RAPID: Region-Based Pointer Disambiguation

KHUSHBOO CHITRE, IIIT-Delhi, India
PIYUS KEDIA, IIIT-Delhi, India
RAHUL PURANDARE, University of Nebraska–Lincoln, USA

Interprocedural alias analyses often sacrifice precision for scalability. Thus, modern compilers such as GCC and LLVM implement more scalable but less precise intraprocedural alias analyses. This compromise makes the compilers miss out on potential optimization opportunities, affecting the performance of the application. Modern compilers implement loop-versioning with dynamic checks for pointer disambiguation to enable the missed optimizations. Polyhedral access range analysis and symbolic range analysis enable $O(1)$ range checks for non-overlapping of memory accesses inside loops. However, these approaches work only for the loops in which the loop bounds are loop invariants. To address this limitation, researchers proposed a technique that requires $O(log\ n)$ memory accesses for pointer disambiguation. Others improved the performance of dynamic checks to single memory access by constraining the object size and alignment. However, the former approach incurs noticeable overhead due to its dynamic checks, whereas the latter has a noticeable allocator overhead. Thus, scalability remains a challenge.

In this work, we present a tool, RAPID, that further reduces the overheads of the allocator and dynamic checks proposed in the existing approaches. The key idea is to identify objects that need disambiguation checks using a profiler and allocate them in different regions, which are disjoint memory areas. The disambiguation checks simply compare the regions corresponding to the objects. The regions are aligned such that the top 32 bits in the addresses of any two objects allocated in different regions are always different. As a consequence, the dynamic checks do not require any memory access to ensure that the objects belong to different regions, making them efficient.

RAPID achieved a maximum performance benefit of around 52.94% for Polybench and 1.88% for CPU SPEC 2017 benchmarks. The maximum CPU overhead of our allocator is 0.57% with a geometric mean of -0.2% for CPU SPEC 2017 benchmarks. Due to the low overhead of the allocator and dynamic checks, RAPID could improve the performance of 12 out of 16 CPU SPEC 2017 benchmarks. In contrast, a state-of-the-art approach used in the comparison could improve only five CPU SPEC 2017 benchmarks.

CCS Concepts: • **Software and its engineering** → **Compilers**.

Additional Key Words and Phrases: alias analysis, LLVM, optimizations, regions, dynamic checks, memory allocation, allocation site

ACM Reference Format:
Khushboo Chitre, Piyus Kedia, and Rahul Purandare. 2023. RAPID: Region-Based Pointer Disambiguation. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 283 (October 2023), 29 pages. https://doi.org/10.1145/3622859

1 INTRODUCTION

Pointers are often the most feared, and yet, the most useful data types in C/C++. The fear associated with their usage is due to the fact that they can lead to ambiguous memory references. Nevertheless,

Authors' addresses: Khushboo Chitre, Computer Science and Engineering, IIIT-Delhi, Okhla - Phase 3, New Delhi, Delhi, 110020, India, khushbooc@iiitd.ac.in; Piyus Kedia, Computer Science and Engineering, IIIT-Delhi, Okhla - Phase 3, New Delhi, Delhi, 110020, India, piyus@iiitd.ac.in; Rahul Purandare, The School of Computing, University of Nebraska–Lincoln, 256 Avery Hall, Lincoln, NE, 68588, USA, rahul@unl.edu.

2475-1421/2023/10-ART283
https://doi.org/10.1145/3622859

many optimizations such as code vectorization [Karrenberg and Hack 2011], loop invariant code motion, load elimination [Chow et al. 1997; Surendran et al. 2014], global value numbering, and store elimination [Surendran et al. 2014] rely on pointer-related information computed by alias analysis. Alias analysis informs the compiler about whether two pointers (a pointer pair) ever point to overlapping memory locations.

Alias analysis can be performed both interprocedurally (across function boundaries) and intraprocedurally (local to the function). When a pointer is passed to a function, interprocedural analysis needs to be performed to identify how the callsite affects the pointer references. However, interprocedural alias analysis can potentially lead to excessive memory usage and scalability issues. Thus, compiler developers usually opt for intraprocedural alias analysis to obtain alias information. This leads to imprecise alias information, and subsequently, lesser than expected performance improvement.

To deal with the imprecision of the alias analysis, subsequent work [Alves et al. 2015; Chitre et al. 2022] and modern compilers [llv 2023; Naishlos 2004] insert runtime checks for pointer disambiguation. These techniques mostly target loops because the cost of dynamic checks is usually marginal compared to the benefit of the optimized loop. The key idea is to insert dynamic checks for pointer disambiguation during the compilation time and create two versions of a loop – an optimized version and an unoptimized version. The dynamic checks decide which loop version to execute at runtime. This technique is also called loop-versioning and the pointer disambiguation check is called loop-versioning condition. Alves et al. [Alves et al. 2015] compute the loop-versioning condition using a combination of polyhedral [Bondhugula et al. 2008; Bondhugula 2008; Feautrier 1992] and symbolic range [Nazaré et al. 2014; Paisante et al. 2016; Rugina and Rinard 2000] analyses. LLVM compiler uses scalar evolution [Engelen 2000, 2001] to compute the loop-versioning condition. The dynamic checks inserted by these analyses are efficient; however, these analyses only work when the loop bounds are loop invariants. Polyhedral analysis additionally requires all memory accesses inside the loop to be affine.

To enable optimizations for loops that can't be versioned using the polyhedral, symbolic range, and scalar evolution analyses, Alves et al. [Alves et al. 2015] and Chitre et al. [Chitre et al. 2022] proposed a different approach to compute the loop-versioning condition. The key idea in these approaches is that if two pointers point to different objects, they can't overlap. To check if two pointers `x` and `y` belong to different objects, Alves et al. compute the starting addresses of the objects pointed by `x` and `y`. Notice that `x` and `y` can be internal pointers. The starting address refers to the address returned by the memory allocation API. If the starting addresses are different, then `x` and `y` point to different objects, and they can never overlap, no matter how they are accessed within the loop. Alves et al. use a red-black tree to keep track of the starting addresses of objects, and thus identifying the starting address may take $O(log\ n)$ memory accesses, where `n` is the number of live objects. Chitre et al. implemented the approach in a tool named Scout. Scout disambiguates two pointers `x` and `y` by computing the size of the object pointed by any of the pointers, say `len`. In this technique, if `x` and `y` are at least `len` apart, then they can't overlap. Scout can fetch the size of the object from a given pointer address in just one memory access. In this way, the dynamic checks proposed by Chitre et al. are more efficient than Alves et al. Scout showed that their technique can scale to large applications and could improve the performance of a few CPU SPEC 2017 benchmarks.

The major drawback of Scout [Chitre et al. 2022] is the additional CPU and memory overhead of the allocator because of the increased object size. Scout requires all the objects to be of size 2^k. The CPU and memory overheads of this scheme for CPU SPEC 2017 benchmarks lie in the range -2.57% to 8.36% and -2.29% to 34.89% (negative means performance improvement), respectively. The primary reason for these overheads is that all the objects need to constrain the allocation size and alignment even though they are never accessed inside the versioned loops.

In this work, we present our tool Rapid, which further reduces the overheads of allocator and dynamic checks. Rapid works by constraining only those memory allocations that may be accessed in additionally versioned loops in Alves et al. [Alves et al. 2015] and Scout [Chitre et al. 2022] approaches. At the high level, Rapid ensures that the conflicting memory accesses always belong to different heap regions, and the loop-versioning condition simply checks if the pointer addresses belong to different regions. The heap regions are allocated in a way that the top 32-bits in the virtual addresses corresponding to two different regions are always different. Thus, the dynamic check for disambiguation is very efficient and doesn't require any memory access.

Rapid works in two phases. In the first phase, Rapid uses a profiler to correlate conflicting memory accesses in the loops to their allocation sites. In the second phase, Rapid generates code in a way that conflicting memory accesses are allocated from different heap regions (at runtime) and implement loop-versioning.

Unlike Scout [Chitre et al. 2022], Rapid's goal is to constrain only those memory allocations accessed in loops that can be optimized further using dynamic checks. Therefore, Rapid uses a profiler to identify and allocate these memory allocations into different regions. Profilers have also been used in the past for the data-dependence analysis [Chen et al. 2004; Fernandez and Espasa 2002; Huang et al. 1994; Lin et al. 2003]. These works aim to prevent loads, whose values are available, from a critical path. The vector instructions supported by modern processors allow operations to execute in parallel. The compiler can execute four iterations of a loop in parallel (also called loop vectorization) using vector instructions if these iterations don't depend on each other. Because two loop iterations may or may not depend on each other depending on their inputs – a profiler can be used in this context to identify the loops that can execute in parallel most of the time. Due to the substantial performance benefits associated with loop vectorization, the need to use profilers for exploring opportunities to vectorize loops becomes apparent.

Here are the key differences between Rapid and Scout. Scout requires one memory access to disambiguate two pointers, whereas Rapid doesn't require any memory access in the dynamic checks for disambiguation. Scout constrains the size and alignment of all objects, which is the primary source of its overhead. Rapid doesn't change the allocation size and alignment of objects. Also, the different regions are assigned to only those objects that may be accessed in the versioned loop – not to all objects. Therefore, the allocator overhead in Rapid is very low compared to Scout. The allocator's CPU and memory overheads of Rapid for CPU SPEC 2017 benchmarks lie in the range -3.67% to 0.46% and -0.62% to 6.96%, respectively. Due to the low overhead of allocator and dynamic checks, Rapid could improve the performance of 12 out of 16 CPU SPEC 2017 benchmarks. In the remaining four benchmarks, none of the loops were versioned.

This paper makes the following contributions:

(1) It introduces an approach that combines loop-versioning with lightweight dynamic checks. Our approach includes a region-based memory allocator and instruments region-based dynamic checks to decide the version of a loop to execute at runtime.
(2) A profiler to identify conflicting memory accesses inside loops and potential loop candidates that show potential for improvement.
(3) Integration of the approach with LLVM.
(4) Evaluation of the performance benefits of Polybench and CPU SPEC 2017 benchmarks using the approach.

Section 2 gives an overview of our approach using an example. Section 3 covers the approach in detail. Section 4 discusses the implementation details. Section 5 discusses the evaluation of our approach on Polybench and CPU SPEC 2017 benchmarks. Section 6 discusses the limitation of our approach. Section 7 presents the prior work. Section 8 presents the concluding remarks.

2 OVERVIEW

```
1. void foo(int *a, int *b, int *c,
2.          int *size)
3. {
4.   for (int i = 0; i < *size; i++) {
5.     a[i] = b[i] + c[i];
6.   }
7. }
```

(a) Original foo.

```
8. int main() {
9.   int *a = (int*)malloc(36);
10.  int *b = (int*)malloc(36);
11.  int *c = (int*)malloc(36);
12.  int size = 9;
13.  foo(a, b, c, &size);
14.  ...
15.}
```

(b) Original main.

```
16. int main() {
17.   int *a = (int*)malloc_prof(36, 17);
18.   int *b = (int*)malloc_prof(36, 18);
19.   int *c = (int*)malloc_prof(36, 19);
20.   int *size_addr = (int*)malloc_prof(4, 20);
21.   size_addr[0] = 9;
22.   foo(a, b, c, size_addr);
23.   ...
24. }
```

(c) Profiling phase of Rapid for main.

```
25. int main() {
26.   int *a = (int*)rmalloc(36, 1);
27.   int *b = (int*)malloc(36);
28.   int *c = (int*)malloc(36);
29.   int size = 9;
30.   foo(a, b, c, &size);
31.   ...
32. }
```

(d) Final code generated for main using Rapid.

```
33. void foo(int *a, int *b, int *c,
34.          int *size)
35. {
36.   if (start_addr(a) != start_addr(b) &&
37.       start_addr(a) != start_addr(c) &&
38.       start_addr(a) != start_addr(size)) {
39.     log_pair(a, b); log_pair(a, c);
40.     log_pair(a, size);
41.     int t = *size, i;
42.     for (i = 0; i + 3 < t; i += 4) {
43.       a[i:i+3] = b[i:i+3] + c[i:i+3];
44.     }
45.     for (; i < t; i++) {
46.       a[i] = b[i] + c[i];
47.     }
48.   } else {
49.     for (int i = 0; i < *size; i++) {
50.       a[i] = b[i] + c[i];
51.     }
52.   }
53. }
```

(e) Profiling phase of Rapid for foo.

```
54. #define D (1ULL << 32) /* 4GB */
55. void foo(int *a, int *b, int *c,
56.          int *size)
57. {
58.   if ((a ^ b) >= D &&
59        (a ^ c) >= D &&
60.       (a ^ size) >= D) {
61.     int t = *size, i;
62.     for (i = 0; i + 3 < t; i += 4) {
63.      a[i:i+3] = b[i:i+3] + c[i:i+3];
64.     }
65.     for (; i < t; i++) {
66.       a[i] = b[i] + c[i];
67.     }
68.   } else {
69.    for (int i = 0; i < *size; i++) {
70.       a[i] = b[i] + c[i];
71.     }
72.   }
73. }
```

(f) Optimized foo using Rapid.

Fig. 1. An example to discuss Rapid approach.

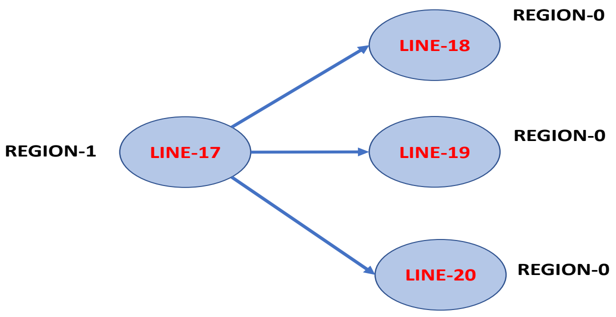


Fig. 2. Interference graph used for assigning regions. Different regions are assigned to nodes connected using an edge.

We now discuss the overview of our approach and compare it with existing techniques using the example in Figure 1. This example is similar to the example used by Scout [Chitre et al. 2022], and Alves et al. [Alves et al. 2015]. We found similar code patterns in CPU SPEC 2017 benchmarks as discussed in Section 5.

The example presented in Figure 1 adds the elements of arrays `b` and `c` and stores the result in array `a` (at line 4-6). The compiler can vectorize this loop if it knows with certainty that `b`, `c`, and `size` don't overlap with `a`. Line 63 shows the vectorized loop body. Here, "`i:i+3`" means four operations at indices `i, i+1, i+2, i+3` are performed in parallel instead of one operation at line 5. Notice that if array `a` overlaps with either array `b`, array `c`, or `size`, then the parallel execution may yield a different result than the sequential execution. In this case, the alias relationships between these objects depend on the values passed by the caller and hence can't be answered using the intraprocedural alias analysis. As discussed before, the polyhedral, symbolic range, and scalar evolution analyses require the loop bounds to be loop invariants, which is not true in this case. The loop bound `*size` is not a loop invariant as `size` can overlap with `a`, which might end up modifying the loop bounds inside the loop body. Therefore, in addition to Rapid, the pointer disambiguation technique discussed in Alves et al. [Alves et al. 2015] and Chitre et al. [Chitre et al. 2022] can be used to version this loop.

Figure 1f shows the versioned loop using Rapid. Lines 58-60 correspond to the loop-versioning condition. Rapid ensures that conflicting memory accesses `(a, b)`, `(a, c)`, and `(a, size)` belong to different heap regions. Two pointer addresses belong to different heap regions if the top 32-bits in the addresses are different. The condition at line 58 is true at runtime if `a` and `b` belong to different regions. Similarly, conditions at lines 59 and 60 hold at runtime if `(a, c)` and `(a, size)` are in different regions. Notice that `b`, `c`, and `size` can be in the same region. If the loop condition fails, the original loop at lines 69-71 executes.

Rapid works in two phases. In the first phase, it profiles the application to find out the allocation sites of pointer variables that require a disambiguation check. In this example, dynamic checks are required for pointer pairs `(a, b)`, `(a, c)`, and `(a, size)`. Figure 1c shows the `main` routine instrumented for profiling. At lines 17-19, `malloc` is replaced with `malloc_prof` that additionally takes the allocation site (a unique identifier for each allocation site, e.g., line number) and stores the allocation site in the metadata corresponding to the allocated object. Lines 36-38 show the loop condition of the versioned loop during the profile phase. It uses the `start_addr` API, which returns the starting address of an object. Two objects cannot overlap if the starting addresses are different. If the optimized path is taken at runtime, the allocation sites of the conflicting pointer pairs are logged. For example, (17, 18), (17, 19), and (17, 20) are logged at lines 39 and 40 (notice that the allocation sites of `a`, `b`, `c`, and `size` are 17, 18, 19, and 20). The next goal is assigning different regions to each conflicting pair of allocation sites. This is done using an interference graph. In an interference graph, there is an edge between two allocation sites if a disambiguation check is required for objects allocated at these sites. Figure 2 shows the interference graph for this example. Rapid assigns regions to each of the nodes in a way that nodes connected using an edge don't share the same region. We have used the standard graph coloring algorithm to assign regions to nodes in the interference graph. In this example, the graph coloring algorithm assigns region-id zero to allocation sites 18, 19, and 20 and region-id one to allocation site 17.

In the second phase, `malloc` is replaced with `rmalloc` if the region-id of the corresponding allocation site is not zero. `rmalloc` additionally takes the target region-id. The default region-id of all allocations is zero. Because, in this case, the region-id for array `a` is one, Rapid replaces the `malloc` with `rmalloc` at line 26. The rest of the allocations are in region-id zero and are not replaced. The optimized code for `foo` is shown in Figure 1f that implements faster checks in the loop-versioning condition.

In Alves et al. [Alves et al. 2015] approach, the allocation sites are not changed. The dynamic checks at lines 58-60 require expensive red-black tree lookups to compute the starting address of `a`, `b`, `c`, and `size`. Scout [Chitre et al. 2022] sets the allocation size and alignment of objects at lines 26-28 to 48 (a power-of-two). The dynamic checks at lines 58-60 require memory access to obtain the size of object `a`.

Rapid enables the compiler to explore useful optimization opportunities missed by the compiler, such as code vectorization, loop invariant code motion, load elimination, and store elimination. In the next section, we describe our approach in detail.

3 RAPID

This section describes the Rapid approach in detail. Figure 3 shows the architecture of Rapid. Rapid uses a region-based allocator that enables faster disambiguation checks. Rapid compiles the application in two phases. Rapid takes an intermediate representation (IR) of the program as input in both phases. The first phase is the profile phase. In the profile phase, Rapid identifies and instruments loops that can be further optimized using loop-versioning. The IR is instrumented to log the allocation sites of the conflicting memory accesses inside the loop. The IR is then transformed into an executable that is executed on a given set of inputs to generate a log file. The second phase also takes the log file generated in the first phase as input. In the second phase, Rapid implements the graph coloring algorithm to infer region-ids for conflicting allocation sites, modifies allocation sites to allocate from inferred region-ids, and implements loop-versioning. The final code generated by Rapid is linked to the region-based allocator during the execution. Now, we discuss both phases in detail.

3.1 Region-Based Allocator

Figure 4 shows a simplified view of the heap layout in the region-based allocator. A region contains a list of segments. A segment is a 4GB (configurable at compile time) contiguous memory area. The starting address of a segment is always aligned to 4GB. Segments in the regions are disjoint. The segments corresponding to a region can be scattered across virtual address space (for simplicity, we showed them contiguous). Two pointer addresses point to different objects if they belong to different segments or regions. Objects in different regions always belong to different segments. Because of the alignment property of segments, if two objects belong to different segments, then the top 32-bits in their virtual addresses can't be the same. This property enables faster checks to disambiguate pointers belonging to different regions, as we use in our dynamic checks.

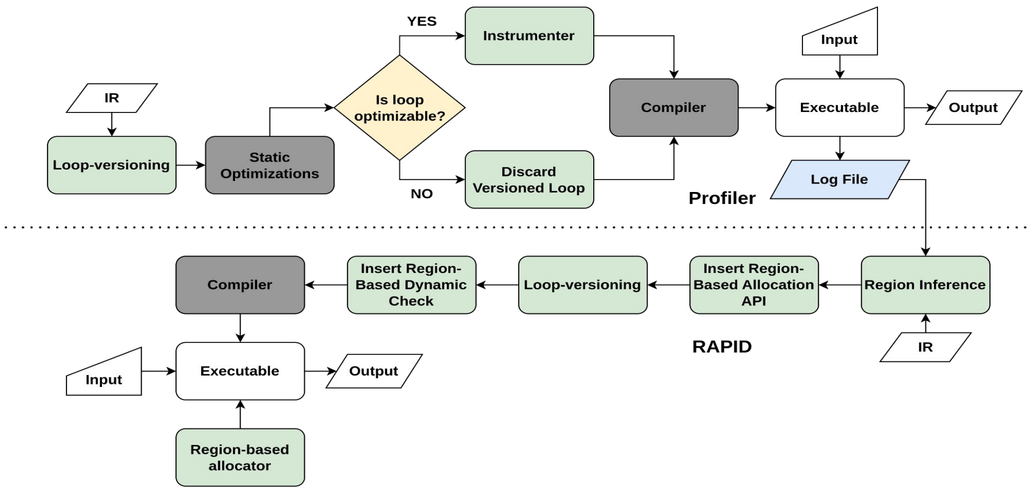


Fig. 3. Architecture of Rapid.

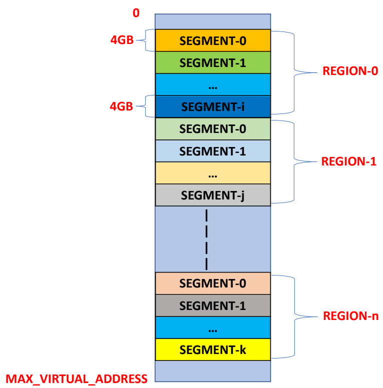


Fig. 4. Heap layout in the region-based allocator.

3.2 Profiling Phase

We use the profiler for the following goals:

(1) To capture allocation sites of objects accessed inside the loops.
(2) To identify potential loop candidates that can be further optimized.
(3) To compute the benefit of loop-versioning by counting the number of iterations of the optimized and unoptimized version of the loop.

3.2.1 Identifying Allocation Sites. To capture allocation sites of objects accessed inside the loop, Rapid inserts an eight-byte object header before the starting address of the object. Rapid assigns a unique integer id to each allocation site, called allocation-id, which is stored in the object header at the time of allocation. From a given pointer address, the allocation site is obtained by first computing the starting address of the object and then reading the allocation id from the object header.

```
Function InstrumentLoop(Function F, Loop L):
    ConflictSet ← ComputeDataDependencies(L);
    /* ConflictSet contains conflicting pointer pairs */
    if ConflictSet.empty() then
        return false;
    end
    L_IsVectorizable ← CanVectorize(L);
    LV ← CreateCopy(L);
    foreach P ∈ ConflictSet do
        assume P doesn't conflict in LV;
    end
    /* check if we can vectorize the versioned loop
    * assuming pointer pairs don't overlap.
    */
    LV_IsVectorizable ← CanVectorize(LV);
    LICM(LV);
    GVN(F);
    DSE(F);
    if LV and L are the same then
        /* LICM, GVN, DSE didn't work */
        if not LV_isVectorizable or L_IsVectorizable then
            /* Loop L can't be optimized further */
            DeleteLoop(LV);
            return false;
        end
    end
    if not LV_IsVectorizable then
        ConflictSet ← RemoveRedundantChecks(ConflictSet, L, LV);
    end
    /* LV is more optimized than L */
    InsertLoopVersioningCondition(L, LV, ConflictSet);
    InsertCodeToLogAllocationIds(LV, ConflictSet);
    InstrumentLoopIterCount(L, LV);
    return true;
```

Algorithm 1: The profiler takes a function and the loop. It returns whether the loop has the potential to be optimized further. The profiler instruments the conflicting memory accesses present in the functions to store the allocation-ids. It also instruments the preheader of the selected loops based on the feedback from static optimizations to log the required information.

3.2.2 Loop-versioning. To identify potential candidates for loop-versioning, Rapid follows an approach similar to Scout [Chitre et al. 2022]. The loop-versioning algorithm is shown in Algorithm 1. `InstrumentLoop` returns true if the loop can be optimized using dynamic checks. `ComputeDataDependencies` collects all the pointer operands that are accessed in a loop. The read-write and write-write pairs for which the static alias analysis fails are added to the `ConflictSet`. It also tries to check if the pointer operands are derived from the same base, in which case they are not added to the ConflictSet. Thus, ComputeDataDependencies returns a set of pointer pairs (ConflictSet) that are accessed inside the loop and follows the following properties:

- Pointer pairs are not the result of pointer arithmetic on the same array in the current function scope.
- At least one of them is involved in a write operation inside the loop body.
- Intra-procedural alias analysis doesn't know precisely if the pointer pairs are alias or not.

CanVectorize uses alias analysis and scalar evolution [Engelen 2000, 2001] to return true if a loop can be vectorized (no read-write or write-write conflict) with/without loop-versioning. We used the existing implementation in LLVM for CanVectorize. Thus, CanVectorize routine returns true if the compiler can vectorize the loop. Notice that InstrumentLoop takes an optimized IR as input that is not vectorized. The reason is that once the loop is vectorized, the LLVM compiler can't optimize it further. At line 8, a copy of loop, LV, is created. At lines 9-11, Rapid assumes that the pointer pairs returned by ComputeDataDependencies don't overlap in LV. Non-overlapping memory accesses inside the loop may enable other optimizations, such as loop invariant code motion (LICM), global value numbering (GVN), dead store elimination (DSE), and loop vectorization. Rapid performs LICM (line 16), GVN (line 17), and DSE (line 18) on LV to check if any of them are enabled. If yes, the loop is versioned. Otherwise, if the original loop (L) was not vectorizable and LV is vectorizable (due to non-overlapping memory accesses), then also the loop is versioned. At line 31, the loop-versioning condition is inserted. For every pointer pair (x, y) in the ConflictSet, InsertLoopVersioningCondition instrument code to obtain the starting addresses of objects pointed by (x, y). The optimized loop version (LV) is executed at runtime if the starting addresses don't match for all pointer pairs in ConflictSet. In LV, Rapid instruments code to log the allocation-id pairs corresponding to all pointer pairs in the ConflictSet. The allocation-id is obtained from the object header of the corresponding object.

Finally, Rapid instruments code to collect the loop-taken statistics for the versioned loop. The loop is versioned in the second phase if the optimized path is taken frequently during profiling. Notice that if LV is not vectorizable, we may not need all dynamic checks. For example, at line 61 in Figure 1, size is moved outside the loop. If the rest of the loop is not vectorizable, we don't need to check if pointer pairs (a, b) and (a, c) overlap. This optimization is sound as long as size doesn't overlap with a. To compute the minimum number of checks, Rapid adopted the approach followed by Scout [Chitre et al. 2022]. At line 28, RemoveRedundantChecks removes the redundant checks using the following approach:

- Initially, mark all pointer pairs in the ConflictSet as invalid.
- If a memory access operation (read/write) of pointer p in L is not present in LV, mark all pointer pairs that contain p in ConflictSet as valid.
- Remove all pointers pairs from the ConflictSet that are invalid.

Finally, the instrumented code for profiling is converted into an executable. The profiler is executed on user-supplied input to generate a log file that is used in the second phase of Rapid.

3.3 Final Code Generation

In the second phase, Rapid does the following tasks.

- Computes region-id for each allocation site.
- Implements loop-versioning for loops identified in the profiling phase.

3.3.1 Region Inference. The goal of the region inference is to compute the region-ids for all allocation sites. The log file generated during the profile phase contains all allocation site pairs that should not belong to the same region. As discussed before, Rapid generates an interference graph in which the nodes are the allocation sites, and an edge between two nodes represents that the nodes should not belong to the same region. Now, the problem of computing region-ids is the same as the graph coloring problem used for register allocation [Chaitin 1982]. To find whether a graph is k-colorable, Chaitin [Chaitin 1982] removes a node with less than k edges from the graph and checks whether the rest of the graph is k-colorable. This approximation algorithm is very efficient. We used the following algorithm to compute the region-ids, where n is the number of nodes in the interference graph.

```
for (i = 2; i <= n; i++) {
  if (graph is colorable using i colors) {
     assign colors starting from zero
     return;
  }
}
```

In this algorithm, colors represent regions. After the region-ids are assigned to each node in the interference graph, all the allocation sites that are present in the interference graph and have region-id other than zero are modified to allocate from the corresponding region.

3.3.2 Handling Conflicting Allocation Sites. The allocation sites of two conflicting memory accesses in a versioned loop may be the same if the corresponding allocations are inside a loop. We explain this using the following example. If we replace a, b, c in Figure 1b with `arr[0]`, `arr[1]`, and `arr[2]` and allocate them inside a loop, there will be only one `malloc` statement. Now, we can't replace `arr[0]` in Figure 1d to allocate from region-1 because the same statement is used to allocate `arr[1]` and `arr[2]`.

In this case, the dynamic check will always fail if we assign a unique region-id to the corresponding allocation site inside the loop. To handle this case, instead of assigning a single region-id, Rapid assigns a range of region-ids to the allocation site. For these cases, during memory allocation, the allocation-id is chosen from the set of assigned region-ids in a round-robin manner. Notice that this approach doesn't guarantee that the dynamic checks for disambiguation always succeed; however, the probability of failure decreases as we increase the set of ids assigned to these allocation sites. Because a large number of regions can degrade an application's performance, we ask the user to provide the maximum number of regions using a compile-time option. After assigning region-ids to non-conflicting allocation sites, we divide the remaining regions equally among all conflicting allocation sites. In our experiments, we found that a small number of ids are sufficient for the existing cases in real-world benchmarks.

3.3.3 Loop-versioning. The loop-versioning algorithm is similar to the algorithm used by the profiler (Figure 1), except for a few changes. The dynamic checks in the loop-versioning condition are very efficient. It checks if the pointer pair belongs to different segments by comparing the top 32-bits. The starting address of a segment is always aligned to 4GB. If the `xor` of two pointer addresses is greater than or equal to 4GB (maximum size of a segment), then they belong to different segments and thus different objects. Rapid doesn't log anything in this phase. The loop is versioned only if the optimized path is taken frequently during the profile run.

4 IMPLEMENTATION

We implemented Rapid as a part of LLVM infrastructure (v10.0.0). Rapid instruments an already optimized IR that is not vectorized. This is because LLVM can't further optimize an already vectorized loop. We have modified Mimalloc (v 2.0.6) [mim 2023; Leijen et al. 2019] to implement

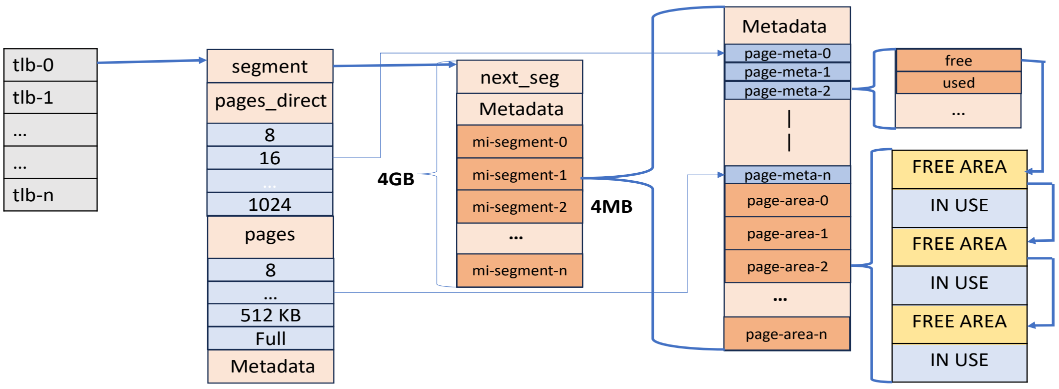


Fig. 5. Heap layout in the region-based allocator.

our region-based memory allocator. Mimalloc is a general-purpose allocator. The performance of Mimalloc is comparable to the state-of-the-art allocators. The primary motivation for picking Mimalloc is that it already supports thread-local heaps that can be easily extended to multiple heap regions.

Region-Based Allocator. By default, Mimalloc uses thread-local heaps. We extended Mimalloc to implement thread-local heaps for multiple regions. In Mimalloc, objects are allocated from pages. Pages are allocated from mi-segments (we use the term mi-segments to distinguish them from the segments in our approach as shown in Figure 4).

The heap layout is shown in Figure 5. A mi-segment is a 4MB contiguous memory area that is divided into pages. A page is further divided into fixed-size slots. At the top, a mi-segment stores metadata for every page. The page metadata contains a reference to the first free slot on the page. A free slot contains a reference to the next free slot on the page. The page sizes are divided into three categories: 64KB pages for small objects under 8KB; for large objects under 512KB, there is one page that spans the whole mi-segment; and huge objects over 512KB have one page of the required size.

A thread-local heap contains references to page metadata stored in mi-segments. Mi-segments are allocated from a segment. A segment is a 4GB contiguous memory area, which is divided into mi-segments. The top of a segment contains a bitmap to keep track of free mi-segments. A segment always belongs to a unique region. A segment also contains a reference to the next segment in the same region. A heap always belongs to a unique region. Because the heaps are local to threads, multiple heaps that correspond to different threads may belong to the same region. Two heaps that belong to a region, say `r`, may allocate mi-segments from the same segment in region `r`. `tlb-0` to `tlb-n` are thread-local variables that point to the heaps in region `0` to `n`.

Initially, Rapid reserves memory area for a segment. The memory area corresponding to the mi-segment is made accessible during allocation. On Linux, Mimalloc eagerly enables read/write access for the entire 4MB area of a mi-segment. However, enabling read/write doesn't significantly change the maximum resident set size because the operating system merely enables demand paging on the memory area during this operation. Once the objects are allocated from a page, the page is reclaimed only when the entire page is free. Because the typical page size in Mimalloc is 64KB, too many partially allocated pages may cause fragmentation. A large number of regions can further escalate fragmentation; however, in our experiments, we found out that the number of regions required for widely used benchmarks is small, and therefore our memory overhead is low.

We export a thread-local variable from Mimalloc that is used by the application to pass the region-id. If the caller doesn't pass the region-id, the allocation is done from the default region. We compared the performance of our region-based implementation using the default region with the original Mimalloc implementation. Our implementation slightly improved the performance of CPU SPEC 2017 benchmarks without additional memory overhead. Therefore, we have used our region-based Mimalloc implementation (that uses region-0 for every allocation) as the baseline in our evaluations.

Finding Starting Address. We implemented an API in Mimalloc to find the starting address of the object, which is used by the profiler. In Mimalloc, mi-segment can be derived from a virtual address using just an bitwise "`and`" operation due to the alignment property of mi-segments. A mi-segment also stores the metadata corresponding to every page in the mi-segment. A page contains fixed-size objects. Page metadata contains starting address of the page and the object size on that page. Our API uses page metadata to compute the starting address of the object. From a given address `x`, the location of page metadata can be efficiently computed using the offset of `x` in the corresponding mi-segment.

Stack Allocations in the Profile Phase. As discussed before, during the profile phase, Rapid stores the allocation site in the object header at the time of heap allocation. The compiler can't distinguish between a heap/stack location statically when a stack address escapes the static scope (i.e., by passing to a function, stored in a memory, or typecasted to an integer). In these cases, disambiguation checks may encounter stack allocations. To handle this case, if a stack address escapes the static scope, Rapid replaces the stack allocation with heap allocation. For these allocations, Rapid instruments `free` when the stack allocation goes out of scope. In disambiguation checks, if an address belongs to the data section, Rapid identifies them as global variables. For global variables, a fixed allocation site is logged in the profile phase.

Stack Allocations in Final Code. If a disambiguation check is needed for the stack and heap location, we might need to allocate the stack object from a different region. We can do it by replacing the stack allocation with region-based heap allocation. However, this may hurt the performance because the stack allocations are faster. To handle this, we try to allocate default regions to stack allocations during region inference, thus eliminating the need to replace them. If we considered that the default stack belongs to a segment by resetting the lower 32 bits, we found that the entire stack always fits within the segment. However, if the default stack may span across multiple segments, then we need to allocate a new stack at the start of the main routine for the default region and switch to the new stack. Before returning from the main, we can switch to the default stack. For this reason, we consider all stack allocations as allocations from the default region. If a stack object conflicts with another heap object, we always allocate the heap object from a region different from the default region.

However, if a disambiguation check is needed between two stack locations, then we can't assign the default region to both of them. We found this case in three CPU SPEC 2017 benchmarks (`imagick_r`, `gcc_r` and `blender_r`). One way of disambiguating such locations would be to replace the second stack allocation with a call to allocator APIs to allocate from a different region. However, we found this to be expensive. To further minimize the overhead of stack allocation replacement, we use per-thread stacks for different regions. This is done for only those benchmarks that require stack allocations in a different region other than the default region. If, during the region inference, we found that a stack allocation, say S, needs to be allocated from another region, say region-2, we pre-allocate a secondary stack from region-2 and store its address in a per-thread variable. The allocations for S are done by adding and subtracting size from the per-thread variable, which is almost as efficient as the usual stack allocations. Rapid identifies whether per-thread stacks for different regions are needed while building the inference graph.

```
Function FindWrappers(FunctionList FL):
    /*Input : FL is the list of all the functions, Output : Set of wrapper functions */
    WrapperFunctions ← (); /*Set of wrapper functions */
    foreach i ∈ Standard allocation API do
        WrapperFunctions.insert(i);
    end
    Changed ← true;
    while Changed do
        Changed ← false;
        foreach Function F ∈ FL do
            if F ∈ WrapperFunctions or
             NumCallsToWrapperFunctions(F) != 1 or
             NumCallsToExternalFunctions(F) > 0 then
                continue;
            end
            Ptr ← GetAllocatedAddr(F);
            StoringAllocatedAddr ← false;
            foreach Store operation S ∈ F do
                ValueStored ← S.GetValueOperand();
                if IsDerivedFrom(ValueStored, Ptr) then
                    GenerateWarning(F, Ptr);
                    StoringAllocatedAddr ← true;
                    break;
                end
            end
            if StoringAllocatedAddr then
                continue;
            end
            foreach Return operation R ∈ F do
                ReturnValue ← R.GetValueOperand();
                if IsDerivedFrom(ReturnValue, Ptr) then
                    WrapperFunctions.insert(F);
                    Changed ← true;
                    break;
                end
            end
        end
    end
    return WrapperFunctions;
```

Algorithm 2: Function `FindWrappers` takes a list of functions. It returns a subset of functions. These functions represent the list of wrapper functions in the application.

Handling Wrappers. In some benchmarks, wrapper functions are used to allocate memory. The problem with the wrapper functions is that memory allocation in the wrapper function is treated as the allocation site for all allocations. Rapid uses an iterative algorithm, Algorithm 2, to detect the wrapper functions automatically. `FindWrappers` takes the list of all the functions in the application as an argument and returns a set of wrapper functions.

If a function does a single allocation (using standard allocation API or a wrapper function), doesn't call external functions, doesn't store addresses derived from the allocated address in memory, and returns an address derived from the allocated address, then it is a wrapper function. All functions except standard library functions are considered external functions. We found that, in the wrapper functions, library functions are used for initializing memory or handling error conditions.

`IsDerivedFrom` routine takes two pointer arguments and returns true if the first argument is derived from the second argument using some arithmetic or typecasts operations or if both pointer arguments are the same. `GetAllocatedAddr` returns the allocated address at the unique allocation site in the function. This iterative algorithm terminates when no new wrapper functions are identified during an iteration.

We could detect most wrapper functions using the algorithm described above. The `perlbench_r` and `gcc_r` benchmarks use wrapper functions (namely `Perl_my_exit`, `Perl_PerlIO_stderr`, `Perl_PerlIO_fileno`, and `xexit`) for the library functions exit, stderr, and fileno in the wrapper functions for allocations. For these benchmarks, we manually added these functions to the list of library functions to detect the wrapper functions for allocations. Apart from these cases, we found that at four places in the `parest_r` benchmark, instead of returning the allocated address, the wrapper function stores the address in a class member. In the `x264_malloc` routine of the `x264_r` benchmark, the wrapper function stores the address of the allocated buffer in the buffer itself before returning the allocated address. Our algorithm generates warnings for these cases. The `GenerateWarning` routine in Algorithm 2 generates a warning if the allocated address is stored once in an address that is derived from the allocated address itself or if the allocated address is stored once in a class member. We found that with these conditions, there are only five functions for which the warning was generated, and they are indeed the wrapper functions. We expect users of our tool to inspect the routines for which warnings are generated manually.

Nevertheless, if the above algorithm misses a wrapper function, we can inspect the profiler's output to identify the wrapper functions. The profiler generates a list of conflicting accesses with the same allocation sites. In addition to the allocations inside loops, this list also contains all wrapper functions. We found that this list is usually small, which makes it easy to find the missing wrapper functions. We have also added a new function attribute in the compiler to annotate these wrapper functions manually. We also verified that we didn't miss any wrapper function by looking at the profiler's output.

Column-2 of Table 1 shows the total number of wrapper functions identified (including the ones for which warnings were generated) by Rapid for Polybench and CPU SPEC 2017 benchmarks using our iterative algorithm. Apart from the wrapper functions that directly allocate memory using the standard APIs, the other wrapper functions call wrapper functions to allocate memory and return the allocated address. Some of them initialize the memory buffer before returning it to the caller. Columns-3 and 4 of Table 1 show the subset of identified wrapper functions and their location in the source code. These functions correspond to the allocation sites of the conflicting memory accesses in the profiler's output.

During the profile phase and final code generation, calls to wrapper functions are treated as allocation sites. As discussed earlier, our region-based allocator receives the region-id in a thread-local variable. During the final code generation, Rapid stores the region-id in the thread-local variable before calling a wrapper function that is eventually passed to the region-based allocator. During the profile phase, Rapid stores the unique identifier for the allocation site in the thread-local

Table 1. Wrapper functions information for Polybench and CPU SPEC 2017 benchmarks. (#WF = Total number of wrapper functions automatically identified by Rapid. The last two columns show the name and source location of the wrapper functions corresponding to the allocation sites of the conflicting memory accesses in the profiler's output. These functions are identified based on the iterative algorithm of Rapid and the profiler's output.)

Benchmark	#WF	Wrapper functions	Source Location
Polybench	1	polybench_alloc_data	polybench.c, line 557
parest_r	9	reinit	vector.h, line 973
nab_r	5	ivector	memutil.c, line 42
perlbench_r	11	Perl_safesysmalloc	util.c, line 133
gcc_r	26	xmalloc xcalloc	xmalloc.c, line 141, xmalloc.c, line 155
xalancbmk_r	7	allocate	MemoryManagerImpl.cpp, line 30
x264_r	10	x264_malloc	common.c, line 1102
blender_r	8	MEM_lockfree_mallocN MEM_lockfree_callocN	mallocn_lockfree_impl.c, line 300, mallocn_lockfree_impl.c, line 279

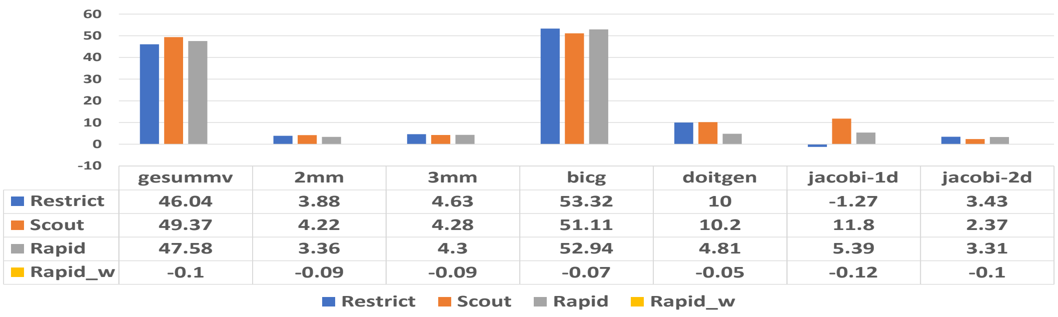
variable before calling the wrapper function. The allocator stores the allocation site passed by the caller (using the thread-local variable) in the object header.

5 EVALUATION

We evaluated Rapid on an Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz 12-core machine with x86_64 architecture, 32 GB primary memory, and the Ubuntu 20.04.1 LTS operating system. We disabled hyper-threading during the evaluation. We used 46 micro and real-world benchmarks for evaluation. These include 30 Polybench (v4.2.1) and 16 C/C++ CPU SPEC 2017 benchmarks. We used the default input set for Polybench (i.e., large input set) and reference input set for CPU SPEC 2017 benchmarks in our evaluation. In our experiments, for all benchmarks, we reported the arithmetic mean of the execution times for five runs. We obtained memory overheads by computing the percentage change in the value of "maximum resident set size" using "/usr/bin/time -v" command. We used Perf [per 2023] tool to generate statistics related to different caches. We used "O3" optimization level to compile the application in both phases. We used the same input sets used for profiling to evaluate the performance of our scheme. We also computed the worst-case overhead of our scheme. In the worst-case, the information gathered during the profile phase never holds at runtime. To compute the worst-case overhead, we insert the dynamic checks in a way that the last check in the loop-versioning condition never holds. Consequently, after executing disambiguation checks for all conflicting pointer pairs in a loop versioning condition, the program always takes the unoptimized path. We have used geometric mean to report the average. Because the percentage change can be both positive and negative and the geometric mean can only be computed for the positive numbers, we converted a negative change into a positive value as follows. We considered one as the baseline. A positive change, say 10%, is considered 1.1, whereas a negative change, say -5%, is considered 0.95. If the final value of the geometric mean is less than one, we reported a negative mean value.

5.1 Polybench Benchmarks

We now discuss the results for Polybench benchmarks. We obtained the performance benefits by computing the percentage decrease in the execution time of these benchmarks. The positive and



	gesummv	2mm	3mm	bicg	doitgen	jacobi-1d	jacobi-2d
🟦 Restrict	46.04	3.88	4.63	53.32	10	-1.27	3.43
🟧 Scout	49.37	4.22	4.28	51.11	10.2	11.8	2.37
⬜ Rapid	47.58	3.36	4.3	52.94	4.81	5.39	3.31
🟨 Rapid_w	-0.1	-0.09	-0.09	-0.07	-0.05	-0.12	-0.1

Fig. 6. Polybench performance benefits (in percentage) for `restrict` keyword, Scout and Rapid. (Rapid_w= Worst case performance for Rapid when the dynamic checks always fail, excluding the allocator's overhead (in percentage). Positive and negative numbers for performance benefits and worst-case performance represent improvement and degradation, respectively.)

negative numbers for performance benefits represent the improvement and degradation over the native execution time of the benchmark, respectively.

At first, we compiled and executed these benchmarks with `restrict` keyword with option `POLYBENCH_USE_RESTRICT`, provided by Polybench. This option attaches `restrict` keyword to the function arguments. Users can annotate the function arguments with `restrict` to provide additional aliasing information to the compiler. Compilers trust these annotations during code generation. The `restrict` keyword informs the compiler that the pointer does not alias with any other pointer. We performed this experiment to ensure that Rapid does not miss out any optimization opportunities enabled by the user annotation using `restrict`. Figure 6 shows the performance benefits obtained using Scout and Rapid. The first column shows the performance benefits of these benchmarks using the `restrict` keyword. The second and third columns show the results obtained when compiled with Scout [Chitre et al. 2022] and Rapid, respectively. The fourth column, `Rapid_w`, represents our worst-case performance, which is essentially the overheads of our dynamic checks. Notice that in the worst case, the program always takes the unoptimized path after executing the dynamic checks. Figure 6 shows the benchmarks with more than 3% performance benefits in execution time with Rapid.

For four out of seven benchmarks, namely, `gesummv`, `2mm`, `3mm`, and `bicg`, the performance of Rapid, `restrict`, and Scout are similar. We observed that the `restrict` annotation slowed down the performance of `jacobi-1d`. Scout [Chitre et al. 2022] also observed similar behavior for this benchmark. This is because `jacobi-1d` failed to preserve the non-aliasing annotations across the optimization passes, as also observed by the Scout.

For `jacobi-1d`, the improvement with Rapid is lesser than Scout, whereas, for `jacobi-2d` Rapid performs better than Scout. We observed that the reason behind this is the different allocators used in these benchmarks. Scout used Jemalloc allocator. For `jacobi-1d`, Mimalloc performs better than Jemalloc, and for `jacobi-2d`, Jemalloc performs better than Mimalloc. To further investigate this, we ran Rapid with Jemalloc. For this experiment, we modified the loop-versioning condition to check if two addresses are different (this condition is always true) rather than checking if the top 32-bits are different. Notice that the original loop-versioning condition is not valid for Jemalloc because it doesn't know anything about regions. In this experiment, Rapid reported 11.92% and 2.68% performance benefits for `jacobi-1d` and `jacobi-2d`, respectively. Scout also reported a similar improvement.

The performance benefit for `doitgen` with Rapid is less than using `restrict` keyword and Scout. We found that during compilation with Rapid, loop invariant code motion moved instructions to perform "xor" operations and checking of top 32-bits to the outermost loop. This caused register pressure during the register allocation, and a local variable was spilled resulting in additional memory accesses inside the loop. To verify this, we disabled loop invariant code motion after inserting the dynamic checks. We obtained performance benefits of around 9.02% for this benchmark similar to `restrict` keyword and Scout. We also found that the number of loops versioned by Scout is one, not two, as reported in the Scout [Chitre et al. 2022] paper.

We observed that the slowdown introduced by the dynamic checks (`Rapid_w` in Figure 6) was always less than 0.13% for these six benchmarks. Moreover, the slowdown was always less than 0.3% for all the 30 Polybench benchmarks. These results indicate that the overheads of our dynamic checks are not high for the Polybench benchmarks.

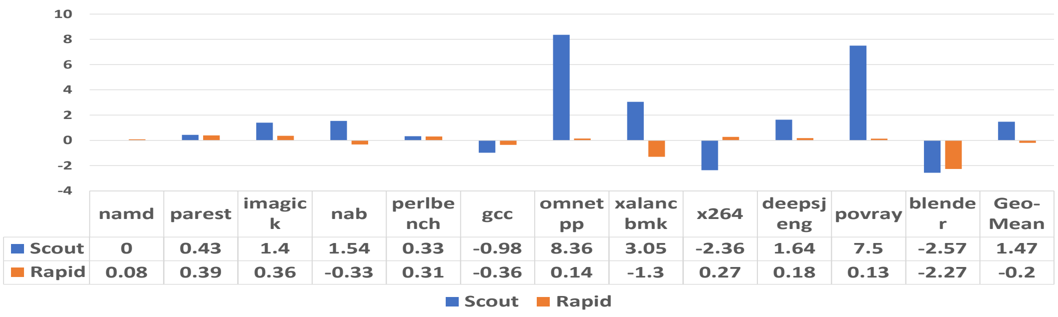
Table 2. Polybench speedups using `restrict` keyword, hybrid approach [Alves et al. 2015], Scout [Chitre et al. 2022] and RAPID. (S_{r3} = Speedup using `restrict` keyword with LLVM-3.6.0, S_{r10} = Speedup using `restrict` keyword with LLVM-10, S_h = Speedup using hybrid approach, S_s = Speedup using Scout, S_r = Speedup using RAPID.)

Benchmark	S_{r3}	S_{r10}	S_h	S_s	S_r
gesummv	1.86	1.86	2.5	1.98	1.91
bicg	2.14	2.14	2.7	2.05	2.13
gramschmidt	1.01	1	1.4	1.01	1.01

We compared our results with the hybrid approach (discussed in [Alves et al. 2015]). The hybrid approach uses both polyhedral [Bondhugula et al. 2008; Bondhugula 2008; Feautrier 1992] and symbolic range [Nazaré et al. 2014; Paisante et al. 2016; Rugina and Rinard 2000] analyses to compute the range based dynamic checks in $O(1)$. Table 2 shows the speedups for three benchmarks that resulted in substantial performance benefits using the hybrid approach . We found that the benefits reported by the hybrid approach are better than our tool. Since the hybrid approach uses LLVM-3.6.0, we performed another experiment using `restrict` keyword with LLVM-3.6.0 to estimate the runtime of these benchmarks using the hybrid approach on our machine. Table 2 shows the speedups obtained when `restrict` keyword is used with LLVM-3.6.0 (Column-2), `restrict` keyword with LLVM-10 (Column-3), the hybrid approach (Column-4), Scout [Chitre et al. 2022] (Column-5), and RAPID (Column-6). We observed that these benchmarks show similar speedups as that of Scout and `restrict` keyword with RAPID. We believe that the performance improvement is lesser than the hybrid approach due to the different processor versions. Alves et al. evaluated the hybrid approach on the second-generation (Sandy Bridge) Intel dual-core i5 @ 1.7GHz, which differs from the processor used in our experiments.

For six benchmarks, namely, `trmm`, `durbin`, `trisolv`, `floyd-warshall`, `nussinov` and `seidel--2d`, RAPID failed to insert the dynamic checks. Five of these benchmarks perform load/store to the same array inside the loop, and the remaining benchmark did not show any potential to be optimized further. A similar observation is mentioned in Scout [Chitre et al. 2022]. We further verified that the enabled set of optimizations for all of Polybench benchmarks was similar to that of Scout by inspecting the IR.

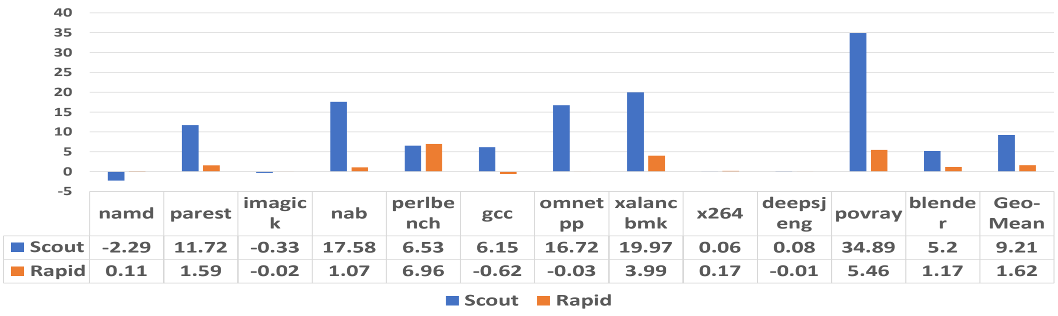
Performance improvements for 23 benchmarks (other than those shown in Figure 6) are in the range of 0% to 2.52%. The geometric mean for the performance improvements of all 30 benchmarks is 0.22%. The maximum number of regions needed for these benchmarks is three. Two regions are sufficient for the majority of the benchmarks. The number of versioned loops is the same as that of Scout for all the benchmarks, except `doitgen` as mentioned above. Our approach enables optimization opportunities such as loop invariant code motion, global value numbering, store elimination, and efficient vectorization for these benchmarks, leading to performance benefits.



	namd	parest	imagic k	nab	perlbe nch	gcc	omnet pp	xalanc bmk	x264	deepsj eng	povray	blende r	Geo-Mean
🟦 Scout	0	0.43	1.4	1.54	0.33	-0.98	8.36	3.05	-2.36	1.64	7.5	-2.57	1.47
🟧 Rapid	0.08	0.39	0.36	-0.33	0.31	-0.36	0.14	-1.3	0.27	0.18	0.13	-2.27	-0.2

🟦 Scout 🟧 Rapid

Fig. 7. CPU overhead of Scout's and Rapid's allocators for CPU SPEC 2017 benchmarks. Negative numbers represent improvement.



	namd	parest	imagic k	nab	perlbe nch	gcc	omnet pp	xalanc bmk	x264	deepsj eng	povray	blende r	Geo-Mean
🟦 Scout	-2.29	11.72	-0.33	17.58	6.53	6.15	16.72	19.97	0.06	0.08	34.89	5.2	9.21
🟧 Rapid	0.11	1.59	-0.02	1.07	6.96	-0.62	-0.03	3.99	0.17	-0.01	5.46	1.17	1.62

🟦 Scout 🟧 Rapid

Fig. 8. Memory overhead of Scout's and Rapid's allocators for CPU SPEC 2017 benchmarks. Negative numbers represent improvement.

For Polybench benchmarks, the memory overhead of our custom allocator lies in the range of -0.68% to 0.09%, with a geometric mean of -0.03%. The CPU overhead lies in the range of -0.73% to 0.5%, with a geometric mean of -0.03%. Scout [Chitre et al. 2022] reported CPU overheads in the range of -2.55% to 5.3% for Polybench benchmarks. The geometric mean CPU overhead in our approach is better than the 0.21% geometric mean overhead reported by Scout. The absolute memory overhead for Polybench benchmarks is always less than 1% for both Scout and our approach.

5.2 CPU SPEC 2017 Benchmarks

We now discuss the results for CPU SPEC 2017 benchmarks. We use two metrics throughout this section,

(1) Overhead represents the percentage increase. Positive and negative numbers represent the degradation and improvement, respectively.
(2) Performance benefits represent the percentage decrease in the benchmark's execution time. Positive and negative numbers represent the improvement and degradation in the execution time, respectively.

5.2.1 CPU and Memory Overhead of the Allocator. We first discuss our region-based allocator's CPU and memory overheads. Figures 7 and 8 respectively show the Rapid allocator's CPU and memory overheads for CPU SPEC 2017 benchmarks. In both figures, the first and second columns correspond to Scout [Chitre et al. 2022] and Rapid. We obtained the CPU overheads by computing the percentage change in the execution times of the benchmarks when the allocator always allocates memory in the default region and when the allocator allocates memory in the regions identified by the `Region Inference` (Section 3). We disabled loop-versioning in this experiment.

The CPU overhead of our allocator is always less than 0.5% with a geometric mean of -0.2%, shown in Figure 7. The memory overhead of our allocator is always less than 7% with a geometric mean of 1.62%. This overhead is more than 3% for only three out of 12 benchmarks. Mimalloc allocates at least one `mi-segment` (a large contiguous area) for every heap region. `mi-segment` is further divided into pages that are used to allocate objects of different sizes. In the case of multiple regions, Mimalloc needs to allocate at least one `mi-segment` for every heap region. Multiple `mi-segments` are adding additional memory overhead in some benchmarks. For `xalancbmk_r` and `blender_r` benchmarks, the multi-region allocation approach improved the performance by 1.3% and 2.27%, respectively. We believe that this might be due to the presence of multiple per-heap allocator caches with different regions. A similar speedup was reported by Scout for the `blender_r` and `x264_r` benchmarks.

Additionally, Figures 7 and 8 respectively show the Scout [Chitre et al. 2022] allocator's CPU and memory overheads for CPU SPEC 2017 benchmarks. Scout reported 3%, 7.5%, and 8.3% CPU overheads for `xalancbmk_r`, `povray_r`, and `omnetpp_r`. Due to the high overheads, Scout can't be used for these benchmarks despite some additional loops being versioned. Apart from these, Scout reported more than 1% overheads for three benchmarks, which was more than the benefits of loop-versioning in these benchmarks. On the other hand, due to the low overhead of our allocator, we could improve the performance of all these benchmarks. We also verified the CPU overhead of Scout's allocator on our machine, which was high for all these benchmarks. The memory overheads in Scout are also substantial. For five benchmarks, the memory overhead in Scout's allocator is more than 10%. For these benchmarks, the memory overhead of Rapid's allocator is always less than 6%. The main reason behind high memory overhead in Scout's allocator is because they add additional padding to satisfy the constraint on the allocation size and alignment of objects. Rapid doesn't change the allocation size and alignment of the objects. Based on the results, we can conclude that our allocator outperforms the allocator used by Scout for most of CPU SPEC 2017 benchmarks.

Table 3. Number of regions required and the number of loops versioned for CPU SPEC 2017 using Rapid. (#R_r = Number of regions required, #L_v = Number of loops versioned.)

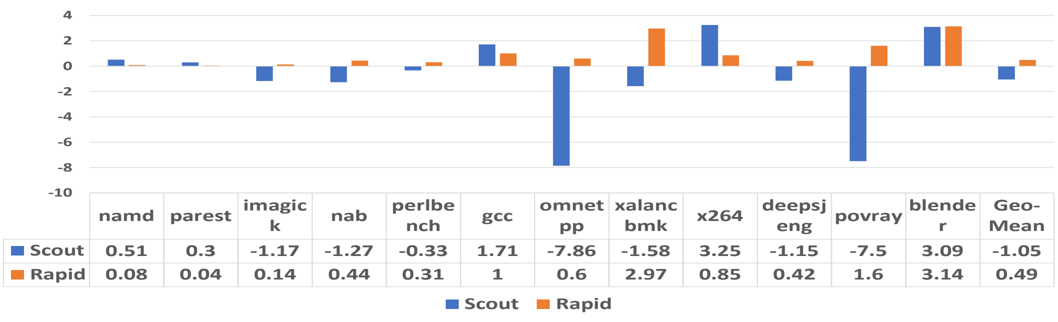
	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender
#R_r	2	4	2	4	3	4	2	3	3	2	2	4
#L_v	9	68	4	21	12	35	6	34	32	6	2	30

5.2.2 Performance Benefits. We now discuss the performance benefits for the CPU SPEC 2017 benchmarks. We compared the performance benefits obtained using Rapid with Scout [Chitre et al. 2022]. Figure 9 shows the results for these benchmarks using Rapid and Scout. Figure 9b shows the performance benefits excluding the allocator's overhead. The first, second, third, and fourth columns correspond to the Scout, Rapid, Scout's worst case, and Rapid's worst case, respectively.

Figure 9a shows the overall performance benefits, including the allocator's overhead. The first and second columns correspond to the Scout and Rapid, respectively. The overall performance benefits for Scout are approximate and computed based on the allocator's overhead and performance benefits reported in the paper.

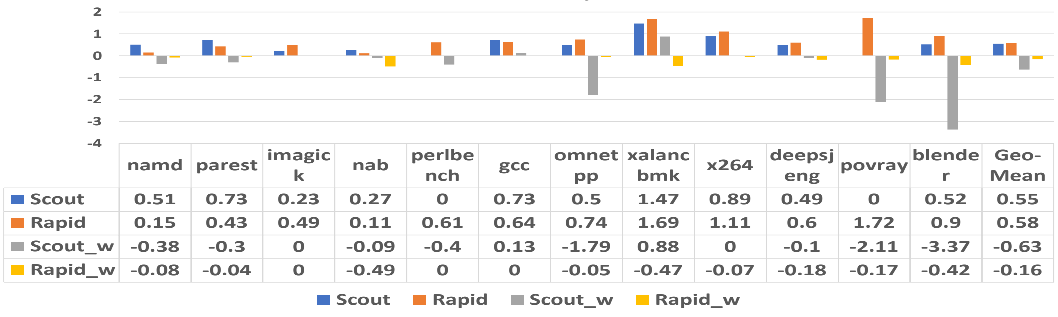
Figure 9 shows results for 12 benchmarks out of 16. The remaining four benchmarks did not show any performance benefits due to the following reasons,

(1) None of the loops shows potential to be optimized further (`lbm_r`, `mcf_r`).
(2) Versioned loops never get executed (`leela_r`, `xz_r`) during the profile phase.



	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender	Geo-Mean
🟦 Scout	0.51	0.3	-1.17	-1.27	-0.33	1.71	-7.86	-1.58	3.25	-1.15	-7.5	3.09	-1.05
🟧 Rapid	0.08	0.04	0.14	0.44	0.31	1	0.6	2.97	0.85	0.42	1.6	3.14	0.49

(a) Performance benefits including allocator's overhead.



	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender	Geo-Mean
🟦 Scout	0.51	0.73	0.23	0.27	0	0.73	0.5	1.47	0.89	0.49	0	0.52	0.55
🟧 Rapid	0.15	0.43	0.49	0.11	0.61	0.64	0.74	1.69	1.11	0.6	1.72	0.9	0.58
⬜ Scout_w	-0.38	-0.3	0	-0.09	-0.4	0.13	-1.79	0.88	0	-0.1	-2.11	-3.37	-0.63
🟨 Rapid_w	-0.08	-0.04	0	-0.49	0	0	-0.05	-0.47	-0.07	-0.18	-0.17	-0.42	-0.16

(b) Performance benefits excluding allocator's overhead.

Fig. 9. CPU SPEC 2017 performance benefits (in percentage) for Rapid and Scout [Chitre et al. 2022]. (Scout_w = Performance benefits over native excluding the allocator's overhead for Scout in the absence of the profiler (in percentage), Rapid_w = Worst-case performance for Rapid when the dynamic checks always fail, excluding the allocator's overhead (in percentage). Positive and negative numbers for performance benefits and worst-case performance represent improvement and degradation, respectively.)

For comparison, we took the maximum improvement reported by Scout [Chitre et al. 2022] for each benchmark across all configurations. Scout reported a performance improvement in the range of 0.23% to 1.47% (Figure 9b) for ten benchmarks, excluding the allocator overhead. However, after considering the allocator overhead, only five benchmarks showed better performance (Figure 9a) . Rapid improved the performance of all 12 benchmarks in which the versioned loops execute at runtime. After excluding the allocator overhead, the benefits are in the range of 0.11% to 1.72% (Figure 9b). The overall improvements (including allocator overhead) for these 12 benchmarks are in the range of 0.04% to 3.14% (Figure 9a). For some benchmarks, the overall improvement is higher because our allocator performs better than the native in these cases. From these results, we conclude that because of the low overhead of our allocator, Rapid can improve the performance of a larger set of benchmarks than Scout.

Table 3 shows the number of regions required and versioned loops for these benchmarks. The maximum number of regions needed for these benchmarks is four. In this experiment, we didn't version loops that require checks for objects allocated at the same site. Apart from `xalancbmk_r`, the dynamic checks never fail for other benchmarks. For `xalancbmk_r`, the dynamic checks fail 90 out of 252267 (0.03%) times across five loops. In these loops, most of the time, the conflicting accesses belong to different regions. However, sometimes the allocation sites of the conflicting memory accesses are the same, so the checks fail. Notice that we version a loop if the dynamic checks succeed most of the time. After handling memory allocations in the loops, none of these

Table 4. Rapid's allocator CPU and memory overhead for CPU SPEC 2017 after handling memory allocations inside the loops. ($CO_R(\#r)$ = CPU overhead (in percentage) for #r regions, $MO_R(\#r)$ = Memory overhead (in percentage) for #r regions. Negative numbers represent improvement.)

Benchmark	CO_R **(8)**	CO_R **(16)**	CO_R **(32)**	CO_R **(128)**	CO_R **(256)**	MO_R **(8)**	MO_R **(16)**	MO_R **(32)**	MO_R **(128)**	MO_R **(256)**
parest_r	0.18	0.25	0.46	0.57	0.57	2.8	3.85	5.14	9.83	15.75
xalancbmk_r	-1.76	-1.67	-1.67	-1.76	-2.13	4.06	4.11	4.22	4.39	4.99
blender_r	-3.67	-3.38	-2.91	-3.9	-3.96	1.98	2.25	3.01	5.88	6.71

Table 5. CPU SPEC 2017 performance benefits after handling memory allocations inside the loops. ($\#L_v$ = Number of loops versioned, P_{wr} = Worst case performance for Rapid across all configurations when the dynamic checks always fail excluding the allocator's overhead (in percentage), $P_b(\#r)$ = Performance benefits over native excluding the allocator's overhead (in percentage) for $\#r$ regions. Positive and negative numbers represent improvement and degradation, respectively.

Benchmark	$\#L_v$	P_{wr}	$P_b(8)$	$P_b(16)$	$P_b(32)$	$P_b(128)$	$P_b(256)$
parest_r	69	-0.25	-0.08	0.15	-0.08	-0.08	0.29
xalancbmk_r	34	-0.48	-0.19	-0.38	-0.48	-0.38	-0.48
blender_r	31	-0.37	-0.25	-0.19	0.3	-0.37	-0.37

Table 6. CPU SPEC 2017 performance benefits using our loop filtering mechanism with the threshold for the number of iterations as 64. ($\#L_v(\#r)$ = Number of loops versioned for #r regions, $P_b(\#r)$ = Performance benefits over native excluding the allocator's overhead (in percentage) for #r regions, $P_{ba}(\#r)$ = Performance benefits over native including the allocator's overhead (in percentage) for #r regions. Positive and negative numbers for performance benefits represent improvement and degradation, respectively.)

Benchmark	$\#L_v$ **(0)**	$\#L_v$ **(> 0)**	P_b **(0)**	P_b **(8)**	P_b **(16)**	P_b **(32)**	P_b **(128)**	P_b **(256)**	P_{ba} **(0)**	P_{ba} **(8)**	P_{ba} **(16)**	P_{ba} **(32)**	P_{ba} **(128)**	P_{ba} **(256)**
parest_r	58	59	0.43	0.08	0.18	0.39	0.67	0.36	0.04	-0.11	-0.08	-0.08	0.11	-0.22
xalancbmk_r	14	14	1.88	0.95	0.85	1.7	1.51	1.33	3.15	2.69	2.5	3.34	3.25	3.43
blender_r	29	30	1.19	0.07	0.19	0.54	-0.07	-0.13	3.43	3.72	3.55	3.43	3.84	3.84

checks failed. Therefore, none of the objects accessed inside the versioned loops actually overlap. Dynamic checks are needed for soundness.

In Figure 9b, `Rapid_w` represents the worst case performance of CPU SPEC 2017 benchmarks with Rapid. Chitre et al. [Chitre et al. 2022] (Scout) uses a profiler to filter loops that don't show improvement at runtime. They provide results for both with and without the profiler. We compare our worst-case performance with the performance of Scout without the profiler represented by `Scout_w` in Figure 9b. This is not the worst-case for Scout because the optimized loops are executed in this setting. We picked this setting because Scout didn't report the worst-case performance. The performance degradation ranges from 0.09% to 3.37% and 0.04% to 0.49% for Scout and Rapid, respectively. Based on the results, the slowdown introduced by Scout in the absence of the profiler could be as high as 3.37%. On the other hand, the slowdown introduced by Rapid was always less than 0.5%, even when the dynamic checks always led to a failure. We can conclude that even in the worst-case scenario, the maximum slowdown introduced by Rapid is not substantial and less than Scout.

We found that in three benchmarks `parest_r`, `xalancbmk_r` and `blender_r`, the allocation sites for conflicting accesses are the same. As discussed in (Section 3.3.2), for conflicting allocation sites, we ask the user to provide the maximum number of regions at compile time that is distributed equally among the conflicting allocation sites after assigning regions to non-conflicting allocation sites. We executed these benchmarks with the maximum number of regions 8, 16, 32, 128, and 256. Table 4 and Table 5 show the results of this experiment. In Table 5, column-2 shows the number of versioned loops, column-3 shows the worst-case performance when the dynamic checks always fail, and columns-4, 5, 6, 7, and 8 show the performance benefits after excluding the allocator overhead for 8, 16, 32, 128, 256 regions, respectively. In `xalancbmk_r`, objects with conflicting and non-conflicting allocation sites are accessed at different times. Therefore, the number of versioned loops in Table 3 and Table 5 are the same.

As discussed, we allocate region-ids for these allocation sites from a set of ids in a round-robin manner in the hope that the conflicting accesses would belong to different regions. As we increase the number of regions, the probability of dynamic check failure decreases. For `parest_r`, with eight regions, the dynamic checks failed 454792 times out of 67025062 (0.67%). However, none of the checks failed with 16 and more number of regions. Similarly, for `xalancbmk_r`, the checks failed for 90 out of 252267 (0.03%) times with our default approach. This number was reduced to 16 (0.01%) with eight regions. For 16 and more number of regions, none of the checks failed. For `blender_r`, none of the dynamic checks failed for 8 and more number of regions. After handling conflicting allocation sites, Rapid could version all the loops versioned by Scout [Chitre et al. 2022] that are executed at runtime. The maximum slowdown introduced by Rapid in the worst-case scenario while versioning all the loops for `parest_r`, `xalancbmk_r` and `blender_r` is still less than 0.5% (Column-3 of Table 5).

Table 4 shows the CPU (CO_R) and memory (MO_R) overheads of the allocator for different numbers of regions. For `parest_r` and `blender_r` benchmarks, the memory overheads of Rapid's allocator increase significantly with a large number of regions. This is because of the delayed reclamation of pages across the heaps in different regions, as discussed in Section 4. The CPU overheads remain low (less than 0.6%) even with a large number of regions. The maximum memory overhead is 15.75% for the `parest_r` benchmark with 256 regions.

The performance degraded (Columns-4, 5, 6, 7, 8 in Table 5) for all of these benchmarks when we enabled loop-versioning (excluding the allocator overhead). Further investigation revealed that versioning some loops with fewer iterations resulted in the generation of relatively unoptimized code for the rest of the function.

To improve further, we use a loop filtering mechanism. We disable loop-versioning for loops with the total number of iterations is less than a specific threshold. Table 6 shows the performance benefits for the different number of regions and loop filtering threshold of 64. The zero number of regions is the case when we don't handle the conflicting allocation sites (as in Figure 9). All the benchmarks showed positive improvement after excluding the allocator overhead (Columns-4, 5, 6, 7, 8, 9), except `blender_r`, which shows slight degradation with regions 128 and 256. The overall performance of `parest_r` was negative (Columns-11, 12, 13, 15) due to allocator overhead except with region 128. The performance of `xalancbmk_r` further improved from 1.69% (Figure 9b) to 1.88% when we filtered loops with fewer iterations.

We performed the same experiment for other benchmarks as well. We also experimented with different thresholds for loop filtering. However, there was no noticeable improvement over the benefits shown in Figure 9. We can conclude that wisely filtering out the loops with small iterations can also improve the performance of the benchmarks.

5.2.3 Impact of Data and Instruction Cache. To understand the impact of the change in the memory layout, we computed the number of L1 data cache loads as shown in Figure 10. The first, second, and third columns correspond to the baseline, Rapid's allocator without dynamic checks, and Rapid, respectively. As expected, we didn't observe any significant change in the number of loads due to the allocator. However, when the dynamic checks are enabled, we observed a reduction in the number of loads for nab_r and x264_r. This is due to additional loop-vectorization and loop invariant code motion (LICM) optimizations enabled by Rapid. The vectorized code can load a 16-byte value using a single instruction. Such loads are counted as a single L1 cache load. The corresponding non-vectorized loop may use four 4-byte reads or two 8-byte reads to read 16 bytes, which are counted as four and two L1 cache loads, respectively. The LICM optimization reduces the number of L1 cache loads because the load is moved outside the loop. Figure 11 shows the percentage miss in the L1 data cache. The cache misses are slightly low for the xalancbmk_r benchmark with Rapid's allocator. We believe that this is due to the change in the memory layout. For xalancbmk_r benchmark, the Rapid's allocator also performs better than the baseline as shown in Figure 7. For other benchmarks, the percentage of cache misses using Rapid's allocator are almost similar.

For parest_r, xalancbmk_r, and blender_r, we also computed the number of L1 data cache loads and percentage miss for 256 regions. The number of loads in billions and percentage misses for these benchmarks using Rapid's allocator were 1183 (9.79), 366 (10.84), and 522 (2.36), which is not very different from the numbers for small regions. Even with the dynamic checks, the numbers are similar for 256 regions.

To understand the impact of the change in the code layout, we computed the number of L1 instruction cache loads as shown in Figure 12. The first, second, and third columns correspond to the baseline, Rapid's allocator without dynamic checks, and Rapid, respectively. Except imagick_r and blender_r, we didn't observe any significant change in the number of loads due to the allocator. Both these benchmarks use per-thread stacks because of the conflicting stack allocations. Allocation and deallocation from per-thread stacks add additional instructions that change the behavior of the instruction cache. With dynamic checks, the number of instruction cache loads varies for more benchmarks because of the optimizations enabled for the loop. The percentage miss in the instruction cache misses is shown in Figure 13. The cache misses reduced significantly for deepsjeng_r benchmarks using dynamic checks. The number of instruction cache loads was also reduced from 503 billion to 493 billion for this benchmark with dynamic checks (Figure 12). We believe this might be due to the reduced code size of the optimized version of the loop. The overall increase in the code size is negligible for most benchmarks except for nab_r (1.09%) and deepsjeng_r (3.1%).

For parest_r, xalancbmk_r, and blender_r, we also computed the number of L1 instruction cache loads and percentage miss for 256 regions. The number of loads in billions and percentage miss for these benchmarks using Rapid's allocator were 465 (0.07), 207 (1.05), and 409 (0.28), which is not very different from the numbers for small regions. For 256 regions, the numbers were similar with dynamic checks.

5.2.4 *Profiler Overhead.*

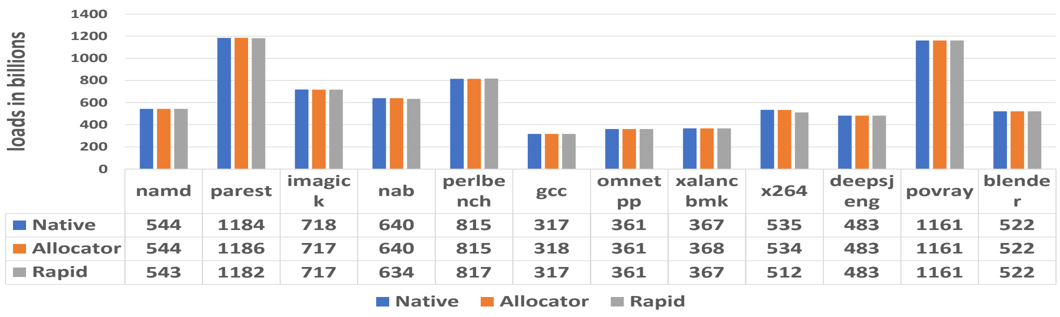
The overhead of the profiler includes the following:

(1) Replacing stack allocations with Mimalloc allocation API and free (Section 4).
(2) Finding object headers at runtime (Section 4) to identify objects and their allocation sites.
(3) Storing the loop-related information to the log file (Section 3).

The slowdown for the CPU SPEC 2017 lies in the range of 1.1% (nab_r) to 751.87% (povray_r). We noticed that replacing stack allocations with Mimalloc allocation API and free contributed significantly to the slowdown for these benchmarks. The slowdown lies between 0.55% (nab_r) to 50.28% (x264_r), excluding the overhead incurred due to replacing stack allocations.

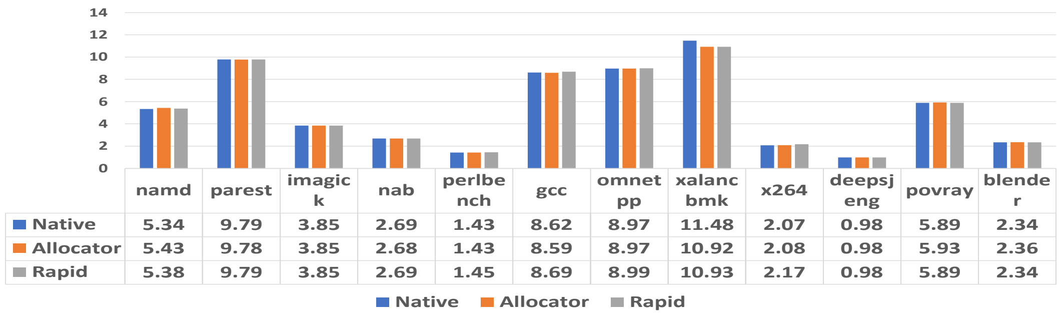
5.2.5 *Code Snippets of Versioned Loop.*

Scout [Chitre et al. 2022] provided the snippets of versioned loops that showed good performance improvement. In addition to the loops reported in Scout,



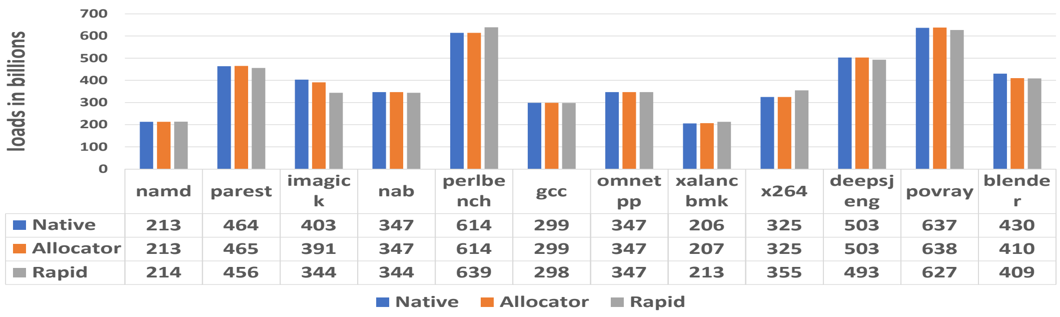
	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender
🟦 Native	544	1184	718	640	815	317	361	367	535	483	1161	522
🟧 Allocator	544	1186	717	640	815	318	361	368	534	483	1161	522
⬜ Rapid	543	1182	717	634	817	317	361	367	512	483	1161	522

Fig. 10. Number of loads (in billions) in data cache (L1) for native, allocator, and Rapid.



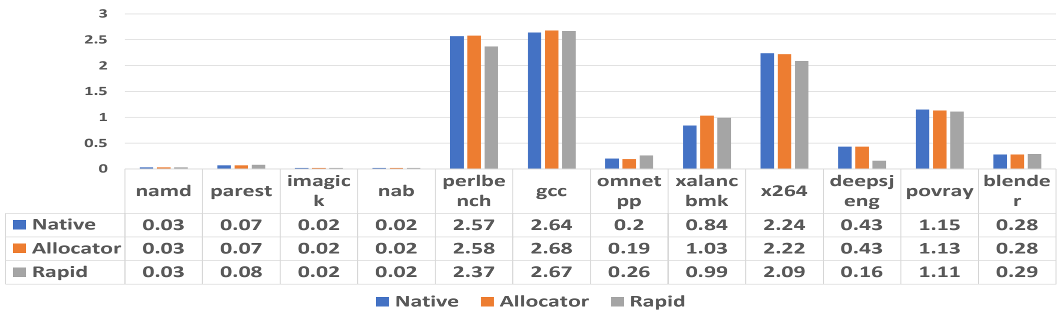
	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender
🟦 Native	5.34	9.79	3.85	2.69	1.43	8.62	8.97	11.48	2.07	0.98	5.89	2.34
🟧 Allocator	5.43	9.78	3.85	2.68	1.43	8.59	8.97	10.92	2.08	0.98	5.93	2.36
⬜ Rapid	5.38	9.79	3.85	2.69	1.45	8.69	8.99	10.93	2.17	0.98	5.89	2.34

Fig. 11. Percentage miss in data cache (L1) for native, allocator, and Rapid.



	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender
🟦 Native	213	464	403	347	614	299	347	206	325	503	637	430
🟧 Allocator	213	465	391	347	614	299	347	207	325	503	638	410
⬜ Rapid	214	456	344	344	639	298	347	213	355	493	627	409

Fig. 12. Number of loads (in billions) in instruction cache (L1) for native, allocator, and Rapid.



	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender
🟦 Native	0.03	0.07	0.02	0.02	2.57	2.64	0.2	0.84	2.24	0.43	1.15	0.28
🟧 Allocator	0.03	0.07	0.02	0.02	2.58	2.68	0.19	1.03	2.22	0.43	1.13	0.28
⬜ Rapid	0.03	0.08	0.02	0.02	2.37	2.67	0.26	0.99	2.09	0.16	1.11	0.29

Fig. 13. Percentage miss in instruction cache (L1) for native, allocator, and Rapid.

```
//nab_r
//sff.c 1193
//Improvement 29%
//prm is a pointer
//to structure.
for(i = 0; i < prm->Natom;
    i++){
  pairlistnp[i] = NULL;
  lpairsnp[i] =
    upairsnp[i] = 0;
}
```

```
//nab_r
//prm.c 1217
//Improvement 30%
//prm is a pointer
//to structure.
for(i = 0; i < prm->Natom;
    i++) {
  if(i + 1 ==
    prm->Ipres[res + 1])
    res++;
  prm->AtomRes[i] = res;
}
```

```
//xalancbmk_r
//ElemStack.cpp 235
//Improvement 42%
//curRow is a pointer
//to structure.
for (; index < curRow->
  fChildCount; index++)
  newRow[index] = curRow->
    fChildren[index];
```

```
//xalancbmk_r
//BaseRefVectorOf.c 282
//Improvement 69%
//fCurCount is a
//class member.
for (; index < fCurCount;
  index++)
  newList[index] =
    fElemList[index];
```

```
//xalancbmk_r
//ValueVectorOf.c 247
//Improvement 29%
//fCurCount is a
//class member.
for(unsigned int index = 0;
  index < fCurCount; index++)
   newList[index] =
     fElemList[index];
```

```
//x264_r
//macroblock.c 388
//Improvement 40%
//h is a pointer
//to structure.
for( int i = 0; i < h->
  i_ref1; i++ )
  h->fdec->ref_poc[1][i] =
    h->fref1[i]->i_poc;
```

Fig. 14. Loops showing a performance improvement of more than 20% from CPU SPEC 2017 benchmarks in addition to the code snippets shown by Scout [Chitre et al. 2022].

Rapid could substantially improve the performance of some additional loops shown in Figure 14. For these code snippets, either the loop bounds are loaded from a pointer argument or represent a class member or structure field. The approach followed by Rapid can handle such loops, which frequently occur in real-world benchmarks.

Rapid always improved the performance of the CPU SPEC 2017 benchmarks without handling conflicting memory allocations. The loop filtering mechanism further enhanced the performance of the three benchmarks. However, the performance benefits obtained were less when a large number of regions were used to handle conflicting memory allocations. We conclude that Rapid can improve the performance of most benchmarks without creating too many regions.

6 LIMITATION

The default segment size in our approach is 4GB, which is also the maximum allocation size supported by our tool. Application developers can change the segment size using a compile time parameter if the application may allocate an object larger than 4GB.

7 RELATED WORK

Alias analysis is a highly explored research field. Many static alias analyses have been proposed in the past [Andersen and Lee 2005; Cooper and Kennedy 1989; Hardekopf and Lin 2007, 2009, 2011; Hind et al. 1999; Lattner et al. 2007; Pearce et al. 2007; Pereira and Berlin 2009; Steensgaard 1996; Sui and Xue 2016; Zheng and Rugina 2008]. Alias analysis can be performed in two ways, intraprocedural and interprocedural. Intraprocedural analyses are scalable but imprecise. Interprocedural analyses are more precise, but they are not scalable [Andersen and Lee 2005; Sui and Xue 2016]. Some attempts have been made to make these analyses scalable [Hardekopf and Lin 2007; Pereira and

Berlin 2009; Steensgaard 1996], at the cost of losing precision. The imprecision leads to missing out on some optimization opportunities.

To deal with imprecision associated with the scalable alias analysis, another class of approaches combines loop-versioning with statically generated dynamic checks [llv 2023; Alves et al. 2015; Naishlos 2004]. Alves et al. [Alves et al. 2015] uses polyhedral [Bondhugula et al. 2008; Bondhugula 2008; Feautrier 1992] and symbolic range [Nazaré et al. 2014; Paisante et al. 2016; Rugina and Rinard 2000] analyses to insert $O(1)$ dynamic checks based on the range of memory accesses inside the loop body. The LLVM compiler uses a scalar evolution analysis [Engelen 2000, 2001] to compute the dynamic check for loop-versioning. However, these approaches work when the loop bounds are loop invariants.

Alves et al. [Alves et al. 2015] also proposed a purely dynamic approach that works for loops in which the loop bounds are not loop invariants. The basic idea in this approach is that if two pointers point to different objects, they can't be aliases, no matter how they are accessed. To uniquely identify an object, they compute the starting address of the object from a pointer address. In their scheme, a red-black tree stores the range of addresses for every object. Finding the starting address of the object requires a red-black tree lookup costing $O(log\ n)$ fetch operations. Therefore, the overhead of this approach is substantially high for real-world benchmarks. Our work aims to improve the performance of dynamic checks in this approach.

Chitre et al. [Chitre et al. 2022] further reduced the cost of dynamic checks in their purely dynamic approach by constraining the object size and alignment. In this scheme, only one memory access is required in a dynamic check, which is very efficient; therefore, it could improve the performance of some real-world benchmarks. However, the problem is not fully solved by this approach because constraining the object size and alignment introduced significant CPU and memory overheads for several benchmarks.

Campos et al. [Sperle Campos et al. 2016] implements code versioning at the function granularity. The optimized version assumes that the pointer arguments don't overlap. Based on the alias relationships among the pointer arguments known at a call site, the relevant version of the function is called. At a call site, alias relationships among function parameters are computed using a combination of static analysis and dynamic checks. Our tool can be improved using the ideas presented in this paper.

Our technique relies on a profiler to reduce the cost of dynamic checks for data-dependency. The idea of profiling to compute data-dependency information has been proposed by several works [Chen et al. 2004; Fernandez and Espasa 2002; Huang et al. 1994; Lin et al. 2003]. These approaches use a profiler to identify alias information that is very likely to hold at runtime. The primary motivation of these works is to use the profiled information for speculative execution – such as removing loads from the critical path. If the data-dependency assumptions don't hold at runtime, the control is transferred to a recovery code that emulates the correct behavior before returning to the original execution. For example, a check is performed before every load instruction to verify that the previous load value is still valid. If the check fails, the recovery code reloads the value before executing the rest of the code. These kinds of optimizations yielded good performance during the time these tools were developed because of the high memory latency during that time. However, due to the large cache size and better data prefetching by the hardware, these techniques might not offer similar performance improvements on modern hardware.

Our technique builds an interference graph to compute the regions for conflicting memory accesses. An interference graph is also used for register allocation. Chaitin et al. [Chaitin et al. 1981] use a graph coloring algorithm to assign colors to the nodes in the interference graph. Here, color represents a register in the register allocation algorithms and a region in our algorithm. In general, the graph coloring problem is NP-complete. Chaitin [Chaitin 1982] further improved the performance of register allocation by using an approximation algorithm for graph coloring, which is very efficient. We also use the same algorithm in our scheme. The linear scan algorithms [Poletto and Sarkar 1999; Sarkar and Barik 2007; Traub et al. 1998; Wimmer and Franz 2010] are considered a simple and fast alternative to graph coloring for register allocation. These techniques rely on the control flow of the routine. In our case, because there is no notion of control flow, these techniques can't be used.

8 CONCLUSION

We have presented Rapid, a tool that combines loop-versioning with constant-time region-based dynamic checks. Rapid uses a profiler to identify the objects that require dynamic checks for disambiguation and allocates them in different regions. We found that the number of regions needed for the real-world benchmarks is small, and thus the additional overhead of our region-based allocator is very low. In contrast, previous techniques reported high CPU overheads for either allocator or dynamic checks, making them unsuitable for many real-world benchmarks. Rapid enabled many optimization opportunities such as load/store elimination, loop invariant code motion, and code vectorization. Consequently, Rapid could improve the overall performance of all 12 CPU SPEC 2017 benchmarks in which the versioned loops were executed at runtime. These results show that our technique is effective and can scale to large real-world benchmarks.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback, that helped improve the presentation of the paper. The first author would like to acknowledge TCS Foundation for supporting her research through the TCS Research Scholar Program.

DATA AVAILABILITY STATEMENT

The source code repository [Rap 2023a,b] of Rapid is available at URLs https://doi.org/10.5281/zenodo.8321488 and https://github.com/khushboochitre/artifact_rapid.git.

REFERENCES

2016 (accessed Aug 12, 2023). Tutorial-Perf Wiki. https://perf.wiki.kernel.org/index.php/Tutorial

2023 (accessed Apr 11, 2023). Mimalloc source code. https://github.com/microsoft/mimalloc

2023 (accessed Apr 11, 2023). Runtime Checks of Pointers. https://llvm.org/docs/Vectorizers.html#runtime-checks-of-pointers

2023 (accessed Sep 2, 2023)a. Rapid Artifact DOI. https://doi.org/10.5281/zenodo.8321488

2023 (accessed Sep 2, 2023)b. Rapid Artifact Github Repository. https://github.com/khushboochitre/artifact_rapid.git

Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime Pointer Disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 589–606. https://doi.org/10.1145/2814270.2814285

Lars Ole Andersen and Peter Lee. 2005. Program Analysis and Specialization for the C Programming Language.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

Uday Kumar Bondhugula. 2008. *Effective automatic parallelization and locality optimization using the polyhedral model.* Ph. D. Dissertation. The Ohio State University.

Gregory J Chaitin. 1982. Register allocation & spilling via graph coloring. *ACM Sigplan Notices* 17, 6 (1982), 98–101. https://doi.org/10.1145/872726.806984

Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. 1981. Register allocation via coloring. *Computer languages* 6, 1 (1981), 47–57.

Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. 2004. Data Dependence Profiling for Speculative Optimizations, Vol. 2985. 2733–2733. https://doi.org/10.1007/978-3-540-24723-4_5

Khushboo Chitre, Piyus Kedia, and Rahul Purandare. 2022. The Road Not Taken: Exploring Alias Analysis Based Optimizations Missed by the Compiler. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 153 (oct 2022), 25 pages. https://doi.org/10.1145/3563316

Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. 1997. A new algorithm for partial redundancy elimination based on SSA form. *ACM Sigplan Notices* 32, 5 (1997), 273–286. https://doi.org/10.1145/258916.258940

K. D. Cooper and K. Kennedy. 1989. Fast Interprocedual Alias Analysis. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 49–59. https://doi.org/10.1145/75277.75282

Robert Engelen. 2000. Symbolic Evaluation of Chains of Recurrences for Loop Optimization. (03 2000).

Robert van Engelen. 2001. Efficient Symbolic Analysis for Optimizing Compilers. In *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*. Springer-Verlag, Berlin, Heidelberg, 118–132.

Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming* 21, 5 (1992), 313–347. https://doi.org/10.1007/BF01407835

M. Fernandez and R. Espasa. 2002. Speculative alias analysis for executable code. In *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*. 222–231. https://doi.org/10.1109/PACT.2002.1106020

Ben Hardekopf and Calvin Lin. 2007. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 290–299. https://doi.org/10.1145/1250734.1250767

Ben Hardekopf and Calvin Lin. 2009. Semi-Sparse Flow-Sensitive Pointer Analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 226–238. https://doi.org/10.1145/1480881.1480911

Ben Hardekopf and Calvin Lin. 2011. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 289–298.

Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural Pointer Alias Analysis. *ACM Trans. Program. Lang. Syst.* 21, 4 (jul 1999), 848–894. https://doi.org/10.1145/325478.325519

A. S. Huang, G. Slavenburg, and J. P. Shen. 1994. Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* (Chicago, Illinois, USA) *(ISCA '94)*. IEEE Computer Society Press, Washington, DC, USA, 200–210. https://doi.org/10.1145/192007.192012

Ralf Karrenberg and Sebastian Hack. 2011. Whole-Function Vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 141–150.

Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 278–289. https://doi.org/10.1145/1250734.1250766

Daan Leijen, Benjamin G. Zorn, and Leonardo Mendonça de Moura. 2019. Mimalloc: Free List Sharding in Action. In *APLAS*.

Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. 2003. A Compiler Framework for Speculative Analysis and Optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 289–299. https://doi.org/10.1145/781131.781164

Dorit Naishlos. 2004. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*. 105–118.

Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. 2014. Validation of Memory Accesses through Symbolic Analyses. *SIGPLAN Not.* 49, 10 (Oct. 2014), 791–809. https://doi.org/10.1145/2714064.2660205

Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. 2016. Symbolic range analysis of pointers. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 171–181. https://doi.org/10.1145/2854038.2854050

David J. Pearce, Paul H.J. Kelly, and Chris Hankin. 2007. Efficient Field-Sensitive Pointer Analysis of C. *ACM Trans. Program. Lang. Syst.* 30, 1 (nov 2007), 4–es. https://doi.org/10.1145/1290520.1290524

Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave Propagation and Deep Propagation for Pointer Analysis. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. IEEE Computer Society, USA, 126–135. https://doi.org/10.1109/CGO.2009.9

Massimiliano Poletto and Vivek Sarkar. 1999. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (sep 1999), 895–913. https://doi.org/10.1145/330249.330250

Radu Rugina and Martin Rinard. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Sigplan Notices* 35, 5 (2000), 182–195. https://doi.org/10.1145/358438.349325

Vivek Sarkar and Rajkishore Barik. 2007. Extended Linear Scan: An Alternate Foundation for Global Register Allocation. In *Proceedings of the 16th International Conference on Compiler Construction* (Braga, Portugal) *(CC'07)*. Springer-Verlag, Berlin, Heidelberg, 141–155.

Victor Hugo Sperle Campos, Péricles Rafael Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. 2016. Restrictification of Function Arguments. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 163–173. https://doi.org/10.1145/2892208.2892225

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

Rishi Surendran, Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. 2014. Inter-iteration Scalar Replacement Using Array SSA Form. In *CC*.

Omri Traub, Glenn Holloway, and Michael D. Smith. 1998. Quality and Speed in Linear-Scan Register Allocation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 142–151. https://doi.org/10.1145/277650.277714

Christian Wimmer and Michael Franz. 2010. Linear Scan Register Allocation on SSA Form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 170–179. https://doi.org/10.1145/1772954.1772979

Xin Zheng and Radu Rugina. 2008. Demand-Driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 197–208. https://doi.org/10.1145/1328438.1328464

Received 2023-04-14; accepted 2023-08-27