PreFix: Optimizing the Performance of Heap-Intensive Applications

Chaitanya Mamatha Ananda
University of California
Riverside, USA
cmama002@ucr.edu

Rajiv Gupta
University of California
Riverside, USA
rajivg@ucr.edu

Sriraman Tallam
Google
Mountain View, USA
tmsriram@google.com

Han Shen
Google
Mountain View, USA
shenhan@google.com

Xinliang David Li
Google
Mountain View, USA
davidxl@google.com

Abstract

Analyses of heap-intensive applications show that a small fraction of heap objects account for the majority of heap accesses and data cache misses. Prior works like HDS and HALO have shown that allocating hot objects in separate memory regions can improve spatial locality leading to better application performance. However, these techniques are constrained in two primary ways, limiting their gains. First, these techniques have *Imperfect Separation*, polluting the hot memory region with several cold objects. Second, *reordering* of objects across allocations is not possible as the original object allocation order is preserved. This paper presents a novel technique that achieves near perfect separation of hot objects via a new context mechanism that efficiently identifies hot objects with high precision. This technique, named **PreFix**, is based upon **Pre**allocating memory for a **Fix**ed small number of hot objects. The program, guided by profiles, is instrumented to compute context information derived from dynamic object identifiers, that precisely identifies hot object allocations that are then placed at predetermined locations in the preallocated memory. The preallocated memory region for hot objects provides the flexibility to reorder objects across allocations and allows colocation of objects that are part of a hot data stream (HDS), improving spatial locality. The runtime overhead of identifying hot objects is not significant as this optimization is only focused on a small number of static hot allocation sites and dynamic hot objects. While there is an increase in the program's memory foot-print, it is manageable and can be controlled by limiting the size of the preallocated memory. In addition, **PreFix** incorporates an object recycling optimization that reuses the same preallocated space to store different objects whose lifetimes are not expected to overlap. Our experiments with **13** heap-intensive applications yield reductions in execution times ranging from **2.77%** to **74%**. On average **PreFix** reduces execution time by **21.7%** compared to **7.3%** by HDS and **14%** by HALO. This is due to **PreFix**'s precision in hot object identification, hot object colocation, and low runtime overhead.

CCS Concepts: • **Software and its engineering → Software notations and tools**; **Compilers**; **Runtime environments**;

Keywords: Hot heap objects, Hot data streams, Object colocation, Cache locality

ACM Reference Format:
Chaitanya Mamatha Ananda, Rajiv Gupta, Sriraman Tallam, Han Shen, and Xinliang David Li. 2025. PreFix: Optimizing the Performance of Heap-Intensive Applications. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3696443.3708960

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708960

1 Introduction

Heap-intensive applications allocate large number of heap objects whose allocation order frequently differs from their access order. To achieve spatial locality, prior works have developed techniques that separate hot objects from other (colder) objects by allocating them in separate memory regions. While this has helped with locality, it is still limited by the fact that objects across allocations are not reordered and appear in the same order as they are allocated. In [8], a *hot data stream* (HDS) is defined as a set of hot objects that are accessed together and allocated in a separate memory region, increasing the likelihood of objects in a HDS set to be colocated, yielding improved inter-object spatial locality. In HALO [21], allocation site instances are disambiguated based on calling contexts and grouped based on access affinity. *Allocations from the same group* are then placed into a distinct memory pool to improve spatial locality of grouped objects. However, these techniques have following limitations:

- *Imperfect Separation.* Both HDS [8] and HALO [21] are imperfect in separating objects in each HDS set and HALO allocation group from all other objects respectively. Object signatures used are not unique and hence spurious objects, potentially a large number of them, are directed to memory regions dedicated to HDS or HALO allocation groups.
- *Inability to Reorder Objects.* The objects belonging to a single HDS or a HALO allocation group are placed in the memory region in the same order as they were allocated. That is, their order is not rearranged to exactly match the desired layout and further improve spatial locality.

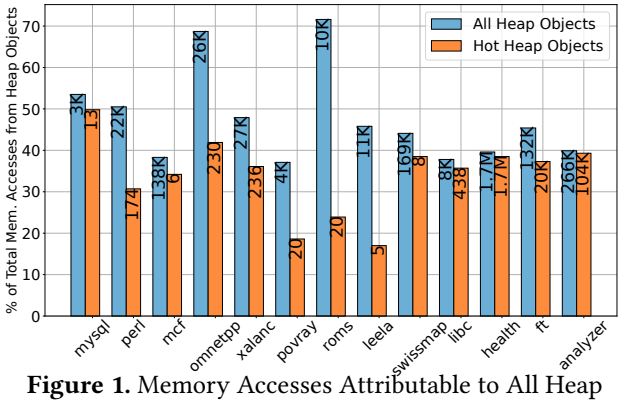

Figure 1. Memory Accesses Attributable to All Heap Objects vs. Hot Heap Objects in Profiling Runs.

In this paper, we present **PreFix** that overcomes both these limitations and substantially improves performance. The **PreFix** technique **Pre**allocates memory for a **Fix**ed small number of hot objects. By using object ids that are unique, **PreFix** achieves ***perfect separation*** of hot objects from others and hence the preallocated memory region is not polluted by additional spurious objects. The program is instrumented to compute context information in the form of unique dynamic object instances, which is utilized to identify hot objects and place them in predetermined locations. Within the preallocated memory region the objects can be placed in any order; that is, objects can be ***reordered*** to colocate objects belonging to HDSs or HALO allocation groups within the preallocated memory region. Via reordering, we can maximally exploit spatial locality among the objects. In summary, **PreFix** overcomes the above two limitations present in both HDS [8] and HALO [21].

The increase in the program's peak memory demand due to preallocation is small and controllable as only a chosen small number of objects are targeted. In addition, **PreFix** incorporates an ***object recycling*** mechanism that reuses the same preallocated space to store different objects whose lifetimes are not expected to overlap. The data in Figure 1 shows the percentage of memory accesses from all heap objects and hot heap objects with the absolute number of dynamic objects numbered within each bar. The figure shows that only 5 to 438 dynamic objects need to be preallocated for 10 of 13 benchmarks to account for most of the heap object memory accesses even though the programs allocate thousands of heap objects across all benchmarks.

Our experiments with 13 heap-intensive applications yields reductions in execution times ranging from **2.77%** to **74%**. On average, **PreFix** reduces execution time by **21.7%** compared to **7.3%** by HDS and **14%** by HALO. This is due to **PreFix**'s precision in hot object identification, hot object colocation, and low runtime overhead.

The key contributions of **PreFix** as can be observed from the comparison summary in Table 1 are as follows:

- The column Hot Object Detection explains that only **PreFix** sends fixed predetermined number of hot objects to the separate (preallocated) memory region while other techniques send an unknown often very large number of objects;
- The column Layout Optimization explains that only **PreFix** can perform object reordering within the (preallocated) memory region for hot objects. In other techniques, the hot object layout is in the order in which those objects are allocated;
- The column Runtime Overhead explains that **PreFix** is efficient due to its use of limited lightweight instrumentation;
- Finally, our evaluation shows that **PreFix** delivers significant reductions in runtimes across 13 benchmarks and substantially outperforms HDS [8] and HALO [21].

The remainder of this paper is organized as follows. In Section 2 we present our approach along with details on how context is defined to generate unique ids for detecting hot objects, performing object reordering for spatial locality, and performing object recycling. In Section 3 presents experimental results. Sections 4 and 5 present the related work and our conclusions.

2 Our Approach: PreFix

In designing **PreFix** we set out to overcome the two major limitations of prior techniques, that is, imperfect separation leading to *pollution*[1] of memory regions meant to hold hot objects and inability to reorder hot objects within the separated memory region limiting inter-object spatial locality.

Like other techniques, using program profiling and tracing, **PreFix** first selects a fixed number of hot objects that will be exploited. A program optimized via **PreFix** **Pre**allocates a separate memory region to hold the chosen **Fix**ed number

[1]Pollution is defined as the number of irrelevant objects in the hot memory region.

Table 1. Comparison of **PreFix** with Prior Works: **HDS** and **HALO**.

Technique	Hot Object Detection	Layout Optimization	Runtime Overhead
HDS [8]: Hot Data Streams	**Signatures are Static Ids** of malloc sites allocating HDS objects (profile-guided). **Number of objects captured** is not fixed and can grow very large.	**Significant Pollution** from chosen malloc sites also allocating non-HDS objects. **Layout** of captured objects is in allocation order (i.e., no object reordering).	**Hot Object Check.** No checks and no overhead. **Hot Object RAM Management.** malloc / free overhead similar to other heap objects.
HALO [21]: Heap Layout Optimization	**Call Stack Signatures** of hot object allocations (profile-guided) used to detect hot objects at runtime. **Number of objects captured** is not fixed and can grow very large.	**Significant Pollution** from irrelevant objects having the same call stack signatures. **Layout** of captured objects is in allocation order (i.e., no object reordering).	**Hot Object Check.** Get the call stack of the malloc instance and check against a signature. **Hot Object RAM Management.** Reserved regions, grown on demand. Managed chunked deallocation of reserved region.
PreFix: Our Approach	**Signatures are Dynamic Allocation Instances** (identifiers) of selected malloc sites allocating hot objects (profile-guided). **Number of objects captured** is fixed as objects ids are unique.	**No Pollution** as specific hot object instances are mapped to a preallocated region. **Layout** of captured objects **controlled** via **pre-determined object placement**. HDS[8] and HALO[21] cannot reorder objects.	**Hot Object Check.** Dynamic allocation instances matched to predetermined hot instance ids. **Hot Object RAM Management.** Space for a fixed number of hot objects is preallocated once and freed at the end, minimal overhead.

of objects at the beginning and then uses this region to avoid pollution and perform object reordering as follows.

- *Layout Determination With Object Reordering* (§ 2.1) A new data layout algorithm is presented that determines where each hot object will be located within the preallocated memory. With no restriction on the order in which the objects appear; thus, inter-object spatial locality can be exploited.
- *Context Determination and Lightweight Instrumentation* (§ 2.2) We employ a strategy that generates unique ids for objects allocated by relevant malloc sites and then uses them to identify chosen hot objects and locates them in predetermined position in the preallocated memory region. Since ids are unique, pollution is avoided.
- *Object Recycling* (§ 2.4) We have observed that, in some applications, though a large number of objects is allocated at runtime, only a small number of them are simultaneously live. Therefore, we preallocate a memory region for only a small fixed number of objects and recycle the memory among the very large number to reduce memory footprint and memory allocation/deallocation overhead.

Thus, the combination of preallocation of a fixed size memory region and unique object ids avoids pollution and permits reordering. Finally, object recycling limits memory usage. All of the above require lightweight instrumentation and involve safe code transformations.

2.1 Layout Determination Via Object Reordering

The program is profiled and memory access traces[2] are obtained which point to the interesting malloc sites where hot objects are allocated. Further, groups of objects that represent hot data streams (HDS [8]) are identified from these traces. This is then used to determine the size of the preallocated memory and the order in which the hot objects must be placed in this region to maximize spatial locality.

Our layout determination algorithm takes as its input an ordered list of HDS (OHDS), sorted in the descending order of memory accesses based on profiles. The algorithm outputs an ordered list of Reconstituted HDS (RHDS) and the objects will be placed in the preallocated memory region in the same order as they appear in RHDSs. Any hot objects that are not part of any reconstituted HDS will be placed at the end of the preallocated region.

It is important to note that all OHDS are not exploitable because same object may appear in multiple HDS in OHDS. However, RHDS are constructed such that all HDS in it are exploitable, that is, no object appears in more than one HDS

[2]profiling & tracing used interchangeably implies memory tracing

Algorithm 1 Reconstituting HDSs for Layout Optimization

Input: OHDS - Set of all observed HDSs in descending order of memory references.
Output: RHDS - Reconstituted HDSs for optimized layout.

```
Globals
    current : Current HDS set processed
    remaining : Set of Objects
    Objects() : Returns objects in a HDS or set of HDS.
    Make_HDS() : Constructs an HDS from an object set.
end Globals

for ohds ∈ OHDS do
    merged(ohds) ← false                          ▷ Initialization
end for
RHDS ← Next(OHDS)                                 ▷ Initialization

while OHDS ≠ ∅ do                                 ▷ Iterate over all HDS
    current ← Next(OHDS)
    remaining ← Objects(current) − Objects(RHDS)
    if remaining == ∅ then
        continue                                  ▷ Nothing to do!
    end if

    if Objects(current) ∩ Objects(RHDS) == ∅ then
        RHDS ← RHDS ∪ current                     ▷ Merge current
        continue
    end if

    done ← False                                  ▷ |remaining| > 0
    for rhds ∈ RHDS do
        exists ← Objects(current) ∩ Objects(rhds)
        if !merged(rhds) && exists ≠ ∅ then
            merged(rhds) ← True
            rhds ← rhds ∪ remaining
            done ← True
            break
        end if
    end for

    if !done && |remaining| > 1 then
        RHDS ← RHDS ∪ Make_HDS(remaining)
    end if
end while
```

belonging to RHDSs. Next we describe how RHDS are constructed – presented in Algorithm 1.

This algorithm always adds the first HDS in the set of OHDS, the one with the most memory accesses, to the RHDS set. Then, for every other HDS in OHDS, the algorithm takes one of the following actions to determine if it can be included either fully or partially in the RHDS set:

- *Unchanged Inclusion*: The HDS is added to RHDS in its original form if there are no shared objects with the current set of RHDS;
- *Merging*: The given HDS may be completely merged with another existing HDS in RHDS with common objects; or
- *Splitting*: The given HDS may be split, merging a subset of its objects with another HDS in RHDS. Any remaining objects will be treated as an HDS if there are at least two objects. If only one object remains, it is added to a set of singleton objects that are placed at the end of the preallocated memory region.

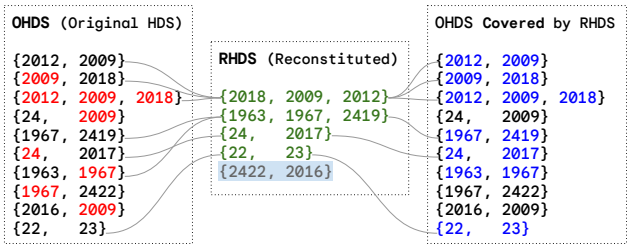

Figure 2. An illustration of layout determination from the trace of **cc1** benchmark.

Note that we do not fully merge more than two HDS with each other because an effective layout for two HDS can always be found by placing them adjacent to each other with common objects in the middle; however, the same cannot be achieved for three HDS or more in general. That is why we resort to splitting of an HDS so that part of it can be merged while the remaining objects can be treated as a separate HDS to be added to RHDS or simply discarded (i.e., left unexploited). Also note that a HDS must have at least two objects for it to be useful. Thus after splitting, if a singleton object results, it will be simply added at the end of preallocated memory region along with other hot singleton objects.

We illustrate the benefit of this algorithm with an example in Figure 2. This example is based on a trace collected from a short run of cc1 program. From the pruned trace, we identified 10 hot HDS shown on the extreme left (OHDS) in sorted order of memory references. Each HDS is a set of objects whose object ids are listed. For example object ids 2012 and 2009 are part of a single HDS that accounts for the most memory accesses. The object ids that are shown in red appear in multiple HDS. Therefore not all the original HDS are exploitable in their current form. The reconstituted HDS are also shown in the middle. The new HDS are shown in green while the last set highlighted represents singleton objects. We observe the following about computed RHDS:

- *Nearly all objects are covered by HDS in RHDSs* – of the total of 12 hot objects in OHDS, 10 are covered by reconstituted HDS and remaining 2 (highlighted) are treated as singleton objects.
- *Majority of HDS from OHDS are Fully Covered or Partially Covered* in RHDS enabling exploitation of spatial locality. The covered HDS are shown in blue on the right side in Figure 2.
- *Object Reordering Enabled Layout* in the preallocated memory region is as follows: {2018, 2009, 2012, 1963, 1967, 2419, 24, 2017, 22, 23, 2422, 2016}.

If we were to use the HDS [8] algorithm or the HALO [21] methods, the opportunities exploited would have been limited. For the above example in Figure 2, the HDS [8] algorithm would only be able to colocate objects to exploit one HDS: {22, 23}. This is because the interesting malloc sites that create the HDS objects also allocate other objects that also get colocated with the HDS objects, thereby polluting the memory region. HALO [21] may perform better than HDS [8] because it introduces less pollution. However, it is still constrained in its ability to colocate relevant hot objects, thereby limiting spatial locality.

Once the RHDS has been constructed and the layout of the objects has been determined, we can compute the precise offsets at which these objects will be stored in the preallocated memory region using the sizes of the objects. The sizes of some hot objects are fixed and hence known at compile-time whereas the sizes of other objects are input dependent and only known during execution. The object sizes that are used are based on the traces collected from the profiling run. A mapping between objects and offsets in preallocated memory is generated and used at runtime to place objects in appropriate locations.

2.2 Context Definition and Identification

Previous works [6, 21, 22, 31] use the dynamic calling context of the allocation call-site to determine hot object allocations. HDS [8] directs all objects allocated by an interesting call-site to a separate memory region. To motivate the need for a new context definition that can accurately capture hot objects at runtime, we describe the drawbacks of existing techniques used in HDS [8] and HALO [21] that pollute the memory regions for hot objects. Figure 3 is a simplified code fragment based on the pattern observed in the **mcf** benchmark with a loop that contains a single malloc site that creates 5 objects (O_1, O_2, O_3, O_4, O_5). Two of the objects are hot as they are repeatedly accessed later in the code (not shown) causing them to form a HDS or HALO allocation group.

Since HDS [8] directs all five objects to a separate memory region, O_1 and O_5 are not colocated and thus the transformation does not capture the spatial locality between O_1 and O_5. Similarly, since HALO [21] uses call-stack signatures that will be identical for all five objects, they will all be directed to the same memory region and once again O_1 and O_5 will not be colocated, limiting spatial locality. The above situation was frequently observed in the benchmarks used in our experiments.

```
while (...) { // high trip count
    ...
    // Objects Created: O1, O2, O3, O4, O5
    malloc(...)
    // Accessed repeatedly: O1, O5
}
```

Figure 3. Simplified code similar to the **mcf** benchmark illustrating drawbacks of previous approaches [8, 21].

Stability of calling context based approaches. We collected data for the benchmarks to understand the effectiveness of using the calling context (dynamic call-stack) in uniquely identifying hot object allocations. For example, for the **mcf** benchmark, 6 static hot allocation sites allocated 1 hot object each, corresponding to a total of 6 interesting hot objects. Of these 6 allocation sites, only 3 sites uniquely identified the hot object with the calling context. The remaining 3 sites had a total of 30 other object allocations also having the same call-stack signature. We observed similar patterns in other benchmarks. We experimented with other approaches to more precisely capture hot object allocations and came up with an approach that uses object identifiers.

2.2.1 New Context Definition.

Here, we describe our approach for identifying hot dynamic objects of interest at runtime and then assigning an appropriate portion of the space in the preallocated memory region to place them. Since we only preallocate space for a fixed number of RHDS objects, it is important that the relevant objects are captured precisely and efficiently. Next we describe our solution.

Our solution is based upon assigning unique ids to objects created by malloc sites as a combination of:

- ***Static malloc site*** – each object is identified by the malloc site that allocates it; and
- ***Dynamic allocation instance*** – the dynamic allocation instance of the malloc site.

The runtime generated object ids are compared with object ids that were identified during the profiling run. Upon a match, they are placed in the preallocated region at a precomputed spot using the mapping between objects and memory region offsets.

The object ids used to identify hot objects generated by a malloc site $\mathbf{M_i}$ fall into the following three categories:

- **Fixed :** A set of **fixed** dynamic instances. For example, {1,3,8} representing the first, third, and eighth dynamic objects created by $\mathbf{M_i}$.
- **Regular :** A set of dynamic instances that can be captured via a **regular pattern**. For example, {1,3,5,..15} represent the first eight odd numbered dynamic instances created by $\mathbf{M_i}$.
- **All :** Every dynamic instance generated by $\mathbf{M_i}$ is a hot object, and thus no check is needed to detect the hot objects and the object id is simply used for placement of the object in the preallocated memory region.

We automatically identify the pattern that must be used to detect hot objects generated by a malloc site. The dynamic instance ids of the hot objects for a given malloc site are inspected and are categorized into one of the above three

patterns. For instance, if the ids fall into an arithmetic progression, the regular pattern will be used. Then, we create a counter per malloc site that generates one or more hot objects of interest, say $\mathbf{C_i}$ for static malloc site $\mathbf{M_i}$. Each time $\mathbf{M_i}$ is encountered, the $\mathbf{C_i}$ is incremented to compute the dynamic allocation instance of the object.

Note that by using different counters for different malloc sites, and associating the code that increments the counters with the malloc sites, the static id is ***implicitly matched*** and only runtime checks for matching the dynamic ids are needed. The code size increase, shown in Figure 14, is small for most benchmarks, because the number of malloc sites that are instrumented to detect hot objects is small. The number of counters that are needed is even smaller because multiple malloc sites that work in tandem can ***share a counter***. Two malloc sites can share a counter if sharing results in object ids for the sharing malloc sites that follow one of three supported patterns (i.e., Fixed, Regular, and All). To automatically identify counter sharing opportunities, sharing is simulated over the allocation trace and updated object ids are generated. If the sequence of object ids of the sharing malloc sites in the allocation trace reveal a pattern, sharing is employed.

Table 2. Context Used.

Benchmark	[type, (#sites, #counters)]
mysql	[fixed ids, (10, 6)]
perl	[regular & fixed, (15, 7)]
mcf	[fixed ids, (6, 2)]
omnetpp	[fixed ids, (52, 6)]
xalanc	[fixed ids, (2, 2)]
povray	[all ids, (8, 1)]
roms	[all ids, (20, 1)]
leela	[all ids, (4, 1)]
swissmap	[all ids, (1, 1)]
libc	[fixed ids, (6, 2)]
health	[fixed & all ids, (3, 2)]
ft	[fixed & all ids, (3, 2)]
analyzer	[fixed & all ids, (5, 3)]

Table 2 presents the contexts that were used in the various benchmarks considered in this work. For each benchmark we provide the types (type: fixed, regular, and/or all) of object ids that were used, the number of static malloc sites involved (#sites), and number of counters (#counters) that these sites together used. The number of relevant malloc sites that allocate hot objects range from 1 to 52 and the numbers of counters that were used ranged from 1 to 7.

We also examined the source code of these applications to understand what the hot objects represent and why the simple context prediction we propose works. Let us consider the mcf benchmark which involves a network flow problem which has six objects allocated by six distinct malloc sites. The first three objects created by this program are hot and they represent the input network graph itself – the first object stores all the nodes of the graph, the second object stores the maximum number of edges or arcs that the network can handle, and the third object is for storing additional dummy_arcs. The other three hot objects are part of an optimization algorithm used by the compute intensive psimplex algorithm [14]. The two groups of hot objects have **fixed** ids and each group of three can share a single counter. In swissmap there is a single malloc site that generates large number of objects to which object recycling can be applied as a small group of objects are created, used, and freed, and this pattern is repeated. Thus, in this case **all** ids are of interest and a single counter is used to decide where to store the object in preallocated region.

2.2.2 Case for a hybrid approach.

For the benchmarks we considered, object ids identified hot objects really well. The ids used to identify a hot object or a HDS in the profiling run using training inputs also largely corresponded to the actual run. On the other hand, our analysis of several previous approaches using calling context showed that it can suffer from high imprecision in several call sites leading to polluting the memory regions allocated for hot objects with several cold objects. However, it is important to note that calling context does work precisely on a subset of call sites.

In the larger picture, any of these techniques could produce undesired results when applied to non-deterministic programs, like large data center server applications. In such cases, it could make sense to use both mechanisms together, object IDs and calling context, to accurately target hot objects. As part of future work, we intend to study and scale our approaches to work on such large applications.

2.3 Instrumentation of malloc and free sites

Next we show how malloc and free sites are instrumented to obtain the optimized program.

(a) malloc: As shown in Figure 4, using the Counter corresponding to the malloc site, the dynamic instance ObjectId is generated which is checked against the preallocated hot objects. Upon a match, using the mapping between ObjectId and address in preallocated space, the ObjectId is assigned an appropriate ObjectAddress. Otherwise, that object is allocated using a call to malloc. Only relevant malloc sites that generate objects of interest are instrumented.

(b) free: Any free that can potentially deallocate a preallocated memory object must be instrumented with code that checks if the ObjectAddress is from the preallocated memory. If a preallocated object is freed by the user program, we simply mark that the object has been deallocated but we do not call free as no space is returned to the heap. However, if the object was allocated from the heap then a call to free is made as shown in Figure 5.

```
ObjectID = Counter + 1;
// if an object is preallocated and size fits
if (ObjectID ∈ PreallocObjectIDS &&
    ObjectSize ≤ PreallocSize[ObjectID]) {
    // Return position in Preallocated memory
    ObjectAddress = PreallocMemory[ObjectID];
} else {
    // Default call to the memory allocator
    ObjectAddress = malloc(ObjectSize);
}
```

Figure 4. Instrumentation of **chosen** `malloc` sites.

```
if (ObjectAddress ∈ PreallocMemory) {
  Mark ObjectAddress as free;
} else {
  free(ObjectAddress);
}
```

Figure 5. Instrumentation for **all** free sites.

```
if (ObjectAddress ∈ PreallocMemory) {
  if (NewSize ≤
      PreallocAddrSize[ObjectAddress]) {
    NewObjectAddress = ObjectAddress;
  } else {
    NewObjectAddress = malloc(NewSize);
    // Copy contents from old object to new
    memcpy(NewObjectAddress,
                ObjectAddress, ObjectSize);
    Mark ObjectAddress as free;
  }
} else {
  // Default call to the memory reallocator
  NewObjectAddress =
        realloc(ObjectAddress, NewSize);
}
```

Figure 6. Instrumentation for **all** realloc sites.

(c) realloc: Similar to frees, reallocs must also be instrumented with code that checks if an object being reallocated is from the preallocated memory. As shown in Figure 6, if a preallocated object is reallocated in the program, we check if the new size of the object is within the bounds of the space allocated in the preallocated memory region. If the size does not fit, we move the object away from the preallocated region. However, the common case for preallocated objects is that the new size is less than the preallocated size as this information was already obtained from the original trace.

Correctness of above Transformations. The above instrumentation based transformations preserve program semantics. Instead of allocating some objects individually from the heap, they are simply placed in a large preallocated heap memory region. Calls to free simply skip returning the object

```
Counter++;
Map = (Counter - 1) % N;
// if mapped object is freed and size fits
if (isFree(PreallocMemory[Map]) &&
    ObjectSize ≤ PreallocSize[Map]) {
    ObjectAddress = PreallocMemory[Map];
} else {
    // Default call to the memory allocator
    ObjectAddress = malloc(ObjectSize);
}
```

Figure 7. Instrumentation for Object Recycling.

memory to the heap if it was allocated in the preallocated memory region. Since the program's logic does not depend on where in the heap an object is allocated memory, the program semantics is preserved.

2.4 Object Recycling

For recycling **N** preallocated objects, a modulo operator is used to increment and update the Counter associated with the malloc site to determine the space where the object must be mapped (see Figure 7). For each of the N preallocated objects we remember whether the space is free or not. When placing another object in the same space, we check to make sure no live object currently resides there. If that is the case, the object being created can reuse the space.

Correctness of Object Recycling. The above transformation involves selecting N, that is number of objects that can be held in the preallocated region. During a program run, if the number of objects that are simultaneously live is greater than N, the code still functions correctly since additional objects are not stored in the preallocated memory but rather they are simply allocated via malloc calls. If the number of simultaneously live objects is less than N, it simply alters where the objects are stored, but they will all be held in preallocated memory.

3 Experimental Evaluation

3.1 Implementation

Figure 8 shows our implementation of **PreFix**. From the original baseline executable, the allocation, free and access trace is generated using DynamoRIO [5]. The generated trace is analyzed to identify Hot Singleton objects and HDSs. Unlike the HDS [8] work that uses Sequitur [15] to identify hot data streams, we employ the Longest Common Subsequence [10] algorithm because it is highly efficient and as effective as Sequitur. Once the HDS and the Hot Singleton objects are identified, the original executable is instrumented to perform memory region preallocation, compute object ids, and map the corresponding objects to the right places in the preallocated memory. The instrumentation is achieved

using LLVM-BOLT [16] which transforms the relevant malloc/free sites to obtain the optimized executable. Cache and TLB performance data are collected using DrCacheSim [5].

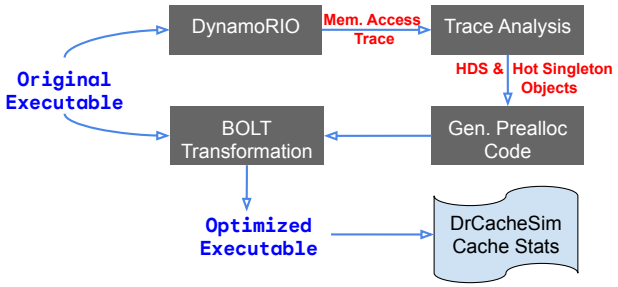

Figure 8. PreFix Implementation.

3.2 Experimental Setup

We evaluate the following algorithms to compare the performance and effectiveness of our approach with prior works.

- **HDS** - Implemented using our framework but it exploits only those HDSs that are constructed by the technique described in [8], that is, HDSs are not reconstituted.
- **HALO** -Code was available [20] and hence we used the authors' original implementation.
- **PreFix:Hot** - All hot objects are placed in preallocated space in the order they are allocated.
- **PreFix:HDS** - Places reconstituted and reordered HDS objects as constructed by our layout determination algorithm in the preallocated space.
- **PreFix:HDS+Hot** - First lays out objects as in **PreFix:HDS** and then places remainder of the hot objects, not part of any HDS, at the end of the preallocated region.

The benchmarks used in the experiments are from Spec [23], Olden [17], Ptrdist [3], FreeBench [19] and fleetbench [1]. In addition, we also use mysql. The baseline runs used in our experiments range from a few seconds to several hundred seconds corresponding to 560 million to 837 billion memory references. The profiling runs used to transform the programs were training inputs involving significantly shorter program runs. Experiments were performed on an Intel machine with the following configuration: 32 KB, 8-way L1 data cache with 64 B line size; 40 MB, 20-way LLC with 64 B line size; and TLB with 64 entry 4-way L1 and 1536 entry 6-way LLC. The baseline binary was generated by compiling the programs using $-O3$ level of optimization.

3.3 Reductions in Execution Times

Table 3 presents the baseline execution times and the reductions (negative numbers) and increases (positive numbers) in execution times relative to the baseline for all five optimization algorithms.

Overall Performance Comparison. **PreFix** outperforms HDS and HALO. This shows that exploiting all hot objects is much better than only handling a subset of hot objects, as is the case with HDS. While HALO can exploit all hot objects, it does not perform object reordering within each allocation group. Best performing versions of **PreFix** reduce execution times by **2.77%** to **74%** with an overall average reduction of **21.7%** across all 13 benchmarks. In contrast, HDS [8] results in average reduction of **7.3%** across 13 benchmarks and HALO [21] **14%** for a subset of 8 benchmarks. We also observe that the standard HDS [8] technique is substantially outperformed by **PreFix:HDS** because of our algorithm for reconstituting HDSs and achieving a superior data layout.

In four benchmarks, povray, roms, leela, and swissmap, opportunities for **object recycling** yield **3.44%**, **17.8%**, **25.3%**, and **11.1%** reductions in execution times respectively. These improvements are due to reductions in memory management overhead and memory footprint. Finally, health's behavior is unique in that it has large number of objects that are equally hot. Therefore **PreFix:Hot**, **PreFix:HDS+Hot**, and HALO perform very well giving **43.4%** reduction in execution time.

On the other hand, **PreFix:HDS** provides only a small improvement because very few objects belong to hot data streams. However, HDS delivers **35.9%** reduction in time because in this case pollution helps performance by sending hot objects, along with HDS objects, to the separate memory region. That is, pollution causes HDS to behave like HALO.

Pollution Analysis. Table 4 shows how many polluting objects are added by HDS and HALO to memory regions allocated for HDSs and HALO allocation groups. As we can see, a large number of objects are directed to these memory regions even though only a few hot objects are present. This is a key limitation of these works that limits their ability to effectively exploit spatial locality.

In contrast to prior work, the data in Table 5 for best performing version of **PreFix** shows that the behavior of objects captured during the long program run is comparable to those determined by analyzing the profiling run. In both runs the percentage of heap accesses (column HA) that are due to objects placed in preallocated memory is high. Vast majority of objects captured during the long run are indeed hot objects (column Hot). Finally, the objects that are observed as forming HDSs (column HDS) are also comparable. The small differences are due to differing program behaviors across the profiling run and long run inputs. That is, spurious objects are not being assigned to the preallocated memory region by **PreFix**. In contrast, as we saw in Table 4, HDS [8] and HALO [21] send thousands of non-hot objects to separate memory regions. Figure 9 plots the heap accesses for the same hot and interesting objects in the baseline and the **PreFix** optimized versions of **leela** benchmark. In the plot, the x-axis is time and the y-axis is heap virtual address offset. For the baseline, the heap footprint is about 10 M for these

Table 3. Execution Times of Baseline and Relative Changes in execution times achieved by HDS, HALO, and PreFix Versions. Positive percentages are increases in execution times while negative percentages are reductions in execution times. Execution time reductions via best performing versions of **PreFix** are in bold. *Execution times for all the benchmarks are averaged over 10 runs and the variations across the runs are small except for the first run.*

Benchmark	Baseline		HDS [8]	HALO [21]	PreFix			
	Time (s)	Mem. Refs.			Hot	HDS	HDS+Hot	Best
mysql	152.7	560 million	+3.9%	*na*	**-13.7%**	-10.2%	-5.2%	-13.7%
perl	106.0	337 billion	-6.3%	*na*	-7.6%	**-8.30%**	-7.8%	-8.30%
mcf	11.74	13.3 billion	+0.8%	-1.20%	-4.9%	-5.10%	**-7.3%**	-7.30%
omnetpp	434.5	556 billion	+0.6%	*na*	-10.6%	**-13.2%**	-10.2%	-13.2%
xalanc	43.38	138 billion	-1.2%	*na*	-4.0%	-3.90%	**-4.3%**	-4.3%
povray	502.3	1.6 trillion	+0.001%	*na*	**-3.44%**			-3.44%
roms	390.2	450 billion	-0.02%	-0.10%	**-17.8%**			-17.8%
leela	555.8	837 billion	+0.9%	-0.80%	**-25.3%**			-25.3%
swissmap	2.275	1.6 billion	+1.1%	-1.50%	**-11.1%**			-11.1%
libc	1.080	630 million	+0.01%	-0.73%	-1.85%	**-2.77%**	-0.93%	-2.77%
health	32.73	5.6 billion	-35.9%	-43.1%	-43.3%	-1.31%	**-43.4%**	-43.4%
ft	5.04	768 million	-42.8%	-47.0%	-73.0%	-1.00%	**-74.0%**	-74.0%
analyzer	18.08	10.1 billion	-15.9%	-17.6%	-57.1%	-18.4%	**-58.9%**	-58.9%

na – HALO [21] could not be run using the BOLT 2018 version required by the BOLT patch used by HALO [21]

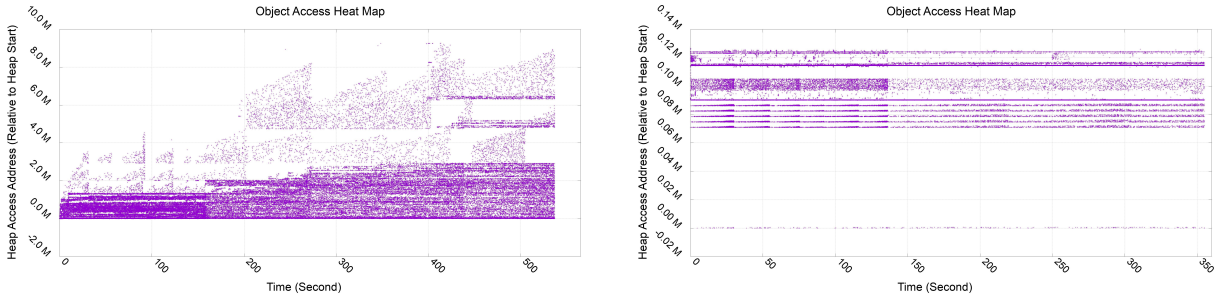

Figure 9. Data Access HeatMaps of Baseline (left) and **PreFix** Optimized (right) binaries. X-axis is time and Y-axis is relative heap offset.

Table 4. Pollution in **HDS** and **HALO**.

Benchmark	HDS		HALO	
	Hot	All	Hot	All
mysql	2	80	na	na
perl	76	32,977,460	na	na
mcf	4	33	10	59,847
omnetpp	67	123,727	na	na
xalanc	54	27,464	na	na
povray	0	16,879	na	na
roms	0	10,690	0	226,552
leela	0	809	1	198,816
swissmap	7	149,191	4	59,864
libc	8	1,072	6	6,639
health	683,334	683,334	1,318,819	1,318,819
ft	13,334	40,000	20,000	59,998
analyzer	2,242	2,242	8,196	8,196

Table 5. PreFix Object Capture in Profiling vs. Long Run: (HA) % of Heap Accesses by Preallocated Objects; (Hot) # of hot objects; and (HDS) # of hot objects in HDSs.

Benchmark	Profiling Run			Long Run		
	HA	Hot	HDS	HAs	Hot	HDS
mysql	93.0%	13	7	86.5%	7	5
perl	60.8%	174	120	53.5%	109	85
mcf	89.3%	6	3	99.9%	6	3
omnetpp	61.1%	230	94	52.1%	153	80
xalanc	75.4%	236	67	72.9%	101	67
povray	50.1%	20	20	28.9%	20	20
roms	33.4%	20	20	74.5%	20	20
leela	37.2%	5	5	70.1%	5	5
swissmap	87.5%	8	8	97.5%	8	8
libc	94.5%	438	384	93.1%	429	376
health	97.2%	1,733,377	213	99.9%	1,733,377	213
ft	82.2%	20,000	868	98.5%	20,000	868
analyzer	98.5%	103,613	3	88.5%	103,613	3

objects, whereas, in the optimized it is about 0.2 M. By using preallocated memory, we have significantly reduced the heap access footprint leading to much better spatial locality.

Relative Benefits of PreFix Versions. As the data in Table 3 shows, different versions of **PreFix** perform the best for different benchmarks. Below, we discuss some of the reasons for these differences.

perl, omnetpp and libc - The **PreFix:HDS** version gives the best speed up. It is because the Hot singleton objects at the end, in the HDS+HOT version, are short lived and their original ordering with the cold object seems to be better for locality compared to them being colocated with other Hot singleton objects. Also, for these benchmarks, performance is better than the HOT-only version, which means that the ordering of HDS objects is also important.

mcf, xalanc, health, ft and analyzer - HDS object reordering and placement of remaining hot objects at the end of the preallocated space improve locality.

povray, roms, swissmap, and leela - The speedups are mainly due to the recycling of objects. In these benchmarks, there are no spurious objects and hence all versions of **PreFix** perform the same.

mysql - The **PreFix:HOT** version gives the best performance. This is because the sizes of the hot objects in mysql are very large with significant intra-object spatial locality. Thus, the impact of object reordering on performance is insignificant.

PreFix on Multithreaded Executions. We also evaluated the benefits of **PreFix** on applications that are multithreaded, particularly mysql and mcf, where the number of threads can be varied. The traces were collected only once from these benchmarks with the number of threads set to the default value. *PreFix*, with the best configuration for that benchmark, was then used to optimize each benchmark. The final optimized binary was then run several times but with different number of threads each time. Figure 10 shows the relative performance improvements of each run with k threads, relative to the baseline run with k threads.

Almost always, **PreFix** optimized runs are faster than their corresponding baseline runs, with improvements ranging from 4.6% to 15.4% in mysql and 1.3% to 10.1% in mcf. With 8 threads, mcf experienced a regression of 1.2%.

In mysql, the hot objects are allocated and accessed by a unique thread, whereas with mcf, a single thread allocates the hot objects and are accessed by several threads. **PreFix** is able to optimize and improve these benchmarks nevertheless.

3.4 Costs and Benefits of PreFix

In this section we further study the costs and benefits of the best performing versions of **PreFix**.

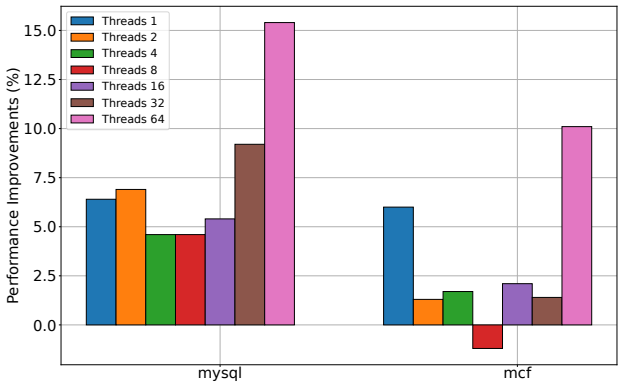

Figure 10. PreFix: Effect of Multithreading.

Table 6. Best **PreFix**: Benefits and Costs.

Benchmark	Calls Avoided	Dynamic Instr. Count Change	Peak Memory Change (MB)
mysql	12	-1.50%	18 ⟶ 426
perl	119	+0.07%	92 ⟶ 94
mcf	5	+0.30%	292 ⟶ 333
omnetpp	93	+1.60%	248 ⟶ 250
xalanc	235	-0.31%	368 ⟶ 405
povray	10,833	-0.2%	8.8 ⟶ 8.6
roms	1,415,999	-0.1%	867 ⟶ 862
leela	30,263,160	-25.2%	28 ⟶ 20
swissmap	148,479	+9.50%	619 ⟶ 318
libc	383	-7.10%	81 ⟶ 88
health	1,733,376	-2.0%	56 ⟶ 43
ft	19,999	-1.1%	7.1 ⟶ 6.5
analyzer	103,612	-0.1%	18 ⟶ 10

Figures 11 and 12 show the cache misses encountered by L1 and LLC (Last level Cache) caches by the baseline binary and the **PreFix** transformed version. This measures the percentage of memory accesses that missed in the L1 and LLC respectively. As we can see, in some programs L1 misses are significantly reduced, while in most benchmarks the benefits come from substantial reductions in LLC misses (shown in log scale). Figure 13 further shows that there is a reduction in the percentage of CPU cycles stalled by the backend [29], a clear improvement in the memory bottleneck, across all benchmarks. TLB miss rates improved significantly and measurably for the health (10% to 0.1%) and the analyzer (0.62% to 0%) benchmarks.

Table 6 shows the number of malloc/free calls eliminated and the net change in the number of instructions executed. Even though the instrumentation added to programs to implement our optimizations causes additional instructions to be executed, this increase is offset to a significant extent by the malloc/free calls that are avoided via the use of preallocated memory. Therefore, in most programs, there is a fairly small reduction in the number of executed instructions. In

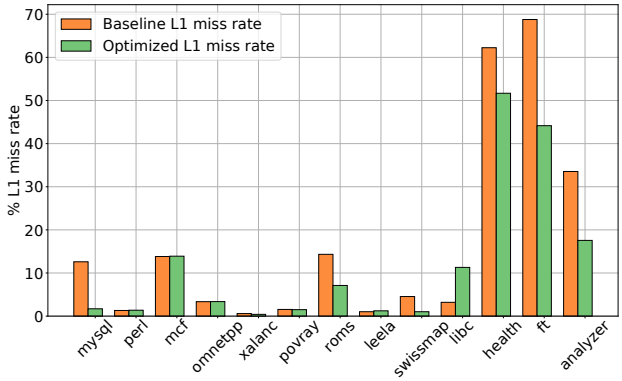

Figure 11. PreFix: Change in L1 miss rate.

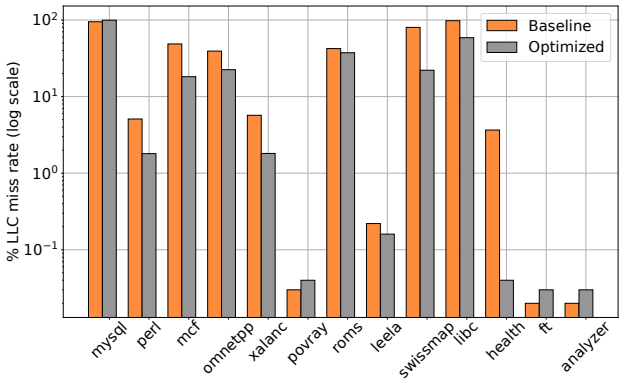

Figure 12. PreFix: Change in LLC miss rate.

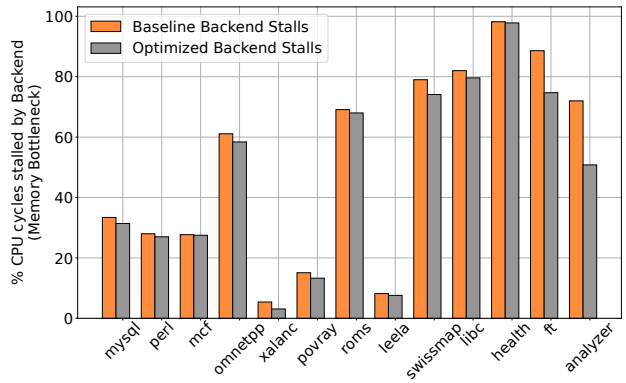

Figure 13. PreFix: Change in Backend Stalls.

leela there is a significant reduction because a very large number of malloc/free calls are avoided. Finally, preallocation can cause an increase in peak memory usage. The last column shows the peak memory usage without (baseline) and with optimization (PreFix). In most benchmarks, there is a modest increase with mysql being an outlier with a significant increase. The exceptions, like leela and swissmap, show substantial reductions in peak memory usage due to the use of object recycling.

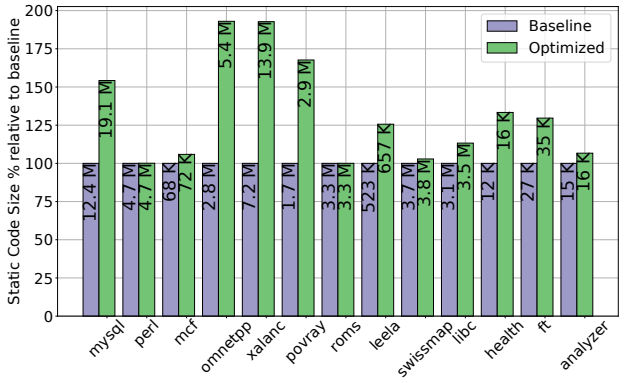

Figure 14. Binary Sizes: Baseline → Best **PreFix**

Figure 14 shows the increases in the static sizes of binaries due to the transformations of the malloc / free sites. Benchmarks where a large number of static malloc sites allocate hot objects also experienced increased code growth as a result of the required instrumentation. For the mysql, omnetpp, xalanc, and povray benchmarks, the excessive bloat in code size is not due to the overheads of the instrumentation. For these benchmarks alone, BOLT retained the original code in a separate section called "*.bolt.orig.text*". This section can be completely deleted and excluding this section makes the code size bloat of these benchmarks similar to the other ones.

Although there are costs associated with **PreFix**, such as the increased memory footprint and static code size, the overall benefits far exceed the costs.

4 Related Work

PreFix performs layout optimization by separating hot objects from other objects as well as changing the order in which they are stored. As we have shown, HDS [8] and HALO [20, 21] which fall in the same category have significant limitations and by overcoming them, **PreFix** greatly outperforms them. Next we describe more related works on enhancing performance of heap-intensive applications.

Data layout optimizations [12, 18, 25, 27] that rearrange fields, objects within objects and perform structure splitting have been proposed to improve cache utilization via better spatial locality. Efficient heap allocation of pointer-based data structures [7, 13] have been proposed where structure elements accessed contemporaneously are colocated or dissimilar instances of such structures are allocated in different pools. These techniques use a combination of the structure's topology, static analysis, and profile information to drive the optimizations. While **PreFix** will allocate and colocate hot data structure instances in the preallocated region guided by profile information, fine-grained layout of individual objects is not within the scope of this work and is orthogonal.

Run-time techniques [26, 31] to track hot call-site allocations and placing them in separate pools have been studied. A run-time framework like DynamoRIO [5] is used to track calling contexts of interesting call-sites and hot allocations are placed in separate pools, guided by either profiles or static analysis. **PreFix** optimizes the binary to be run as standalone and uses simple checks to identify allocations to be placed in the preallocated regions.

Object Lifetimes. Techniques that use lifetimes [4, 22] to place short-lived objects together have been evaluated. **PreFix** uses object lifetimes implicitly to determine allocation decisions. Colocated HDS objects will likely have similar lifetimes as they are more likely to occur together.

Calling context. Several techniques [6, 21, 22, 31] use the calling context of the call-sites to determine hot allocations. In contrast, **PreFix** proposes a highly precise numbering scheme using the static call-site id and its dynamic instance to determine the objects that must be preallocated.

Arena allocators [9, 24, 28] also use preallocated memory called arenas to perform allocations. These are intended to reduce the cost of allocation and deallocation and works very well when objects with largely overlapping lifetimes are grouped into the same arena. The cost of deallocation is reduced to finally garbage collecting the entire arena and the cost of allocation is a pointer increment. *PreFix*, *HALO* and our implementation of *HDS* use preallocated memory similar to arenas and the cost of allocating and deallocating objects is cheap similarly. However, with object recycling, **PreFix** also drastically reduces the memory foot-print. Using several arenas for objects with different lifetime ranges is left for future work.

Other techniques Also, *Other* layout techniques have been explored, like [30], which introduce a class of transformations to modify the representation of dynamic data structures used in programs with the objective of *compressing their sizes*. The compression is made possible by frequently occurring values that have a common prefix or represent narrow-data values. In [2], the authors propose a GC locality optimization that uses heuristics to guide GC threads processing local objects. The design of Temeriare [11] shows how a huge page aware memory allocator can reduce memory usage by improving fragmentation and also improve TLB utilization. In [32], a comprehensive characterization of data center workloads and tuning several aspects of the memory allocators like per-CPU cache sizes, allocation algorithms per workload to achieve performance speedups is presented. It should be noted that all of these techniques are complementary and can be used in conjunction with **PreFix**.

5 Conclusions

Prior works on optimizing heap layout for performance have looked at separating the small fraction of hot dynamic heap objects by allocating them in independent memory regions. Further, the techniques to identify such objects have relied primarily and heavily on the dynamic calling context of the allocation site. In this work we have shown that merely redirecting hot objects to independent memory regions is suboptimal and does not exploit the full inter-object spatial locality (object order across allocation sites) that can maximize performance gains. Instead, **PreFix** uses preallocated memory to separate and group hot objects across allocation sites for maximal locality. Further, while using calling contexts is effective in some cases to identify hot objects, there are several instances where it contributes to significant pollution of the hot memory regions with irrelevant/cold objects. In **PreFix**, we have shown that a simple technique that uses dynamic object IDs can effectively identify hot objects with low static and dynamic instrumentation overheads.

Evaluation of **PreFix** on 13 memory intensive benchmarks, large and small, has shown that our technique has consistently outperformed previous techniques and yielded performance improvements averaging 22%. **PreFix** has clearly demonstrated the performance potential of exploiting spatial locality by reordering hot objects across allocation sites. Future work would look at scaling our techniques to very large warehouse-scale applications, like servers, and using hardware tracing to reduce the overheads of trace collection and analysis.

Acknowledgment

This work was supported by a research award from Google and National Science Foundation Grants CCF-2226448 and CCF-2002554 to the University of California, Riverside.

References

[1] Andreas Abel, Yuying Li, Richard O'Grady, Chris Kennelly, and Darryl Gove. 2024. A Profiling-Based Benchmark Suite for Warehouse-Scale Computers. In *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 325–327. https://doi.org/10.1109/ISPASS61541.2024.00046

[2] Khaled Alnowaiser. 2014. A study of connected object locality in NUMA heaps. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*. Association for Computing Machinery, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/2618128.2618132

[3] Todd Austin. 1995. The Pointer-intensive Benchmark Suite. https://pages.cs.wisc.edu/~austin/ptr-dist.html. (1995).

[4] David A Barrett and Benjamin G Zorn. 1993. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming Language Design and Implementation*. 187–196.

[5] Derek L. Bruening and Saman Amarasinghe. 2004. Efficient, transparent, and comprehensive runtime code manipulation. (2004). AAI0807735.

[6] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-conscious data placement. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. 139–149.

[7] Trishul M Chilimbi, Mark D Hill, and James R Larus. 1999. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. 1–12.

[8] Trishul M Chilimbi and Ran Shaham. 2006. Cache-conscious coallocation of hot data streams. In *Proceedings of the 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. 252–262.

[9] Ryan Fleury. 2022. Untangling Lifetimes: The Arena Allocator. https://www.rfleury.com/p/untangling-lifetimes-the-arena-allocator. (2022). [Online; accessed August-2024].

[10] GeeksforGeeks. Longest Common Subsequence. https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/.

[11] Andrew Hamilton Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 257–273.

[12] Thomas Kistler and Michael Franz. 2000. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 3 (2000), 490–505.

[13] Chris Lattner and Vikram Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. *ACM Sigplan Notices* 40, 6 (2005), 129–142.

[14] Andreas Lobel. 2011. 429.mcf: SPEC CPU2006 Benchmark Description. https://www.spec.org/cpu2006/Docs/429.mcf.html. (2011). [Online; accessed August-2024].

[15] Craig Nevill-Manning and Ian Witten. 1997. Linear-time, incremental hierarchy inference for compression. 3–11. https://doi.org/10.1109/DCC.1997.581951

[16] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: a practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, 2–14.

[17] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. 1995. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.* 17, 2 (mar 1995), 233–263. https://doi.org/10.1145/201059.201065

[18] Shai Rubin, Rastislav Bodík, and Trishul M. Chilimbi. 2002. An efficient profile-analysis framework for data-layout optimizations. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 140–153. https://doi.org/10.1145/503272.503287

[19] Peter Rundberg and Fredrik Warg. 1995. The FreeBench v1.03 Benchmark Suite. https://web.archive.org/web/20020601092519/http://www.freebench.org/. (1995).

[20] Joe Savage and Timothy M Jones. 2019. Research data supporting "HALO: Post-Link Heap-Layout Optimisation". Apollo - University of Cambridge Repository. https://doi.org/10.17863/cam.46071. (2019).

[21] Joe Savage and Timothy M Jones. 2020. Halo: Post-link heap-layout optimisation. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 94–106.

[22] Matthew L Seidl and Benjamin G Zorn. 1998. Segregating heap objects by reference behavior and lifetime. *ACM SIGPLAN Notices* 33, 11 (1998), 12–23.

[23] Standard Performance Evaluation Corporation. 2017. SPEC CPU2017. https://www.spec.org/cpu2017. (2017).

[24] Protocol Buffer Team. 2022. C++ Arena Allocation Guide. https://protobuf.dev/reference/cpp/arenas/. (2022). [Online; accessed August-2024].

[25] Dan N Truong, Francois Bodin, and André Seznec. 1998. Improving cache behavior of dynamically allocated data structures. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*. IEEE, 322–329.

[26] Zhenjiang Wang, Chenggang Wu, and Pen-Chung Yew. 2010. On improving heap memory layout by dynamic pool allocation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 92–100.

[27] Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Di Xu. 2012. On-the-fly structure splitting for heap objects. *ACM Trans. Archit. Code Optim.* 8, 4, Article 26 (jan 2012), 20 pages. https://doi.org/10.1145/2086696.2086705

[28] Chris Wellons. 2023. Arena Allocator tips and tricks. https://nullprogram.com/blog/2023/09/27/. (2023). [Online; accessed August-2024].

[29] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

[30] Youtao Zhang and Rajiv Gupta. 2002. Data Compression Transformations for Dynamically Allocated Data Structures. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, Berlin, Heidelberg, 14–28.

[31] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. 2005. Dynamic memory optimization using pool allocation and prefetching. *ACM SIGARCH Computer Architecture News* 33, 5 (2005), 27–32.

[32] Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. 2024. Characterizing a Memory Allocator at Warehouse Scale *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 192–206. https://doi.org/10.1145/3620666.3651350

Received 2024-09-12; accepted 2024-11-04