On Improving Heap Memory Layout by Dynamic Pool Allocation

Zhenjiang Wang[1][2], Chenggang Wu[1] *

[1]Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2]Graduate University of Chinese Academy of Sciences, Beijing, China
{wangzhenjiang, wucg}@ict.ac.cn

Pen-Chung Yew

Department of Computer Science and Engineering, University of Minnesota at Twin-Cities, Minnesota, USA
Institute of Information Science, Academia Sinica, Taiwan
yew@cs.umn.edu

Abstract

Dynamic memory allocation is widely used in modern programs. General-purpose heap allocators often focus more on reducing their run-time overhead and memory space utilization, but less on exploiting the characteristics of their allocated heap objects. This paper presents a lightweight dynamic optimizer, named *Dynamic Pool Allocation (DPA)*, which aims to exploit the affinity of the allocated heap objects and improve their layout at run-time. DPA uses an adaptive partial call chain with heuristics to aggregate affinitive heap objects into dedicated memory regions, called *memory pools*. We examine the factors that could affect the effectiveness of such layout. We have implemented DPA and measured its performance on several SPEC CPU 2000 and 2006 benchmarks that use extensive heap objects. Evaluations show that it could achieve an average speed up of 12.1% and 10.8% on two x86 commodity machines respectively using GCC -O3, and up to 82.2% for some benchmarks.

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Allocation/deallocation strategies

General Terms Management, Performance

Keywords pool allocation, adaptive partial call chain, data layout, dynamic optimization

1. Introduction

The huge speed gap between modern processors and their memory has long been a main barrier that limits the performance of computer systems. The trend shows no relief in the foreseeable future. A great deal of efforts have been geared toward coping with this problem. They range from architectural support to compiler optimizations. Many try to use the cache hierarchy effectively from various aspects, including affinity-aware cache data placement on CMP, improving the data locality [7] [8] [9] [6] [10] [11] [18] [3], hiding latency using prefetching [18] [13], to name just a few.

* To whom correspondence should be addressed.

CGO'10, April 24–28, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-60558-635-9/10/04. . . $10.00

Improving data locality is essential to bridge the speed gap. It could reduce cache misses and thus reduce the memory bandwidth requirement. In some programs, the heap objects occupy a large portion of their work space. As a result, it is critical to improve the locality of such data objects in the heap.

Some programming languages (e.g., Lisp, Java and C#) have an automatic memory management mechanism, such as *garbage collection* (GC). *Garbage collectors* can copy some or all of the reachable objects into a new area in memory, and update all references to those objects as needed, to improve overall data locality [5]. When using garbage collectors to manage heap objects at run-time, the layout of the objects could be improved by considering their frequent access patterns [10], affinity [6], and other run-time characteristics [16].

However, there exist many applications written in languages that do not have GC support at run-time. Moving allocated objects to improve their memory layout at run-time is a great challenge because it is difficult to update all relevant pointers due to potential aliases. Hence, once data objects are allocated, their memory layout cannot be easily changed unless they are freed or moved explicitly by the programmers. Many efforts have thus been directed toward the layout of heap objects at their *allocation time* [11] [15] [2] [7] [18].

Pool allocation is a technique that aggregates heap objects at the time of their allocation into separate memory pools in order to control their layout. Lattner et al. [11] proposed a compiler technique to improve cache locality by *pool allocation*. They identify distinct data structures such as lists, graphs, or trees in the source programs using sophisticated *static* program analysis and heuristics. Such objects are then segregated into separate pools. It could gain a significant performance improvement by reducing data cache misses.

However, in many cases, source programs are not available for such analysis and optimizations. To support data layout improvement directly on the binaries at run-time, some techniques have been proposed [15] [2] [7] [3]. They collect profiling information in the prior runs, and classify objects into categories by their characteristics such as access pattern, access frequency or the lifetime of the objects. In later runs, objects in the same category are allocated in the same pool to reduce potential cache misses. Such an approach could provide a rather good estimation of the run-time behavior using a representative training input set, and hence, could be used to improve data layout for better memory performance.

In this paper, we present a *Dynamic Pool Allocation (DPA)* scheme. It analyzes the relationship among heap objects and aggregates the affinitive ones into pools guided by adaptive call chain information - all done at the run-time. It aims to eliminate the extra profiling runs and overheads, as well as the difficulty of finding representative training input sets. Experimental results show that our approach improves the performance by an average of 12.1% and 10.8% (up to 82.2%) on two x86 machines for some SPEC CPU 2000 and 2006 benchmarks that contain extensive heap objects and their usage.

Our paper makes the following contributions:

- A transparent dynamic approach to improve the performance of pointer intensive programs by controlling the layout of allocated heap memory space. It is done without the requirement of re-compilation or carrying profiling information from prior profiling runs.
- A new strategy that uses *adaptive partial call chain (APCC)* to eliminate the effect of wrappers around default system memory allocators.
- A scheme that uses heuristics based on affinity analysis to aggregate related objects in the same memory pool even if those objects are allocated at different call sites.
- A study on the multitude of factors that can impact the performance of DPA.
- A lightweight implementation of DPA that shows improvement in memory performance and running time of heap-intensive programs.

The organization of the remaining paper is as follows: Section 2 introduces the related work; Section 3 explains our approach; Section 4 illustrates our system. Section 5 contains some evaluation. Finally, section 6 concludes the paper.

2. Related Work

Since a *garbage collector* can identify all references to all objects, it has the ability to move objects safely at run-time. Many schemes aim to improve garbage collectors through exploiting data locality in the heap [6] [10] [16]. They use online object-instance sampling techniques to discover frequent access patterns. From time to time, GC rearranges data objects according to the identified access patterns. Their results show that great performance improvement could be obtained for some benchmarks.

For applications that do not have GC support, Chilimbi et al. [9] presented a semi-automatic tool, called *ccmorph*, which could reorganize the layout of homogeneous trees at run-time. It relies on annotations provided by the programmers to identify the root of a tree and to indicate whether the layout reorganization is safe. They also describe another tool, *ccmalloc*, which is a *malloc* variant that accepts hints from the programmers to allocate one object near another for better locality. Both tools require the source code and the hints from the programmers.

Lattner et al. [11] proposed a compiler framework that segregates distinct instances of heap-based data structures into separate memory pools. It is driven by their pointer analysis algorithm, called *data structure analysis (DSA)*. The approach also needs the source code.

There exist some profile-based schemes [15] [2] [7] [3] for applications without their source code. Their first step is to collect profile information. Chilimbi et al. [7] presented a profile-based analysis for co-allocating contemporaneously accessed heap objects in the same cache block. The hot data stream, i.e., a regular data access pattern that frequently repeats, is obtained through profiling. In [15], they use heuristics to predict accessing and lifetime behavior of heap objects when they are allocated. They considered a variety of information available at the time of object allocation. It includes information from stack pointer, path pointer and stack contents. Their scheme could reduce the number of page faults. A representative training input set is necessary in these approaches. However, it may not be easy to find such "representative" training input set for some real applications.

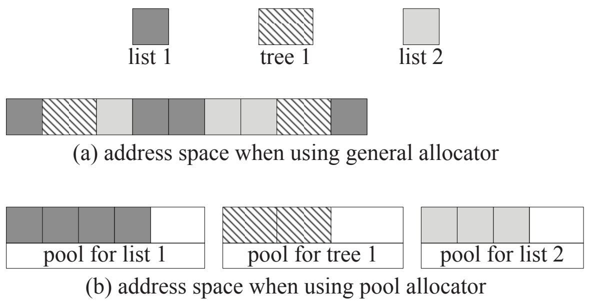

list 1
tree 1
list 2
(a) address space when using general allocator
pool for list 1
pool for tree 1
pool for list 2
(b) address space when using pool allocator

Figure 1. A typical scenario of memory space allocation

What the users prefer is a transparent mechanism that needs no extra information they have to provide, nor extra profiling work they have to do, except the application binary itself. Qin Zhao et al. [18] proposed a dynamic heap allocation scheme in DynamoRIO. They treat each static memory allocation site as a single pool, hence, the pools and the static allocation sites could be mapped one-to-one to each other. It can improve the locality for some programs, but does not work on some others as discussed in Section 3.

3. Dynamic Pool Allocation (DPA)

Most general-purpose heap allocators such as *dlmalloc* [12], a widely used memory allocator in Linux system, focus primarily on reducing the run-time overhead and enhancing the memory space utilization. An object is usually allocated in a best-fit fragment, ignoring the correlation with other objects. Figure 1 shows a scenario of heap memory layout after 3 distinct data structures are allocated.

In general-purpose allocators, only the allocation sequence and/or fragmentation status, as well as the object size are considered. Hence, objects of the same data structure could scatter around in the memory space, as Figure 1a shows. In Figure 1a, traversals of one data structure usually need to access several memory pages and several cache lines. When the working set size is large, it could cause significant TLB misses and data cache misses.

If we could design a mechanism to allocate them in 3 different pools, and form the layout as shown in Figure 1b, the locality can often be substantially improved. Component objects of each data structures are allocated in their corresponding pool next to each other, so that they are likely to reside in the same cache line and memory page. Therefore, traversals of these data structures could cause fewer cache and/or TLB misses. Moreover, hardware or software prefetching becomes easier when traversal matches the allocation sequence (the strides are regular in this case).

The key of dynamic pool allocation is deciding which pool an object should belong to. However, it is not easy to get high-level data structure information without the source code. One possible scheme is to regard all heap objects allocated at the same call site as affinitive, and put them into one object group. An object group is usually corresponding to a pool. This call-site based strategy can work well in many cases [18]. However, there exist two challenging issues that often render such schemes less effective.

The first is caused by using *wrappers* around the memory allocation routines such as *malloc*. Some programmers prefer to use *wrappers* to enhance the reliability of their programs. Figure 2 shows a typical wrapper in 300.twolf of SPEC CPU 2000.

```
char *safe_malloc(size)
unsigned size;
{
   char *p;
   extern char *malloc() ;
   if ((p = malloc(size)) == (char *) 0) {
      cleanupHandler(heap_no_mem,"safe_malloc");
   }
   return p;
}
```

Figure 2. A malloc wrapper in 300.twolf

Almost all heap objects are allocated through the wrapper routine *safe_malloc* in the program. The call sites of *malloc* inside *safe_malloc* become useless for object classification. It could trick the scheme to aggregate all heap objects into one pool. This kind of wrapper is quite common in many programs. For example, 197.vpr, 253.perlbmk and 300.twolf all have more than 80% of their heap allocation through such wrappers. What makes it worse in some cases is that some wrappers could even wrap memory allocation routines under several layers, such as those in 2.3% of heap allocations in 483.xalancbmk.

The second issue is caused by the fact that the heap objects of a data structure could be allocated through several different call sites. Using the above mentioned call-site based schemes, they will be allocated into several different pools and lose some affinity advantage. Two main reasons lead to such a phenomenon.

One is from the programmers themselves. For example, nodes of a linked list may be allocated and inserted from different code regions in a program. The other is caused by the compiler. It may cause a memory allocation call site to be duplicated several times by loop unrolling or recursive function inlining. All these duplicated call sites are from the same original call, and are supposed to build the same data structure. Again, using a call-site based strategy, objects allocated from these call sites will be put into different pools. For example, 55.4% of call sites in 197.parser are processing the same data structures with others.

A good pool allocation policy should accommodate these two challenging issues and distinguish the relationship of these heap objects correctly.

3.1 Adaptive Partial Call Chain

A call chain could be traced by the content of a dynamic call stack (stack unwinding). It starts from the current procedure, followed by its caller, caller's caller and so on. It contains useful context sensitive information. Hence, we could use it to resolve the issue caused by wrappers. An important issue here is how far back we should trace the call chain in order to identify the wrappers that have multiple layers.

One straightforward option is to use the *full call chain*. It can eliminate the problems caused by wrappers because we could use different callers of a wrapper to produce different call chains. However, using the entire call chain may make the calling context over-specialized. It could produce too many object groups with very few objects in each. This could, in turn, lead to too many pools, wasting the pool space, and making the data layout too sparse to take advantage of its spatial locality. Besides, using the entire call chain could incur large overhead at run-time, especially for those programs that have very long call chains, or functions that are recursively invoked.

Another option is to select a *partial call chain* of a fixed length *n*, called *n-PCC* in the rest of the paper. It has less overhead than in the *full call chain* if *n* could be selected appropriately. However, the optimal chain length could vary at different call sites. For example, in our study, 1-PCC is enough for most call sites in 197.parser, while some objects need 4-PCC to eliminate the impact of wrappers in 483.xalancbmk.

In our approach, we design an *Adaptive Partial Call Chain (APCC)* strategy, which uses a *variable length* that is adaptive to the calling context. We start from the allocator's direct caller, called *procedure A* here for the ease of reference. We then analyze the data flow of the allocated memory pointer. In most cases, if procedure *A* does not process the allocated object or link the allocated object with other ones, but simply passes the pointer to its caller by return value or call-by-reference parameter, we could be pretty sure that procedure *A* is a wrapper. The length of the APCC is then increased by 1. Their analysis is repeatedly applied up the call chain until a caller does not show the behavior of a wrapper. Using the APCC strategy, we could keep enough context information to eliminate the impact of wrappers. Later, we could build an object group for each APCC.

The experimental data shows that the average length of APCC is 1.44 in our evaluated benchmarks, and the maximum length is 4 (in 483.xalancbmk). In general, it only needs to analyze dozens of instructions to recognize a wrapper on average. Moreover, after an APCC is analyzed, the result can be used for later allocations without re-analysis. Therefore, the overhead is amortized by all the allocations from the same APCC. Evaluation in Section 5.5 shows that such an overhead is quite small.

3.2 Object Group Merging

As discussed earlier, the objects allocated in several different call sites could belong to the same data structure in some cases. Call-site based schemes will put them into different object groups. To merge them back into one object group, we propose to use the *Storage Shape Graph (SSG)*

3.2.1 Storage Shape Graph

Our SSG is derived from [4] [14]. It is a tuple of the form (V, H, E). V is a set of variable nodes representing the global/stack variables. H is a set of heap nodes which represent the object groups of heap objects. $E \subseteq (V \bigcup H) \times H \times O$ is a set of graph edges, each of which abstracts a set of pointers. When a pointer points from some member field in a structure or class, O is the field offset.

Figure 3a gives a code fragment that builds a linked list, whose corresponding SSG is shown in Figure 3b. The heap nodes in the graph stand for the object groups generated by our APCC scheme. The edge $(h_1, h_2, 0)$ means that the *data* field (offset is 0) of objects in h_1 is a pointer which points to the objects in h_2.

Building an SSG needs the points-to information. The variables are extracted from the binary, and a points-to analysis, like [1]and [17], can be applied.

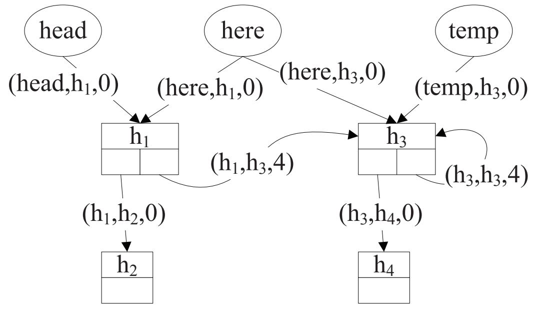
3.2.2 Affinity Recognition

We define two types of object affinity for objects of the same type. 1) Objects are of *type-I* affinity if they are linked to form a data structure, such as a list, a tree, or a graph. These objects are often referenced together when the data structure is being traversed. 2) The member fields in type-I affinitive objects can be pointers, which point to objects of another type. When traversing the data structure and accessing the fields, those objects are usually referenced together. We consider those objects *type-II* affinitive.

An object group consists of a number of objects. The pointers pointing to them could be kept in one of three different styles: (1) These objects may keep their pointer in some member field of themselves to form a data structure, which makes the objects type-I affinitive; (2) Sometimes the pointers of these objects are kept

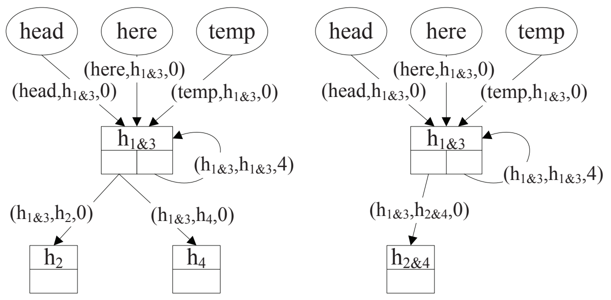
```
typedef struct {
    int *data;
    struct node *next;
} node;
...
node *head;
head = (node*) malloc(sizeof(node));  // heap node h1
head->data = (int*) malloc(sizeof(int)); // heap node h2
head->next = NULL;
…

void foo() {
    ...
    while (…) {
        node *temp;
        node *here = head;
        …                               // traverse "here" to a proper node
        temp = (node*) malloc(sizeof(node));  // heap node h3
        temp->data = (int*) malloc(sizeof(int)); // heap node h4
        temp->next = here->next;
        here->next = temp;
        ...
    }
}
```

(a) a code fragment



(b) SSG of the code fragment

Figure 3. A code fragment and its SSG



(a) h_1 and h_3 merged (b) h_2 and h_4 merged

Figure 4. An example of object group merging

in some member field of another data structure's nodes or another array's elements. In this case, the objects are of type-II affinity; (3) Infrequently, the pointers of these objects are kept in different ways, making some objects not affinitive with the others. We cannot avoid the last style even using a full call chain, but fortunately, it seldom happens. Since object groups are usually accessed using the first two styles, we consider that the objects in an object group are always affinitive.

However, not all affinitive objects are in the same object group because they might have different APCCs due to being allocated at different call sites. The purpose of object group merging is to identify these affinitive object groups and merge them for better locality. We use an SSG to realize the merge process.

In an SSG, different heap nodes whose objects are affinitive have some key attributes. If the objects in two heap nodes are type-I affinitive, the type of these objects are the same and there is an edge connecting them. If the objects are type-II affinitive, the type of these objects are also the same, but they will have edges from the same heap node or from heap nodes whose objects are type-I affinitive, and the edges have the same offset. We merge heap nodes in an SSG using these key attributes.

Take the SSG in Figure 3b as an example. Heap nodes h_1 and h_3 are of the same type, and they have an edge $(h_1, h_3, 4)$, so they are type-I affinitive and could be merged, as Figure 4a shows. Heap nodes h_2 and h_4 have the same type, and edge $(h_{1\&3}, h_2, 0)$ and $(h_{1\&3}, h_4, 0)$ have the same beginning heap node and offset, so h_2 and h_4 are type-II affinitive. The SSG after they are merged is shown in Figure 4b.

To recognize the affinity relationship between two heap nodes, we should first find out if their types are the same. However, type information is usually missing in executable binaries. In the experimental system we implemented, we assume that two objects of the same size will have the same type. Although this assumption may not be always true, it works reasonably well in most cases. Such errors will only affect the layout of heap objects. It will not affect the program correctness.

The merging is important when the affinity objects are left in too many object groups, or the access sequence jumps between the object groups frequently. The experimental data in Section 5.2 shows that 4 out of 12 benchmarks have distinct merging, and 2 of them have obvious speedup.

3.3 Allocate Memory Space for Merged Object Groups

After merging object groups, the next step is to allocate memory pools for them. Since we cannot predict how much memory space an object group needs, we have to find a way to allocate appropriate amount of memory space for its memory pool. A large memory space that is enough for any object group will be a waste of memory, because it uses the worst-case size for all memory pools. Our approach allocates memory space in units called *pool segments*. When the allocated segment of an object group fills up, another segment of the same size will be allocated. In our empirical study, we tested different pool segment sizes and found that a 4096-byte segment works the best, same as the virtual page size.

In the pool segment of an object group, we can use a general free-list based allocator with coalescing of adjacent free objects. This allocator aims to process objects of various sizes, so it uses object headers to keep the management information. However, in most cases (78.6% in our statistics), objects in an object group are of the same size. Therefore, we can use a lightweight allocator for these fixed size objects [18], and the object headers can be eliminated. A typical allocation/deallocation operation just needs to change a pointer or a free list. Both space and time can be saved. The effect of object header elimination is shown in Section 5.5.

However, allocating memory pools to all object groups may not be necessary and beneficial. Here, we consider two factors that could affect our decision to allocate a memory pool to an object group: number of objects in the object group and the size of the objects.

3.3.1 The Number of Objects

In many programs, some object groups have a very small number of objects, for example, a short linked list with only a few nodes allocated. We call them *small* object groups. Allocating memory pools to such small object groups is not very useful for improving locality. Besides, allocating memory pools to them will produce pool segments that are only sparsely populated. It will not only hurt the memory utilization but also the data locality.

To avoid this, we set a threshold on the number of objects in an object group before memory pools are allocated to them, and leave the allocation of memory space for the small object groups to the operating system.

A proper threshold is needed to filter out small object groups. A low threshold may allocate too many sparsely populated memory pools. Setting the threshold too high may lose too many optimization opportunities. Experiments in Section 5.3 show that a threshold of 100 is suitable for most programs on the systems we tested.

3.3.2 Object Size Threshold

The main benefit of pool allocation is from the improvement of locality. However, a large object can occupy several cache lines, which makes the same field of adjacent objects in different cache lines. A large object also makes the object header insignificant. As a result, the larger an object is, the less benefit we will gain from allocating to memory pools. Besides, for a pool segment of size s and objects of the fixed size n, since s may not be a multiple of n, a remainder of $[0, n)$ bytes may be wasted. Therefore, a larger object size may waste more memory. Our experiments in Section 5.4 show that the threshold of 128 bytes is suitable.

4. Implementation

We implemented our approach as a dynamic optimizer on Linux operating systems (IA32 ISA), named *DigitalBridge-dopt* . In this section, we present our system in detail.

The first step is to take over the memory allocation functions: the default system allocators need to be replaced with our DPA allocator at the beginning of the program execution. Several approaches are possible for this purpose. In our current implementation, we use the LD_PRELOAD environment variable [13] to intercept the main function, and modify the global offset table entries of the allocators (including *new* and *delete*). After that, the main function is resumed and the execution starts. In this way, when the program executes a memory allocation call, it will call DPA allocator instead.

The DPA allocator has the same interface as the system allocator: the request allocation size is passed to the allocator, and then the allocator returns a free memory object to the program. What DPA allocator controls is where (from which pool segment) to allocate the requested memory. As a result, DPA allocator does not affect the correctness of the program.

Figure 5a shows the basic implementation of our DPA allocator. When the DPA allocator receives an allocation request, the first step is to determine the APCC by the *APCC Generator* as described in Section 3.1. After that, the APCC is used to look up a hash table, the *APCC-OG Table*. Each entry of the table contains the analyzed APCC and its associated object group ID. If the APCC appears for the first time, it has no entry in the table. We then use the *Affinity Analyzer* to add a new heap node in the SSG and try to merge it with other heap nodes by the heuristics described in Section 3.2.2.

If the heap node is successfully merged with another heap node whose object group ID is g, a new entry <APCC, g> is inserted in the *APCC-OG Table*. If the heap node cannot be merged with other heap nodes, we build a new object group with ID *g'* and insert <APCC, *g'*> into the table. The object group ID and the request

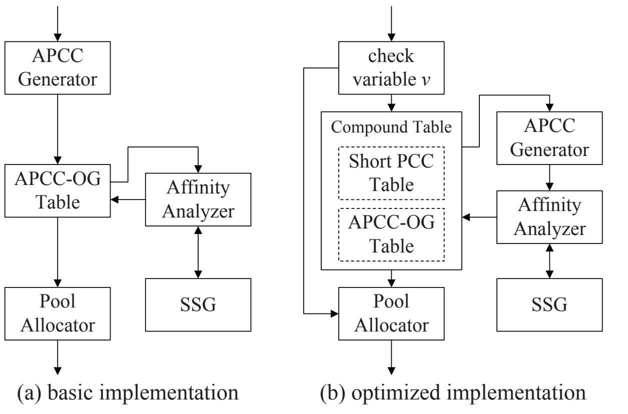


Figure 5. Structure of DPA allocator

are sent to the *Pool Allocator*, which allocates memory space as described in Section 3.3.

Note that this basic implementation has to analyze the length of APCC by the *APCC Generator* and look up the *APCC-OG Table* for *every* memory allocation request. It could incur a large overhead when the requests for allocation are frequent. Hence, we optimize it as shown in Figure 5b. The two optimizations are on the critical path to reduce the overhead.

(1) To reduce the overhead of getting APCC for each incoming request, we add an *Short PCC Table*. After *APCC Generator* gets an APCC, it can also get its corresponding shorter n-PCCs. These shorter n-PCCs are recorded in the *Short PCC Table*. When a request comes, we first look up the *Short PCC Table* with its 1-PCC. If the 1-PCC matches an entry in the table, we look up the table again with its 2-PCC. The looking up repeats until no entry is matched. In this way, we can get a PCC and check whether it is in the *APCC-OG Table*. If it does not hit, it means a new APCC appears. The *APCC Generator* is invoked to obtain the new APCC and update the *Short PCC Table*.

For incoming requests, the *Short PCC Table* can get the correct APCC so long as the APCC has been analyzed because all of its corresponding shorter n-PCCs are in the table. The overhead of looking up the *Short PCC Table* is much smaller than the overhead of the analysis using *APCC Generator*. Finally, since there is no identical n-PCC in the two tables, we combine them together as the *Compound Table*.

(2) To reduce the overhead of looking up the *Compound Table*, we instrument some code at the outermost call site of an APCC to provide its associated object group ID directly. We do this only when the next allocation (after executing the instrumented code) has surely the very APCC we want. For example, an APCC of length 2 has an outermost call site *S*. The call site *S* is not in a wrapper function, but it calls a wrapper *safe_malloc*. If the call at *S* is not an indirect call, and *safe_malloc* does not call any other functions before it calls the allocator, and the call to the allocator in *safe_malloc* cannot be skipped by the control flow transfer, we can conclude that when *safe_malloc* is called from *S*, the next allocation must have the very APCC. Hence, we instrument at the call site *S* to provide the associated object group ID by a global variable *v*. The DPA allocator first checks whether *v* is a valid object group ID. If so, the DPA allocator calls the *Pool Allocator* with the ID and reset *v* to an invalid ID. This can avoid looking up the *APCC-OG Table* in many cases.

	#1	#2
CPU Family	Intel Northwood	Intel Harpertown
Cores	1	4
Frequency	2.40GHz	2.33GHz
L1I Cache Size	32kB	32kB
L1D Cache Size	32kB	32kB
L2 Cache Size	512kB	6144kB
Memory Size	2GB	16GB
OS	Linux 2.6.27	Linux 2.6.26

Table 1. Experiment platforms

Name	Lang	Description
175.vpr	C	FPGA circuit placement and routing
197.parser	C	word Processing
253.perlbmk	C	PERL programming language
300.twolf	C	place and route simulator
197.art	C	image recognition / neural networks
183.equake	C	seismic wave propagation simulation
188.ammp	C	computational chemistry
473.astar	C++	path-finding algorithms
483.xalancbmk	C++	XML processing
447.dealII	C++	finite element analysis
453.povray	C++	image ray-tracing
482.sphinx3	C	speech recognition

Table 2. Evaluated benchmarks

5. Evaluation

In this section, we evaluate different strategies and factors discussed in Section 3. We evaluate our system on two different IA32 architectures. The hardware and operating system information are shown in Table 1. We elide some data on #2 when they are similar to those on #1. The C and C++ library on the system implements *malloc/free* using a modified *Lea allocator* [12], which is a high quality general-purpose allocator. All the runtimes shown in this section are the average time of three executions of the program.

The benchmarks we use are SPEC CPU 2000 and SPEC CPU 2006. They are compiled by GCC 4.3.2 at -O3 and with the reference input set. We select the benchmarks shown in Table 2 by the policy that their objects allocated in memory pools occupy more than 1% of all heap data in use on average. The runtimes of the other benchmarks are not affected by our optimizer. As an experiment, we replace the customized allocator in 197.parser with direct calls to *malloc/free*, because its customized allocator has semantics identical to *malloc/free*.

Our system uses APCC with object group merging. The pool segment size, object number threshold, and object size threshold are 4096 bytes, 100 and 128 bytes respectively. When analyzing the impact of one factor, the others are fixed at the above values.

5.1 Adaptive Partial Call Chain

DPA uses the Adaptive Partial Call Chain (APCC) strategy to build the object groups before object group merging. Alternative strategies include the fixed length partial call chain (n-PCC) and full call chain (FCC).

Figure 6 shows the normalized runtime of the benchmarks when using different strategy to generate object groups. The baseline is the runtime without pool allocation. The bars illustrate the performance of 1-PCC, 2-PCC, 3-PCC, 4-PCC, FCC and our APCC, respectively. The figure shows the trend that the average performance decreases when the length of PCC increases. The reason is that the identification of the longer call chain incurs more overhead. The trend is especially obvious for 197.parser because it has quite a few allocations (nearly one billion), and its call chain is often very long. n-PCC (n>1) is usually worse than 1-PCC, but it outperforms 1-PCC in 300.twolf and 483.xalancbmk when wrappers are used. Our APCC strategy has the best average performance (3% to 13% better than others) because it is adaptive to different call sites.

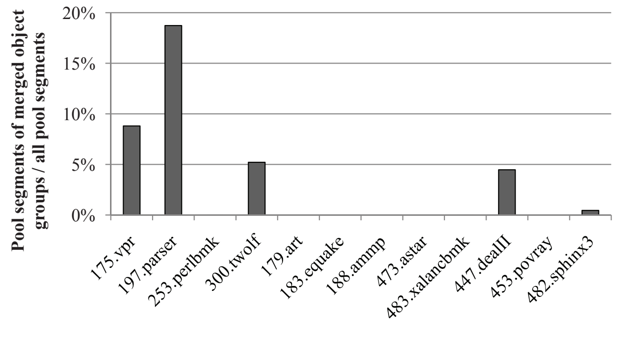


Figure 7. Pool segments of merged object groups

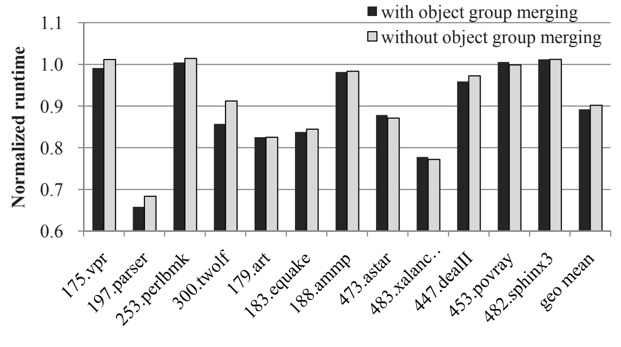


Figure 8. Normalized runtime with/without object group merging

5.2 Object Group Merging

The purpose of object group merging is to aggregate affinitive objects into one object group if they have different APCCs. The pool segments of merged object groups are shown in Figure 7. The y-axis shows the percentage of these pool segments from all the pool segments. Four benchmarks are affected by the object group merging. Figure 8 shows the runtime of our approach (normalized to the runtime without pool allocation) with and without object group merging.

Two out of the four benchmarks (197.parser and 300.twolf) have a modest 3% and 6% improvement in their runtime. 175.vpr and 447.dealII have no obvious improvement because their merged object groups are used for looking up hash tables and searching in large red-black trees. The frequent access behavior is to visit some selective objects, but not traversing the entire data structures. The average runtime with object group merging has a modest 1% improvement.

5.3 Number of Objects in Object Groups

As discussed in Section 3.3.1, we set a threshold to filter out small object groups that cannot make full use of the pool segments. Figure 9 shows the impact of such filtering. The baseline is the runtime without pool allocation. When a larger threshold is selected, fewer object groups are allowed to have their pool segments. It can make the objects in the pools denser, but may lose some po-

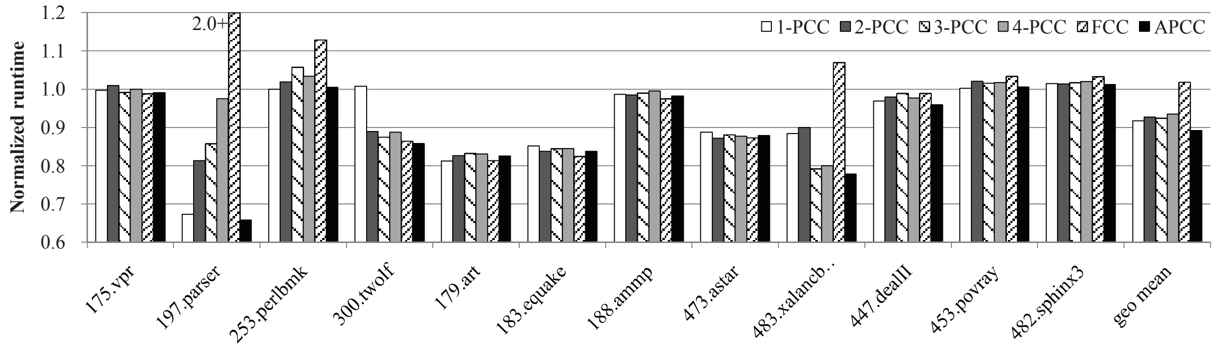


Figure 6. Normalized runtime for different strategies

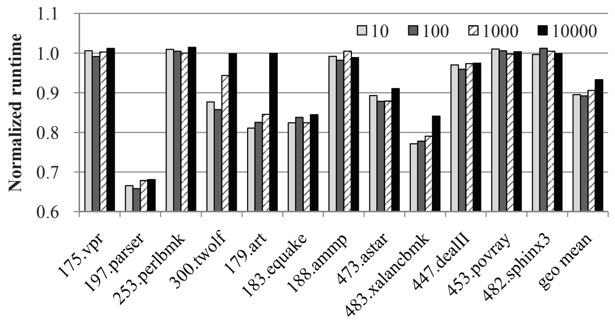


Figure 9. Normalized runtime for different object number threshold

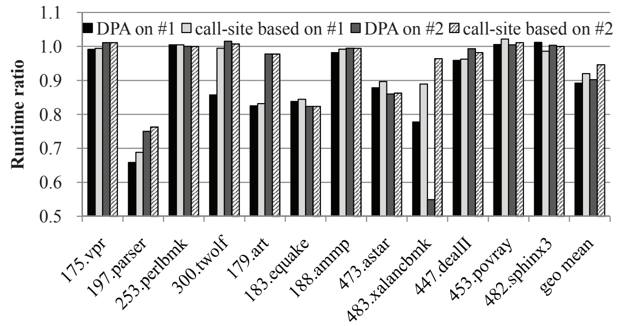


Figure 11. The runtime ratio of DPA and the call-site based approach

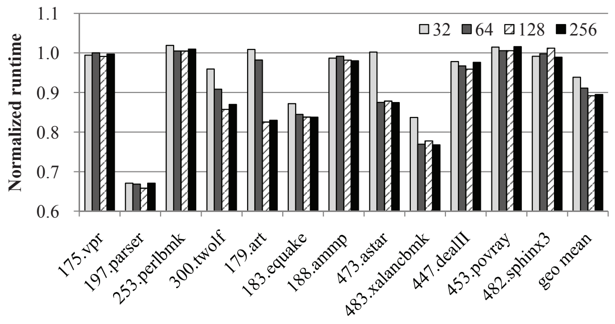


Figure 10. Normalized runtime for different maximum object size

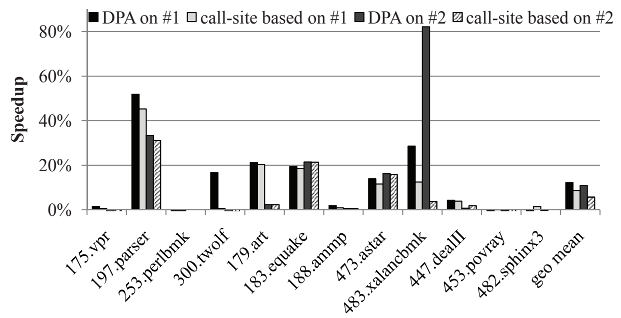


Figure 12. The speedup of DPA and the call-site based approach

tential improvement. The loss is especially obvious for 300.twolf and 179.art, because their critical object groups have only several thousand objects. In our evaluated benchmarks, a threshold of 100 makes a good trade-off and shows the best performance for almost all the benchmarks.

5.4 Object Size

A large object cannot benefit much from improved pool allocation, but may waste some space in a pool segment due to internal fragmentation. We set an object size threshold that prevents the pool allocation for large objects. As a contrast, a small threshold is too conservative and may exclude critical object groups. The impact of different thresholds is shown in Figure 10. The baseline is the runtime without pool allocation. We can see that the threshold of 32 or 64 bytes works poorly for 300.twolf, 179.art, 473.astar and 483.xalancbmk. The threshold of 128 bytes is adequate and a larger one does not help further.

5.5 Overall Performance

Now, we have explored the design space of our DPA allocator. The overall performance on the two platforms is shown in Figure 11 and Figure 12, compared with the call-site based strategy (i.e., 1-PCC without object group merging). The baseline is the runtime without pool allocation. Our approach can accelerate the benchmarks by an average of 12.1% and 10.8% on the two platforms. 483.xalancbmk shows significant speedup, 28.6% on #1 and 82.2% on #2. The reason is that several object groups in it have 188,718 small objects each, which need only 3Mb-6Mb space when using DPA allocator, but the objects are scattered in a 200M memory space when using the system allocator. For 197.parser, our approach is 1% faster than

Name	Base	DPA	Ratio
175.vpr	1370M	1365M	100%
197.parser	765M	437M	57%
253.perlbmk	139M	142M	102%
300.twolf	2304M	1863M	81%
197.art	14837M	13677M	92%
183.equake	996M	472M	47%
188.ammp	2847M	2830M	99%
473.astar	8480M	6660M	79%
483.xalancbmk	4992M	2846M	57%
447.dealII	5137M	4718M	92%
453.povray	8M	14M	169%
482.sphinx3	12623M	12840M	102%

Table 3. Cache misses on #1

	L2D misses			TLB misses		
Name	Base	DPA	Ratio	Base	DPA	Ratio
175.vpr	145M	147M	101%	1009M	998M	99%
197.parser	11M	5M	43%	1376M	1017M	74%
253.perlbmk	15M	16M	106%	162M	164M	101%
300.twolf	11K	9K	86%	815M	759M	93%
197.art	77K	15K	20%	508M	245M	48%
183.equake	188M	116M	62%	50M	39M	78%
188.ammp	56M	52M	94%	849M	846M	100%
473.astar	2584M	996M	39%	8994M	5895M	66%
483.xalancbmk	1671M	549M	33%	16920M	1472M	9%
447.dealII	770M	734M	95%	1253M	1275M	102%
453.povray	82K	56K	68%	1041M	1049M	101%
482.sphinx3	1673M	1658M	99%	2020M	2022M	100%

Table 4. Cache and TLB misses on #2

its custom allocators. Some benchmarks have a slight slow down because their benefit is less than the run-time overhead incurred.

Compared to the call-site based strategy, our approach outperforms on three benchmarks on #1 (6.6% for 197.parser, 16.1% for 300.twolf, and 16.1% for 483.xalancbmk). 100% of the memory allocation request in 300.twolf and 6.5% in 483.xalancbmk have wrappers, and APCC strategy can handle them properly. 197.parser and 300.twolf can gain benefit from object group merging, as Section 5.2 shows. 197.parser and 483.xalancbmk have similar outperform on #2. 300.twolf has no improvement on #2 because it has few cache misses (about 11K) before applying DPA, as we will show soon.

Table 3 shows the number of cache misses on platform #1 with and without applying DPA. DPA can reduce the cache misses by 15% on average. 453.povray gets more cache misses with the DPA allocator, but the total amount is quite small compared to the other benchmarks. Benchmarks with distinct cache miss reduction all have obvious speedup. We believe that the TLB misses are also reduced, but we cannot get the data because of the lack of hardware support.

Fortunately, platform #2 has more detailed performance monitors, and Table 4 shows the cache and TLB miss number on it. Due to the larger cache, the benchmarks have less cache miss number than that on #1. DPA reduces the cache misses by 37%, and TLB misses by 30% on average. The great miss reduction of 483.xalancbmk (67% and 91% respectively) leads to its significant speedup. 300.twolf and 179.art have no improvement on this machine because they have few cache misses before applying DPA.

Figure 13 shows the run-time overhead (compared to the base runtime) of our system. We evaluated the overhead by forwarding all allocation requests to the default system allocator when the objects are about to be allocated from pools. In this case the memory layout is similar to the base. The overhead includes all in our system except the pool allocator (in fact, our pool allocator has a shorter execution time than the system allocator). The graph shows that our DPA allocator has negligible overhead (0.7% on average) for most benchmarks.

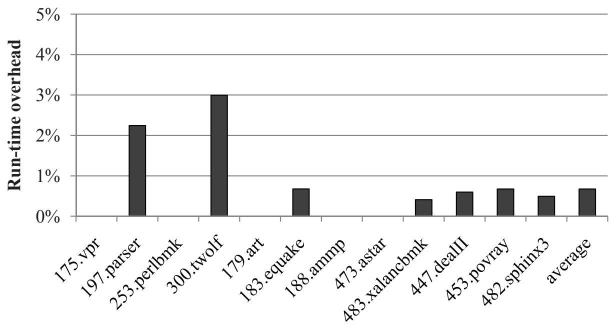


Figure 13. Run-time overhead of DPA

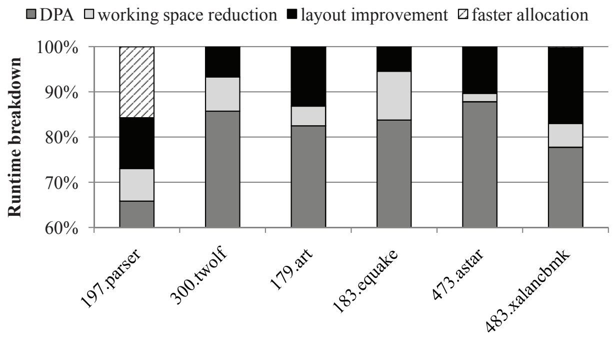


Figure 14. Breakdown of benefit

Figure 14 illustrates the breakdown of the benefit gained in our system for the benchmarks that have obvious speedup. We separate the total benefit into three parts: the faster allocation/deallocation, the rearrangement of memory layout, and the reduction of working space. To measure the benefit of the first two parts, we use modified pool segments which always allocate objects with their object headers. We can get the impact from only the two parts by comparing this runtime with the base runtime. The incremental impact of working space reduction is the difference between this runtime and that of standard DPA. It is difficult to measure the benefit of faster allocation/deallocation alone, so we estimate it by small test cases. The test cases have the same number of allocation/deallocation in pools as the benchmarks, but have no data accesses to the allocated objects. This part of benefit is obvious for 197.parser because it has quite a lot of allocations. Most benchmarks (197.parser 179.art 473.astar, and 483.xalancbmk) benefit more from the improvement of layout than the working space reduction, but some benchmarks show the opposite is true (183.equake).

6. Conclusions

Accessing heap objects occupies a large portion of the workloads in many programs. It is critical to improve the locality of data in heap. An important issue is to control the heap data layout on executable binary transparently for a better performance. Existing techniques either need source code, or prior execution to collect profiling information.

In this paper, we studied the two important issues in dynamic pool allocation: widely used wrappers can group unrelated objects together, while different call sites can separate related objects into different pools. We have also studied the affinity of objects and its key attribute in the storage shape graph (SSG).

We also proposed an approach to control the layout of heap data dynamically. It eliminates the effect of wrappers using an adaptive partial call chain (APCC) strategy, and merges object groups using their key attributes in SSG. In order to reduce the waste of pool space, it uses a set of proper thresholds to filter out object groups that have a small object number or have a large object size. It also compresses the objects which have fixed size in the pool segment.

We also did a lightweight implementation of our approach, and optimized it to get the APCC and object group ID of an object with little overhead. Our approach gets a speedup of 12.1% and 10.8% on average on two commodity machines, and up to 82.2% for some benchmarks.

Acknowledgments

This paper is supported in part by the National Natural Science Foundation of China (NSFC) under the grant 60736012, the Innovation Research Group of NSFC (60921002), the National Basic Research Program of China (2005CB321602), the National High Technology Research and Development Program of China (2007AA01Z110), the National Science and Technology Major Project of China (2009ZX01036-001-002), and by the U.S. National Science Foundation under the grant CNS-0834599 and a gift grant from Intel.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[2] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Conference on Programming Language Design and Implementation*, pages 187–196, 1993.

[3] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Architectural Support for Programming Languages and Operating Systems*, pages 139–149, 1998.

[4] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[5] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[6] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *International Symposium on Memory Management*, pages 37–48, 1998.

[7] T. M. Chilimbi and R. Shaham. Cache-conscious coallocation of hot data streams. In *Conference on Programming Language Design and Implementation*, pages 252–262, 2006.

[8] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Conference on Programming Language Design and Implementation*, pages 13–24, 1999.

[9] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[10] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage, improving program locality. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 69–80, 2004.

[11] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Conference on Programming Language Design and Implementation*, pages 129–142, 2005.

[12] D. Lea. A memory allocator. *The C++ Report*, 1989.

[13] J. Lu, A. Das, and W. Hsu. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *International Symposium on Microarchitecture*, 2005.

[14] M. Marron, D. Kapur, and M. Hermenegildo. Identification of logically related heap regions. In *International Symposium on Memory Management*, pages 89–98, 2009.

[15] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1998.

[16] M. J. Serrano and X. Zhuang. Placement optimization using data context collected during garbage collection. In *International Symposium on Memory Management*, pages 69–78, 2009.

[17] B. Steensgaard. Points-to analysis in almost linear time. In *Annual Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[18] Q. Zhao, R. Rabbah, and W. Wong. Dynamic memory optimization using pool allocation and prefetching. *ACM SIGARCH Computer Architecture News*, 33(5):27–32, 2005.