# Using Co-Access Information for Modifying Dynamic Memory Allocation Workloads for Improved Cache Performance

Valerie Choung
*Carnegie Mellon University*

Daniel Ramos
*Carnegie Mellon University*

## Abstract

In this paper, we introduce the notion of object co-accesses to improve data placement by dynamic memory allocators. With the key insight that co-accessed objects have strong temporal locality, we construct a workload for the dynamic memory allocator that exhibits both temporal and spatial locality. This improves cache performance for generic programs without modifying underlying cache policies.

## 1 Introduction

Modern systems use a variety of techniques to improve program performance. Much of this revolves around reducing data movement, which is known to be expensive. Some techniques for reducing data movement include caching and memory allocation optimizations. Standard caching techniques take advantage of applications' memory access locality and are an extensive area of research.

In the meantime, general-purpose allocators such as standard *malloc* need to work in a wide variety of cases, allocating chunks of memory whose sizes range from one byte to several gigabytes or even more. However, every program has specific needs, so custom memory allocators written expressly for one application tend to perform better. These custom memory allocators cater to specific workloads that the application is expected to handle.The downside to using custom memory allocators is that the memory allocator must be re-written or re-designed for every new application. Because of the difficulty of writing optimal memory allocators, there are entire papers written on how to design and write them.

Our key insight is that instead of writing a new memory allocator for every application, we can transform an application's workload into one that is naturally more cache-friendly while reducing memory fragmentation. Most implementations of *malloc* are not cache-aware, although there is some work on optimizing *malloc* to reduce cache thrashing for multi-core systems. Our solution is novel because our approach to using *malloc* makes use of application-level semantics to help *malloc* play nice with caching.

To achieve our goal, we introduce the notion of *object co-accesses*, which are accesses to objects with strong temporal locality. Since co-accessed objects have similar lifetimes and access patterns, allocating and caching co-accessed objects together will create a workload that has improved locality. In essence, we would like to use temporal locality to force spatial locality in a program.

### 1.1 Contributions

1. A framework for detecting co-accesses of variables within functions.

2. An optimizer pass for coalescing *mallocs* of constant/dynamic size based on co-access information.

3. Preliminary results showing how coalescing co-accessed objects can affect run-time performance.

## 2 Design

### 2.1 Co-Access

On a high level, we say two objects or variables are co-accessed if an application accesses those variables within a short time duration. This is an interesting property to explore, because objects that are strongly co-accessed exhibit, by definition, strong temporal locality but not necessarily strong spatial locality. Co-accessed objects also share similar access patterns - after all, if two objects are strongly co-accessed, we can expect that an access to one of those objects will be closely followed by an access to the other. With similar reasoning, we can also say that co-accessed objects will have similar lifetimes - when one object in a co-access pair will never be accessed again, we can make a wise guess that the other object in the pair will likewise never be accessed again.

## 2.2 Malloc Transformation

In this paper, we examine a compiler optimization that benefits from having co-accessed data. Our insight is that strongly co-accessed variables generally share the same lifetimes and the same hotness. Using this insight, we expect that allocating strongly co-accessed variables contiguously in memory could benefit performance. In particular, small-enough variables could fit in a single cache line, reducing the number of cache misses. Additionally, less-hot variables would be flushed from the cache together, while hot variables stay in the cache together. Overall, this results in less data movement. This can be done by replacing individual calls to `malloc` for co-accessed variables with a single call to `malloc` that would allocate enough memory for the entire clique of strongly co-accessed variables. Figure 1 illustrates this idea. In this case, the three objects are marked as coaccessed and therefore their memory allocation is coalesced together.

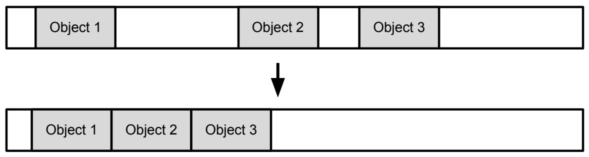

Figure 1: Heap memory allocation transformation

Consider as an example that variables $a$ and $b$ are strongly co-accessed with separate calls to `malloc`, we would replace the calls with a single large `malloc` to obtain a pointer $p$. Every time a is accessed, we dereference $p+n$, where $n$ depends on the size of $a$. This approach has the benefit of reducing calls to `malloc` as well, which should additionally improve program performance.

In principle, coalescing multiple `malloc`'s into one sounds fairly simple, but if the allocation sizes are not known until runtime, we will need to employ a JIT-style dynamic analysis to coalesce the calls to `malloc`.

```
1  a = ?
2  p1 = malloc(a)
3  b = ?
4  p2 = malloc(b)
```

Figure 2: Source code

```
1  a = ?
2  b = ?
3  p1 = malloc(a + b)
4  p2 = p1 + b
```

Figure 3: Target code

For example, consider the source program in Figure 5. Our goal is to coalesce `malloc`'s in lines 1 and 3 to a single malloc, as shown in Figure 3. To make sure this is possible, we need make sure that the computation of $b$ (line 3, Figure 5) does not depend on any operations made on top of $p_1$.

To coalesce mallocs in loops, we also need to do code transformations. For instance, consider the program from Figure 4. Suppose all the $p$'s are co-accessed. If the loop runs $N$ times but $N$ is unknown, we can replace the $i$'th call to `malloc` (if $i$ is a power of 2) with `malloc`$(2^{(i-1)} \cdot C)$ here for worst case $\frac{1}{2}$ utility with $log(N)+1$ calls to `malloc`.

```
1    loop {
2      p = malloc(C)
3      (...)
4    }
```

Figure 4: Example loop with `malloc` of fixed sized C, where all the $p$ variables are co-accessed

# 3 Implementation

## 3.1 Identifying Co-Accesses

There are a few ways we can formally define a co-access. For example, the simplest definition of a co-access could be a pair of addresses that are each dereferenced within N instructions of the other with no exceptions. Even with this simple definition, there are multiple considerations to take when defining a co-access (e.g., what is the size of $N$?, should it be a constant or a variable dependent on types or object sizes?). Alternatively, we can consider co-accesses with a finer-granularity, ignoring if accesses to addresses are within an object.

Another aspect of co-accesses to consider is whether a co-access should be a uni- or bi-directional relation: if an access to variable a is always followed by an access to b, but an access to a does not always precede an access to b, then we need to choose whether we still call the pair (a, b) a co-access. This problem can be partially resolved by giving each co-access a notion of strength. A very strong co-access could be one where two variables are always accessed together and their accesses are very close and very frequent. A weaker co-access would be one where two variables are mostly accessed together, and their accesses are somewhat frequent.

## 3.2 Co-Access Strength

Algorithm 1 illustrates our method for calculating co-access pairs. The algorithm takes as input a basic block, and a sliding window of fixed size (initially without any concrete values), and outputs a list $\mathcal{S}$ of co-access pairs and their respective strength $(v_i, v_j, s_{i,j}), \ldots,$ where $v_i$ and $v_j$ are variables and $s_{i,j}$ is the respective strength. For each instruction in the basic block (line 2), and for each operand in the instruction (line 3), we will strengthen the relation between each operand and the variables in the window. In our implementation STRENGTHEN simply increments a counter. After processing the instruction, we slide the windows SIZE($I$) times (i.e., the number of operands in Instruction $I$). The idea is then to append the new operands to the window (line 10) to be processed in the future iterations.

**Algorithm 1** GENCOACCESSESPAIRS($\mathcal{B}$,$\mathcal{W}$)

```
Input: $\mathcal{B}$: Basic block
Input: $\mathcal{W}$: Window of variables of fixed size
Output: $\mathcal{S}$: Pairs of co-accesses and their strength
 1: $\mathcal{S} := \{\}$
 2: for each $I \in \mathcal{B}$ do
 3:     for each $op \in I$ do
 4:         for each $v \in \mathcal{W}$ do
 5:             $\mathcal{S}$ := STRENGTHEN($\mathcal{S}, v, op$)
 6:         end for
 7:     end for
 8:     $\mathcal{W}$ := SLIDEWINDOW($\mathcal{W}$, SIZE($I$))
 9:     for each $op \in I$ do
10:         $\mathcal{W}$ := APPENDTOWINDOW($\mathcal{W}, op$)
11:     end for
12: end for
13: return $\mathcal{S}$
```

Our current implementation of STRENGTHEN is simply incrementing a counter for the number of times each pair of variables are seen together. However, there are different alternatives such as:

$$Strength(a,b) = \frac{\text{\# accesses to a followed by b}}{\text{\# accesses to a}}$$

We currently have an unmerged pull request for making this change to our co-access strength calculation.

Our algorithm for GENCOACCESSPAIRS in 1 also has a parameter $\mathcal{W}$, so that we can calculate co-accesses that may cross basic block boundaries.

## 3.3 Co-Access Sets

After computing co-access pairs, we can build a notion of co-access sets using transitivity. Our idea is that if two pairs of co-accesses $(v_i, v_k, s_{i,k})$ and $(v_k, v_j, s_{k,j})$ are above a certain certain strength $\mathcal{T}$ (that is, if $s_{i,k} \geq \mathcal{T}$ and $s_{k,j} \geq \mathcal{T}$, then we can say (using transitivity) that all variables $v_i, v_k, v_j$ form a co-access set and their memory allocation can be coalesced together. We do this process iteratively until we have merged all co-access pairs into co-access sets using a union-find data structure, as illustrated in Algorithm 2.

**Algorithm 2** GENCOACCESSESSETS($\mathcal{L}$,$\mathcal{T}$)

```
Input: $\mathcal{L}$: List of co-accesses pairs and strength
Input: $\mathcal{T}$: Strength threshold
Output: $\mathcal{S}$: Sets of co-accesses to malloc together
1: $\mathcal{S} := \{\}$
2: for each $(v_1, v_2, s) \in \mathcal{L}$ do
3:     if $s \geq \mathcal{T}$ then
4:         $s_1$ := GETSET($\mathcal{S}, v_1$)
5:         $s_2$ := GETSET($\mathcal{S}, v_2$)
6:         MERGESETS($s_1, s_2$)
7:     end if
8: end for
9: return $\mathcal{S}$
```

## 3.4 Choosing Subsets of Co-access Sets for Malloc Transformation

The algorithm in 2 retrieves global co-access sets given a threshold strength. Because it is hard to figure out which call to *malloc* comes first in a set of *mallocs* that span multiple blocks and which call to *free* comes last in the corresponding set of *free*s, we split the global co-access sets into co-access sets for which all mallocs are contained within the same basic block and all corresponding frees are contained within the same basic block, which we call a *per-mallocblock co-access set.* This way, we can obtain a mapping of *(mallocblock, freeblock)* to one or more per-malloc co-access sets.

Then, for each basic block, we find all the per-mallocblock co-access sets corresponding to the basic block and run the malloc transformation on each of those per-mallocblock co-access sets.

So, for the code in 6, if objects a, b, c, and d are all co-accessed, then a and b would be *malloc*ed

```
a = malloc(C)
b = malloc(C)
c = malloc(C)
d = malloc(c)
(...)

if (cond) {
  free(a)
  free(b)
}

free(d)
free(c)
```

Figure 5: Source code

```
a = malloc(C)
b = a + C
c = malloc(C)
d = c + d
(...)

if (cond) {
  free(a)
}

free(c)

.
```

Figure 6: Target code

together, while c and d would be *malloc*ed together, since their frees are in differing basic blocks.

## 4 Future Work

### 4.1 Control Flow

We have some infrastructure set up for instrumenting the source code to handle control flow when identifying co-accesses, necessary for handling the scenario in 7.

```
loop {
  p = malloc(C)
  (...)
}

loop {
  free(q)
  (...)
}
```

Figure 7: Example loop containing `malloc` and another loop with corresponding frees

The scenario in 7 is a particularly annoying scenario, because the order of the freed objects may not be the same as the order of the malloced objects.

To handle this, we can either instrument the program code to keep track of *malloc*ed objects and use reference counting to find the last call to *free* in a cset.

Another interesting approach might be to modify *malloc* so that it can track its own callsites and infer whether a call to *malloc* is probably in a loop, and handle the case in 4 accordingly. This essentially leaves a lot of co-access set identification logic to *malloc*, which is convenient because *malloc* already keeps track of *malloc*ed objects, to some extent.

However, we ran out of time to implement these due to implementation complexity.

### 4.2 Weakening Constraints onf Co-Access Sets

The constraints described in 3.4 are fairly tight. With analysis information about basic blocks that dominate other basic blocks, we could conceivably loosen the constraints for per-malloc co-access sets so that the frees don't have to be in the exact same basic block, as long as the ordering of the frees is still known at compile time.

## 5 Evaluation

For our `malloc` optimizations, we aim to answer the following research questions.

1. What is the perfomance of programs with our optimization compared against the baseline?

2. How does coalescing `malloc` calls affect performance?

3. How does our optimization affect code-size?

### 5.1 Experimental setup

To evaluate our approach, we implemented our optimization using LLVM 12.0.1 as a function-level pass. All results presented in this section were obtained on macOS Monterey running on a 2.3 GHz 8-Core Intel Core i9 with 32 GB of RAM.

### 5.2 Results

#### 5.2.1 Running Time of Randomly-Generated Benchmark

We generate benchmarks by randomly generating $n$ variables, *malloc*ing them and freeing them in a random (valid) order while inserting uses of random *malloc*ed variables before they are freed. We could probably have introduced more complex programs with interesting basic blocks if we used a more sophisticated code generator such as CodeAlchemist, but that type of code generation does not result in realistic-looking code either.

Because we did not finish implementing co-access identification for *malloc*s with loops (and other complex control flow), we could not test on actual applications, which mostly used *malloc* in more complex situations.

Figure 8 illustrates shows the run time of our benchmarks in function of the number of malloc / free blocks for.

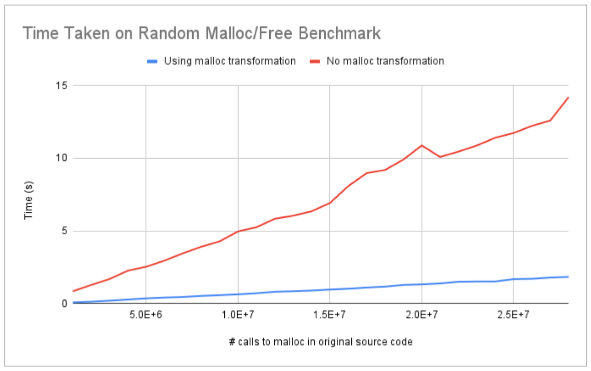

Figure 8: Caption

### 5.2.2 Code Size

We also notice that the number of instructions reduces after running our optimization. This is because the number of instructions to call malloc and store the pointer is the same as calculating the offset of a pointer and assigning it to a variable, while we remove $n-1$ frees for every $n$ mallocs that we coalesce.

## 6 Related Work

Prior research related to our domain can be decomposed into 2 main approaches: 1) hint-based data layout optimizations, 2) hardware-software cooperative techniques.

As data movement starts to become a major bottleneck within large scale systems, data layout optimizations improve performance by enhancing data locality across the hardware stack. Prior works have proposed techniques to provide semantic hints in order to improve the data layout of different program executions and workloads. One technique proposed by Peled et al. uses a program analysis pass to capture access patterns for data structure and transform them into semantic hints for a context-based memory prefetcher [4]. Similarly, Geomancy is a tool that can find efficient data layouts for file systems by using reinforcement learning with the I/O traces of past workloads as semantic hints [1]. Lastly, Whirlpool, a data placement system for non-uniform cache architectures, provides optimal data placement using a combination of static information and dynamic policies [2]. These semantic hints aren't just limited to improving the execution time of systems. In [3], the authors demonstrate a new practical way to reuse JIT profile data across virtual machine executions to improve the warm up time of subsequent virtual machines.

In more recent work, Vijaykumar et al. propose Metasys, a hardware software co-design that enables new cooperative techniques to enable cross-layer communication from an executing program to the hardware stack [5]. By communicating program metadata to the hardware architecture, Metasys enables new approaches to enhance memory prefetching performance and security defenses.

## 7 Conclusions

Overall, this approach to making uses of *malloc* more cache-friendly seems promising. Without a working control flow analysis, however, it is rather difficult to perform a more useful evaluation, since, as it turns out, many applications tend to call malloc in loops and branches. So, we hope the work outlined in 4 will help with this.

## 8 Project Logistics

1. 50-50 work distribution

## References

[1] Bel, O., Chang, K., Tallent, N. R., Duellmann, D., Miller, E. L., Nawab, F., and Long, D. D. E. Geomancy: Automated performance enhancement through data layout optimization. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2020), pp. 119–120.

[2] Mukkara, A., Beckmann, N., and Sanchez, D. Whirlpool: Improving dynamic cache management with static data classification. *ACM SIGARCH Computer Architecture News 44*, 2 (2016), 113–127.

[3] Ottoni, G., and Liu, B. Hhvm jump-start: Boosting both warmup and steady-state performance at scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2021), pp. 340–350.

[4] Peled, L., Mannor, S., Weiser, U., and Etsion, Y. Semantic locality and context-based prefetching using reinforcement learning. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)* (2015), pp. 285–297.

[5] Vijaykumar, N., Olgun, A., Kanellopoulos, K., Bostanci, N., Hassan, H., Lotfi, M., Gibbons, P. B.,

and Mutlu, O. Metasys: A practical open-source metadata management system to implement and evaluate cross-layer optimizations. *CoRR abs/2105.08123* (2021).