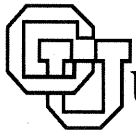
Predicting References to Dynamically Allocated Objects

Matthew L. Seidl
Benjamin G. Zorn

CU-CS-826-97



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Predicting References to Dynamically Allocated Objects

Matthew L. Seidl and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

CU-CS-826-97 January 1997

University of Colorado at Boulder

Technical Report CU-CS-826-97
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA

Predicting References to Dynamically Allocated Objects*

Matthew L. Seidl and Benjamin G. Zorn

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430 USA

Telephone: (303) 492-4398
FAX: (303) 492-2844
E-mail: {seidl,zorn}@cs.colorado.edu

January 1997

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 architectures are including deeper cache memory hierarchies to hide the increasing latency to main memory. In this paper, we investigate efforts to predict which heap objects will be highly referenced at the time they 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. Our results, based on measurements of four allocation intensive programs, show that program references to heap objects are highly skewed and that our prediction methods can successfully predict the program objects that will be the most highly referenced. We show that using this technique, a large fraction of a program's heap references can be directed at a part of the heap that is small enough to fit in the cache (e.g., 8–64 kilobytes).

*This research supported by NSF Grant CCR-9404669 and an equipment donation from Digital Equipment Corporation.

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 ours comparing C and C++ programs showed 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].

At the same time, while basic processor cycle time has increased dramatically, the speed of bulk memory has not increased as fast. As a result, modern architectures now include several levels of cache between the processor and main memory to hide this latency. The result is that it is more important than ever for applications to fit well in the cache on modern systems, and to avoid main memory references as much as possible.

These two trends suggest that providing high reference locality for programs with large amounts of dynamically allocated memory is important now and will be increasingly important in the future. Surprisingly little work has been done with the specific goal of improving locality of reference (especially in the cache) in programs with explicit storage management (e.g., use `malloc`). Recent work of ours surveyed existing `malloc` implementations with the goal of understanding what techniques they provide to support cache locality [10]. Our 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.

We suspected that reference locality could be increased beyond what is currently being done. Our approach exploits the significant non-uniformity of references to heap-allocated objects that we measure and present here. In particular, we observed that a small fraction of the heap objects account for a large percentage of the heap references in all the programs we considered. This skewed access pattern suggests that traditional allocator implementations introduce significant inefficiencies in the cache.

Existing `malloc` implementations allocate memory with little awareness of the specific object address being generated. 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 the cache memory. As a result, addresses are mapped to objects in a mostly random fashion (with perhaps a sequential bias if large amounts of memory are allocated without subsequent frees). We show later that a small number of objects receive a majority of total references in significant programs, and as a result, assigning a random address to such objects may result in two problems in the cache.

First, if two highly referenced (HR) objects are mapped to two addresses that conflict in the cache, then unless the cache is set associative, a significant number of conflict misses will occur. Second, because HR

objects account for a small percentage of the total objects allocated, most objects are not HR objects. As a result, since the object addresses are assigned randomly, on a given cache line only a small fraction of the line may be devoted to an HR object, while the rest of the line is essentially wasted because the objects that occupy it are infrequently referenced.

Organizing program code to improve its locality of reference in the virtual memory (e.g., see [14]) and cache (e.g., see [12]) 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 [18]. In this paper, we investigate the analogous problem of predicting frequently referenced heap objects at the time they are allocated. We refer to this prediction as HR (highly referenced) prediction, and to the algorithms used for prediction has HR predictors. The technique we use is also analogous to a technique we previously published that predicts the lifetimes of heap-allocated objects [2].

Our approach uses profile-based optimization to first identify frequently referenced heap objects based on training inputs, and then to predict when such objects are allocated based on information present at the time of allocation. In this paper we consider a variety of such information including the object's size, the contents of the call stack, the value of the stack pointer, and the depth of the stack.

Based on measurements of four heap-allocation intensive programs, our results show that many of the HR objects in a program can successfully be predicted at the time they are allocated. Using this technique, a large fraction of a program's heap references can be directed at a part of the heap that is small enough to fit in the cache (e.g., 8–64 kilobytes).

This paper has the following organization. In Section 2 we discuss related work. In Section 3 we describe the methods we use to predict HR 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

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 [11] has prompted the improved "better fit" methods [17] that are now often used. As mentioned, in our own previous work, we looked at existing `malloc` implementations with an eye to what impact they had on cache locality [10]. This previous 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 [6, 13, 19], including several recent papers that specifically consider the effect

of garbage collection on cache performance [7, 15, 20, 21]. 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 [6], in which the working set for an entire Lisp system is obtained by performing a "training" of the system, is the closest in this group to our work. However, our focus is specifically on optimizing for cache locality while theirs was on VM performance, and our prediction techniques differ completely from theirs as well.

There has also been related work in the different dimension of attempting to predict the other aspects of behavior of allocation-intensive programs. Perhaps the closest work in spirit to our work is the previous work of Barrett and Zorn, in which we attempted to predict object lifetimes (and short-lived objects in particular) using information present at the time of allocation [2]. While the methods used in this paper are similar, our results are quite different. In particular, in the current work, we investigate a wider variety of predictions methods to solve the more important problem of improving reference locality. Further, we find that the technique used in the previous work, that of combining size and stack context information, is not as effective in predicting reference locality as the methods we consider here. In very recent work, Cohn and Singh 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 [5].

3 Algorithms

The section discusses the optimization process we use to do HR prediction, and describes the specific HR predictors we have considered and their implementation.

3.1 Overview

Figure 1 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 [16], described in the next section). The instrumented program is then run with a number of training inputs that are intended to be representative of the program in actual use.

In our case, these training executions result in trace files that contain a great deal of information about the heap objects allocated by the program. In particular, the instrumentation requires that all loads and stores to heap objects are recorded. This instrumentation is costly, as we describe in the next section, and may explain in part why this particular optimization has not been considered before. 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 potentially interesting to other researchers doing

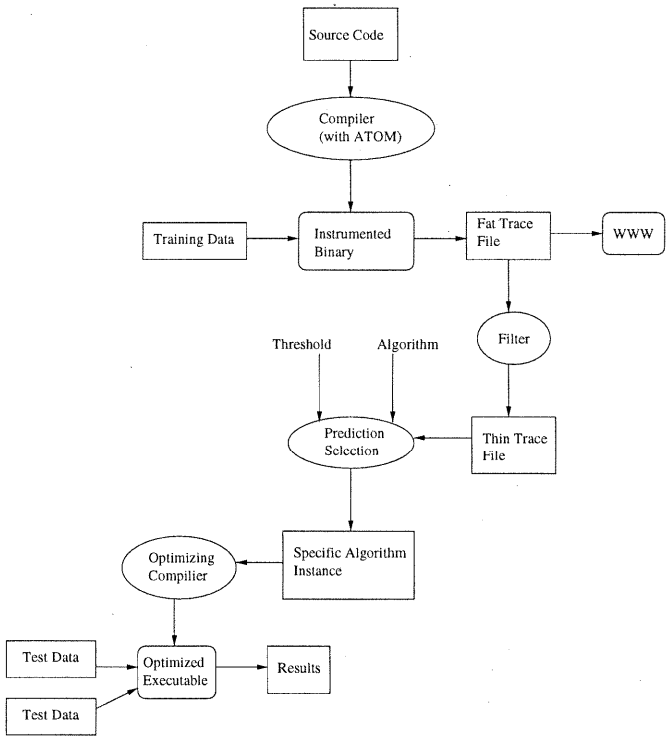


Figure 1: Overview of the Optimization Framework

memory management studies. In the near future, we intend to make these "fat" trace files available publicly on the WWW.

After the unnecessary information is removed from the trace, the result of program instrumentation and training is the thin trace file indicated 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 data in the thin trace file, 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. The current approach we are using is to classify objects as either highly referenced or not (a binary decision), and then to find the best correlation between information available at the time the object is created and the fact that the object is highly referenced.

Categorizing an object as highly referenced can be done in different ways, and we consider two of those approaches here. The most natural way is based on a normalized "references per word of object" (e.g., a reference density) metric, where all objects with greater than a threshold number of references per word are classified as highly referenced. Another possible approach is to ignore the object size in reference calculations, and set a threshold for the total references to the object regardless of its size. Later we show that thresholds based on reference density work much better than thresholds based on total references. Other categorizations are possible, including looking only at loads to objects, instead of both loads and stores. Such a distinction would be of value in systems where store buffers substantially reduce the overhead of store instructions.

Another important decision that must be made before a predictor instance can be created is that the correct threshold value 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 actual threshold value chosen, while being selected automatically, may depend on implementation parameters related to the hardware on which the program is likely to be run, such as the size of the L1 or L2 cache.

Finally, the most important input to the predictor selection process is a set of potential prediction algorithms. Before describing the predictor algorithms in depth, we first outline the rest of the process shown in Figure 1. To make this discussion concrete, suppose the most effective prediction algorithm is determined to be based on object size, and the specific instance of that algorithm for a given program is to predict all objects with size 34 bytes as highly referenced.

After the predictor instance is determined, a modified version of the allocation runtime system is automatically created that implements that instance. In the example mentioned, the malloc code would be modified to include a test for objects of size 34 bytes. If the allocated object is predicted to be highly referenced, then the object is allocated in a different part of the heap, which we call the *HR segment*. The HR segment is a small section of the heap reserved for HR objects. Placing the HR objects in this smaller space, instead of spread throughout the heap, should improve the program's reference locality, and thus its runtime performance. Finally, an optimized version of the program is created that incorporates the predictor-based allocator, and the optimized version of the program is executed with new inputs.

3.2 The Prediction Algorithms

We looked at a number of different predictors, ranging from the simple to the complex. The predictors we have spent the most time considering are based on the value of the stack pointer, the allocated object size, the stack contents, and stack contents plus object size. These predictors have varying degrees of success and varying degrees of implementation overhead. We also considered the call stack depth and when the object was allocated, but neither of these predictors showed as good performance as the ones listed above, so we will not discuss them in depth.

- *Object Size* The object's size is a very easy predictor to implement, but one that uses very little information about the object being allocated. The trouble with using size as a predictor is that unless different sized objects have fundamentally different access patterns, it will end up miss-predicting a large number of objects. Some programs are predictable in this manner, though, such as CFRAC, and for these programs this predictor can be a very efficient way of optimizing the program. A simple case statement inside the `malloc` call can determine if the size being requested is for an HR object and allocate the storage appropriately.

- *Stack Pointer* The stack pointer at the time of object allocation is also a fairly easy prediction to implement, but again offers little information about the object being allocated. The stack pointer does encode the call stack somewhat, and in general provides enough information to be a medium grade predictor for most programs. Because this predictor is fairly easy to implement by placing a wrapper around `malloc`, checking the stack pointer and placing the object in the HR segment if it is called for, it can be useful to some programs. 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. The stack pointer does not encode as much information about the object as the predictors to follow, but for some programs it can do a fairly good job with a low overhead.

- *Stack Contents* The stack contents predictor considers the call chain at the time of the object's creation. This predictor will be harder to implement, but uses more context sensitive information about the object allocated and therefore tends to make better predictions. This kind of optimization is referred to as k-CCP by Chambers [4] (where k is the stack depth used). The depth the predictor looks to is of particular importance with this predictor. If the depth is set too deep, the predictor can over-specialize the prediction and not generate something that is useful across data sets. If the depth is not set deep enough the predictor may not capture what is actually going on. 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 down the stack past these masks, and get into the actual meat of the program while avoiding over-specification. Fortunately, because we are using profile-based optimization, the correct depth to use can be determined automatically on an application by application basis.

  The stack contents predictor requires that the chain of callers on the stack be determined. This problem is similar in spirit to the problem of determining a trace of basic blocks (or path) within a procedure. There has been recent work in the area of path profiling by Ball and Larus [1] which may be used 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

CFRAC	CFRAC is a program that factors large integers using the continued fraction method. The training and testing inputs were 20-25 digit numbers constructed by multiplying two large primes. The program has approximately 6,000 lines of source code.
ESPRESSO	ESPRESSO, version 2.3, is a logic optimization program. The testing and training inputs were two of the inputs provided with the release code. The program has approximately 15,500 lines of source code.
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.
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.

Table 1: General information about the test programs.

the call site. Our hope is that these shift operations can be hidden in the pipeline stalls of today's superscalar architectures. 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 malloc context.

An alternate method of establishing the calling context is to selectively inline the chains of callers that result in predicted allocations of 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.

4 Evaluation Methods

The infrastructure we utilized for our studies was a network of high performance DEC Alpha multiprocessors. Each computer came equipped with four 300MHz processors and 512 megabytes of RAM. On these computers we used a number of DEC tools, such as DEC's C++ compiler, `cxx`, and ATOM. ATOM is a tool that allows programmers to instrument existing binaries with additional code to gather run time data, without interfering with that program's execution [16]. ATOM has the facility to insert arbitrary procedures into a binary at many different places. Code can be inserted per object, procedure, or even instruction. Our use of this ability will be discussed in detail below.

4.1 Programs

Table 1 lists the four programs in our collection, and gives some information about what they do. These

programs were selected because they are all heap allocation intensive, and CFRAC, ESPRESSO, and SIS have been used in previous related work. A breakdown of basic information about the programs and the data sets we used for training and testing is presented in Table 2. We see that CFRAC is much smaller than the

Source Program	Input	Insts Executed ($\times 10^6$)	Total Bytes ($\times 10^3$)	Total Objects ($\times 10^3$)	Total Loads ($\times 10^3$)	Total Stores ($\times 10^3$)	Maximum Bytes ($\times 10^3$)	Maximum Objects ($\times 10^3$)
CFRAC	small	131	24.1	1.2	10.7	4.3	23.2	1.2
CFRAC	large	347	29.9	1.5	27.1	10.6	28.7	1.5
ESPRESSO	Z5xp1	24	1711.7	24.2	3128.2	887.6	47.9	0.7
ESPRESSO	cps	506	22776.4	187.9	79018.6	20783.3	233.9	2.9
VIS	s208	46	3426.9	66.3	4200.0	1407.6	1106.2	12.6
VIS	sbc	179	17553.5	265.2	19748.0	5907.5	3476.7	86.5
SIS	small	59	1091.5	34.2	7081.2	1950.5	556.6	15.7
SIS	speed	360	47996.1	781.2	24288.0	10167.7	995.6	26.2

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. Insts Executed refers the the total number of instructions executed. Total Loads/Stores refers to the total number of heap references the programs executed.

other three programs, and will probably not be included in future work because of its size. The other three programs all allocate between 15 and 50 megabytes of memory over their lifetimes and access this memory actively. ESPRESSO is the most memory intensive, but also has the smallest maximum memory size, and uses the fewest objects at one point. VIS uses the most memory and objects at any point, but ends up allocating the least amount of memory over the run of the program. SIS has the largest total memory usage and falls in between the other programs with respect to how intensely it utilizes its memory, and how large the program is at any one time.

4.2 Program Data Sets

In our study we needed both testing and training data for all of our programs. Of the data sets presented here, we used the smaller of the two as the training set for all of our programs, while testing with the larger set. We felt this would better simulate the real use of this technology, as no matter how much data is used to train the system, the testing is likely to uncover some data paths that were not trained.

One problem we had with some of our training-testing pairs, especially with SIS, is that the training sets and testing sets ended up exercising very different code sections in the program. SIS is a wrapper program, that allows the programmer to call a number of other logic and circuit design libraries, one of which implements the same algorithm used in ESPRESSO. This means that unless the same library within

SIS is used for both the training and testing data, the best that can be done is to optimize the behavior of the shell program, which does very little memory allocation. Therefore, our lower effectiveness in SIS is a result of training on one library component of SIS, and testing on another, and not a fundamental weakness in our algorithms. The optimal curve shows that SIS still has a great potential for this suite of optimizations. If we had trained and tested on the same libraries, we may have seen results closer to those of ESPRESSO and VIS.

4.3 Details

The first step in our data gathering was to instrument our test programs. We did this by inserting procedures at the beginning and end of the program, at each call site, at each procedure's end, at the beginning and end of each memory management routine, and at each load and store instruction.

The procedures at the start and end of the program serve to initialize and finalize the data structures used by our other routines. The procedures at each call site and at the end of each procedure serve to update our internal copy of the program's stack. We have also implemented some optimizations to only output stack values to the trace file if they are necessary for a memory management routine, i.e., calls that do not lead to a memory management routine are not output to the trace file.

The procedures placed at the start and end of the memory management routines update our global object data structure, implemented as an interval set. Each change to this data structure is output to the trace file and, on object deallocation, a number of pieces of reference behavior are also output. The procedures placed at each load and store instruction track the references to the objects in our global object data structure and, on each reference the object's record is updated and the location of the reference stored.

The output file contains enough data to recreate every call to a memory management procedure, what the stack was at the time of this call, and what the reference patterns of the objects created was. We hope to make these trace files available in the future and plan on including complete details along with documentation.

All of these data gathering procedures do add some overhead to the resultant computation. We found that overall the program ran between two and three orders of magnitude slower, and took up one to two orders of magnitude more main memory during its run time. These overheads can be directly attributed to the detail of the data we gathered. The overhead to do a procedure call and manipulate a large data structure every 5–15 instructions is quite large, leading to the large time overhead cited above. The large space overhead results from the fact that we are keeping at least 2 extra copies of every object in memory, in addition to the overhead of the data structure, and other information about the object. These two copies are used to track loads and stores on a per word basis within the object. Overall, these overheads limited the size of inputs we could observe, but fortunately, our substantial hardware infrastructure still allows us to

evaluate our technique using significant programs. Reducing the overhead of data collection and optimizing the information gathering process is one of our goals for future work.

5 Results

In this section, we present the results of applying our prediction methods to four test programs. We first consider the effectiveness of different predictors across all four programs, and we then look at other parameters and the impact of those parameters on our results.

5.1 Predictor Effectiveness

Figure 2 shows the effectiveness of five different HR object predictors across our four test programs. Each figure shows the fraction of total bytes allocated on the X axis, and the fraction of total references on the Y axis. In all figures except Figure 4 the threshold metric used to determine HR objects is the reference density of the objects (e.g., predict an object will be an HR object if it experiences some number of references per byte allocated). Each point on the figures corresponds to a setting of the threshold value and predictor used. To generate the curves shown, we varied the threshold value widely from very high thresholds, in which only the most densely referenced objects are considered HR objects, to very low thresholds, in which almost all objects are classified as HR objects. The shape of the curve shows how effective each predictor is at identifying HR objects.

These curves imply for a given threshold setting the fraction of all program heap references that would be made to the HR segment if HR prediction is used. For example, the ESPRESSO figure shows that a very substantial fraction of all heap references can be directed at a very small HR segment. Note that these figures conservatively estimate the size of the HR segment because the assumption is that objects allocated to the HR segment are never subsequently freed and reclaimed. In practice, if HR objects are freed and the storage reused, then the size of the HR segment would be smaller than indicated on the graph.

We also note that the fraction of bytes classified as HR objects can be very sensitive to the setting of the threshold. As a result, in some of the curves presented, portions of the X-axis are not covered by a particular curve. This implies that further increases or decreases in the threshold setting resulted in classifying all (or no) objects as HR objects, resulting in values of either 1,1 or 0,0 for these settings. In practice, the determination of the threshold setting for a particular program would be performed *automatically* using the training data gathered. The threshold selected would depend on implementation parameters such as cache size, and would be based on a exploration of the self prediction curve, the data for which can be computed from the training run.

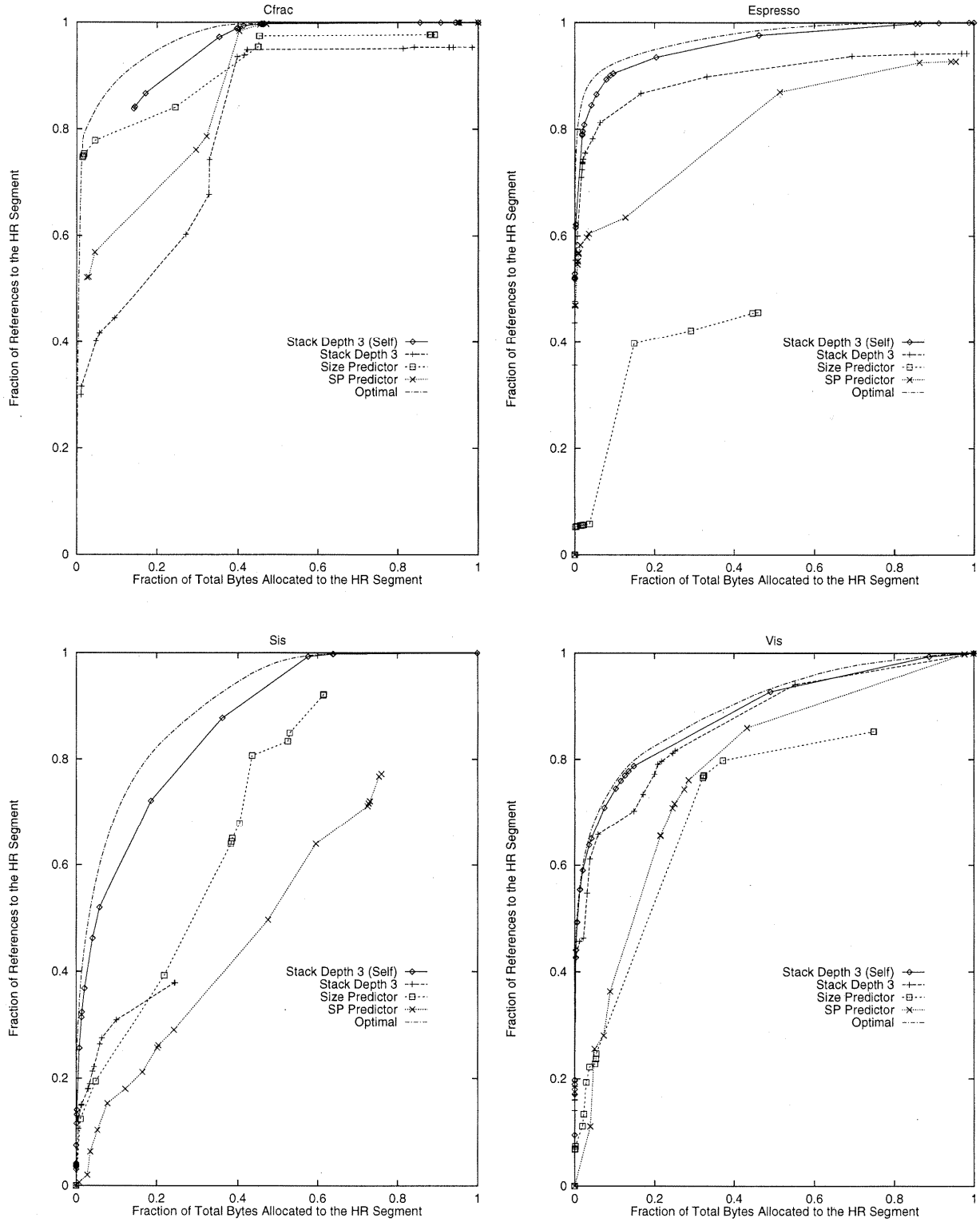


Figure 2: Performance Results for Our Four Test Programs

Each subfigure in Figure 2 includes three curves based on predictors using different training and testing sets, and two curves specifically for comparison. The curve labelled "optimal" shows what happens if all objects allocated by a program are sorted with highest reference density objects first (based on the actual reference counts taken from the test data set). The optimal curve indicates the best possible space allocation to achieve the highest reference density possible. The shape of the optimal curve in all programs indicates that a great deal of opportunity exists in these programs for organizing heap objects to improve reference locality. In particular, the curve indicates that a small number of objects account for a large fraction of total references in all cases, and furthermore, a large fraction of objects often account for almost none of the reference activity (i.e., the curve becomes almost horizontal in three of the four programs).

The curve labelled "Stack Depth 3 (Self)" in the subfigures indicates the effect of using exactly the same input for training and testing (i.e., self prediction). This is another idealized result that is not achievable in practice, but provides an upper bound on the effectiveness of the "Stack Depth 3" predictor. In all cases, we see that self prediction follows the optimal curve quite closely.

The remaining three curves, "Stack Depth 3", "Size Predictor", and "SP Predictor", indicate the effectiveness of HR prediction in practice. From the figures, we see that the size predictor is ineffective in three of the applications, but quite effective in CFRAC. Interestingly, the most effective predictor overall, "Stack Depth 3" performs well in ESPRESSO and VIS, but poorly in CFRAC. One nice result of our profile-based approach is that the particular predictor and threshold to be used in a given application can be selected automatically on an application by application basis. Since size appears to be most effective in CFRAC, and requires less runtime overhead to implement, it is an attractive alternative for that application. We note that the "SP Predictor" works better than size in most cases, however fails to perform as well as the "Stack Depth 3" predictor in all cases except CFRAC.

The "Stack Depth 3" predictor is highly effective in both ESPRESSO and VIS, to the point that it closely follows the self prediction curve, particularly on the far left side of each graph. Unfortunately, none of the predictors appears to work well in SIS, in part because there is less reference locality available to exploit, as indicated by the optimal curve. The self prediction curve in SIS appears much closer to optimal than the "Stack Depth 3" predictor, indicating that at least part of the difficulty in predicting SIS HR objects resulted from using training and test sets that exercised significantly different parts of the program.

In practice, the goal of HR prediction is to identify a very small fraction of the total objects as HR objects and create an HR segment that will fit comfortably in cache memory. As a result, for the programs measured, the interesting portion of the graphs presented is the far left portion, in which the curves are often almost vertical. To understand this fraction of the curve better, we present the data in a slightly different way in Table 3.

Program	% Refs to 8 Kbyte HR Segment	% Refs to 64 Kbyte HR Segment	% Refs to 256 Kbyte HR Segment	HR Seg. Size to catch 50% Refs. (Kbytes)	HR Seg. Size to catch 75% Refs. (Kbytes)	Total Bytes Allocated ($\times 10^3$)
ESPRESSO	40%	60%	65%	41.0	582.1	22776.4
SIS	5%	8%	11%	14686.0	35997.0	47996.1
VIS	30%	40%	46%	456.3	3159.6	17553.5

Table 3: Information about what percentage of a program's references can be grouped into how much memory (8K, 64K, or 256K bytes) and how many kilobytes the program needs to capture what percentage of references.

In the table, we present the relationship between the fraction of total heap references and the size of an HR segment that captures that percentage of references. In the left side of the table, we fix the size and show what fraction of references are made to the HR segment, and in the right side of the table, we fix the fraction of references, and show the size of the HR segment needed to account for that fraction of references. The predictor used in the "Stack Depth 3" predictor in all cases.

First, the table does not include the CFRAC application because the total bytes allocated (30 kilobytes) is not sufficient to be of interest. The table shows that for both ESPRESSO and VIS, HR prediction with a high threshold can result in a very small HR segment (8 kilobytes) and still identify objects that account for 30–40% of all program heap references. If the HR segment is allowed to be larger, then up to 65% of ESPRESSO's heap references can be limited to a 256 kilobyte region of the heap. As indicated in the previous figure, however, only a small fraction of heap references can directed to the HR segment in the SIS program.

5.2 Combining Stack Depth with Size

Barrett and Zorn [2] concluded that combining both size and stack context information resulted in more effective prediction of short-lived objects in C programs. In this section, we consider the effect of using this combination for HR prediction. The problem with this combination is that it results in highly specific context information that may not generalize well from one input set to another. We considered this combination, and the results for two of the four programs are presented in Figure 3. The results for the other two programs are comparable. The figure shows that combining stack and size reduced the effectiveness of the predictor in both cases, and significantly reduced the effectiveness in the ESPRESSO program. Upon further investigation, we noted that the ESPRESSO program performs a large number of calls to `realloc`, growing or shrinking the objects as appropriate. We treat such `reallocs` as `mallocs` followed by `frees`, and as a result, using size as part of the predictor overspecializes the HR prediction and causes it to fail to generalize from the training input to the test input.

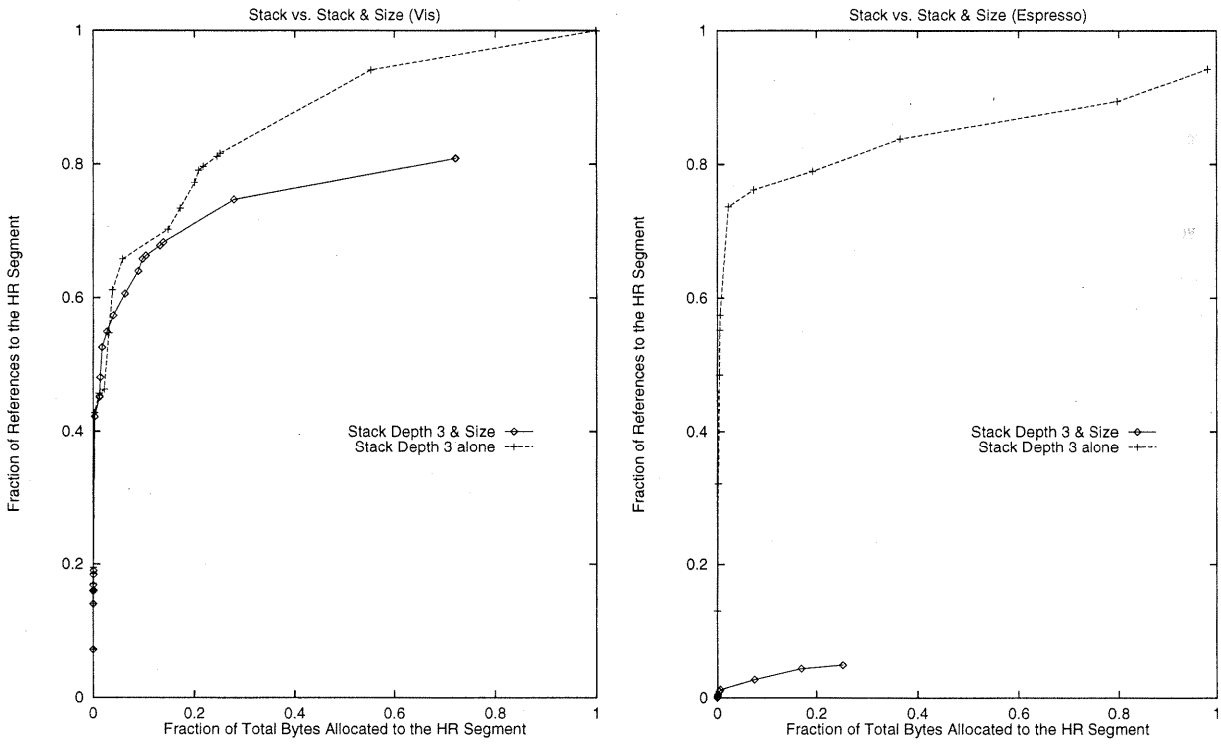


Figure 3: The Effect of Including Both Stack and Size in Two Programs.

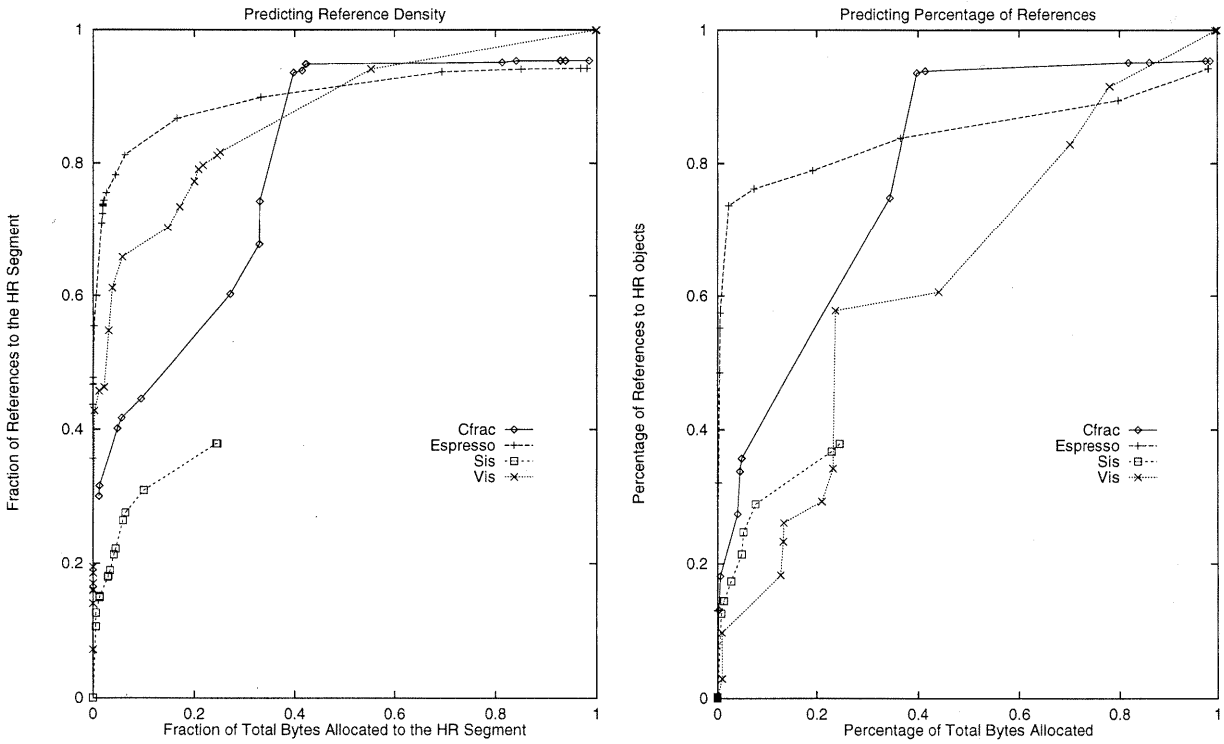


Figure 4: The Effect of Using a Different Threshold Metric in Four Programs using Stack Depth 3.

5.3 Using Different Threshold Metrics

In this section, we consider the effect of using a different threshold metric to predict HR objects. Up to now we have considered basing the threshold on the predicted reference density of objects. We now compare that approach to basing the threshold on the total fraction of program references to the object (i.e., ignoring the object's size in the computation). As expected, this metric is less effective in that it does not account for large objects that are highly referenced overall, but not highly reference on a per word basis.

Figure 4 shows the effect of using both threshold metrics for the four programs. The reference density threshold outperforms the reference percentage threshold in all cases. Considering VIS, for example, we see that the reference percentage threshold results in dramatic increases at approximately 25% of the bytes allocated. This near-vertical line indicates that a collection of objects accounting for a significant fraction of the total references are classified as HR objects only after many other objects that account for less of the total references are classified that way. The equivalent line in the reference density figure indicates that this threshold metric correctly classifies these objects as HR objects before the other less frequently referenced ones.

5.4 The Effect of Stack Depth on Prediction

As with combining stack depth and size, increasing the depth of the "Stack Depth" predictor makes the predictions more specialized and thus less capable of generalizing between program inputs. As described by Chambers et al. "...as the amount of context increases, the precision of the distributions increases, but the applicability of the distributions decreases ..." [4][p 25]. On the other hand, with respect to calls to `malloc`, we know that many applications hide `malloc` behind application specific abstractions, and thus in many cases some context is needed.

Figure 5 illustrates the effect of varying the stack depth from one to five in the Stack Depth predictor in the VIS application. With a stack depth of 1, the predictor has no information, because calls to `malloc` are all indirected through another interface. With a stack depth of two, we see better performance, but less effectiveness than depths of 3 or 4, which are quite close in performance. Finally, at a stack depth of 5, the performance again decreases, probably because too much context is being used to perform the prediction.

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 highly referenced objects when they are allocated and segregates those objects to improve a program's locality of reference.

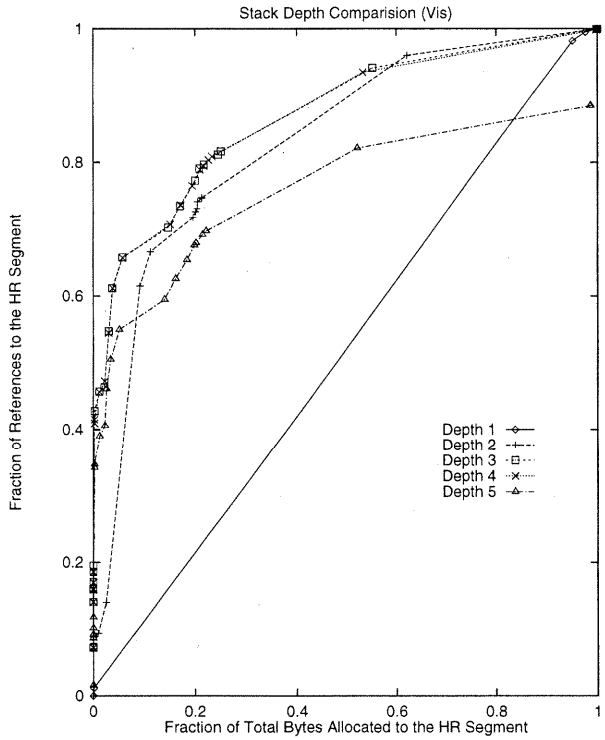


Figure 5: The Effect of Stack Depth on Stack Depth Predictor Performance

Our results show that the opportunity to exploit highly referenced objects exists in four allocation-intensive programs. We further show that the prediction techniques we develop are effective in three of the four programs considered. Using the technique of profile-based optimization, we are able to identify and segregate objects so that a large fraction of a program's heap references can be directed at a part of the heap that is small enough to fit in the cache (e.g., 8–64 kilobytes). We further investigate several different kinds of prediction techniques, and show that the correct context and information to use to predict highly referenced objects varies from program to program but can be identified as we process the training data.

In the future, we plan to implement an allocator based on predicting highly referenced objects and measure the improvement in program reference locality and overall performance using this technique. We will also carefully consider the costs of implementing our prediction algorithm, including the resulting code expansion if selective inlining is used to specialize particular allocation calling contexts. While our study used C programs that perform explicit storage allocation, our technique is also applicable to garbage collected languages, and we intend to investigate such implementations in the future. Finally, our results indicate that there might be value in classifying objects into several categories (e.g., highly referenced, referenced, and not referenced). We also plan to investigate this possibility.

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] Craig Chambers, Jeffrey Dean, and David Grove. Whole-program optimization of object-oriented languages. Technical Report UW-CSE-96-06-02, University of Washington Department of Computer Science and Engineering, Seattle, WA, 1996.

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

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

[7] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs using copying garbage collection. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (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 *SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 177–186, Albuquerque, June 1993.

[11] 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.

[12] S. McFarling. Program optimization for instruction caches. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, Boston, MA, April 1989.

[13] 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.

[14] 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.

[15] Mark B. Reinhold. Cache performance of garbage-collected programs. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 206–217, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices*, 29(6), June 1994.

[16] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, June 1994.

[17] 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.

[18] 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.

[19] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Ontario, June 1991. ACM. Published as *SIGPLAN Notices 26*(6), June 1992.

[20] 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.

[21] Benjamin 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.