Profile-guided Tuning of Heap-based Memory Access

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 number of cache misses are caused by accesses to dynamically allocated memory and that a small number of heap objects account for a large percentage of heap misses.

In this paper we describe two methods that attempt to increase cache utility using profile-guided allocation of heap objects. In our first apprach, we have modified an existing malloc library to allocate heap objects with the aim of reducing first level data cache conflicts. Our allocation routines utilize information about the target cache architecture. We use program behaviour obtained from profiling to classify objects and allocate them to regions in the cache where they will potentially cause fewer cache conflicts. We perform our work on a Compaq Alpha 21264 processor as our target architecture.

In our second approach, we explicitly guide allocation of objects to increase spatial locality. We maintain *Temporal Relationship Graphs* (TRGs) for subsets of objects and allocate objects that are observed to have strong temporal interaction into close regions in the heap to increase spatial locality, and to different regions (i.e., lines) of the data cache. We introduce the concept of allocation *phases* to connect context of consecutive allocation of objects to the temporal relationship displayed by accesses to those objects.

To motivate this work, we provide an evaluation of the differences between heap-based and nonheap-based accesses. We show that by using a procedure-stack based predictor as input to the allocator, we can achieve speedups of up to 5%.
[1]

1 Introduction

Memory latency has become increasingly important as the gap between processor speeds and memory speeds grows. Many methods have been proposed to overcome this disparity, 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

[1]This is a revised version of a paper that was submitted to the PACT 2001 conference. In this paper we describe the use of a TRG to attempt to improve upon previous results. We also provide cache miss rates in this paper.

percentage of the memory accesses result in cache hits. As the gap between processor and memory speeds continues to grow, this percentage must also increase.

For the purposes of this paper we divide memory into two categories:

1. heap, and
2. nonheap.

Heap objects are allocated dynamically. Accesses to the heap region differ in nature from statically allocated objects (e.g., arrays). References to heap objects access nonconsecutive elements at noncontiguous memory locations. Operations such as sorting and insertion/deletion can alter the overall structure of the linked data structures at runtime. This inherent lack of spatial locality reduces the efficiency of conventional prefetching methods in link intensive applications. Temporal locality may also be lacking [9], as a traversal through a linked data structure may involve visiting enough nodes to displace a node from the cache before it is revisited (i.e., exhausting the cache's capacity).

When we look at the characteristics of memory accesses to the heap region as compared to accesses to other regions (mainly to the data segment and the stack segment), several issues stand out. First, a disproportionate number of cache misses are caused by accesses to the heap region, as seen in Table 1. For example in equake, a simulation of sysmic wave propagation from the Spec2000fp suite, around 90% of all cache misses are caused by accesses to dynamically allocated memory. However, number of accesses to the heap region only form 19% of all accesses.

Another interesting characteristic of accesses to dynamically allocated memory is that increases in cache size do not reduce collision and capacity misses as they do for statically allocated structures. In experiments involving several benchmarks suites, we found that accesses to linked data structures that miss in the cache actually make up an increasingly

Program	twolf	equake	ammp	power	tsp	em3d
Percentage of heap accesses over all accesses	13.3	19.2	21.8	0.5	11.9	43.9
Percentage of heap-based cache misses over all misses	49.6	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. This data was obtained by simulating the execution of our benchmarks on a 64K L1 environment.

disproportional percentage of the total misses with increasing cache size, as seen in Figure 1.

When we look at the source of the misses in the heap region (the blocks that are accessed when a cache miss occurs 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) might lead to problems. When multiple objects with high reference counts are mapped to addresses that conflict in the cache and the cache is not associative, a significant number of misses will occur.

In general existing allocation routines tend to balance allocation speed and memory usage. Preserving locality has not been a major concern [15]. Also, the footprint of dynamically allocated objects tends to be larger than the size of the data segment, even though accesses to the heap region are generally less than accesses to the data segment. As observed in the benchmarks listed in Table 1 (executed with mid-sized inputs), the heap footprints would not fit in even the largest L1 caches currently available. Among other reasons, allocator mechanisms and programming practices (not to mention memory leaks) significantly influence memory layout. This larger footprint leaves heap accesses more exposed to cache conflicts. The spatial locality of heap data structures thus have a great deal of room for improvement. These observations are the premise of our work.

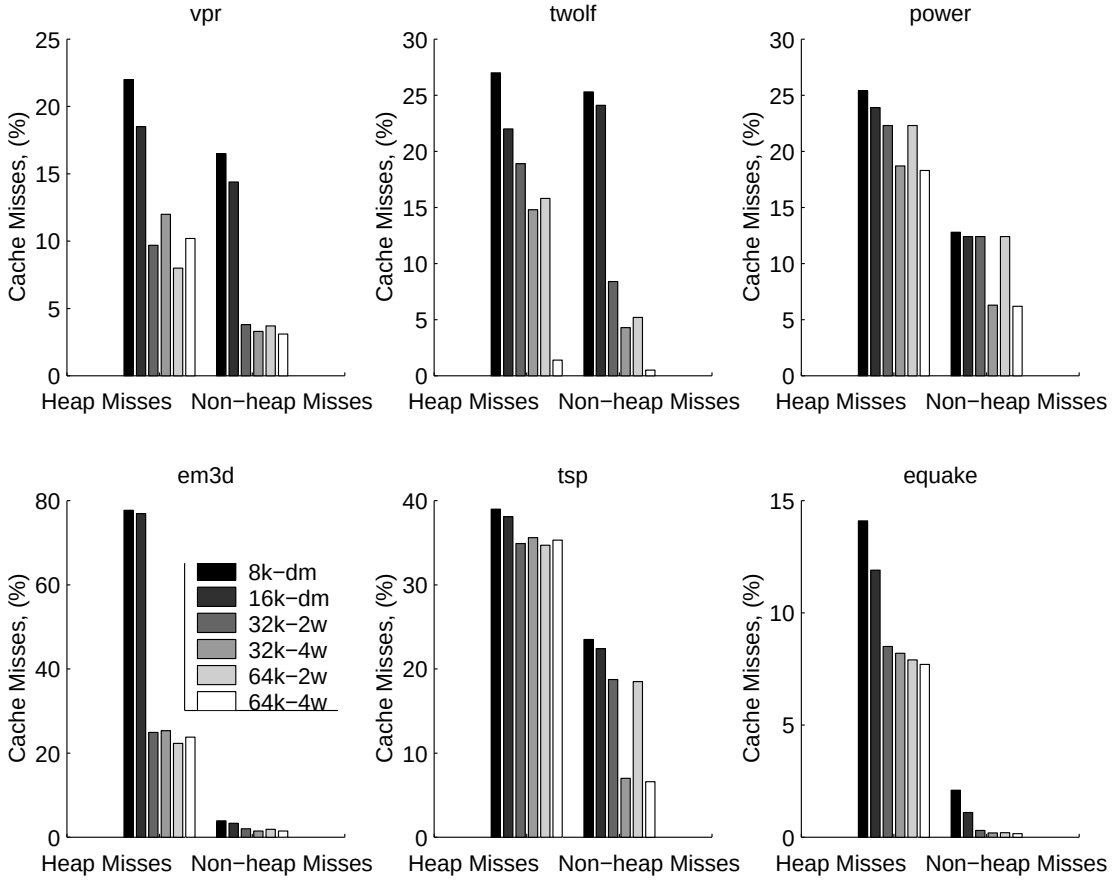

Figure 1: Cache miss rates for non-heap segment and heap segment accesses with varying cache sizes and associativities.

We have developed two different profile-guided approaches to allocating heap objects to improve heap behaviour. As previously mentioned, mapping two blocks with high reference counts to the same cache line will cause a significant number of cache misses. The objective of the first method is to identify and reduce potential cache collisions. In the second method, we attempt to develop a novel way of classifying consecutive allocations (i.e., *phases*) and use this information to determine the state of the program at both profiling and allocation state.

Both of our approaches use profile-guided optimization. Our profile-guidance is based on characteristics obtained during a profile run. All of our analysis uses different training and testing inputs. To tie the profiled state to an actual program runtime state, we develop a *state predictor*. For the purposes of identify program states associated with heap

Program	Total Allocation Bytes	Maximum Footprint Byte	Final Footprint 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
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 "Final Footprint" column refers to memory still residing in the heap at program termination.

allocations, we utilize informatoin abou the program call stack and the allocated heap object size.

This paper is organized as follows. Section 2 briefly reviews related work. Section 3 explains our state predictor and discusses how we use it to guide our cache-conscious memory allocator. Section 4 presents runtime and simulation results. Section 5 summarizes the contributions of this work and suggests directions for future work.

2 Related Work

There has been a considerable amount of research on the effects of garbage collection on cache performance and locality [7, 8, 12, 16], though we have not encountered any work that attempts to improve cache performance by reducing conflict misses between heap objects. Chilimbi and Larus [5] have studied reordering the objects themselves in memory during a garbage collection cycle.

Kistler and Franz [11] put forward a method of improving the memory-hierarchy performance at runtime by continuously adapting the internal storage layout of heap objects.

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

3 Algorithms

For the first part our experiments 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 libg++, the GNU C++ library. For the TRG-guided allocation part of our research we have written our own malloc library, one that was more suited to the specific allocation strategy we wished to pursue.

We tried to use a broad base of benchmarks with different properties. Table 3 lists these programs and their brief descriptions. All profiling was performed on Alpha 21164 workstations. We used the ATOM [14] binary instrumentation tool to instrument binaries for profile extraction. ATOM allows us to insert analysis procedures to executables to get run-time data without affecting the program's normal execution. Results were obtained by running the executables with the modified allocation libraries on Alpha 21264 workstations with a 64K 2-way set associative L1 cache, which was our target platform.

3.1 State Predictor Implementation

State predictors form the interface between the profile data and the target program execution. The effectiveness of profile-guided reordering is based on the premise that the profile is representative of the target execution. To produce a profile, a program in run with a training input set to train predictors are are used to identify similar program states in the target (same program, different input set) execution. We then detect performance prob-

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
equake	SPEC2000fp	Simulation of seismic wave propagation in large basins
ammp	SPEC2000fp	Modeling large systems of molecules

Table 3: General benchmark information.

lems (i.e., cache conflicts) that were encountered during the profile run and then anticipate (and potentially remedy) these problems when they are detected in the target execution. Detection is dependent on our ability to identify a similar program state.

We will make our decisions to reorder at heap allocation time. Among the possible sets of program information we could obtain at allocation time include the *stack pointer*, the *procedure call stack*, the *allocation size* and the *allocation call site*. We have evaluated these possible predictors used singly as well as in combination.

The value of the stack pointer at allocation time is easy to implement as a predictor. The call stack is harder to implement, but provides a great deal of information about the program state. The implementation of a low-overhead call stack-based predictor should actually be trivial; using lightweight binary instrumentation to implement such a mechanism could provide negligible overhead as the instrumentation granularity would be at procedure level. There have also been work done by Ball and Larus [1] on efficient path profiling.

The object size is also easy to implement, but used alone, provides very little information. When used together with the call stack, it can outperform most other state predictors.

We have also experimented using the*malloc* call site within a procedure as a very accurate predictor mechanism. This has proved to be costly to implement, and combined usage of the call stack and allocation size has given comparable results. In our experiments we have obtained the best results with the call stack state prodictor combined with the allocation size.

3.2 Phases of Allocation

In the TRG-guided allocation part of our work, one of our premises has been that the sequence of allocation events can be separated into distinct slices, capturing unique patterns of accesses. It follows that when one object among a sequence of consecutively allocated objects is accessed, the other objects will also be accessed (a form of *sequence locality*).

We decided to exploit this fact by identifying *phases* of allocation. We keep a FILO-buffer of predictor states, and whenever a phase is entered into the buffer (a phase is entered at each allocation) that is not currently in the buffer, we identify this as the beginning of a new phase. All the allocations are entered as *belonging* to this phase. A phase can contain a large number of alocations, but can have at most n - *1* number of distinct state types, where n is the size of the state buffer. We selected a buffer length of 10 (based on experimentation).

These phases are used extensively in the TRG-guided allocation part of our work, both in the profiling and execution stages. It is essential that we maintain a TRG among the states of each phase. We define the *signature* of a phase as the contents of the state buffer at the time the new phase is entered and use these signatures to identify multiple occurences of phases.

3.3 Profile Creation

We used the ATOM binary instrumentation tool to obtain our profiling results. We have also used ATOM to emulate run-time call stack tracking for the target execution. During the target exection we access a separate indexed em State Table, used to track and guide allocation states of the target execution.

3.3.1 Miss-Profile Creation

A *miss profile* is created during the profiling step, which will be indexed by a *State Table.* This is essentially a conflict table, holding conflict information between each predictor state for the cache under consideration. In these experiments our target architecture was the Alpha 21264, which has a 64K 2-way set associative level-1 data cache.

During the creation of this graph we also found it useful to also keep a separate TRG graph to prune the conflict table of entries that do not have a lot of temporal interaction. The probability of a cache conflict repeating itself in the target execution when the associated states do not interact significantly was found to be low.

3.3.2 TRG-Profile Creation

As previously mentioned, we keep a TRG graph for each phase during the profiling stage. We had seen that multiple occurences of the same phase during allocation time have the very similar temporal access patterns. Therefore we update the weights of the TRG edges of a phase for all instances of the phase; however, we only identify the temporal interaction between the states of a certain phase. So if states 2 and 3 of a phase exhibit high temporal interaction, we update the TRG graph. For another phase with the same signature (thus an instance of the same phase), we update the same temporal relationship graph. What we do not look for is interaction between states of two separate instances of a phase. This

happens very infrequently.

3.4 Implementation of Miss-Profile Guided Allocation

As mentioned previously, we used the dlmalloc allocation library as a basis to implement our first strategy. 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 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 free block.

If a conflict is found in the Conflict Table, the block is released 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 be in the Conflict Table (and thus posing a collision risk later on), our routine checks the following cache line to see if it is free of conflict. This is repeated until a cache line is reached which is not among the addresses in the current state's Conflict Table entry. If during this step we find that the end of the wilderness block 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 at immediately after this region. The previously allocated buffer is immediately freed, causing minimal space wastage (though possibly some limited fragmentation).

3.5 Implementation of TRG-Profile Guided Allocation

For the TRG-profile guided allocation we have written our own complete malloc library, one that was more suited to the explicit placement of selected objects. As in the Miss-Profile guidance, we maintain a table of objects, but a table which holds the states within various phases that showed strong temporal interaction and should be allocated as close as possible. We maintain a simplified version of the State Table used in the Miss-Profile part, this time holding only the allocation number of highly accessed states of selected phases.

Our allocator keeps separate bins for directed allocation, into each bin is allocated an arena where we allocate objects that we would like to be in proximity with each other. From the profiling stage we have obtained the list of states that should be allocated close together, when we come across one of these states we direct its allocation to the bin reserved for it and its related states. When the arenas in the bin are filled, we simply request more memory from the OS for the arena. Since we know beforehand the required number of such arenas, the allocation headers are allocated at compile-time and we are assured of having as many arenas as are needed.

4 Results

The results we obtained from our algorithms have left us with a good degree of confidence in our policies and mechanisms. Despite the overhead introduced to the allocation mechanisms, in each case we were able to achieve actual speedup figures. The TRG profiling method is a novel method, one that exhibits strong opportunity for improvement.

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

Program	twolf	equake	ammp	power	tsp	em3d
Speedup,Miss Profling	3.1%	5.5%	2.8%	1.9%	1.0%	0.85%
Speedup,TRG Profiling	1.0%	3.1%	2.5%	2.1%	1.3%	1.9 %

Table 4: Execution time speedups relative to the original execution. All runtimes obtained on a Compaq Alpha 21264.

Program	twolf	equake	ammp	power	tsp	em3d
Miss Rate, Original	14.6%	9.2%	18.0%	27.3%	12.6%	29.2%
Miss Rate,TRG Profiling	14.3%	7.1%	16.2%	16.2%	12.5%	27.3%

Table 5: Miss rates for a 64KB 2-way set associate data cache. Results obtained using Atom.

Table 4 show cache miss rates for the original and TRG optimized executions. Profile-guided allocation provides a significant reduction in miss rates for a number of the programs.

5 Conclusion

In this paper we have proposed and evaluated two techniques that attempt to increase cache performance through guided allocation of dynamically heap memory. The profile-guidance is dependent on our ability to map the a profiled state to the target execution state.

We attempt to identify blocks or machine states (indicators to conflict-causing blocks) by using a profile and perform compile-time optimizations using the profile output and machine cache size to direct the memory allocator. In our first method we have modified an existing malloc routine to use a profile and attempt to allocate objects a addresses where

they will cause fewer cache conflicts. In the second method we restricted the reallocation address of an object based on a TRG graph, avoiding mapping temporally local heap objects to the same portion of the cache.

Our preliminary results show we can obtain satisfactory speedup figures using efficient greedy algorithms to partition the conflict graph and perform cache-conscious memory allocation. Our results give us confidence in the validity of our policies and provide motivation to carry the work further.

Our TRG-profile based allocation strategy especially holds a great deal of room for improvement and we are confident of obtaining much better performance increases in the future. We also plan to employ more aggressive graph partitioning algorithms and attempt to implement our state level profile-guided allocation. We plan to take into account the hot and cold regions within highly referenced states and allow for conflicting states' cold regions to overlap in the cach. We also plan to look at 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, Orando, 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.