Microarchitecture-aware Profile-guided Heap Allocation

Efe Yardımcı
David Kaeli
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, 02115
{eyardimc,kaeli}@ece.neu.edu

Abstract

As memory latencies continue to grow, effective use of cache memories is necessary. A disproportionate percentage of data cache misses are caused by accesses to dynamically allocated memory blocks, and accesses to a small number of heap objects are responsible for a majority of the heap misses. There is an opportunity to improve cache access performance to these objects.

We have modified an existing malloc library to allocate heap objects with the aim of reducing data cache conflicts. We have designed our allocation strategy considering how this scheme would be incorporated into a dynamic optimization system. Our new allocation routine utilizes microarchitecture information and program behavior obtained from profiling accesses to heap objects. This profile is then used to guide allocation on subsequent runs, mapping conflicting heap objects to regions in the cache where they will potentially cause fewer cache conflicts. The target microarchitecture for our work is the Compaq Alpha 21264.

In this work we tradeoff the cost of profiling with the accuracy of modeling cache collisions. We build a state-based scheme that considers the call path leading up to an allocation, along with the size of the allocated block, and attempts to use this information to identify potential conflicts at runtime. The standard malloc allocator is used to obtain an address, and before returning this address to the application we check for potential cache conflicts with other allocated objects. When a potential conflict is found we remap the allocation to a non-conflicting region.

We have implemented our profiling and remapping scheme and obtain actual runtime improvements for our profile-guided allocator. Using a range of benchmarks taken from SPEC2000int, SPEC2000fp and the Olden suite, we can achieve speedups of up to 5.5%.

1 Introduction

Memory latency becomes increasingly important as the gap between processor and memory performance grows. Many methods have been proposed to overcome this latency, such as the design of sophisticated cache memory hierarchies and cache prefetching algorithms. A memory hierarchy can hide much of the memory latency only if a large percentage of the memory accesses are in the cache when requested.

The widespread usage of object-oriented applications written in Java and C++ has

meant that usage of dynamic memory allocation has increased dramatically. It has been observed in a previous study [3] comparing C and C++ that over a range of applications dynamic allocation is done ten times as often in C++ than in C.

Objects allocated dynamically to the heap region generally take the form of linked data structures (LDS), which are formed by linking the dynamically allocated nodes to one another through pointers to form a complete structure. These structures differ from statically allocated arrays as they do not necessarily have consecutive elements at contiguous memory locations. Operations such as sorting and insertion/deletion can also alter the overall structure of LDS at runtime. This inherent lack of spatial locality reduces the efficiency of conventional prefetching methods in LDS intensive applications. Temporal locality may also be lacking [9], as a traversal through an LDS may involve visiting enough nodes to displace a node from the cache before it is visited again (i.e., exhausting the cache's capacity).

When we compare the characteristics of memory accesses made in the heap region to those made in other program data regions (mainly to the data segment), several differences stand out. First, a disproportionate number of data cache misses are caused by accesses to the heap region, as seen in Table 1. For example in equake, a simulation of seismic wave propagation from the SPEC2000fp suite, around 90% of all cache misses are caused by accesses to dynamically allocated memory, though accesses to the heap region make up only 19% of all accesses.

Another interesting characteristic of dynamically allocated memory accesses is that increases in cache size do not significantly reduce collision and capacity misses the same way they do in statically allocated structures. Experiments involving several benchmarks suites showed that LDS cache misses actually make up an increasingly disproportional percentage of the misses with increasing cache size, as shown in Figure 1.

Program	twolf	vpr	equake	ammp	power	tsp	em3d
Percentage of heap accesses over all accesses	13.3	24.1	19.2	21.8	0.5	11.9	43.9
Percentage of heap-based cache misses over all misses	49.6	91.8	88.8	86.2	18.1	30.5	92.2

Table 1: Number of heap accesses and heap misses as a percentage of the total number of memory accesses and cache misses, respectively, modeling a 64K 2-way L1 cache using medium-sized benchmark inputs.

When we look at the source of the misses in the heap region, we notice that a small number of objects account for a large percentage of the misses [13]. This is significant in that it shows why assigning a random address to heap objects (which in practice approximates the case in most malloc implementations) may lead to problems. When multiple objects with high reference counts are mapped to addresses that conflict in the cache, a significant number of misses can occur. While associativity can reduce this effect, it can not eliminate all conflict misses.

In general, existing allocation routines tend to balance allocation speed and memory usage, though locality preservation has not been a major concern. Eliminating boundary tags and providing rapid front-ends for object reuse has been of some benefit, but the same issues still remain.

Usage of some more aggressive allocation mechanisms may in fact inadvertently reduce cache locality. An example of such a memory allocator mechanism is the usage of a *roving pointer* for allocation [15]. This mechanism avoids the traversal of splinters (i.e., small freed blocks) in the free list by cycling through the free lists in successive searches, but also ensures that the whole list will have to be traversed before the same block is revisited. It may also reduce the locality of the program by scattering objects

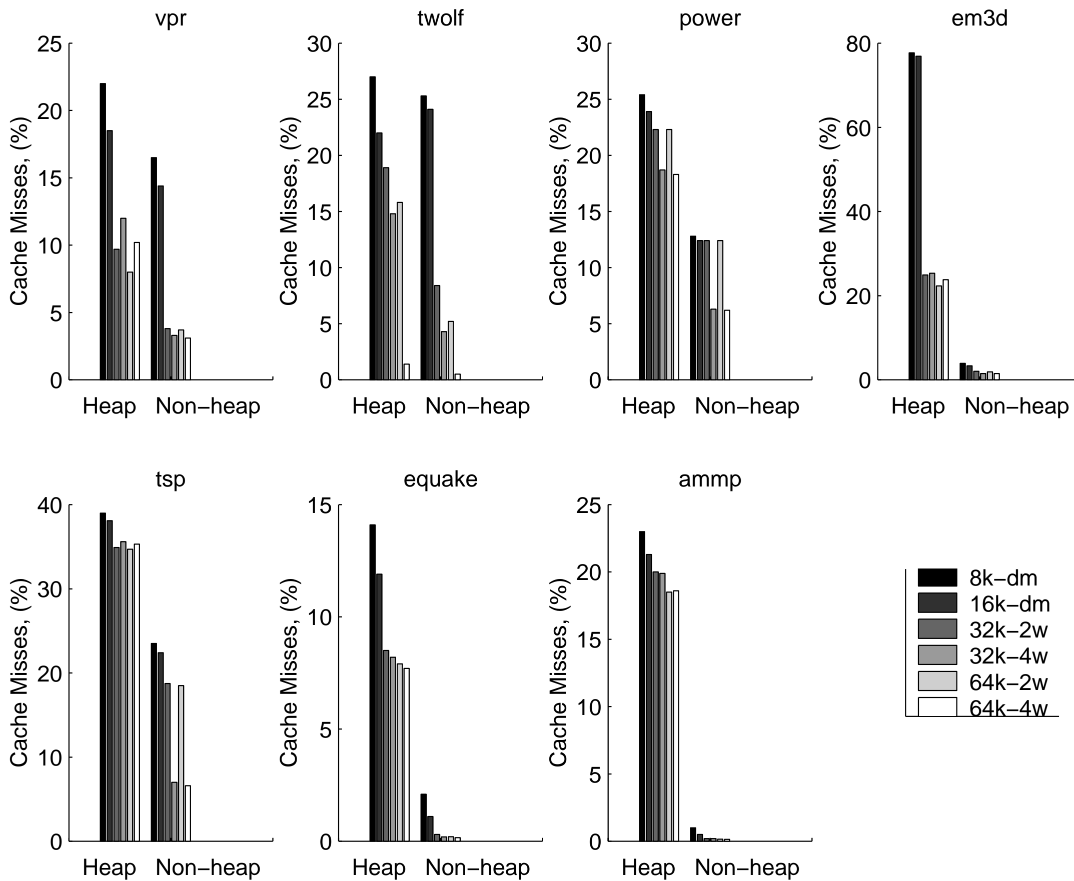

Figure 1: Cache miss rates for non-heap and heap segment accesses with varying cache sizes and associativities.

used by certain program phases among those used by other phases.

Also worth mentioning is the observation that the footprint of dynamically allocated objects tends to be larger than the size of the data segment, even though the number of accesses to the heap region is generally smaller than the number of accesses to the data segment.

The heap region begins from the top of the data segment and expands upwards towards the stack segment (which starts from the top of the address space and expands downwards). The region lying between the stack segment and the heap region is unused. If the heap region is filled and more space is required, the heap region extends to the unused region typically through system calls such as sbrk() and brk(), which request memory from the system for the calling process.

Program	Total Allocation (in bytes)	Maximum Footprint (in bytes)	Final Footprint (in bytes)
em3d	1,289,800	1,289,800	244,880
power	468,944	468,944	468,944
tsp	1,836,592	1,836,592	1,835,752
twolf	10,245,722	2,470,941	1,092,089
vpr	74,778,508	19,789,548	18,159,766
ammp	30,626,152	26,400,656	23,479,132
equake	13,230,080	9,244,200 0	7,142,543

Table 2: Utilization of heap memory during program execution. The Total Allocation column refers to the amount of memory allocated during the total execution of the program. The Maximum Footprint refers to the maximum amount of heap that was allocated at any point in time during the program execution. The Final Footprint column refers to memory still residing in the heap at program termination.

As observed in the benchmarks listed in Table 1 (executed with medium-sized inputs), the heap footprints would not fit in even the largest L1 caches currently available. This is actually expected since an allocated block may have space overhead such as block headers and footers that are not present in statically allocated memory. More importantly, allocator mechanisms and programming practices (not to mention memory leaks) greatly influence memory layout. Programs possessing large heap footprints are subject to many potential cache conflicts.

We have developed a profile guided approach to allocating heap objects to improve heap behavior. As previously mentioned, mapping of blocks with high reference counts to the same cache line causes a significant number of cache misses. Our objective is to identify and reduce potential cache collisions.

Our approach uses profile-guided optimization. We perform separate training and testing runs (with different program inputs) to evaluate our profiling accuracy. The results we obtained in the training program guide allocation in the actual program execution through the use of *state IDs*, which are comprised of a snapshot of the procedure call stack (ignoring that last three calls since they are generally associated with the present call) and *object size* information. We use these states to map conflicts between the training and actual program executions.

The rest of this paper is organized as follows: Section 2 reviews related work. Section 3 explains our policy and the methods we use to implement a general profile-guided cache-conscious memory allocator. Section 4 presents our results. Section 5 summarizes the contributions of this work and suggests future directions for this work.

2 Related Work

There has been a considerable amount of research doen on the effects of garbage collection on data cache performance and locality [7, 8, 12, 16]. Chilimbi and Larus [5] have studied reordering objects in memory during a garbage collection cycle.

Kistler and Franz [11] describe a method of adapting the internal storage layout of heap objects. They attempt to increase cache performance by modifying the layout of fields within an object. Their main objective is to identify fields that are frequently accessed within a certain time and colocating them on the same cache line. They place individual fields in a single cache line in an order to increase spatial locality. Their work relies on the process reaching a threshold after which the overhead introduced by the continuous profiling and code regeneration will be offset by the improved locality.

The work perhaps closest to ours is that of Seidl and Zorn [4], in which they use pro-

filing to classify objects into categories (highly referenced, not referenced) and attempt to reduce page faults by allocating objects in the same category into the same segment (thus improving reference locality). They use combinations of *call stack*, *object size* and the *stack pointer* in their predictor algorithm with varying degrees of success.

Barrett and Zorn [2] studied lifetime prediction and attempted to place objects in the heap region according to the predicted lifetime. They utilized the call stack and object size to identify an *allocation site* and extrapolated the future behavior of the program by the results obtained with training inputs. In many cases they achieved increases in performance. Cohn and Singh [6] used a similar decision tree based lifetime prediction in which they attempted to distinguish *short-lived* and *permanent* objects. They send each short-lived object to a *quick malloc*; in addition they use a specialized routine to handle those objects they classified as permanent.

Our work targets a middle ground, using a program state (based on call stack state and allocation size) to improve data cache performance by reducing conflicts between different heap objects. We have studied the causes of data cache misses in the heap access stream and found that a majority of the misses are due to collisions between accesses to different object versus to accesses within a single object. The only instances of when intra-object references became an issue were for large objects. In our reordering algorithm, we only move objects that are smaller than 25% of the target cache size (this value was arrived at through experimentation).

3 Profiling and Allocation Algorithms

To implement our ideas we obtained and modified dlmalloc, a version of malloc/free/realloc written and released to the public domain by Doug Lea who was a primary author of

Program	Benchmark Suite	Description
em3d	OLDEN	Electromagnetic wave propagation in a 3D object
power	OLDEN	Power pricing system optimization problem solver
tsp	OLDEN	Traveling salesman problem solver using a partitioning algorithm
twolf	SPEC2000int	Place and route simulator using simulated annealing
vpr	SPEC2000int	Integrated Circuit Computer-Aided Design Program
equake	SPEC2000fp	Simulation of seismic wave propagation in large basins
ammp	SPEC2000fp	Modeling large systems of molecules

Table 3: General benchmark information.

libg++, the GNU C++ library. It is a highly versatile malloc, being satisfactorily fast, memory-conserving, tunable and portable at the same time.

We used a range of benchmarks with different properties. We selected 3 benchmarks from the OLDEN benchmark suite [4] which are used in pointer-intensive and heap-intensive research, two benchmarks from the SPEC2000int suite and two benchmarks from the SPEC2000fp suite. Table 3 lists these seven programs and their brief descriptions.

All profiling was performed on Alpha workstations. We used the ATOM [14] binary instrumentation tool to instrument binaries for profile extraction. ATOM allows us to insert analysis procedures into executables to obtain run-time data with minimal perturbation of the memory profile. Runtime improvement results were obtained by running the executables with the modified allocation libraries on Alpha 21264 workstations with a 64K 2-way set associative L1 data cache, which was our target platform.

3.1 State Identification

We need to perform profiling both during a training session (this will allow us to map conflicts to program states), and again during runtime execution (we need to use the profile to guide where to allocate heap data structures). We refer to the first pass of profiling as *miss profiling* and the second pass (during target execution) as *runtime profiling*. We will monitor program state to identify the instance of a malloc. We then use this information to index into a miss profile, identifying candidate cache locations where conflicts may occur for that instance.

One of the contributions of our work is to propose a profiling scheme that would introduce minimal runtime overhead, while also providing an accurate picture of program state to guide future allocations. We use *state predictors* to interface between the miss profile generation and the runtime profiling. To examine the effectiveness of our profiling, we use different program inputs during miss profiling and runtime profiling.

Among possible sets of state information we can use to identify a particular allocation, we have tried using the *stack pointer*, the *procedure call stack*, *allocation size* and *allocation call site*. We have evaluated each of these possible predictors used singly, as well as in combination with each other. The predictors we have chosen to study for our application have also been used by others in related work [2, 6].

To judge our ability to map from a miss profile state to a runtime profile state, we develop a measure of *profile state coverage*. If we identify n states for the miss profile, $n - m$ of those states are found during the runtime profile execution and there are $n + k$ distinct states for the runtime profile execution, we say we have a correlation of $1 - (m + k)/(n + k)$ with the predictor used.

Implementation overhead of obtaining state information during program runtime is

obviously a very important factor as it will limit the runtime benefits of our scheme. Some predictors may capture more context-sensitive information and thus make better predictions, though they will also incur high runtime overhead (e.g., multiple table lookups and updates).

Capturing the stack pointer at allocation time is easy to implement as a predictor, but did not prove to be useful when state coverage is considered. Capturing the call stack incurs additional overhead, but also provides much higher coverage. We can utilize efficient path profiling techniques, as proposed by Ball and Larus [1], to reduce this overhead.

Capturing the object size is simple to implement, but, used alone, provides very poor state coverage. When used together with the call stack state, it provides us with a high level of state coverage. We have also experimented with capturing the *malloc* call site within a procedure. This is a very accurate predictor mechanism, though has proven to be too costly to implement, and does not provide a significant advantage over using the combined call stack and object size state predictor. For the remainder of the paper we will be assuming a predictor which records call stack state (discarding the 3 most recently accessed procedures since these tend to be wrapper routines for malloc), combined with the object size.

3.2 Profile Creation

We have used the ATOM binary instrumentation tool to obtain our miss profiling results [14]. We have also used ATOM to emulate monitoring of the call stack for the runtime execution. We had in fact implemented a instrumentation-based call-stack monitor, but ATOM places a 30% runtime overhead on the program runtime. Since capturing this information at runtime in either a hardware or software mechanism is

trivial, we decided to assume this information is captured by the dynamic optimization system (as emulated by ATOM), and then use it in a final run of the program, measuring total execution time.

ATOM captures each unique call to malloc (recorded with a unique malloc call number), and the corresponding state information. This information will be used to index into the miss profile which is stored in a conflict table. The conflict table is indexed by the malloc number, which indexes to a state, which hashes to a number of potential cache conflicts.

During the creation of the conflict table we found it useful to use a temporal relationship graph (TRG) to prune any potential conflicts that that have no temporal relationship with one another (do not occur close in time). This can help us focus our attention on those situations which have a higher potential for encountering conflicts. Figure 2 shows how the malloc number is used to lookup a state table entry, which is used to hash into the conflict table to find conflicting cache blocks.

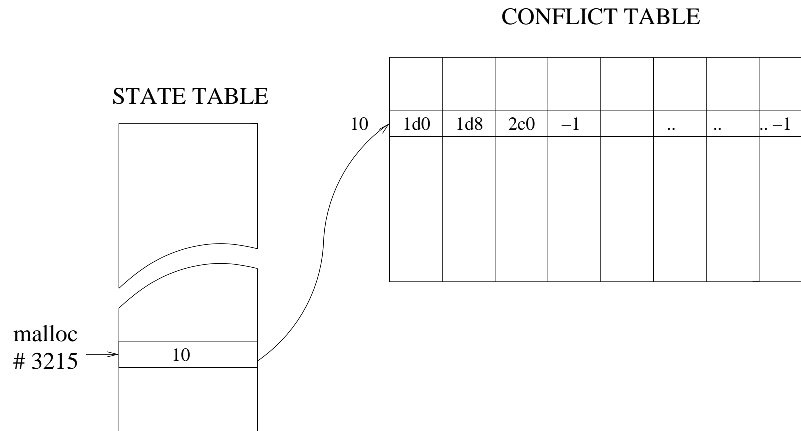

Figure 2: State Table to determine the predictor state and the corresponding entry in the Conflict Table to guide the allocation.

3.3 Implementation of Profile-Guided Allocation

As mentioned previously, we used the dlmalloc allocation library as the basis for our implementation. This allocator holds several bins and tries to match allocation requests with each bin before trying to allocate in the *wilderness block* (the highest address heap block, placed between the heap region and the unmapped region; this block is always free and is the only block whose size can grow freely). Whenever a call is made to malloc our modified malloc waits until a free space is found. Then the Conflict Table is checked with the predictor number obtained from the State Table. The entry in the Conflict Table corresponding to the predictor state is checked if there is a potential collision with the candidate block. Figure 3 shows pseudocode for our modified allocation algorithm.

If there is a conflict, the candidate block is not allocated and the next available bin is checked until either a conflict-free address is found or the wilderness block is reached. If we are at the wilderness block and the immediate region is also found to result in a conflict, we check the following cache line to see if it is free of conflict. This is repeated until a conflict-free cache line is reached. If during this process, the end of the wilderness region is reached, a call is made to sbrk() and the wilderness block is extended.

Once an address is found, the region starting from the original address up to the non-conflicting address is allocated. The actual allocation is done immediately after this region. The previously allocated buffer is immediately freed, causing minimal space wastage (though possibly some limited fragmentation).

An important point to consider is the treatment of large objects. Objects with size greater than a quarter of the cache size are not considered by this algorithm, they are allocated as they would be otherwise. There are two main reasons for ignoring them: 1) objects greater than a certain size tend to register a very large number of

```
FUNCTION ProfileGuidedMalloc(Input Size){
   IF (Input Size > Cache Size / 4){
      RETURN Result From Standard Allocation;
   }ELSE{
      WHILE(1){
           Victim := Address of Next Available Block;
           IF (Wilderness Block reached)
               break;
           IF (Victim Has No Conflict in Conflict Table)
               return Victim;
      }
      // If we get here, we are in the Wilderness Block
      Victim := Top of Wilderness Block;
      WHILE (Victim Has a Conflict in the Conflict Tal
           Victim := Victim + Cache Line Size;
      }
     Return Victim;
}
```

Figure 3: Pseudocode for our Profile-guided allocation algorithm.

collisions with all the other states, and 2) the large size of the block also decreases the accuracy of the profiling. One assumption that we have made is that the entry address of a block is representative of the rest of the block when we need to register conflicts. This is obviously true with blocks of sizes less than a few cache lines; it also holds for somewhat larger blocks as well, probably because the entry address of any block is the most heavily traversed address in the whole block.

4 Results

Table 4 presents the actual runtime of the profile-guided allocated benchmarks. Improvements vary from .85% to 5.5% These results are promising, and we feel we can further improve upon these numbers if we are more careful where we remap a block to. Also, despite the overhead introduced by lookups into the conflict table at allocation time, we are still able to achieve reasonable speedup figures.

Program	twolf	vpr	equake	ammp	power	tsp	em3d
Speedup	3.1%	1.5%	5.5%	2.8%	1.9%	1.0%	0.85%

Table 4: Experiment results.

The figures in Table 4 were obtained by running the optimized and unoptimized versions of the benchmarks 10 times, disregarding the highest and lowest recorded times, and averaging the remaining execution times.

5 Conclusion

In this paper we propose and evaluate techniques that attempt to reduce data cache conflicts caused by accesses to the heap region. We attempt to identify blocks or machine states (used to identify conflict-causing blocks) by a profiling pass and use information on the cache microarchitecture to direct the memory allocator. We have modified a well-known malloc routine to use the profiling results and attempt to place selected blocks into addresses where they are believed to cause a fewer number of cache conflicts with other blocks.

Our preliminary results show we can obtain satisfactory speedup figures using a low cost state identifier (call stack state and object size). Our results give us confidence in the validity of our policies and provide motivation to carry our work further.

In the future, we plan to employ a more elaborate block placement algorithm that uses cache line coloring that remembers past allocations. We also plan to consider block splitting, taking into account the hot and cold regions within highly referenced states and allow for conflicting states' cold regions to overlap in the cache. We also plan to study further the sensitivity of our policies for different inputs.

6 Acknowledgements

This work has been supported by NSF Grant CCR-9900615 and by Mercury Computer Systems, Chelmsford, MA.

References

[1] T. Ball and J. R. Larus. *Efficient path profiling.* In IEEE/ACM International Symposium on Microarchitecture (MICRO), Paris, France, Nov. 1996.

[2] D. A. Barrett and B. G. Zorn *Using lifetime predictors to improve memory allocation performance.* in Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation, pp. 187–196, June 1993.

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

[4] M. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines.* PhD Thesis, Princeton University Department of Computer Science, June 1996.

[5] T. M. Chilimbi and J. R. Larus. *Using Generational Garbage Collection To Implement Cache-Conscious Data Placement.* ACM. ISMM 1998 pp.37–48.

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

[7] R. Courts. *Improving Locality of Reference in a Garbage-Collecting Memory Management System.* CACM 31(9): 1128-1138 (1988)

[8] A. Diwan, D. Tarditi, and E. 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.

[9] R. Ghiya. *Putting Pointer Analysis to Work.* PhD Thesis, School of Computer Science, McGill University, Montreal, 1998

[10] D. Grove, J. Dean, C. Garret, C. Chambers. *Profile-Guided receiver class prediction.* In Proceedings of the 1995 Conference on Object-Oriented Programming Systems, Languages, Applications (OOPSLA), pages 108-123, Austin, TX, October 1995

[11] T. Kistler and M. Franz. *The Case for Dynamic Optimization: Improving Memory-Hierarchy Performance by Continuously Adapting the Internal Storage Layout of Heap Objects at Run-Time.* Technical Report No. 99-21, Department of Information and Computer Science, University of California, Irvine; May 1999 (revised September 1999).

[12] M. Reinhold. *Cache Performance of Garbage-Collected Programs.* In PLDI '94 Conference Proceedings, pp. 206-217, Orlando, FL, June 1994. Published as SIGPLAN Notices 29(6), June 1994.

[13] M. Seidl and B. Zorn. *Predicting References to Dynamically Allocated Objects.* Technical Report CU-CS-826-97, Department of Computer Science, University of Colorado, Boulder, January 1997.

[14] A. Sristava and A. Eustace. *ATOM: A System for building customized program analysis tools.* In Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, pp. 196-205, Orlando, FL, June 1994

[15] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. *Dynamic storage allocation: A survey and critical review.* In Proceedings of the 1995 International Workshop on Memory Management, volume 986 of Lecture Notes in Computer Science, Kinross, United Kingdom, Sept. 1995. Springer-Verlag.

[16] P. R. Wilson, M. S. Lam and T. G. Moher. *Effective "static-graph" reorganization to improve locality in garbage-collected systems.* In Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), pages 177-191, Toronto, Ontario Canada, 26-28 June 1991. SIGPLAN Notices 26(6), June 1991.