Segregating Heap Objects by Reference Behavior and Lifetime

Matthew L. Seidl and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA
{seidl, zorn}@cs.colorado.edu

Abstract

Dynamic storage allocation has become increasingly important in many applications, in part due to the use of the object-oriented paradigm. At the same time, processor speeds are increasing faster than memory speeds and programs are increasing in size faster than memories. In this paper, we investigate efforts to predict heap object reference and lifetime behavior at the time objects are allocated. Our approach uses profile-based optimization, and considers a variety of different information sources present at the time of object allocation to predict the object's reference frequency and lifetime. Our results, based on measurements of six allocation intensive programs, show that program references to heap objects are highly predictable and that our prediction methods can successfully predict the behavior of these heap objects. We show that our methods can decrease the page fault rate of the programs measured, sometimes dramatically, in cases where the physical memory available to the program is constrained.

1 Introduction

Due to the widespread success of C++ and more recently Java, object-oriented applications now dominate the commercial marketplace. As a result, the use of dynamic storage allocation in application programs has increased dramatically. A recent study of C and C++ programs shows that over a range of application domains, heap objects are allocated almost ten times more frequently in C++ than in C [3]. Because all objects in Java must be allocated on the heap, dynamic storage allocation in Java is likely to be even more frequent than in C++ [8].

As program sizes have increased, so have main memory sizes. But, because of rapid changes in computer technology, a larger amount of outdated hardware is currently in use than ever before. For these older computers to be of value, they need to be able to run more memory intensive programs. The secondary storage used for virtual memory swap space has not increased significantly in speed, so programs need to make better use of physical memory pages or they will slow fast processors to the level of the slower storage media.

These trends suggest that program reference locality, especially in programs with dynamically allocated memory, is important now and will continue to be increasingly important. Surprisingly, little research has been devoted to the specific goal of improving locality of reference in programs with explicit storage management (e.g., that use `malloc`).

We see three solutions to improve locality of reference in these programs: user-driven optimizations, compiler-driven optimizations, and profile-driven optimizations. User-driven optimizations require the user to have intimate knowledge of how objects will be used, and for large programs with multiple programmers, this process is often fraught with error. Compiler-driven optimizations are limited by the amount of information available at compile time, information which may not be sufficient to predict object behavior.

In this paper, we propose a profile-driven approach to organizing heap-allocated objects that substantially improves the spatial locality of reference to those objects. The specific focus of the work reported in this paper is to decrease a program's usage of virtual memory pages. We do not explicitly attempt to decrease the cache miss rate in this work. Our technique attempts to classify dynamically allocated objects into different behavior categories at the time they are allocated. Objects in different categories are placed in different areas of the heap, which we call segments. In the results we present, the four segments we identify are: highly referenced (HR), not highly referenced ($\overline{\text{HR}}$), short-lived (SL), and other.

In the training phase of our profile-driven optimization, a training input from the program is used to find a correlation between information that is present when each object is allocated and the segment into which the object should be placed. Based on these measurements, we generate a customized version of an allocator that predicts which segment each allocated object should be placed into based on the information available. In this paper, we describe and evaluate several approaches to predicting which segment an object should be placed in.

Our results are based on measurements of six allocation-intensive programs. We evaluate our approach using binary instrumentation based on ATOM [17], and vmalloc, a region-based general purpose storage allocator [19]. Us-

This paper appeared in the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, October, 1998.

ing a cross-validated experimental method, we show that our technique is uniformly effective at increasing the spatial locality of program references in the programs measured. Further, we show that our methods can decrease the page fault rate of the programs measured, sometimes dramatically, in cases where the physical memory available to the program is constrained. Finally, we show that our object placement does not significantly affect the cache locality of the programs.

This paper has the following organization. In Section 2 we discuss related work. In Section 3 we describe the methods we use to predict the behavior of objects. In Section 4 we describe our evaluation methods, including the programs measured and the instrumentation performed. Section 5 presents our results and Section 6 concludes and suggests directions for future work.

2 Background and Related Work

Organizing program code to improve its locality of reference in the virtual memory (e.g., see [15]) and cache (e.g., see [13]) has been of interest for many years. These methods work because frequently executed code segments can be readily discovered using program profiling and/or static profile estimation [20]. In this paper, we investigate the analogous problem of predicting heap object behavior at the time objects are allocated.

Existing `malloc` implementations allocate memory with little awareness of the other objects that have been allocated on the target page. In general, these implementations are tuned to allocate and free objects as fast as possible and are not aware of overall application performance with respect to main memory. As a result, substantial opportunity exists to increase the spatial reference locality of objects in such systems. In previous work, we showed that for some programs, a small number of objects receive a majority of the references, while other objects receive almost no references [16]. That previous result acts as a starting point for the current paper, which goes beyond the previous work by quantifying the performance effect achievable using object segregation.

The issue of locality of reference of heap-allocated objects has been investigated extensively, although more so for garbage-collected languages than for languages with explicit storage allocation. The poor reference locality characteristics of first-fit storage allocation [12] has prompted the improved "better fit" methods [18] that are now often used.

Grunwald et al. surveyed existing `malloc` implementations with the goal of understanding what techniques they provide to support cache locality [10]. Their conclusion was that the existing methods, including eliminating boundary value tags, and providing a fast allocator front end to rapidly reuse freed objects, did provide substantially better reference locality than the simple first-fit algorithm. Their work did not consider the more speculative issue of predicting reference locality as we do in this paper.

There have also been a number of papers investigating the effect of heap organization on reference locality in garbage collected languages [5, 14], including several recent papers that specifically consider the effect of garbage collection on cache performance [7, 21, 23]. This work differs from ours in its focus. While much of the related garbage collection work has investigated how generational garbage collection interacts with processor cache architecture, none of the previous work we are aware of has attempted to classify objects using profiles and segregate them as we do. The work of Courts [5], in which the working set for an entire

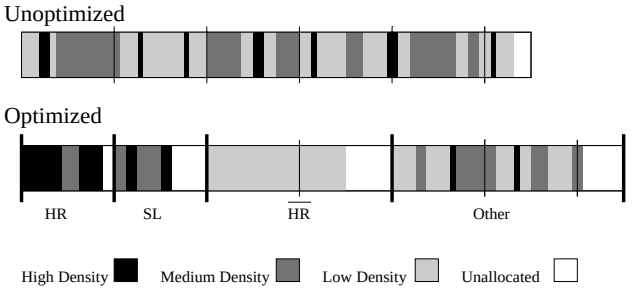

Figure 1: Illustration of our goal of memory segmentation

Lisp system is obtained by performing a "training" of the system, is the closest in this group to our work. While our goals are similar to theirs, our prediction techniques are very different.

Perhaps the work that is closest in spirit to our work is that of Barrett and Zorn, who attempted to predict short-lived object lifetimes using information present at the time of allocation [2]. Cohn and Singh also showed that object lifetimes in allocation-intensive programs can be predicted using decision trees to extract relevant static features present at an object's allocation [4]. While the methods used in our paper are similar to this previous work, this paper goes beyond that work in several dimensions. First, we attempt to predict object reference behavior as well as lifetime. Second, we consider new predictors that previous work did not consider. Finally, we present performance results based on a prototype implementation of our methods, including measurements of page fault rates.

3 Algorithms

Our approach uses profile-based optimizations to predict the behaviors of objects at the time they are allocated. The specific behaviors we are predicting are highly referenced (HR), not highly referenced ($\overline{\text{HR}}$), short-lived (SL), and other. By separating these different kinds of objects, we attempt to achieve the following results:

- Highly referenced objects will be co-located and densely packed on a small set of pages.
- Short-lived objects will be densely packed on a small set of different pages, which are also part of the working set. A secondary effect will be that long-lived objects will be more densely packed, potentially reducing fragmentation, since the SL objects have been segregated.
- Not highly referenced objects will be co-located on pages that are not part of the working set.

A potential disadvantage of segregating objects into four segments is that the heap will be fragmented. Figure 1 shows a hypothetical memory layout before and after optimization.

Figure 2 presents a diagram of the optimization framework that we use. The process starts by instrumenting the program to be optimized, which can be done either with a special compiler, or as we do, with a executable transformation tool (e.g., ATOM [17]). The instrumented program is then run with a number of training inputs that are intended to be representative of the program in actual use.

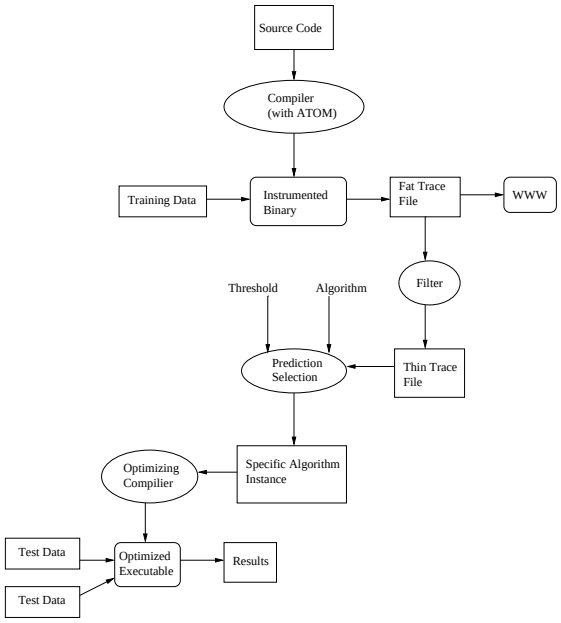

Figure 2: Overview of the Optimization Framework

In our case, these training executions result in trace files that contain information about all the heap objects allocated by the program, including the number of loads and stores to each object. In the figure, the output from the instrumentation is called a "fat" trace file because it contains additional information beyond what is needed for the research described here, but of potential interest to other researchers. After the unnecessary information is removed, the result is the "thin" trace file in the figure. The actual data contained in the "thin" trace file depends on the predictors being considered, but in general contains information about each object including the number of references that were made to it, what the call stack was at the time it was allocated, what the size of the object was, etc.

Based on the "thin" trace file data, predictor selection takes place. The process of predictor selection involves comparing the effectiveness of the different predictors over the training data gathered, and generating a specific instance of the predictor based on the specifics of the program being optimized.

The underlying assumption of this approach is that the program behavior observed in the training runs is representative of the behavior that is likely to occur in actual use. As a result, the goal of the selection process is to determine what data that is available at the time an object is allocated will be most effective in predicting that the object will be highly referenced, short-lived, etc.

An important decision that must be made before a predictor instance can be created is that correct threshold values must be chosen. The training data allows a large number of alternative performance scenarios to be considered (as we show in Section 5) across a variety of threshold settings. The current work uses four threshold values: the maximum age of an SL object, the minimum age of an HR object, the minimum reference density (references per byte) of an HR object, and the maximum reference density of a $\overline{\text{HR}}$ object. Any object not classified as one of these three types (HR, $\overline{\text{HR}}$, and SL) is placed in the "other" segment. As long as the maximum lifetime of an SL object is less than the minimum age of an HR object, these segments are non-overlapping and we do not have to decide which of the two segments to place the object in.

Choosing the appropriate threshold values for a program requires a search of the parameter space. We chose the values presented here by selecting an initial position in the threshold space, and then exploring outward from that initial location. Our experience is that while some threshold values are either particularly good or bad, the majority provided moderate improvements. Therefore, for the results we present, the search of the space was concerned primarily with avoiding bad values, and did not involve searching exhaustively for a global maxima. Also, because each program runs for a different amount of time and references objects more or less densely, each program uses its own threshold values.

The most important input to the predictor selection process is a set of potential prediction algorithms, but before describing the predictor algorithms in depth, we first outline the rest of the process shown in Figure 2. After the predictor instance is determined, a modified version of the allocation runtime system is automatically created that implements that instance. Finally, an optimized version of the program is created that incorporates the predictor-based allocator. This optimized program should exhibit substantially increased spatial locality if our segregation technique is successful.

3.1 The Prediction Algorithms

We looked at a number of different predictors, ranging from simple to complex. The predictors we focus primarily on here are based on the value of the stack pointer, the call path leading to the allocation, and the stack contents. These predictors have varying degrees of effectiveness and varying degrees of implementation overhead.

3.1.1 Stack Pointer

The stack pointer predictor (Sp) uses the value of the stack pointer at the time of heap allocation to predict which segment the object should be placed in. The stack pointer encodes the call stack to a limited extent and is easy to implement, but also offers less information about the object being allocated. This predictor can be implemented by placing a wrapper around `malloc`, checking the stack pointer and placing the object in the correct segment. The main cost of the implementation is the lookup process that maps a particular stack pointer value to a prediction. Depending on the number of "significant" stack pointer values, either a simple test or a hash table lookup could be performed. As an example, assume objects in the training set allocated with a stack pointer of 0xABCDABCD were short-lived objects. When an object in the final program is allocated, the current stack pointer is looked up in a hash table. If the stack pointer was 0xABCDABCD, the object would be allocated to the SL segment. Although the stack pointer predictor is a fairly easy predictor to implement, we are not presenting results for it here due to space concerns and the fact that its performance was well below the other two prediction algorithms discussed next.

3.1.2 Path Point

The intuition of the path point predictor (Path) is that certain call sites in a program are highly correlated with the behavior of objects allocated in procedures that occur below

the call site in the dynamic call graph. These sites are found by taking each allocated object's behavior, and attaching the behavior to each call site in the call path leading to the object's allocation. Once all of the objects have been seen, we can then look at all the call sites leading to an allocation and try to find sites where the behavior patterns attached are uniform. For example, if call site 12345 was part of the call chain for objects that were only referenced 10 times or less, call site 12345 would be classified as a $\overline{\text{HR}}$ path point.

To implement this predictor, at program runtime, when we enter call site 12345 we know that any object allocated until we left call site 12345 should be classified as a $\overline{\text{HR}}$ object. The implementation would require that we instrument the appropriate path points in the optimized program to modify a global variable indicating what prediction the allocator should currently make. Based on our experience with this predictor, few such path points exist in typical programs. As a result, the runtime cost of the path point predictor is low.

3.1.3 Stack Contents

The stack contents predictor (STACK) uses a subset of the call chain at the time of the allocation (starting at the call to the allocator and going up) to predict which segment to allocate the object in. For example, in the training set, assume that allocations with call sites 1234, 2345, and 3456 as the last three call sites in the call chain always allocated HR objects. In the optimized executable, if an allocation occurs with call sites 1234, 2345, and 3456 as the last call sites in the call chain, the object allocated will be placed in the HR segment. This predictor is more costly to implement than the previous two, but uses more context sensitive information about the object allocated and therefore tends to make better predictions.

The number of entries in the call chain that the predictor uses is of particular importance with this predictor. If the number is too large, the predictor can over-specialize the prediction and not generate something that is useful across data sets. If the number is too small, the predictor may not capture meaningful information. Many programs put in layers of abstraction around `malloc`, such as C++ object constructors or array allocators. The depth needs to be set deep enough to go up the stack past these layers, and get into the actual meat of the program while avoiding over-specification. In previous work [16], we showed that using the last three entries from the call chain is effective for the programs measured, and so a depth of three is the number used in all the results presented here.

The stack contents predictor requires that the chain of callers on the stack be determined at runtime. This problem is similar in spirit to the problem of determining a trace of basic blocks (or path) within a procedure. Work in the area of path profiling by Ball and Larus [1] may be of use as a starting point for the stack contents implementation. One technique that can be used is called "bit-pushing", where before each call bits are pushed into a shift register that indicate the identity of the call site. Using control-flow analysis similar to the path profiling analysis of Ball and Larus, it may also be possible to significantly reduce the number of shift operations needed to compute the call-chain information necessary to uniquely identify each allocation context. Also, these shift operations can potentially be hidden in the otherwise dead cycles.

An alternate method of establishing the calling context is to selectively inline the chains of callers that result in predicted allocations of, for example, highly referenced objects. If the number of such call chains is small, then the impact on program code size may be negligible. Research by Chambers et al. [9] suggests that such selective inlining may be effective for certain optimizations. In future work, we intend to investigate the effectiveness of such inlining in the context of storage allocation optimizations.

3.1.4 Other Predictors

Beyond the predictors mentioned, in previous work, we have also considered other predictors such as object size and some combinations of predictors, including combining the object size with the stack contents predictor. Object size was found to be a poor predictor, so it is not included in this study. Combining the object size and the stack contents predictor was found to be effective, but for our example programs the results were very similar to those obtained using the stack contents predictor alone, so those results are also not presented.

4 Evaluation Methods

The infrastructure we use in our studies is a network of high performance DEC Alpha workstations. As mentioned, we instrument the programs used in our studies with the ATOM toolkit [17]. ATOM allows programmers to instrument existing binaries with additional code to gather runtime data, without interfering with that program's execution.

4.1 Programs

Table 1 lists the six programs in our collection, and gives some information about what they do. These C programs were selected because they are all allocate heap objects extensively, and ESPRESSO and SIS have been reported on in previous related work (e.g., see [6]). A breakdown of basic information about the programs and the data sets we used for training and testing is presented in Table 2. In addition to the standard size metrics presented, we include some information on the number of allocation sites in the programs. The Depth 1 column shows the number of sites that call `malloc`, while the Depth 3 column shows the number of sites considered by the STACK predictor (Section 3.1.3).

In our study we used training, threshold and testing data for all of our programs to cross validate our results. Specifically, by cross validation, we mean that for all the results presented in Section 5, we used the training data set to generate the predictors, the threshold data to set the thresholds, and the testing data to evaluate the effectiveness of our methods. Using the same input for both training and testing would have provided an idealized understanding of the potential performance of a profile-based approach, but it is not informative about the effectiveness of the method in practice.

4.2 Evaluation

To evaluate the effectiveness of our predictors, we again instrumented the original programs using ATOM with a set of different procedures than those used to collect the data for the training phase. This instrumentation was necessary to drive the tycho cache simulator [11], and the basic LRU virtual memory simulator [22] we used to gather the data presented in Section 5.

CHAMELEON	CHAMELEON an N-level channel router. The training and threshold inputs were two of the inputs provided with the release code (ex4 and ex1 respectively) and the testing input was ex3. The program has approximately 8,000 lines of source code.
ESPRESSO	ESPRESSO, version 2.3, is a logic optimization program. The training, threshold and testing inputs were three of the inputs provided with the release code (mlp4, cps, and largest, respectively). The program has approximately 15,500 lines of source code.
GS	GS 5.03, This program is a PostScript interpreter and display engine. It has approximately 196,000 lines of source code. The training, threshold, and testing inputs were ms-thesis, ud-doc, and whole-program, respectively.
SIS	SIS, Release 1.1, is a tool for synthesis of synchronous and asynchronous circuits. It includes a number of capabilities such as state minimization and optimization. The program has approximately 172,000 lines of source code. The training, threshold, and testing inputs were sis-markex, sis-speedup, and simplify, respectively.
VIS	VIS is a platform used to run test cases on verification. It performs a number of tasks such as reachability analysis and model checking. The program has approximately 160,000 lines of source code. The training, threshold, and testing inputs were s444, sbc, and eight-queens, respectively.
YACR	YACR 2.1 "Yet Another Channel Router". This program has approximately 10,000 lines of source code. The training, threshold, and testing inputs were ex1, ex4, and seidl1, respectively.

Table 1: General information about the test programs.

Source Program	Instrs Executed ($\times 10^6$)	Total Bytes ($\times 10^6$)	Total Objects ($\times 10^3$)	Total Loads ($\times 10^6$)	Total Stores ($\times 10^6$)	Max. Bytes ($\times 10^6$)	Max. Objects ($\times 10^3$)	Dynamic Alloc. Sites	
								Depth 1	Depth 3
CHAMELEON (tr)	17	0.783	19.8	1.76	0.27	0.78	19.8	6	64
CHAMELEON (thresh)	93	3.71	103.6	10.70	1.74	3.49	103.4	6	70
CHAMELEON (test)	15	0.68	16.4	1.63	0.23	0.68	16.4	6	71
ESPRESSO (tr)	72	5.47	61.3	9.23	2.40	0.047	0.5	83	358
ESPRESSO (thresh)	511	23.32	190.0	78.93	20.67	0.24	3.0	90	413
ESPRESSO (test)	2130	186.26	1668.3	292.26	100.25	0.36	4.4	87	418
GS (tr)	8680	1311.18	101.5	913.84	615.70	3.19	0.2	7	31
GS (thresh)	1267	58.00	3.3	127.12	38.42	3.12	0.2	7	31
GS (test)	823	81.28	2.0	81.68	27.92	3.00	0.2	7	31
SIS (tr)	581	34.53	370.6	83.96	14.88	0.84	22.3	14	936
SIS (thresh)	368	48.89	791.0	23.57	9.67	0.99	26.2	14	827
SIS (test)	2513	231.13	3.68	200.84	48.86	1.64	43.6	14	562
VIS (tr)	157	8.06	170.9	1.427	4.03	2.22	24.8	19	784
VIS (thresh)	183	17.76	266.1	19.45	5.63	3.84	86.4	19	778
VIS (test)	990	95.04	1.29	121.68	33.57	9.81	342.2	15	534
YACR (tr)	6	0.22	4.6	0.78	0.09	0.22	4.6	40	94
YACR (thresh)	11	0.56	12.0	1.69	0.21	0.41	11.9	39	109
YACR (test)	38	1.17	15.2	8.51	0.63	1.17	15.2	35	78

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. Max. Bytes and Max. Objects show the maximum number of bytes and objects, respectively, that were allocated by each program at any one time. Instrs Executed refers the the total number of instructions executed. Total Loads/Stores refers to the total number of heap references the programs executed.

To implement the multiple segments of memory necessary to segregate objects, we used vmalloc [19], a customizable storage allocator implemented by K. Phong Vo. Vmalloc allows its users to define separate regions of storage and manage those regions with different management policies. The original intent of vmalloc was that programmers would explicitly allocate objects in specific regions, based on the programmer's knowledge about the object. Our approach, however, is naturally implemented as a wrapper around vmalloc that presents a standard malloc interface to the programmer. The purpose of the wrapper code, in this case, is to determine what segment to place objects in based on the predictors described in the previous section.

For our evaluation, we used vmalloc to determine where objects allocated to different segments should be placed, but we have not yet implemented the wrapper code that implements the different predictors described. As a result, we are able to present memory performance data that would correspond to the actual implementation of our methods, but we are not able to provide estimates of the CPU overhead necessary to implement the predictors. The CPU overhead of the stack contents predictor, which has the highest CPU overhead of the predictors we describe here, is also considered by Barrett and Zorn [2]. They computed the per allocation cost to be between 9 and 94 instructions.

The memory layout we chose to use for the segments was dictated in part by using vmalloc. We wanted each of the four memory segments (HR, $\overline{\text{HR}}$, SL, and other) to be contiguous in memory so that we had more control over the range of addresses allocated to each segment. Because vmalloc currently uses sbrk to increase the size of a memory segment, we used an initial alignment of two megabytes per memory segment. This means that each segment started with two megabytes of space, and if that space was exhausted, the heaps were grown in two megabyte chunks.

Because each heap segment started at an address two megabytes greater than the address of the previous segment, we were initially concerned with increasing the number of cache conflicts in the lower cache lines. To avoid this, we offset each segment 64 kilobytes from the start of the previous heap. For caches over 64 kilobytes, this offset keeps the first lines of each heap from conflicting in the cache.

5 Results

In this section we present cache and virtual memory results for our six test programs. We first consider how effective our methods are at segregating objects by looking more closely at the placement of objects in segments. Then we present and discuss the effects our methods have on the virtual memory performance of the programs, both in general and then by looking in depth at each program. In the last section we consider how our object segregation affects cache performance.

5.1 Placing Objects into Segments

In this section, we consider how effective our methods are at placing objects in different segments. Throughout the section, we present absolute results for the DEFAULT allocator (an unsegmented allocator implemented by using vmalloc and a single memory segment), and results relative to default for two of the predictor schemes we considered.

Table 3 shows the number of physical pages each program uses in the DEFAULT case, the number of pages relative to this number in each of the four segments using our predictors, and the relative total number of pages. A sum greater than one means our predictors required more physical pages of memory than the DEFAULT allocation scheme; a number less than one means our predictors reduced the total number of physical pages needed.

The first thing to note is that the different predictors perform differently, and sometimes quite badly. For example, The PATH predictor in one case (VIS) makes almost no predictions at all. For the PATH predictor, ESPRESSO has allocated 4.6 times as many pages to the "other" segment as the DEFAULT allocator allocated for the whole program (also note that ESPRESSO is the smallest application we measured).

The table also shows that, overall, the size of the HR segment using the different predictors tends to be small, which is the intent of creating an HR segment. Except for GS, the HR segment never used more than 13% of the pages needed by the default allocator in both the predictors. GS had a very large HR segment though, but in the case of both the PATH and STACK predictors, all the rest of the objects in GS fell into the $\overline{\text{HR}}$ segment.

In the case of the SL segment, it is used only infrequently by any of the predictors or programs. One reason for this is that both YACR and CHAMELEON never allocate short-lived objects. Both of these programs free memory rarely if ever, so all of the objects they allocate live for most of the life of the program.

Table 4 shows the average number of references per page for each segment, in absolute numbers for DEFAULT, and relative to DEFAULT for the rest of the segments in each predictor. This metric, references per page, allows us to quickly see if our predictors are successfully segregating HR objects onto some pages, and $\overline{\text{HR}}$ objects onto other pages. What we would like to see are numbers that differ from one. A number higher than one means a page has an increased reference density, while a number less than one means a reduced density (relative to the reference density in DEFAULT). Because the number of bytes allocated and the number of references stays constant across the allocation schemes, A good result would be some regions with a number much greater than 1 and some with a number much less than 1. If all the regions had a number close to 1, or every region had a number less than one, our techniques would have failed to increase the reference density of the program.

The table shows that the STACK predictor is successful at segregating objects onto pages that are more highly referenced than the pages allocated using the DEFAULT allocator. The HR segment in the STACK predictor (for all programs except ESPRESSO) shows a much higher reference density than the DEFAULT segment. ESPRESSO, due to its small size, is anomalous in a number of respects, which we discuss in detail in a later section. The STACK predictor has also been successful at moving unreferenced objects into the $\overline{\text{HR}}$ segment, where the reference density, in all cases, is lower than DEFAULT. Finally, the table shows that the SL segment has value in increasing reference density, but only in specific cases. In particular, the STACK predictor is very effective at increasing the reference density of short-lived objects in the SIS program. For VIS, it is important to note, that even though the SL segment has a density less than one, its density is still twice that of the "other" segment.

At first glance our results appear to contradict Barrett and Zorn's results [2] about the number of short-lived objects in a program. Barrett concluded that over 90% of the objects allocated by their test programs were short-lived.

8-Kilobyte Pages per Segment													
	Absolute	Relative to Default											
Program	Default	Path						Stack					
		other	SL	HR	(abs.)	nonHR	Total	other	SL	HR	(abs.)	nonHR	Total
Chameleon	116	0.974	0.000	0.009	(1)	0.026	1.017	0.560	0.000	0.129	(15)	0.319	1.017
Espresso	54	4.593	0.519	0.093	(5)	0.000	5.222	1.000	0.000	0.037	(2)	0.000	1.074
Gs	395	0.000	0.000	1.091	(431)	0.585	1.681	0.000	0.000	0.681	(269)	0.585	1.271
Sis	318	0.906	0.000	0.003	(1)	0.053	0.965	0.374	0.028	0.050	(16)	0.742	1.195
Vis	1993	1.000	0.002	0.000	(0)	0.000	1.003	0.930	0.028	0.064	(127)	0.000	1.022
Yacr	174	0.891	0.000	0.109	(19)	0.000	1.011	0.891	0.000	0.109	(19)	0.000	1.011

Table 3: Number of 8-kilobyte pages per memory segment, absolute and per segment relative to Default.

References per 8-Kilobytes Page by Segment									
	Absolute	Relative to Default							
Program	Default	Path				Stack			
	$\times 10^3$	other	SL	HR	nonHR	other	SL	HR	nonHR
Chameleon	16.1	1.012	0.000	1.189	0.168	0.550	0.000	4.960	0.158
Espresso	7268.8	0.154	0.531	0.170	0.000	0.978	0.000	0.605	0.000
Gs	277.5	0.000	0.000	0.915	0.003	0.000	0.000	1.462	0.008
Sis	785.2	1.098	0.000	0.000	0.103	0.755	17.369	1.112	0.229
Vis	77.9	1.000	0.018	0.000	0.000	0.378	0.740	9.861	0.000
Yacr	52.5	0.544	0.000	4.720	0.000	0.544	0.000	4.718	0.000

Table 4: The number of references per 8-kilobyte page, absolute and per segment relative to Default.

Our results do not share this conclusion however, and there are a number of reasons for this. First, Barrett was classifying objects based on the number of bytes allocated between birth and death, while we use number of instructions between these two events. Second, we consider only two of the five programs Barrett did, Gs and Espresso. For Gs, Barrett placed hooks directly into the PostScript interpreter to track objects, while our work looks only at the `malloc` interface. For Gs, this is particularly important, as this program does its own internal memory management. Espresso uses a large number of reallocs to dynamically resize memory objects. We believe that Barrett modeled a realloc as a free followed by a malloc. This leads to seeing a large number of short-lived objects, instead of one longer-lived object that has been realloced, which is our model for reallocs.

5.2 Virtual Memory Performance

In this section we consider the virtual memory impact of object segregation, first in the ideal case and then with respect to page faults. Throughout this section, in addition to the Default allocator and our two predictors, we also consider Random, a predictor that allocates objects to the four segments at random. Random is included as a control to test the validity of our hypothesis that our prediction methods are the cause of the performance improvements we see.

To understand how effective our methods are at increasing spatial reference locality, we measured the total number of references to each page for each program, using Default and the different predictors. We then sorted the pages in each program for each predictor by reference count, with the most frequently referenced page first. We then computed the cumulative references to the first j pages, ranging j from 1 to n, the maximum number of pages (for the given program/predictor). Finally, to facilitate a comparison, we plotted the relative cumulative reference function by dividing the cumulative reference function for each predictor at point j by the cumulative reference function for Default at point j, ranging j from 1 to n.

Figure 3 shows the relative cumulative reference function for each predictor, computed as described above. Since each graph is normalized, the value for Default is always one. Also, since the functions are cumulative, the different curves always converge to the value one since the number of references is constant across all predictors. We continue to present a relative value for the allocators that use more pages then Default by observing that we can continue to add pages to default even if it never used those pages. Thus the points on the graph after the Default line stops are relative to the total number of references the Default allocator saw. Note also that the y-axis begins at 0.5 to make the figures easier to read.

To interpret the graphs, if the value of the curve is above one, there are more references to the most highly referenced set of pages than there are with the Default allocator; if below one, there are fewer references. The interesting part of these graphs is the left side. For four of our six programs there is a sizable increase in the number of references to these most highly referenced pages (up to 15.6 times default for Vis), showing that our predictors (and Stack in particular) are packing references better than the Default allocator. This is not a measure of performance improvement, just a demonstration that our techniques do pack more references onto the most highly referenced pages. The right side of Figure 3 has all the graphs tending toward one because when all of the pages are considered for each allocator, we see the same number of total references, thus the relative value is one.

Figure 4 shows relative page fault miss rates for our six programs. Like the previous figure, the curves show the miss rates of our methods relative to Default for a given memory size. As with the previous figure, Default is always set at one, and our predictors vary from less than one, meaning we have reduced the number of page faults, to more than

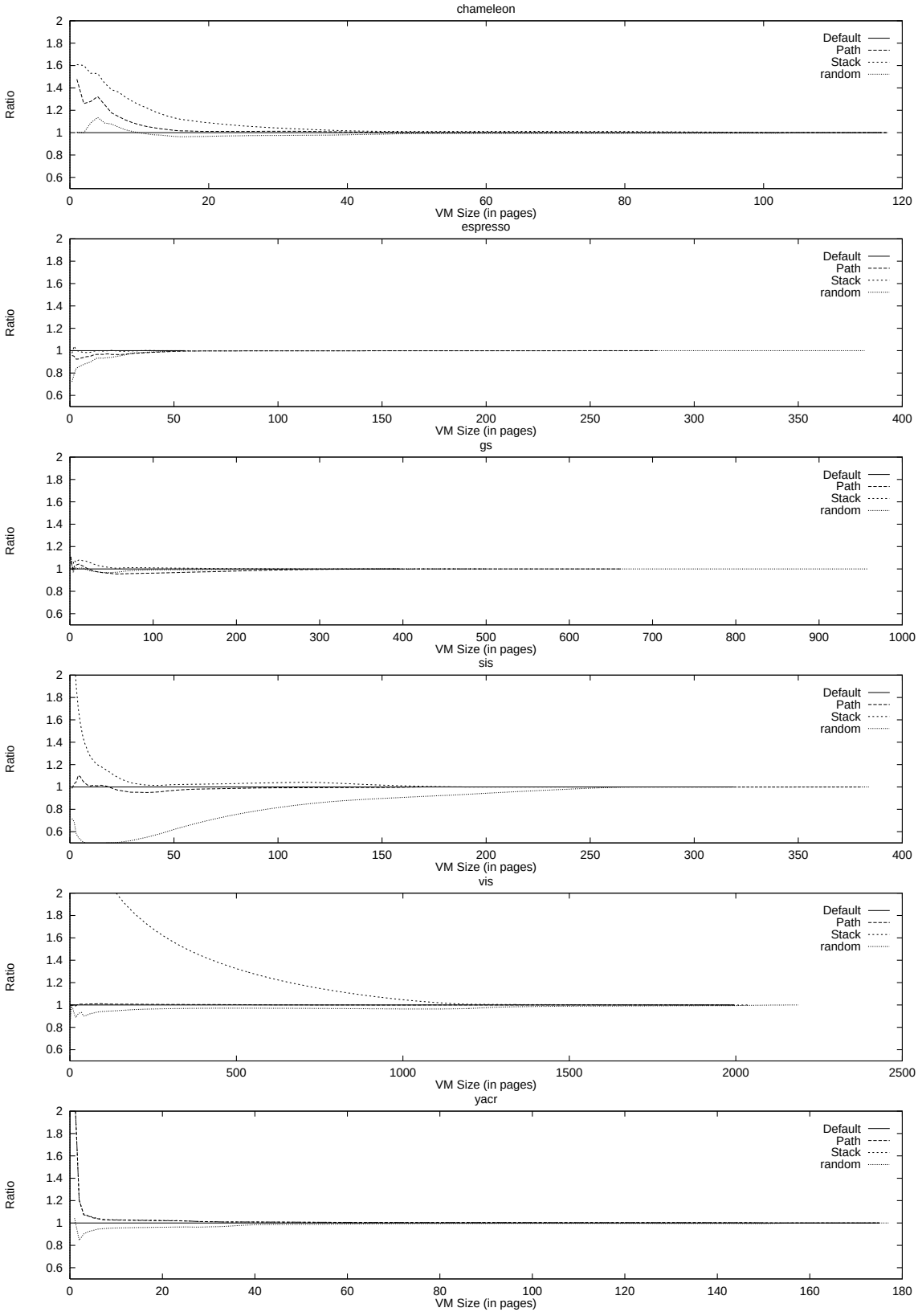

Figure 3: Cumulative percentage of references per page, relative to default, for each of the test programs.

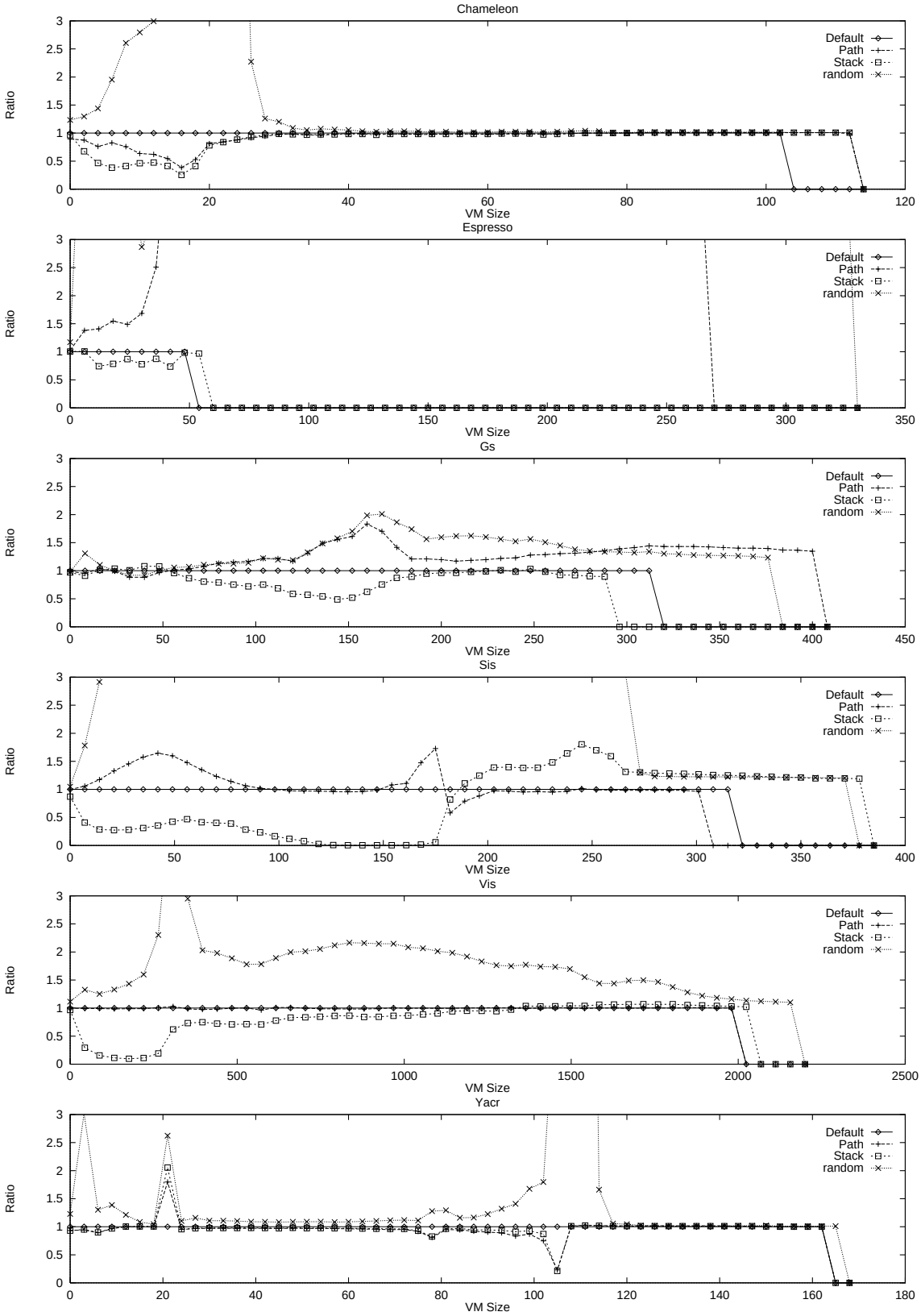

Figure 4: Page fault rate, relative to default, for each of the test programs.

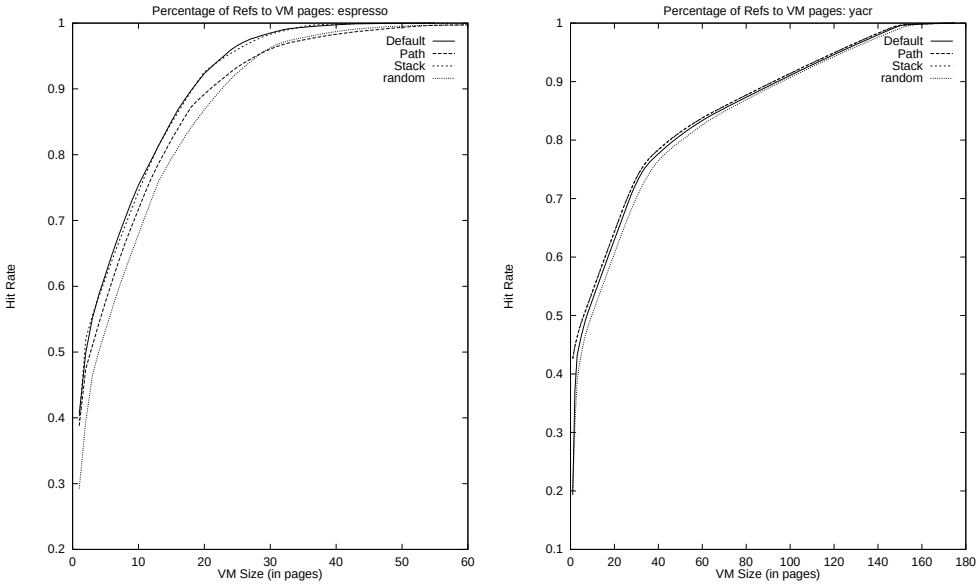

Figure 5: Cumulative reference function for Espresso and Yacr, not relative to default.

one, meaning we have increased the number of page faults. Each curve also exhibits a "drop-down" point where the value of the curve becomes zero. This point indicates the memory size at which the only page faults that occur are fill page faults. To continue to graph the other allocators after Default has "dropped-down" we present results relative to the last non-zero VM miss result from Default.

We do not present graphs for any fixed size of memory, because all of that information in encompassed in Figure 4. For any fixed x-axis value, the graph will show the relative page fault numbers for each allocator.

In the figure, both Sis and Vis have a sizable performance benefit from using the Stack predictor across a range of memory sizes. This can be seen by the fact that Stack reduced the number of page faults by 80% or more, depending on the amount of physical memory available.

In the results for Sis, you see somewhat aberrant behavior on the right side of the graph, where our predictors can vary widely from the Default predictor. This variance is caused by the fact that these graphs are relative to Default. At the far right side of the graphs, the number of page faults incurred by the Default allocator is very small, and even a small number of extra page faults for a particular predictor can cause the relative rate for that predictor to increase dramatically.

Finally, Random does uniformly poorly, which is what we would expect. This poor performance means that randomly distributing objects across four segments has a decidedly negative effect on the virtual memory performance.

5.3 Program by Program Analysis

CHAMELEON For the Chameleon application, Figure 3 shows that our predictions place slightly more references on the highly referenced pages than the Default predictor. Stack does the best, with roughly 1.6 times as many references to the first page, and roughly 1.1 times as many to the first 25 pages. Figure 4 shows that in a very memory constrained environment the Stack predictor improves performance significantly, but if the program has more than a certain minimum number of pages (roughly 40 pages), all of the algorithms (even Random) do equally well.

The other interesting feature of Chameleon in Figure 4 is the fact that the Default predictor uses fewer total pages than the other predictors. This is caused by an interaction between the vmalloc allocator, the objects being allocated, and the segmentation, which results in some amount of internal or external fragmentation.

ESPRESSO Due to its small size, Espresso appears to have very little opportunity for spatial locality optimizations. Figure 5 shows that even in the Default case, the program heavily references a very small number of pages (54 total, for Default). Espresso also has a fragmentation problem—with the Path and Random predictors taking many more pages of memory than the other predictors. For Espresso, this behavior is caused by the large number of reallocs used in the program. With a single memory segment, it is easier to reuse space made available by dead objects or objects that have been moved in order to allow them to grow.

GS The Gs part of Figure 3 is relatively uninteresting. Our techniques do improve the number of references to highly referenced memory pages by some small amount, but less so than in other programs.

Figure 4 is much more interesting with respect to Gs. For a fairly wide range of memory sizes the Stack predictor reduced the number of page faults up to 50%. The other interesting section of this graph is the "drop-down" point for the Stack predictor. This point is less than the Default predictor's "drop-down", even though Table 3 shows the Stack predictor using almost 1.3 times as many pages. The reduction in "drop-down" point comes from segregating objects which are never referenced onto $\overline{\text{HR}}$ pages.

SIS Sis responds well to our object segregation techniques. In Figure 3, we show that the Stack predictor increases the number of references to the most highly referenced pages for Sis. It places almost 1.2 times the number of references that the Default allocator does to the 14 most highly referenced pages.

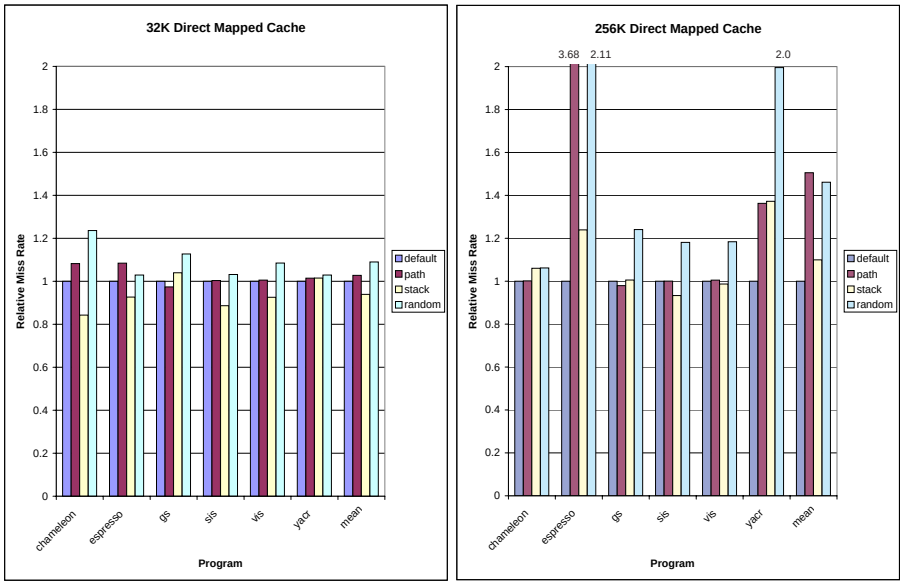

Figure 6: 32K and 256K direct mapped cache. Miss rate relative to DEFAULT.

This increased reference density is also apparent in the page fault rate for SIS (Figure 4). The STACK predictor dramatically decreases the number of page faults encountered by the program at any memory size up to 182 pages, reducing this number by as much as 95%.

On the right hand side of the SIS graph in Figure 4 though, there is some anomalous behavior from the STACK predictor. STACK reduces the number of page faults for most memory sizes, but around 200 pages it has a large increase in the relative number of faults. This spike in the graph is caused by the fact that the graph is relative, and at the far right side a few extra page faults can cause a dramatic increase in miss rate.

VIS VIS has its reference density greatly improved by the STACK predictor as well. This increase in density carries over to the virtual memory performance, resulting in up to an 90% reduction in page faults.

The RANDOM predictor appears to have a strange behavior around the 300 page mark. RANDOM dramatically increases in miss rate, then quickly drops back down. This behavior results from RANDOM placing important objects so they occupy a couple of more pages than they do in the DEFAULT case. At the point where DEFAULT gets all of these important objects into memory but RANDOM does not, the relative miss rate increases dramatically. With the addition of a few more pages of memory though, the RANDOM predictor's miss rate falls back to normal. This is because RANDOM can use these extra few pages for important objects, while DEFAULT is placing less highly referenced objects onto those pages.

YACR The most interesting feature of YACR is the multiple spikes in Figure 4. These spikes, first described with respect to VIS, above, are much more noticeable here. With 21 physical memory pages, DEFAULT places all the important objects so they fall in memory, while all the other algorithms fail and need an additional page. This results in the abrupt spike in the graph. At the 105 page mark, we see the inverse of this behavior. STACK and PATH manage to place all of a second category of objects into memory, while DEFAULT needs four additional pages in order to get the same performance. The different categories of objects can be clearly seen in Figure 5, the non relative version of the YACR graph from Figure 3. There are two very obvious inflection points in the graph showing that there are three types of objects: the highly referenced ones represented by the few pages at the far left of the graph, medium referenced objects represented by the middle of the graph, and objects with a smaller reference density, which account for the tail on the right side of the graph.

5.4 Cache Performance

Figure 6 shows the impact of our object segregation on the cache performance of our test programs. Figure 6 presents results with a direct-mapped 32KB cache and a direct-mapped 256KB cache with 32-byte lines.

Overall the cache miss rate in the 32KB direct-mapped cache is better with STACK than with DEFAULT. All of our test programs except GS and YACR experienced reduced cache miss rates, and the total for all the programs was approximately an 8% reduction in cache misses. This reduction in miss rate is probably attributable to placing HR objects on contiguous pages in memory. If there are fewer than 32KB of these objects, there will be no conflict cache misses between them. Even if there are more than 32KB of these objects, this sequential layout minimizes conflicts. In the DEFAULT case it's quite possible that one cache line has no conflicts while another line has three or more HR objects mapped to it. The other interesting aspect of the 32KB cache graph is the uniformly poor performance of the RANDOM allocator.

For the 256KB cache graph though, our predictors perform less well. The performance of our test programs on the 256KB cache mirrors their performance in the VM tests. The two programs with the most dramatic VM performance improvement, SIS and VIS, have the largest cache improvement, while the program that had the least improvement, YACR, has the worst performance for the STACK predictor. ESPRESSO, which had fragmentation problems with the PATH and RANDOM predictors, shows cache performance degradation for these predictors. Overall these results are

understandable. If the reference density of the program has not been improved by our optimizations, the cache performance does not improve. Our current implementation does very little to try and improve cache performance though, and since our optimizations are at a much coarser grain than the 32-byte cache line, significant performance improvement would have been unexpected.

In future work, we are considering augmenting our runtime system with more cache information. One extension we are considering is using the fact that we know where in memory the majority of HR objects reside to insure that no other objects conflict with these in the cache.

6 Summary

Computing technology trends indicate that it is increasingly important to achieve good locality of reference in programs that perform significant amounts of dynamic storage allocation. In this paper, we propose and evaluate a technique that identifies object behavior and segregates objects based on that behavior.

Our results show that there is a large opportunity to improve program performance in constrained memory situations in four of our six programs. Furthermore, in two of our programs we were able to reduce the number of page faults substantially. Using the technique of profile-based optimization, we are able to identify and segregate objects based on lifetime and reference density so that the most frequently referenced data pages were referenced up to 8.5 times more frequently than they were using the default allocator. This increase in spatial reference locality reduced the page fault rate up to 95% in some memory sizes.

In the future, we hope to be able to continue this work and extend it into new areas. In the short term, we plan to add more cache awareness to the prediction metrics, such as insuring that no pages conflict with HR pages in the cache. This should help improve our cache performance, which currently is not substantially better than DEFAULT. Also, we will implement an actual instance of a predictor allocator so we can more accurately assess the performance costs and benefits of our approach. We are also considering how these prediction techniques can be of value in a garbage-collected language such as Java.

Acknowledgements

This is work is supported in part by NSF grant number CCR-9711398 and a generious equipment grant from Digital Equipment Corporation. In addition we would like to thank all of the anonymous reviewers for taking the time to give us feedback on this paper.

References

[1] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of MICRO-29*, Paris, France, December 1996.

[2] David Barrett and Benjamin Zorn. Using lifetime predictors to improve memory allocation performance. In *SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 187–196, Albuquerque, June 1993.

[3] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1994.

[4] David A. Cohn and Satinder Singh. Predicting lifetimes in dynamically allocated memory. In *Advances in Neural Information Processing Systems 9*, 1996. To appear.

[5] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[6] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software—Practice and Experience*, 24(6):527–542, June 1994.

[7] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs using copying garbage collection. In *ACM SIGPLAN-SIGACT POPL'94*, pages 1–14, Portland, Oregon, January 17–21, 1994. ACM Press.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison Wesley, Reading, MA, 1997.

[9] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *Proceedings of the 1995 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–123, Austin, TX, October 1995.

[10] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *ACM SIGPLAN PLDI'93*, pages 177–186, Albuquerque, June 1993.

[11] Mark D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California at Berkeley, November 1987. Technical Report UCB/CSD 87/381.

[12] 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.

[13] S. McFarling. Program optimization for instruction caches. In *ASPLOS'89*, pages 183–191, Boston, MA, April 1989.

[14] 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.

[15] Judith B. Peachey, Richard B. Bunt, and Charles J. Colburn. Some empirical observations on program behavior with applications to program restructuring. *IEEE Transactions on Software Engineering*, SE-11(2):188–193, February 1985.

[16] Matthew L. Seidl and Benjamin G. Zorn. Predicting references to dynamically allocated objects. Technical Report CU-CS-826-97, Department of Computer Science, University of Colorado, Boulder, CO, January 1997.

[17] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *ACM SIGPLAN PLDI'94*, pages 196–205, Orlando, FL, June 1994.

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

[19] K. Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice & Experience*, 1996.

[20] Tim A. Wagner, Vance Maverick, Susan Graham, and Michael Harrison. Accurate static estimators for program optimization. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, FL, June 1994.

[21] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *SIGPLAN Symposium on LISP and Functional Programming*, San Francisco, California, June 1992.

[22] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, March 1989. Technical Report UCB/CSD 89/544.

[23] Benjamin G. Zorn. The effect of garbage collection on cache performance. Computer Science Technical Report CU-CS-528-91, University of Colorado, Campus Box 430, Boulder, CO 80309, May 1991.