. We see from the table that this abrupt increase occurs at a relatively short length, always length-4 or less in the programs measured. The intuitive explanation for this behavior is that programs use a layered design. Until enough layers are resolved, the different actual allocators of objects are indistinguishable. For example, many programmers perform memory allocation by calling a function *xmalloc* which then calls *malloc*. The *xmalloc* layer provides greater security by checking that the value returned from *malloc* is not NULL. In such a design, the length-1 call-chain information would be an ineffective predictor.

Table 6 also suggests how effective lifetime prediction can be at improving the reference locality of programs. The New Ref column for each application shows the predicted fraction of total heap references that are made to short-lived objects (i.e., the objects that will be highly-localized using lifetime prediction). We see from that table that heap references to short-lived objects account for 8–70% of all references, typically accounting for around 40%.

From Table 6, we see that the length-4 call-chain effectively predicts a large fraction ($> 90\%$) of the short-lived bytes predicted by the complete call chain. We conclude from this measurement that in practice, length-4 call chains can be very effective at predicting short-lived object lifetimes.

5 Simulation Results

Measurements in the previous section show that allocation sites effectively predict the allocation of short-lived objects. In this section we explore the performance benefits and costs of lifetime prediction. In particular, we describe an algorithm that segregates short-lived objects and we show, through simulation, that all aspects of performance of this algorithm may be improved when lifetime prediction is employed.

5.1 The Algorithm

Using the techniques described in the previous section, training executions of a program are used to generate a set of allocation sites that predict only short-lived objects. This set of sites is stored in a database that is incorporated into an allocation system that is then linked to the program. When an object is allocated, the allocation system uses this database of allocation sites to select one of two strategies: one specific to the short-lived allocation sites, and a general one for all others.

The short-lived objects are allocated using an algorithm very similar to Hanson's [10]. Each short-lived arena has a fixed size that is relatively small (8–32 kilobytes)[1]. In addition to the objects, each arena contains one pointer (alloc) and a count field that identifies how many objects in the arena are currently alive. Objects in the arena have no per-object overhead for tags, sizes, allocated/free bits, etc. Allocation in a short-lived arena proceeds as follows. When a new arena is selected, the count field is set to zero and the alloc pointer is set to the base of the arena. Each time an object is allocated, the following events occur. First, the alloc pointer is checked to see if the arena contains enough space for the request. If it does, the count field is incremented, the alloc pointer is incremented, and the associated space is returned. If there is not enough space, the algorithm scans all short-lived arenas attempting to find one with a zero count field (indicating no live objects). If one is found, its alloc pointer to reset to the base of the arena and allocation proceeds. If one cannot be found, the object is allocated as if it were long-lived. To perform a free, this algorithm simply decrements the count field in the object's arena.

The algorithm requires that we distinguish allocations and deallocations of arena objects and objects in the general heap. For deallocations, the address of the object gives this information (and which arena) because arenas are contiguous and not part of the general allocation heap. For allocations, the presence of the allocation site in the short-lived site database indicates an arena allocation. Because the number of short-lived allocation sites is small, the database may be maintained as a small hash-table. The allocation site is encoded as an integer that is used to index into the hash-table.

We propose two alternatives to determining the allocation site. The measurements in Table 6 suggest that exclusive-ORing the size with the last four return addresses should yield results comparable to using the entire call-chain. As an alternative to calculating the callers at each allocation, Carter [3] suggests the following approach

[1] Objects larger than a specific size are allocated by the general purpose allocator.

Program	Actual Short-lived Bytes (%)	Predicted Short-lived Bytes (%)	Sites Used To Predict
CFRAC	100	0	5
ESPRESSO	91	19	177
GAWK	98	5	64
GHOST	97	36	106
PERL	99	29	26

Table 5: Fraction of total bytes predicted as being short-lived based only upon object size in each of five programs. Self prediction is used in this table. The Actual, Predicted, and Sites columns are as in Table 4. As the table shows, size information alone predicts only a small fraction of the objects as being short-lived.

Call Chain Length	CFRAC		ESPRESSO		GAWK		GHOST		PERL	
	Pred. (%)	New Ref.(%)	Pred. (%)	New Ref.(%)	Pred. (%)	New Ref.(%)	Pred. (%)	New Ref.(%)	Pred. (%)	New Ref.(%)
1	48	52	(41)	7	72	26	40	13	31	23
2	(76)	66	41	7	78	29	40	13	63	33
3	82	70	41	8	(99)	43	47	14	63	33
4	82	70	42	8	99	43	(75)	31	(91)	44
5	82	70	42	8	99	43	80	37	94	45
6	82	70	43	9	99	43	80	37	94	45
7	82	70	44	9	99	43	81	38	95	45
∞	82	70	42	8	99	43	81	38	92	44

Table 6: Effect of call-chain length on short-lived object prediction. This table shows how the fraction of total bytes predicted as being short-lived changes as more callers in the call-chain are considered. Self prediction is used in this table. The predictive ability of the call-chain increases with the number of callers. The numbers in parentheses indicate the length at which an abrupt improvement in prediction occurred. The New Ref columns indicate the predicted fraction of total heap references to the short-lived objects. The ∞ case corresponds to using the complete call-chain. Note that in some cases, the ∞ case predicts less than the length-7 chain (e.g., ESPRESSO). This occurs because we perform recursive loop elimination in the complete site case and not in the length-N case.

that we call *call-chain encryption*. For each function in a program, assign a 16-bit id[2]. At each function call, create an new key by XORing the current function's id with the key of the caller. The function ids should be selected so that the resulting keys being generated by call-chains are likely to be unique; static call-graph analysis may be used to determine the best ids. The overhead of this approach is distributed over all function calls, but would result in a small number of additional instructions per function call (load from caller's stack, XOR with the function's id, and store into callee's stack).

We modeled the CPU cost of each of these methods by multiplying the instruction count for each by the number of allocations and function calls respectively. On a RISC-style architecture, such as SPARC, we estimate that the length-4 call-chain can be computed in 10 instructions. Two assumptions are necessary to achieve this count. First, we assume that a pointer to the previous stack frame is stored in every frame. Compiler optimizations that eliminate this frame-pointer will increase the cost of determining a function's callers. Fortunately, the SPARC architecture, with built-in register windows, eliminates this problem. The second assumption we make is that the C `crt0` routine pushes three dummy stack frames on the stack so that the last four callers are always available. This optimization eliminates checks for the bottom of the stack.

We further estimate that the determination of whether an allocation is short-lived takes approximately 18 instructions, including the 10 to determine the length-4 call-chain.

In contrast, to compute the per-allocation cost of call-chain encryption, we take the total number of function calls in a program execution and multiply by the cost per call (we estimate 3 instructions) and then divide by the total number of allocations. While this cost is program dependent, we observe it to range from 9 instructions to 94 instructions per allocation in the programs measured. A complete comparison of the CPU overhead of this algorithm is presented in Table 9.

5.2 The Simulation

We measured the performance of the test programs with the lifetime prediction algorithm using trace-driven simulation. To measure the performance of a program, we fed a trace of the program's allocation events and a list of short-lived sites into a simulator of the prediction algorithm. The allocation event included the lifetime, size, and an identifier corresponding to the complete call-chain and size of that allocation. The output of the simulator gives operation counts, information about the fraction of objects and bytes allocated in arenas, heap size, and fragmentation measurements.

In our experiments, we compare the performance of a lifetime predicting arena-allocator with a relatively simple first-fit algorithm with enhancements described by Knuth [13]. We chose to compare the arena allocator with the first-fit algorithm for two reasons. First, the first-fit allocator often has relatively good memory utilization characteristics, and as such serves as a fair baseline for comparing the memory usage of the arena allocator. Second, the first-fit algorithm is used by the arena allocator for non-arena objects. In this way, the first-fit algorithm becomes the degenerate case of an arena allocator that allocates no objects in arenas. In all simulations, we present the results of true prediction.

Table 7 shows how effectively the algorithm was able to recognize short-lived objects. In all cases, the total space devoted to arenas was 64 kilobytes, twice the age of the objects predicted as short-lived by the predictor. This size was chosen with the intuition that by the time the last half of the 64 kilobytes are filled, we are confident that objects in the first half of the arena are dead, and thus the space in the first half can be re-used. Furthermore, the 64-kilobyte arena area was divided into 16 distinct 4-kilobyte arenas. This blocking reduces the space consumed by erroneously predicted long-lived objects that tie up the entire arena in which they are allocated.

Table 7 should be compared directly with Table 4. The fraction of bytes allocated in arenas corresponds very closely with the predicted short-lived bytes shown in Table 4. There are two notable exceptions, namely GHOST and CFRAC. In GHOST, we see that while the fraction of arena objects is high (80%), the fraction of arena bytes is much smaller (only 38%). This occurs because GHOST allocates about 5000 6-kilobyte short-lived objects. These objects cannot be allocated in the 4-kilobyte arenas and are allocated in the general heap instead.

CFRAC shows what happens to this algorithm if too many long-lived objects are erroneously predicted to be short-lived. Recall that 3.65% of the objects in CFRAC were incorrectly predicted as short-lived. Furthermore, empirical measurements of CFRAC show that the object lifetime distribution is very highly skewed [23]. That is, while the vast majority of objects allocated by CFRAC are very short-lived, some objects it allocates are extremely long-lived. Therefore, objects incorrectly predicted as short-lived may actually be very long-lived. These objects then tie up all the arenas (*polluting* the arenas), forcing the arena allocator to degenerate to a general-purpose allocator. We speculate that this is what happened in CFRAC, although its poor performance requires more investigation. High error rates degrade performance dramatically and it will be important to identify programs that exhibit them, such as CFRAC, during training. Lifespan prediction may be inappropriate for such programs.

Table 8 shows the effect of arena allocation upon the total memory consumed by each program. For programs that use only a small amount of memory, the arena algorithm consumed more space. For GHOST, which uses much more memory than the other programs, a savings of 28–48% was realized. Table 8 also compares the effectiveness of true prediction with self prediction. In most cases, true prediction results in the same heap size as self prediction. In the GHOST program however, true prediction increased heap size relative to self prediction, but still reduced heap size by 28% over the standard first-fit algorithm. We feel confident that in programs with large heaps, lifetime prediction will be very effective in reducing total heap size.

Table 9 shows the average number of instructions to allocate and free objects in four allocation algorithms. The numbers for the first two algorithms (BSD and First-fit) were computed directly from actual implementations using the QP [1] instruction profiling tool. The numbers for the Arena algorithms were computed using operation counts (e.g., allocations, frees, etc), multiplying them by the estimated cost per operation. In general, even though an estimated 18 instructions are expended to predict object lifetime, this overhead is a relatively small fraction of the total cost of allocation plus deallocation in the BSD and First-fit allocators.

We see from the table that the effectiveness of lifetime prediction translates directly into increased CPU performance. For example, GAWK, in which lifetime prediction was highly successful, shows very low CPU overheads. CFRAC, on the other hand, which suffers from significant prediction errors resulting in arena pollution, shows the poorest CPU performance. It is clear from the table that lifetime prediction has a mixed effect on CPU performance; it can

[2] We use 16-bit ids because existing hardware, such as the MIPS R3000 architecture, supports 16-bit immediate constants.

	True Prediction					
Program	Total Allocs (1000's)	Arena Allocs (%)	Non-arena Allocs (%)	Total Bytes (Kbytes)	Arena Bytes (%)	Non-arena Bytes (%)
CFRAC	3809.2	2.6	97.4	63472	1.8	98.2
ESPRESSO	1654.2	19.1	80.9	102423	18.2	81.8
GAWK	4273.0	98.2	1.8	163401	99.3	0.7
GHOST	924.1	81.3	18.7	87567	37.7	62.3
PERL	1466.8	18.0	82.0	32743	20.5	79.5

Table 7: Fraction of total objects and bytes allocated in the arena and the non-arena part of the heap in a lifetime predicting arena allocator.

		Self Prediction		True Prediction	
Program	First-fit Heap Size (Kbytes)	Arena Alloc Heap Size (Kbytes)	Arena Alloc Heap/ First-fit Heap (%)	Arena Alloc Heap Size (Kbytes)	Arena Alloc Heap/ First-fit Heap (%)
CFRAC	144	208	144.4	208	144.4
ESPRESSO	280	344	122.9	344	122.9
GAWK	56	112	200.0	112	200.0
GHOST	5584	2896	51.9	4048	72.5
PERL	80	144	180.0	144	180.0

Table 8: Maximum heap sizes allocated by a first-fit allocator and a lifetime predicting arena allocator. The arena heap sizes include the 64-kilobyte arena area in the total.

							True Prediction					
	BSD (instr per)			First-fit (instr per)			Arena (len-4) (instr per)			Arena (cce) (instr per)		
Program	alloc	free	a+f	alloc	free	a+f	alloc	free	a+f	alloc	free	a+f
CFRAC	52	17	69	66	64	130	134	62	196	140	62	202
ESPRESSO	55	17	72	65	65	130	76	55	130	84	55	139
GAWK	54	17	71	56	64	120	29	11	40	29	11	39
GHOST	61	17	78	165	57	222	58	18	76	142	18	160
PERL	51	17	68	70	65	136	82	55	137	120	55	175

Table 9: Average number of instructions for calls to allocate and free in the application programs using different allocation algorithms. The numbers for BSD and First-fit were measured counting instructions in actual implementations with the QP tool. Arena (len-4) indicates the cost of lifetime prediction using the last 4 callers. The determination of the length-4 caller costs 10 instructions per allocation. Arena (cce) indicates the per allocation overhead using call-chain encryption and factoring the per-call call-chain encryption as a per-allocation cost. In this case, the encryption key costs from 9 to 94 instructions per allocation in the programs measured.

improve CPU performance significantly and it can also degrade it significantly. Based on these results, we conclude that improved CPU performance is not the primary advantage of this approach.

In comparing the different call-chain identification strategies, we see that the length-4 arena algorithm was usually slightly faster than the call-chain encryption arena strategy, and in one case twice as fast. This suggests that a space-speed tradeoff may be possible by selecting one or another of the algorithms depending upon how effective the length-4 algorithm was in finding short-lived allocation sites.

6 Conclusions

Our goal is to improve all aspects of the performance of dynamic storage allocation. In this paper, we have investigated using information present at the allocation of an object to predict its lifetime. We have measured five significant allocation-intensive C programs to determine the effectiveness of this approach.

Our results show that segregation of short-lived objects into small contiguous areas of memory can improve the reference locality and total memory requirements of memory allocation. In one program, where lifetime prediction was particularly effective, the CPU overhead of storage allocation was also significantly reduced. We have seen that the allocation site (intuitively, an abstraction of the call-stack at the point of allocation) reliably predicts short-lived objects for a broad range of programs; 40–100% of all bytes allocated by a program are predicted to be short-lived by the allocation site.

By allocating short-lived objects, which represent a large fraction of the total bytes allocated, in a small arena area (64 kilobytes), reference locality is improved. Because these objects are not competing for space in the heap, heap fragmentation and size is reduced for programs with large heaps. Lifetime prediction does not come without cost; on the SPARC architecture, we estimate 18 instructions per allocation are required to attempt to predict if an object is short-lived. Our measurements show that the CPU overhead of a lifetime predicting allocator is often comparable to that of a first-fit allocator.

This paper has explored the possibility of lifetime prediction and simulated the performance of one algorithm based on this idea. Further exploration of algorithms based on this idea are required to fully understand its performance implications. In future work, we will build a prototype implementation of the most promising algorithms and measure the performance of lifetime predicting allocators directly.

7 Acknowledgements

We would like to thank John Ellis, Kinson Ho, Dirk Grunwald, Rod Oldehoeft, and the anonymous reviewers for their insightful comments. This material is based upon work supported by the National Science Foundation under Grant No. CCR-9121269 and by a Digital Equipment Corporation External Research Grant.

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] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, pages 807–820, September 1988.

[3] Larry Carter. Discussion of efficient encoding of call-chain information. Personal communication, September 1992.

[4] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the SeventeenthACM Symposium on Principles of Programming Languages*, pages 261–269, January 1990.

[5] John DeTreville. Experience with concurrent garbage collectors for modula-2+. Technical Report 64, Digital Equipment Corporation System Research Center, Palo Alto, CA, November 1990.

[6] A. Diwan, E. Moss, and R. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, volume 27, pages 273–282, San Francisco, CA, June 1992.

[7] Joseph A. Fisher and Stefan M.Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 85–95, Boston, MA, October 1992.

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

[9] Dirk Grunwald and Benjamin Zorn. CUSTOMALLOC: Efficient synthesized memory allocators. Technical Report CU-CS-602-92, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, July 1992.

[10] David R. Hanson. Fast allocation and deallcoation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, January 1990.

[11] Barry Hayes. Using key object opportunism to collect old objects. In *OOPSLA'91 Conference Proceedings*, pages 33–46, Phoenix, AZ, November 1991.

[12] 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.

[13] 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.

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

[15] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

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

[17] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 2nd edition, 1991.

[18] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[19] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[20] David W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 59–70, Toronto, Ontario, Canada, June 1991.

[21] Paul R. Wilson. Opportunistic garbage collection. *SIGPLAN Notices*, 23(12):98–102, December 1988.

[22] Benjamin Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 1993. To appear.

[23] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Notices*, 27(12):71–80, December 1992.