HALO: Post-Link Heap-Layout Optimisation

Joe Savage
University of Cambridge, UK
joe@reinterpretcast.com

Timothy M. Jones
University of Cambridge, UK
timothy.jones@cl.cam.ac.uk

Abstract

Today, general-purpose memory allocators dominate the landscape of dynamic memory management. While these solutions can provide reasonably good behaviour across a wide range of workloads, it is an unfortunate reality that their behaviour for any *particular* workload can be highly suboptimal. By catering primarily to average and worst-case usage patterns, these allocators deny programs the advantages of domain-specific optimisations, and thus may inadvertently place data in a manner that hinders performance, generating unnecessary cache misses and load stalls.

To help alleviate these issues, we propose HALO: a post-link profile-guided optimisation tool that can improve the layout of heap data to reduce cache misses automatically. Profiling the target binary to understand how allocations made in different contexts are related, we specialise memory-management routines to allocate groups of related objects from separate pools to increase their spatial locality. Unlike other solutions of its kind, HALO employs novel grouping and identification algorithms which allow it to create tight-knit allocation groups using the entire call stack and to identify these efficiently at runtime. Evaluation of HALO on contemporary out-of-order hardware demonstrates speedups of up to 28% over `jemalloc`, out-performing a state-of-the-art data placement technique from the literature.

CCS Concepts • **Software and its engineering → Allocation / deallocation strategies**.

Keywords Memory management, dynamic allocation, cache locality, profile-guided optimisation, binary rewriting

ACM Reference Format:
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 (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3368826.3377914

1 Introduction

As the gap between memory and processor speeds continues to widen, efficient cache utilisation is more important than ever. While compilers have long employed techniques like basic-block reordering, loop fission and tiling, and intelligent register allocation to improve the cache behaviour of programs, the layout of *dynamically allocated* memory remains largely beyond the reach of static tools.

Today, when a C++ program calls `new`, or a C program `malloc`, its request is satisfied by a *general-purpose* allocator with no intimate knowledge of what the program does or how its data objects are used. Allocations are made through fixed, lifeless interfaces, and fulfilled by inflexible, relatively conservative back-end allocators [13]. Naturally, this creates inefficiency, and can produce programs whose performance is beholden to the whims of a generic data-layout algorithm, generating unnecessary cache misses, TLB misses, and prefetching failures [11, 35]. While custom allocators and diligent programming practices can resolve these issues with startling efficiency [4], outside of high-performance niches like games programming these solutions can be complex, time consuming, and, on the whole, easy to get wrong [5, 15].

To address these issues, this paper proposes HALO (Heap Allocation Layout Optimiser): an automatic post-link profile-guided optimisation tool and runtime memory allocator that intelligently lays out heap data to reduce cache misses. While other solutions of this kind exist, prior work suffers from a range of issues including poorly fitting models, expensive runtime identification techniques, and complications surrounding fragmentation behaviour. HALO differentiates itself from existing solutions through a number of novel features.

First, it performs intelligent affinity grouping of memory allocations, deriving robust placement information from an affinity graph using a clustering algorithm based on weighted graph density, and filtering edges based on practical runtime constraints like co-allocatability. Second, it employs an adaptive full-context identification mechanism, using information from the *entire* call stack to accurately characterise heap allocations during profiling, and distilling this information to only a small handful of call sites that it must monitor to efficiently identify co-allocation opportunities at runtime. Third, it operates at the binary level without any need for high-level source code, and after all other optimisations have been performed, allowing it to be applied retroactively and for profile data to be used at the highest level of accuracy.

CGO '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377914

At a high level, HALO first profiles the target program to build a model of how the allocations it makes in different contexts are related. Clustering these relationships into groups, it then applies binary rewriting to instrument key points of interest within the program, and synthesises a specialised allocator that uses these instrumentation points to guide allocation decisions – satisfying requests from groups of related allocations from separate pools in order to reduce cache misses. Using these techniques, HALO can reduce L1 data-cache misses by up to 23%, and total execution time by up to 28%, against `jemalloc` on contemporary out-of-order hardware, significantly improving performance on a range of programs where a previous approach [11] fails.

2 Background and Related Work

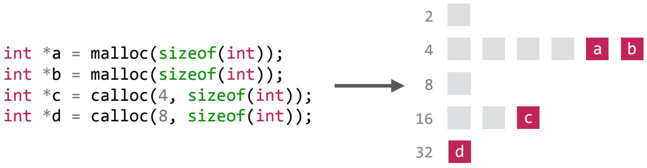
2.1 Memory Allocation

Most applications manage dynamic memory through a general-purpose memory allocator like `ptmalloc2` (glibc), `tcmalloc` [14], or `jemalloc` [12]. While these allocators can provide reasonably good behaviour across a wide range of workloads, their behaviour for any *particular* workload can be highly suboptimal. By catering primarily to average and worst-case usage patterns, these allocators fail to accommodate the specific behaviours of individual programs, and thus deny them the advantages of domain-specific optimisations [13].

One such missed opportunity, of particular importance, concerns the placement of data objects with strong temporal locality. Since general-purpose allocators are traditionally not provided with any intimate knowledge of how values are used in a given program, they can rely only on simple heuristics to place data, and thus may inadvertently scatter highly related data throughout the heap.

In particular, to satisfy allocation requests with reasonable time and space overheads, almost all contemporary general-purpose allocators — including `ptmalloc2`, `jemalloc`, and `tcmalloc` — are based on *size-segregated* allocation schemes. That is, they organise free blocks and housekeeping information around a fixed number of size classes. As a result, allocations are co-located based primarily on their size and the order in which they're made, as depicted in Figure 1.

While a placement strategy of this sort is a reasonable approach in the absence of any additional information, any program in which strongly related and heavily accessed data objects are allocated non-consecutively or belong to different size classes is likely to perform relatively poorly under this model. In such cases, the related objects are unlikely to be placed within the same cache line, or perhaps even the same page, by a size-segregated allocator, and as such may generate unnecessary cache and TLB misses [35], as well as prefetching failures [11]. This can be especially problematic for programs written in languages like C++ that encourage the allocation of many, relatively small objects, the placement of which can be critical to overall performance [6].



```
int *a = malloc(sizeof(int));
int *b = malloc(sizeof(int));
int *c = calloc(4, sizeof(int));
int *d = calloc(8, sizeof(int));
```

Figure 1. An illustration of a simple size-segregated allocator with power-of-two size classes.

2.2 Related Work

In light of these inefficiencies, we propose specialising allocation routines with the goal of making more informed data-layout decisions. While this research area has received relatively little attention, similar ideas have been proposed in the past and can be roughly divided into three categories: runtime, compile-time, and profile-guided solutions.

2.2.1 Runtime Techniques

MO [36] applies an extremely simple form of analysis based on allocation call sites, co-locating objects allocated from each unique `malloc` caller after a certain number of allocations have been made from the call site, all of which are identical with respect to size. When these conditions are met, objects allocated from the site are laid out contiguously in a custom pool, and thus are placed with strong spatial locality. Wang et al. [35] improve on this scheme through the use of static analysis to identify wrapper functions around `malloc` calls and to co-locate related data objects originating from different allocation sites. The authors construct a storage-shape graph based on static points-to analysis, allowing them to merge related call sites into groups in order to co-locate allocations from multiple sources. The benefit of these approaches is their ability to adapt to real events occurring at runtime. As a result, however, these solutions must make difficult decisions about which analyses they can afford to run – ensuring that their placement decisions can offset their analysis costs to provide a net benefit to execution time. In the case of Wang et al. [35], only the simplest types of wrapper functions can be accounted for, and the quality of allocation groups is bounded by the limitations of the static analysis applied and its related simplifying assumptions.

2.2.2 Compile-Time Approaches

Chilimbi et al. [10] describe `ccmorph`, a memory reorganisation tool that utilises topology information provided by the programmer to both cluster and colour tree-like data structures. Similar topology-based layout decisions can even be carried out without programmer intervention in some cases, purely at compile-time, with the aid of static points-to analysis, as has been demonstrated by Lattner and Adve [22]. Techniques of this sort can yield good results, though are ultimately limited by the capabilities of static analysis.

Chilimbi et al. [10] also describe a more general method that relies on hint-based allocation. They provide a custom allocation function which accepts a size and pointer to an object with which to attempt to co-locate. A similar technique is employed by Jula and Rauchwerger [18] in the design of their TP and Medius allocators. While solutions of this kind can achieve good results, they often require significant programmer effort — particularly in cases where hints must be plumbed through existing infrastructure — and must trade off costs borne by the runtime allocator against data locality.

2.2.3 Profile-Guided Solutions

The latency overheads of runtime solutions and limitations of compile-time solutions have led some authors to consider a third class of solution that uses profile information in making placement decisions. Such strategies have been successfully employed in the cache-conscious reordering and splitting of structure members [9, 19, 34], for instance, and have even proven effective in generating specialised allocators that segregate objects based on their predicted lifetimes to improve page locality and reduce fragmentation [3, 31].

Particularly relevant to the goals of HALO, however, are approaches which carry out object-level co-allocation based on predicted temporal locality. One such scheme, from Calder et al. [7], describes a profile-guided technique by which global variables, stack data, and heap objects can be laid out harmoniously in memory to reduce cache misses. At its core, this identifies heap allocations by XORing the last four return addresses on the stack at any given allocation site to derive a unique 'name' around which heap objects are analysed. Placement decisions are then made on the basis of these names using a temporal relationship graph, and are enforced at runtime using a specialised allocator. While the solution as a whole performs well, only a fraction of its observed speedups are a result of improved heap data placement, leading the authors to conclude that their approach is "not as effective for heap objects".

Another approach is described by Chilimbi and Shaham [11], who employ a co-allocation strategy based on the 'hot data streams' representation of data reference locality [8]. In this scheme, a global data reference trace is constructed from heap allocations during a profiling run, and is compressed using the SEQUITUR algorithm [25] to find minimal 'hot' sequences (*streams*) that make up more than a certain percentage of the trace. Viewing each of these sequences as a set of allocation contexts which the heap allocator might use to place objects together at runtime, an analysis pass then evaluates the projected cache miss reduction from the various object groupings suggested by each stream, and selects a profitable placement policy using an approximation algorithm to the weighted set packing problem [16]. A specialised allocator then enforces this policy at runtime. While performance improvements of up to 20% are observed, in considering only those placement opportunities that arise in perfectly repeated strings of accesses this approach can miss important relationships *between* streams, and may perform especially poorly when its stream formation threshold artificially separates otherwise related objects (see Section 5).

2.3 Summary

Existing solutions to improving heap-object locality show significant promise, but suffer from a wide range of issues including poorly fitting models, expensive runtime-identification techniques, and complications surrounding fragmentation behaviour. To better illustrate one such class of issues, we next consider how existing schemes perform on a real-world benchmark, motivating the need for a new type of profile-guided allocator, which we then proceed to develop.

3 Motivation

Figure 2 shows a heavily simplified version of a pattern arising in povray from SPEC CPU2017 [32]. The code reads tokens from an input stream, allocating a heap object for each through a procedure specific to the token type. When the entire input is consumed, the program then traverses some subset of these objects, in this case accessing objects of types A and B while leaving aside those of type C.

A typical allocator might place these objects as shown in Figure 3(a). However, this lays out objects purely according to the order in which they were allocated, when the way in which they are actually accessed is heavily dependent on their type. This detail, to which traditional allocators are oblivious, means the allocator unwittingly produces a layout with poor spatial locality, scattering unrelated objects of type C between those of types A and B, reducing the effectiveness of the cache and resulting in suboptimal performance.

Locality-improving allocators, in contrast, seek to utilise exactly these kind of out-of-band regularities to make improved data layout decisions. In this case, such an allocator could notice that since each of the three types of allocations arises from a separate invocation of malloc (within each of the create_* procedures), each is likely to have different access characteristics. Analysing the nature of these characteristics using a profiling run, the allocator could then divert allocations of types A and B to a separate memory pool from those of type C, as illustrated in Figure 3(b). In theory, this ought to allow the access loop in Figure 2 to operate without bringing any objects of type C into the cache, improving cache efficiency and thus performance.

Unfortunately, while existing solutions can tackle inefficiencies in simple examples like this with ease, real programs can introduce additional complexities that can ultimately result in their failure. If we examine the example from Figure 2 in its original context in povray, for instance, where types A and B correspond to geometry objects such as planes and CSG composites, we see that almost all heap data is allocated through a wrapper function, pov::pov_malloc, thwarting approaches that look to characterise allocations using only

```
// Allocate
Object *list = NULL;
while (!eof) {
        Token token = get_token();
        if (token.type == A) {
                Object *obj = create_a();
                obj->sibling = list;
                list = obj;
        } else if (token.type == B) {
                Object *obj = create_b();
                obj->sibling = list;
                list = obj;
        } else {
                Object *obj = create_c();
                do_something(obj);
        }
}

// Access
Object *obj = list;
while (obj) {
        process(obj);
        obj = obj->sibling;
}
```

Figure 2. A simple C program based on code from `povray` which allocates three different types of objects on the heap.

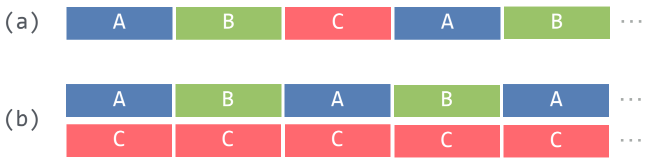


Figure 3. Two possible heap layouts for the data in Figure 2.

the call site to `malloc`. Further, even if these wrapper functions are stripped, geometry objects allocated from different places in the program are used in different ways, with some types accessed far less heavily than others. As such, detailed context information is required in order to segregate objects most efficiently, beyond that which can be reasonably extracted by walking the dynamic call stack at runtime.

This motivates the design of a new solution that can resolve some of the issues described with prior work and can more accurately characterise different types of allocations in large, complex programs. To this end, we propose the **H**eap **A**llocation **L**ayout **O**ptimiser (HALO). Profiling the target binary to understand how allocations made in different contexts are related, HALO specialises memory-management routines to allocate groups of related objects from separate pools in order to increase their spatial locality. Unlike other solutions of its kind, HALO employs novel grouping and identification algorithms which allow it to create tight-knit allocation groups using the *entire* call stack and to identify these at runtime with extremely low overhead.

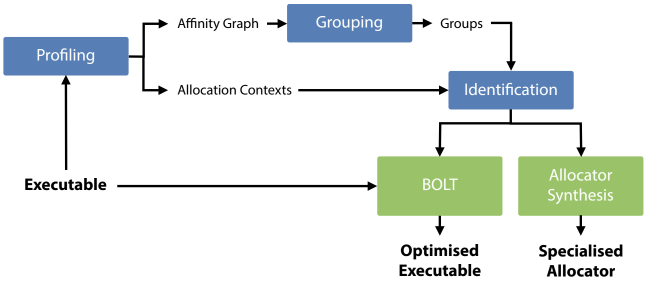


Figure 4. A high-level overview of the HALO optimisation pipeline.

4 Design and Implementation

A high-level overview of HALO's design is presented in Figure 4. Broadly, the target program is first *profiled* to construct a representation of how the allocations it makes in different contexts are related. It then has its allocation contexts *grouped* into tight-knit clusters that might benefit from improved spatial locality, undergoes *identification* and is rewritten using BOLT [27] to allow these groups to be efficiently identified at runtime, and has a *specialised allocator* generated that it should be linked against at runtime to allow a new group-centred layout policy to be enforced.

4.1 Profiling

To begin, the target program is run under a custom instrumentation tool written using Intel's Pin framework [24] to generate a model of how it accesses heap data. To determine how allocations are related, this tool instruments calls to all POSIX.1 memory-management functions, tracking live data at an object-level granularity. While the overhead of this can be considerable, slowing execution by up to 500×, we do not apply any optimisations to this process, such as sampling, to trade off accuracy against a faster profiling stage.

For each allocation, the tool keeps track of the *context* in which the allocation was made by way of the call stack. To do this, it maintains a *shadow* stack that differs from the true call stack by design. For each call instruction (or other cross-function control transfer), we add an entry to this stack *only* if the target of the call is statically linked into the main binary, or is one of a handful of externally traceable routines like `malloc` or `free`. Within the shadow stack, the tool then tracks the exact call sites from which each function was invoked. In order to prevent these from referring to undesirable locations such as linker stubs and library procedures, however, call sites may be *indirect*, and are traced back to their nearest points of origin in the main executable. In addition, stacks containing recursive calls are transformed into a canonical 'reduced' form in which only the most recent of any (function, call site) pair is retained to avoid overfitting without imposing any fixed size constraints.

Having established which allocations are being made and in what context, our instrumentation tool then looks to

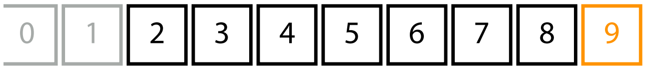


Figure 5. A visual depiction of the affinity queue. In this example, a program iterates over 10 objects making 4-byte accesses, each depicted as a box. With $A = 32$, the newest element (orange) would be considered affinitive to the seven others to its left (black).

model the relationships *between* contexts by analysing the target program's access patterns. To this end, we generate a *pairwise affinity graph*, the nodes of which are 'reduced' contexts as described above, and the edges of which are weighted according to the number of contemporaneous accesses to objects allocated *from* these within some fixed window.

As the target program executes, its loads and stores are instrumented. When an access is made to a heap object tracked by the tool, this access is added to the *affinity queue*. Depicted in Figure 5, this holds the IDs of the most recently accessed data objects. We describe any pair of elements for which the sizes of the entries between them in the affinity queue sums to less than A bytes as being *affinitive*, where A is a parameter which we call the *affinity distance* by which the queue is implicitly sized. When an access a to a heap object u allocated from context x is added to the affinity queue as a result of a load or store, the queue is traversed to identify all affinitive relationships with this new access. As a result, the weight on the edge (x, y) is incremented for each object v allocated from context y in the last A bytes worth of accesses in the queue, subject to the following four constraints:

Deduplication: consecutive machine-level accesses to a single object are considered to be part of the same macro-level access, and thus do not re-trigger queue traversal.

No self-affinity: objects cannot be affinitive to themselves ($u \neq v$), as they occupy only a single memory location.

No double counting: each unique object v can be affinitive with u at most once within a single queue traversal.

Co-allocatability: no allocations made between u and v chronologically can originate from either x or y.

Of these, co-allocatability is perhaps the least intuitive. Inspired by the 'co-allocation sets' employed by Chilimbi and Shaham [11], this ensures it would be possible to actually co-locate u and v at runtime if all objects originating from x and y were allocated contiguously from a shared pool.

Once the target program has finished executing, the nodes of its affinity graph are iterated through from most to least accessed. In doing so, their access counts are added to a running total, and after 90% of all observed accesses have been accounted for, any remaining nodes are discarded and do not contribute to the generated graph. This helps to reduce noise by eliminating extraneous contexts from the graph.

```
def group(graph, args):
    groups ← $\varnothing$
    graph.edges ← $\{ (u, v) \mid (u, v) \in$ graph.edges,
                  $w(u, v) \geq$ args.min_weight $\}$
    avail ← graph.nodes
    while avail $\neq \varnothing$:
        # Form a group around the hottest node
        # in the strongest available edge
        edge ← $\text{argmax}_{(u,v) \in \texttt{graph[avail].edges}}(w(u, v))$
        group ← $\{ \text{argmax}_{u \in \texttt{edge}}(u.\text{accesses}) \}$
        avail ← avail \ group

        # Grow the group
        while |group| < args.max_group_members:
            best_score ← 0.0
            best_match ← None
            for stranger $\in$ avail:
                benefit ← merge_benefit(graph, group,
                  stranger, args.merge_tol)
                if benefit > best_score:
                    best_score ← benefit
                    best_match ← stranger
            if best_match is None:
                break
            group ← group ∪ best_match
            avail ← avail \ group

        # Add the group to the list if it
        # exceeds the minimum group weight
        weight ← $\sum_{(u,v) \in \texttt{group.edges}} w(u, v)$
        if weight ≥ graph.accesses × args.gthresh:
            groups ← groups ∪ group
    return groups
```

Figure 6. Pseudocode for our context grouping algorithm, in which w is the affinity graph's edge weight function and `merge_benefit` is calculated using the formula in Figure 8.

4.2 Grouping

Having generated a representation of the actionable temporal relationships in the target program, we must now devise some scheme by which these relationships can be exploited to yield improved performance. For this purpose, we partition the set of allocation contexts into *groups*, such that members of each group can be allocated from a common memory region to improve cache locality. In order to establish such groups of related allocations, we describe a simple greedy algorithm that generates clusters we find to be more amenable to region-based co-allocation than standard modularity [26], HCS [17], or cut-based clustering techniques. The pseudocode for this algorithm is presented in Figure 6.

At a high level, the algorithm operates by repeatedly growing tight-knit clusters around the most promising opportunities in the graph. Starting with one of the two nodes that participate in the strongest ungrouped edge, a singleton group is formed. This group is then cultivated by considering

$$s(G) = \frac{\sum_{(u,v)\in E} w(u,v)}{|L| + |V|(|V|-1)/2}$$
$$\text{where } L = \{\, (u,v) \mid (u,v) \in E, u = v, w(u,v) > 0 \,\}$$

Figure 7. The score function s by which we evaluate group quality, where $G = (V, E)$ is the input graph, for which w is the associated weight function.

$$m(A,B) = S_c - (1-T)\max(S_a, S_b)$$
$$\text{where } S_a = s(G[A]),\ S_b = s(G[B]),\ S_c = s(G[A \cup B]))$$

Figure 8. The merge-benefit function m by which we evaluate whether a candidate node should be merged into a group, where G is the input graph and the square bracket operator yields the subgraph containing only the specified nodes.

each remaining ungrouped node in turn and calculating the 'merge benefit' of adding this candidate to the group. The node with the largest merge benefit is selected for addition, and this process continues until the merge benefit is less than or equal to zero, after which the group is considered complete and another group is formed starting from the next ungrouped node with the strongest edge. While the asymptotic complexity of this process is quadratic in the number of nodes in the graph, this value is unlikely to grow large and can be made arbitrarily small through filtering.

In order to generate high-quality groups, the merge-benefit metric is carefully designed such that if a candidate node is not well-connected enough to other nodes in the group, or is better off in a group of its own, the merge operation will not take place. To this end, merge benefit quantifies the quality of a given group via its *score* (Figure 7). This is a variant of *weighted graph density*, and thus encourages the formation of tight-knit groups with strong inter-member connections. As the standard formulation of weighted density does not account for loop edges, however, our score metric is a variation on this that distributes weight among loops *only* when they are present in the graph. In combination with the edge thresholding that we apply to reduce noise, we find this provides an effective objective function by which to guide grouping decisions.

With this score, merge benefit is calculated as in Figure 8 to give a positive value only if merging is beneficial to both parties. In order for a candidate node B to be considered beneficial, the graph formed by its addition to an existing group A must produce a higher score than either the group or the candidate node in isolation. The only exception to this rule is if the score of the combined graph is only fractionally lower than that of the separated graphs, in which case merging is permitted to encourage group formation. Without this proviso, merging behaviour would be too strict, and the majority of groups would consist only of one or two nodes around the strongest edges. This slack in the merge-benefit calculation can be controlled through the tolerance parameter T, which we find performs well at around 5%.

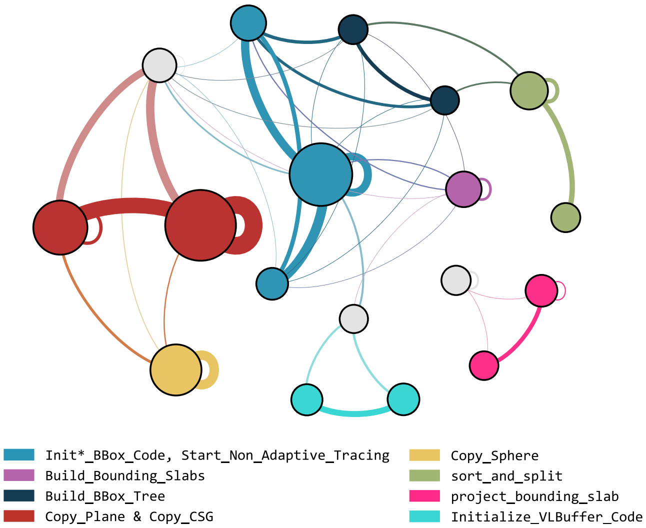


Figure 9. Allocation groups generated by instrumenting the povray test workload from SPEC CPU2017 [33]. Edges with weight less than 200,000 are hidden to reduce visual noise.

Figure 9 shows the groups formed by applying this process to the povray test workload from SPEC CPU2017. In this, each node corresponds to a single allocation context, coloured according to its group, with the thickness of the edges between nodes denoting weight. Nodes marked in grey are ungrouped due to insufficient merge benefit. In spite of the complexity of the access patterns in this case, our algorithm produces groups with high semantic value — grouping, for example, highly related allocations from Copy_CSG and Copy_Plane, as was observed to be beneficial in Section 3.

4.3 Identification

Having established a number of groups around which data placement decisions can be made, we now devise a scheme by which these groups can be identified at runtime. While much of the existing work in this area relies on the dynamic call stack for this purpose, our solution employs a different approach, utilising binary rewriting to identify group membership based on *control-flow* behaviour around a handful of salient locations in the target program.

More specifically, our solution determines group membership on the basis of *selectors*: logical expressions which determine whether or not a particular allocation belongs to a particular group based on whether the flow of control has passed through a certain set of call sites. To generate these, we define a simple greedy algorithm, the pseudocode for which is listed in Figure 10. At its core, this builds up selectors in disjunctive normal form by combining conjunctive expressions to distinguish each of a group's members from unrelated contexts. In spite of its simplicity, and the

```
def identify(groups, contexts):
    ignore ← ∅
    for group ∈ sort_by_popularity_desc(groups):
        ignore ← ignore ∪ group.id

        # Construct a selector to identify
        # members of this group
        selector ← λx.False
        for member ∈ group.members:
            # Build an expression to identify
            # this group member
            expr ← λx.True
            conflicts ← ∞
            while conflicts:
                chains ← { $c$.chain | $c$ ∈ contexts,
                                      expr($c$.chain),
                                      $c$.group ∉ ignore }
                opts ← { (addr, $\sum_{c \in \texttt{chains}}$ [addr ∈ $c$ ]) |
                          addr ∈ member }
                opts ← { $(a, m)$ | $(a, m)$ ∈ opts,
                          $\nexists\, (b, n) \in$ opts, $n = m \implies$
                          $a$ is lower in the stack than $b$ }
                (site, m) ← $\operatorname{argmin}_{(a,m) \in \texttt{opts}}(m)$

                # Add the new constraint only if
                # it reduces conflicts
                if m = conflicts:
                    break
                expr ← λx.(expr($x$) ∧ (site ∈ $x$))
                conflicts ← m
            selector ← λx.(selector($x$) ∨ expr($x$))
        selectors ← selectors ∪ selector
    return selectors
```

Figure 10. Pseudocode for our group-identification algorithm.

sub-optimality that can result from considering the conjunctive expressions for each group member independently, we find its results to be more than sufficient for our prototype.

To capture and act upon the control-flow behaviour around these call sites, we then rewrite the target binary using the BOLT post-link optimisation framework [27]. Constructing a custom pass specifically for heap-layout optimisation, we insert instructions around every point of interest in the target binary, setting and then unsetting a single bit in a shared 'group state' bit vector to indicate whether the flow of control has passed through this point.

4.4 Allocation

Finally, having established several groups of related allocation contexts and a state vector through which these can be identified, we generate a specialised allocator that can act upon this information to co-locate data at runtime. The high-level design of this allocator is depicted in Figure 11.

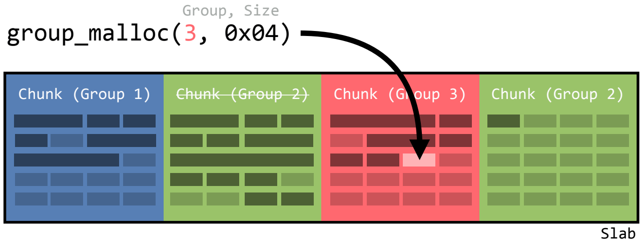


Figure 11. A visual depiction of the allocation strategy employed in HALO's specialised group allocator, which combines the efficiency and contiguity guarantees of bump allocation with a chunk-based reuse model.

After the allocator is loaded into memory, its first task is to locate the address of the group state vector described in the previous section. When an allocation request is made, this state is used to determine whether or not the allocation belongs to a group. To do this, the allocator compares the size of the allocation with the maximum grouped object size, and checks the contents of the group state vector against the set of selectors obtained in group identification. If the allocation does not belong to a group, and thus should not be satisfied by our specialised allocator, it is forwarded to the next available allocator through `dlsym`. If, however, the allocation size is less than the page size and the group state vector matches a group selector, the allocation is satisfied using our `group_malloc` and `group_free` procedures.

Memory is reserved from the OS in large, demand-paged *slabs* to amortise `mmap` costs, and is managed in smaller group-specific *chunks* from which *regions* can be allocated. Whenever a grouped allocation is made, the allocator first attempts to reserve a region of the requested size and alignment from the 'current' chunk associated with the group. This occurs through straightforward bump allocation with no per-object headers, and thus guarantees contiguity between the vast majority of consecutive grouped allocations. If the chunk has insufficient remaining space or if this is the first allocation from a given group, a new chunk is carved out from the current slab and is assigned as the 'current' chunk for the target group. In turn, if the current slab has insufficient remaining space, a new slab is reserved from the OS and assigned as the current slab. All allocations are made with a minimum alignment of 8 bytes [20].

Whenever a region is freed or reallocated, the allocator must determine whether this region was originally group allocated, or whether the free request should be forwarded to the default allocator. In the case that a region being freed was group allocated, its reservation is released from its respective chunk by way of the chunk's header. As chunks are always aligned to their size in memory, the header of any particular chunk can be trivially located from a region pointer by way of simple bitwise operations. In the case of a `free` operation, the `live_regions` field of the header, which is incremented after every allocation from a given chunk, is decremented. If, after doing so, its value is zero, the chunk is empty, and thus can be reused or freed.

As regions are always reserved from chunks using bump allocation, encouraging contiguity over compaction, fragmentation behaviour in our allocator is dependent solely on the chunk size and the degree to which consecutive grouped allocations have similar lifetimes. In the worst case, if a chunk is put into a state where it is empty aside from a single region, almost the entirety of the chunk will be kept unused as external fragmentation. This could prove to be problematic in some use cases. We do not believe, however, that this is a fundamental limitation of this kind of approach. There are many areas in which our prototype could be more sophisticated, and fragmentation behaviour is just one of these. One could equally imagine extending the model described here to support multi-threaded allocations [14], to reduce allocator-induced conflict misses [1], or to employ more sophisticated dirty-page purging techniques [13].

5 Evaluation

Having described the design of a new tool to improve the cache locality of heap data, we now set out to evaluate how well this tool works. To this end, this section describes a series of experiments to evaluate the practical performance of HALO and compares the results to those from a replication of a high-performing approach from the literature.

5.1 Experimental Setup and Methodology

Benchmarks Focusing particularly on the SPECrate CPU 2017 benchmark suite [33], we examined the behaviour of a number of candidate programs to select a small subset by which our prototype should be measured. From this investigation, we produced a list of 11 programs around which the performance of our solution is evaluated. The first five, `povray`, `omnetpp`, `xalanc`, `leela`, and `roms`, are the SPECrate CPU2017 benchmarks that made, on average, more than one heap allocation per million instructions in their `train` workloads. Due to lack of space, this excludes a handful of benchmarks that were unaffected by either of the optimisations we examine. Following this, the remaining six workloads are programs that showed stand-out opportunities for layout improvement in prior work [22, 36]. These include `health`, `ft`, and `analyzer` from the Olden [29], Ptrdist [2], and FreeBench [30] suites, as well as `ammp`, `art`, and `equake` from the SPEC CPU2000 suite [32].

All workloads are compiled with the `-g -O3 -fno-tree-loop-vectorize -fno-unsafe-math-optimizations` compiler flags. We also utilise the `-no-pie -falign-functions=512` flags in accordance with current limitations of our BOLT pass, and omit the `-march=native` flag to avoid generating code which BOLT cannot disassemble. Workloads that make use of custom allocators are compiled without modification, and as such have many of their regularities obscured.

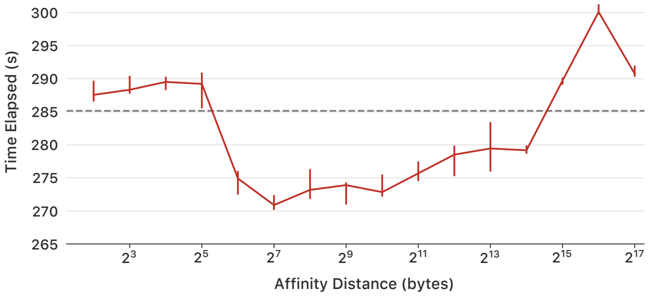


Figure 12. Time taken by `omnetpp` at various affinity distances. The dashed line indicates the median execution time of the original unmodified program under `jemalloc`.

All workloads are compiled with gcc 7.3.0, rewritten using our custom variant of BOLT (based on `e37d18e`, with all other optimisations disabled), and run in single-threaded configurations on a 64-bit Xeon® W-2195 running Ubuntu 18.04.2 LTS with 32KiB per-core L1 data caches, 1,024KiB per-core L2 caches, and a 25,344KiB shared L3 cache. Workloads are profiled on small `test` inputs and measured using larger `ref` inputs, mirroring the methodology employed by Chilimbi and Shaham [11] and preventing profiling runs from consuming excessive compute resources.

HALO Configuration To configure HALO for evaluation, we set the parameters empirically as described in Section 4. Binaries are profiled with a maximum grouped-object size of 4KiB, undergo allocation context grouping with a merge tolerance of 5%, and have their grouped allocations reserved at runtime from 1MiB chunks. With the exception of `omnetpp` and `xalanc`, for which group chunks are always reused due to a limitation of our current implementation, all benchmarks have their specialised allocators configured to keep a single spare chunk for reuse when purging dirty pages, as early versions of `jemalloc` did [13]. In order to establish a reasonable value for the affinity distance parameter, we examined the performance of `omnetpp` at various power-of-two parameter values. The results of this experiment are shown in Figure 12. In light of this, we select an affinity distance of 128 for use in our evaluation, as this appears to yield reasonable performance gains at a relatively low profiling overhead.

Comparison Technique To establish how our solution performs in comparison to other similar techniques, we also examine the performance of each benchmark under the hot-data-stream-based co-allocation technique described by Chilimbi and Shaham [11]. To this end, we utilise the same specialised allocator as HALO, but with groups that are generated through hot-data-stream analysis and identified at runtime using the immediate call site of the allocation procedure. In replicating this technique, we attempt to mirror the approach described in the original paper as much as possible. As such, we configure our analysis to detect minimal hot data streams that contain between 2 and 20 elements, with the stream threshold set to account for 90% of all heap accesses. Hot data streams are extracted using a modified SEQUITUR implementation [25], and are converted into allocation groups using our own implementations of the algorithms described in the original paper and its related work [8, 16, 21]. This process has been validated against the examples in the original paper, as well as a small test suite, on which we find it performs as expected.

Measurement For each unique configuration of our benchmarks, we run 11 trials, discarding the results of the first to ensure that the system is in a steady state prior to measurement. Measured characteristics include time taken, L1 data-cache misses, and fragmentation behaviour. All reported figures represent the median of the 10 recorded trials, with error bars calculated using the 25th and 75th percentiles. We do not present geometric or arithmetic averages across our results, as we believe that these would provide little informational value in light of our rather eclectic mix of workloads.

Performance characteristics are measured for our prototype system, our hot-data-stream-based system, and a baseline configuration using the unmodified binary. Unlike much existing work in this area, all configurations use `jemalloc` 5.1.0 as the default allocator. Initial experiments show that this universally outperforms `ptmalloc2` from glibc 2.27, reducing L1 data-cache misses by as much as 32%, and thus provides a more aggressive baseline against which to measure the benefits of cache-conscious heap-data placement.

5.2 Results

Figures 13 and 14 show the high level results of our performance evaluation, showing that HALO consistently outperforms our comparison technique, especially for more modern workloads. To explain these results in detail, we first focus on the hot-data-stream-based approach described by Chilimbi and Shaham [11].

Hot Data Streams Under this scheme, the six benchmarks from prior work see a sizeable reduction to both cache misses and execution time, while the five more recent benchmarks see little improvement, or in some cases slight degradation. As such, while some benchmarks yield performance improvements in line with those reported in the original work, others yield results that are at best somewhat disappointing.

The reason for this, it seems, largely comes down to the fixed-sized contexts through which this solution characterises heap allocations. In most cases, the six benchmarks from the existing literature invoke `malloc` directly from domain-specific code, and do so in relatively distinct locations with minimal surrounding abstractions. As such, they represent relatively easy targets around which semantically valuable allocation groups can be formed. In contrast, the benchmarks from the CPU2017 suite are typically more complex. `xalanc`, for instance, displays significant indirection its call chains, requiring the traversal of tens of stack frames to properly appreciate the context in which allocations have been made. `povray`, meanwhile, allocates a significant portion of its heap data through wrapper functions, while `leela` allocates memory exclusively through C++'s `new` operator. These complexities impede the naïve identification mechanisms employed by the hot-data-stream-based solution, preventing it from achieving any meaningful improvement on these benchmarks.

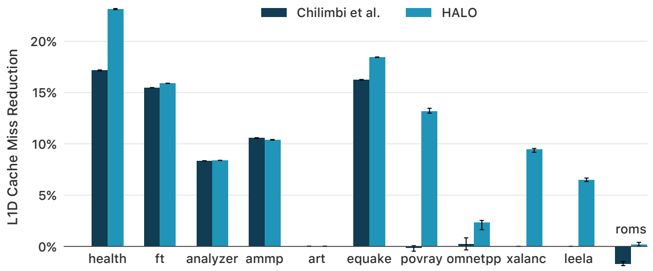


Figure 13. The percentage by which both HALO and hot-data-stream-based co-allocation [11] reduce L1 data-cache misses across a range of 11 programs.

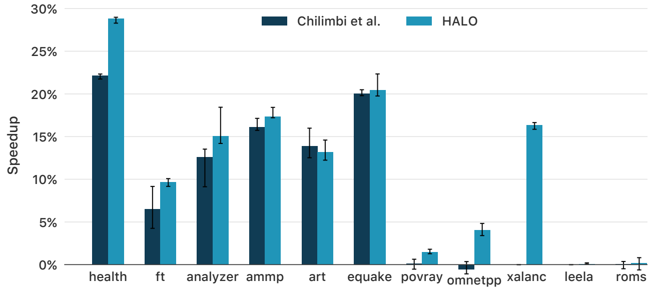


Figure 14. The percentage by which both HALO and hot-data-stream-based co-allocation [11] improve execution time across a range of 11 programs.

The exception to this rule is `roms`, which we find tends to call `malloc` directly, and thus lends itself much better to characterisation by this solution. In this benchmark, however, we actually uncover a more significant problem. While HALO's affinity graph can represent over 90% of all salient accesses in this program using only 31 nodes, the hot-data-stream-based approach requires over 150,000 streams.

This suggests a significant downside in the methodology described by Chilimbi and Shaham [11] in which even highly regular programs may require many streams to represent their behaviour. By capturing access patterns at an *object level* granularity, this approach can inadvertently scatter regularities at the *allocation context* level across many hot data

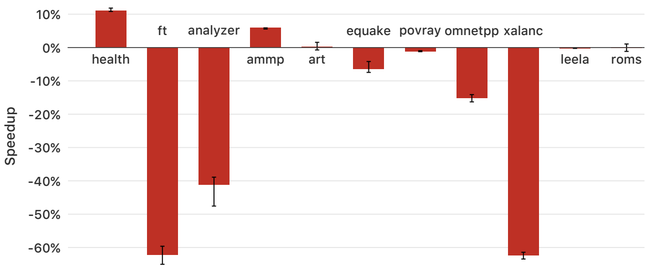


Figure 15. The percentage by which an allocator that randomly assigns small objects to one of four bump allocated pools improves execution time across a range of 11 programs.

streams. If any patterns of this kind make up a large proportion of a program's access trace, the stream formation threshold must be significantly reduced in order to capture the many streams amongst which the pattern is distributed. As a direct result, the hot data streams for other areas of the program's behaviour may be cut short, and their corresponding co-allocation sets rendered near-useless. A similar problem arises in programs with large, widely accessed objects, in which such objects cause almost any access pattern in which they are present to exceed the stream formation threshold, and thus to immediately terminate. This behaviour may explain why the hot-data-stream-based solution actually *increases* L1 data-cache misses for `roms` in Figure 13. The truncated co-allocation sets that it produces as a result of a deflated stream threshold may separate data that would otherwise naturally be co-located by a size-segregated allocator.

HALO HALO, in contrast, utilises a more robust representation of reference locality and more sophisticated strategies to cut through deep abstractions. As such, it performs well not only on the six programs from the existing literature, but also on the large, complex programs from the CPU2017 suite — producing a roughly 4% speedup in `omnetpp` and 16% speedup in `xalanc`. This all despite the fact that some benchmarks, like `xalanc`, already utilise custom allocators, and as such prevent HALO from exploiting many of their regularities. Even for the benchmarks on which the hot-data-stream-based approach performs best, we find that HALO can achieve better results. In `health`, for example, extracting an additional seven percentage points of improvement in execution time, bringing total speedup to around 28%. This additional improvement is a product of HALO's full-context identification mechanism, which can extract better allocation groups from more specific context information.

In other cases still, HALO is able to achieve a sizeable reduction in L1 data-cache misses even when these do not result in significant reductions to execution time. `povray` and `leela`, for example, display roughly 5–15% fewer misses under HALO, compared with only around 2% using hot data streams, even though their overall execution times remain largely unchanged by the optimisation. While we have not examined the source of this behaviour in detail, we suspect that these benchmarks may be primarily compute rather than memory bound. In more realistic environments with greater external cache pressure, or on less sophisticated machines, the observed speedups may be significantly larger.

Table 1. Fragmentation behaviour of grouped objects at peak memory usage across nine benchmarks.

Benchmark	**Frag. (%)**	**Frag. (bytes)**
`health`	0.01%	31.98KiB
`equake`	0.05%	12.08KiB
`analyzer`	0.13%	4.31KiB
`ammp`	0.20%	40.97KiB
`art`	0.62%	11.70KiB
`ft`	2.06%	4.05KiB
`povray`	26.47%	37.06KiB
`roms`	93.60%	29.95KiB
`leela`	99.99%	2.05MiB

For other benchmarks, meanwhile, such as `roms`, and for almost all of the SPEC CPU2017 benchmarks we examined outside of those shown in Figure 13, we find that HALO has essentially no effect. Critically, however, its optimisations do not degrade performance in these cases, but rather simply fail at improving it. In examining the performance of a configuration in which each BOLT-instrumented binary is run *without* its specialised allocator, we find that noise from the surrounding system is far greater than the effects of HALO's instrumentation. Instrumentation overhead, then, appears not to be responsible for the failure to improve performance.

Instead, it seems HALO performs poorly in these cases simply because these programs are not terribly sensitive to the placement of small heap objects. To illustrate this, Figure 15 shows the results of running each benchmark under an allocator that randomly allocates objects smaller than the page size from four 'groups', much in the same way that a variant of HALO with an extremely poor grouping algorithm might. The benchmarks with the largest change in behaviour in response to this rather extreme allocation policy align well with the benchmarks for which our technique proves most effective. The benchmarks unfazed by it, meanwhile, are generally those for which the placement of small heap objects appears not to be terribly important, and thus for which our layout decisions are of little consequence.

Fragmentation Behaviour Table 1 shows the relationship between live and resident data at peak memory usage in our specialised allocator, listed for each of the benchmarks where it could be easily examined. While the percentage of unused resident memory in our allocator can far exceed what is typically considered reasonable — averaging at above 30% — the absolute number of bytes wasted in each case is actually relatively small. As grouped data objects typically make up a small fraction of all allocations, overall fragmentation behaviour in each case should see only a marginal decline. Nonetheless, behaviour in this area is not ideal, and this potential shortcoming provides fruitful ground for future work.

6 Conclusion

General-purpose memory allocators provide a convenient, one-size-fits-all solution to the problem of dynamic memory management. However, without intimate knowledge of how values are used in a program, they can rely only on simple heuristics to place data, and may inadvertently scatter highly related data throughout the heap, generating unnecessary cache misses and load stalls. To help alleviate these issues, we developed HALO: a post-link profile-guided optimisation tool and runtime memory allocator that can intelligently rearrange heap data to reduce cache misses. Unlike other solutions of its kind, HALO employs novel grouping and identification algorithms which allow it to create tight-knit allocation groups using the entire call stack, and to identify these groups at runtime with extremely low overhead.

Evaluation of HALO on contemporary out-of-order hardware demonstrates speedups of up to 28% over `jemalloc`, out-performing a state-of-the-art placement technique from the literature. In many cases HALO's layout optimisations combat inefficiencies that would be difficult to find manually, such as those shrouded in abstraction, that arise from the interactions between components that the programmer may not directly control, or that occur due to common characteristics of typical inputs. Future work could explore how additional program state could contribute to allocation characterisation decisions, how the grouping and allocation algorithms could be improved to more accurately model and exploit the expected cache behaviour of generated groups, or how techniques such as free list sharding [23] and meshing [28] could be used in place of bump allocation to improve practical fragmentation behaviour.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1 and EP/P020011/1. Additional data related to this publication is available in the data repository at https://doi.org/10.17863/CAM.46071.

References

[1] Yehuda Afek, Dave Dice, and Adam Morrison. 2011. Cache Index-aware Memory Allocation. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 55–64.

[2] Todd Austin. 1995. *The Pointer-intensive Benchmark Suite*. http://pages.cs.wisc.edu/~austin/ptr-dist.html

[3] 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 (PLDI '93)*. ACM, New York, NY, USA, 187–196.

[4] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2001. Composing High-performance Memory Allocators. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 114–124.

[5] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering Custom Memory Allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 1–12.

[6] Brad Calder, Dirk Grunwald, and Benjamin Zorn. 1994. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages* 2, 4 (1994), 313–351.

[7] 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 (ASPLOS VIII)*. ACM, New York, NY, USA, 139–149.

[8] Trishul M. Chilimbi. 2001. Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 191–202.

[9] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. 1999. Cache-conscious Structure Definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, New York, NY, USA, 13–24.

[10] 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 (PLDI '99)*. ACM, New York, NY, USA, 1–12.

[11] Trishul M. Chilimbi and Ran Shaham. 2006. Cache-conscious Coallocation of Hot Data Streams. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 252–262.

[12] Jason Evans. 2006. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of BSDCan 2006*.

[13] Jason Evans. 2015. Tick Tock, Malloc Needs a Clock. In *Applicative 2015 (Applicative 2015)*. ACM, New York, NY, USA.

[14] Sanjay Ghemawat. 2007. *TCMalloc: Thread-Caching Malloc*. https://gperftools.github.io/gperftools/tcmalloc.html

[15] Jason Gregory. 2009. *Game Engine Architecture*. A K Peters.

[16] Magnús M. Halldórsson. 1999. Approximations of Weighted Independent Set and Hereditary Subset Problems. In *Computing and Combinatorics*, Takano Asano, Hideki Imai, D. T. Lee, Shin-ichi Nakano, and Takeshi Tokuyama (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 261–270.

[17] Erez Hartuv and Ron Shamir. 2000. A clustering algorithm based on graph connectivity. *Inform. Process. Lett.* 76, 4 (2000), 175–181.

[18] Alin Jula and Lawrence Rauchwerger. 2009. Two Memory Allocators That Use Hints to Improve Locality. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. ACM, New York, NY, USA, 109–118.

[19] Thomas Kistler and Michael Franz. 2000. Automated Data-member Layout of Heap Objects to Improve Memory-hierarchy Performance. *ACM Trans. Program. Lang. Syst.* 22, 3 (May 2000), 490–505.

[20] Bradley C. Kuszmaul. 2015. SuperMalloc: A Super Fast Multithreaded Malloc for 64-bit Machines. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 41–55.

[21] James R. Larus. 1999. Whole Program Paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, New York, NY, USA, 259–269.

[22] Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the

Heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 129–142.

[23] Daan Leijen, Ben Zorn, and Leonardo de Moura. 2019. *Mimalloc: Free List Sharding in Action*. Technical Report MSR-TR-2019-18. Microsoft. https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200.

[25] C. G. Nevill-Manning and I. H. Witten. 1997. Linear-time, incremental hierarchy inference for compression. In *Proceedings of the 1997 Data Compression Conference (DDC '97)*. 3–11.

[26] M. E. J. Newman and M. Girvan. 2004. Finding and evaluating community structure in networks. *Phys. Rev. E* 69 (Feb 2004), 026113. Issue 2.

[27] 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, Piscataway, NJ, USA, 2–14.

[28] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. 2019. Mesh: Compacting Memory Management for C/C++ Applications. *CoRR* abs/1902.04738 (2019). arXiv:1902.04738 http://arxiv.org/abs/1902.04738

[29] 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 (March 1995), 233–263.

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

[31] Matthew L. Seidl and Benjamin G. Zorn. 1998. Segregating Heap Objects by Reference Behavior and Lifetime. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. ACM, New York, NY, USA, 12–23.

[32] Standard Performance Evaluation Corporation. 2000. *SPEC CPU2000*. https://www.spec.org/cpu2000

[33] Standard Performance Evaluation Corporation. 2017. *SPEC CPU2017*. https://www.spec.org/cpu2017

[34] D. N. Truong, F. Bodin, and A. Seznec. 1998. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. 322–329.

[35] 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 (CGO '10)*. ACM, New York, NY, USA, 92–100.

[36] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. 2005. Dynamic Memory Optimization Using Pool Allocation and Prefetching. *SIGARCH Comput. Archit. News* 33, 5 (Dec. 2005), 27–32.

A Artefact Appendix

A.1 Abstract

The artefact submitted in association with this report is composed primarily of three elements: the source code for the HALO optimisation pipeline, a small test program on which its functionality can be validated, and scripts to assist in applying our optimisations and extracting performance results. With these, we allow for the reproduction of the primary performance results presented in Figures 13 and 14, as well the execution of custom experiments, including those that build on or modify our technique.

A.2 Artefact check-list (meta-information)

- **Algorithm:** HALO profile-guided heap-layout optimisation.
- **Program:** Benchmarks from SPEC 2000, SPEC 2017, Olden, Ptrdist, and FreeBench (not provided).
- **Transformations:** BOLT binary rewriting.
- **Run-time environment:** The only supported environment is x86-64 Linux. Dependencies include `perf`, `patchelf`, and Python 2.7.
- **Hardware:** Reported results were obtained using an Intel Xeon® W-2195 with 32KiB per-core L1 data caches, 1024KiB per-core unified L2 caches, and a 25344KiB shared L3 cache. Similar systems should yield comparable results.
- **Output:** Provided scripts can generate speedup and L1 data cache miss reduction graphs against a `jemalloc` baseline, corresponding to the light blue bars in Figures 13 and 14.
- **Experiments:** Python scripts are provided to automate the process of applying the entire optimisation pipeline, taking performance measurements, and plotting results. Some variation from the reported results is expected according to the cache parameters of the system.
- **Disk space:** 10GB should be more than sufficient.
- **Expected preparation time:** Around an hour, excluding any time required to prepare benchmarks and their inputs.
- **Expected run time:** To obtain all results, around a day. Profiling should take less than 20 minutes per benchmark. Most time is consumed by repeated reference runs for performance measurement.
- **Publicly available:** Yes.
- **License:** BSD 3-clause.

A.3 Description

A.3.1 How delivered

The artefact can be downloaded as a tarball from the following URL: https://doi.org/10.17863/CAM.46071

A.3.2 Hardware dependencies

We recommend testing on an x86-64 platform with similar cache parameters to the Intel Xeon® W-2195. Due to a quirk of the current implementation, running programs must be able to map at least 16GiB of virtual memory, overcommited or otherwise. As all evaluated benchmarks are single-threaded, a large number of cores is not necessary.

A.3.3 Software dependencies

Reported results were obtained on Ubuntu 18.04 with glibc 2.27, but any x86-64 Linux system should suffice. We recommend running on a platform of this type, having installed `cmake`, `ninja`, `perf`, `patchelf`, and Python 2.7, in addition to the `numpy`, `pandas`, `networkx`, and `matplotlib` Python packages.

A.4 Installation

As the HALO pipeline involves a number of components developed across separate codebases, including LLVM, BOLT, and Pin, its setup is not completely trivial. To avoid distributing an opaque binary blob to carry out our optimisations, which may not execute the procedure as claimed, we allow the transparent patching and setup of each component to be carried out manually. To make this process as straightforward as possible, a `README.md` file in the root directory of our artefact lists a full set of suggested commands to set up a working HALO environment.

A.5 Experiment workflow

Unlike the version of the artefact presented to the evaluation committee, this publicly-available version does not include binaries or command lines to aid in replicating the exact set of performance results presented in the paper, primarily due to licensing issues. As a result, users are expected to run their own experiments from scratch, following the instructions provided in `README.md`. This describes the usage of the `halo baseline`, `halo run`, and `halo plot` commands, which can be used to carry out the baseline and HALO-optimised runs for each workload and to plot results.

A.6 Evaluation and expected result

Using the procedure described above, JSON files should be generated in the output directory containing the specific data points for each run, and graphs of these generated in PDF format. These can be compared with the corresponding figures in the paper to evaluate the reproducibility of our results. Depending on the parameters of the memory subsystem used for evaluation, improvements of comparable magnitude to those reported should be observed.

A.7 Experiment customisation

In addition to the experiment workflows described in Section 5, different programs and parameters can be tested by varying the flags passed to each `halo` command. For more details, see `README.md` and the full parameter list in the `halo` source file.

A.8 Notes

In order to accommodate quirks in our current implementation, some benchmarks require additional flags to be passed to `halo run` in order to be processed without encountering errors. These include `--chunk-size 131072 --max-spare-chunks 0` for `omnetpp`, `--max-spare-chunks 0` for `xalanc`, and `--max-groups 4` for `roms`.

A.9 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging