



# The Road Not Taken: Exploring Alias Analysis Based Optimizations Missed by the Compiler

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

Context-sensitive inter-procedural alias analyses are more precise than intra-procedural alias analyses. However, context-sensitive inter-procedural alias analyses are not scalable. As a consequence, most of the production compilers sacrifice precision for scalability and implement intra-procedural alias analysis. The alias analysis is used by many compiler optimizations, including loop transformations. Due to the imprecision of alias analysis, the program's performance may suffer, especially in the presence of loops.

Previous work proposed a general approach based on code-versioning with dynamic checks to disambiguate pointers at runtime. However, the overhead of dynamic checks in this approach is $O(log\ n)$, which is substantially high to enable interesting optimizations. Other suggested approaches, e.g., polyhedral and symbolic range analysis, have $O(1)$ overheads, but they only work for loops with certain constraints. The production compilers, such as LLVM and GCC, use scalar evolution analysis to compute an $O(1)$ range check for loops to resolve memory dependencies at runtime. However, this approach also can only be applied to loops with certain constraints.

In this work, we present our tool, Scout, that can disambiguate two pointers at runtime using single memory access. Scout is based on the key idea to constrain the allocation size and alignment during memory allocations. Scout can also disambiguate array accesses within a loop for which the existing $O(1)$ range checks technique cannot be applied. In addition, Scout uses feedback from static optimizations to reduce the number of dynamic checks needed for optimizations.

Our technique enabled new opportunities for loop-invariant code motion, dead store elimination, loop-vectorization, and load elimination in an already optimized code. Our performance improvements are up to 51.11% for Polybench and up to 0.89% for CPU SPEC 2017 suites. The geometric means for our allocator's CPU and memory overheads for CPU SPEC 2017 benchmarks are 1.05%, and 7.47%, respectively. For Polybench benchmarks, the geometric mean of CPU and memory overheads are 0.21% and 0.13%, respectively.

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

Additional Key Words and Phrases: LLVM, alias analysis, dynamic checks, loop-versioning, optimizations.

**ACM Reference Format:**
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 (October 2022), 25 pages. https://doi.org/10.1145/3563316

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, Computer Science and Engineering, IIIT-Delhi, Okhla - Phase 3, New Delhi, Delhi, 110020, India, purandare@iiitd.ac.in, The School of Computing and University of Nebraska–Lincoln, 256 Avery Hall, Lincoln, NE, 68588, USA, rahul@unl.edu.


2475-1421/2022/10-ART153
https://doi.org/10.1145/3563316

## 1 INTRODUCTION

Alias analysis is the backbone of many compiler optimizations such as automatic vectorization [Karrenberg and Hack 2011], loop invariant code motion, various loop transformations, and scalar promotion that includes dead store elimination and load elimination [Chow et al. 1997; Surendran et al. 2014], etc. These transformation passes rely on information concerning memory dependencies. This is where alias analysis plays its role. Alias analysis determines whether a pair of pointers has overlapping memory addresses, i.e., whether the pointers depend on each other. Alias analysis is, therefore, an essential aspect of the compiler. A variety of algorithms have been proposed in the past to perform alias analysis [Andersen and Lee 2005; Cooper and Kennedy 1989; Diwan et al. 1998; Hardekopf and Lin 2009, 2011; Hind et al. 1999; Jaiswal et al. 2018; Lattner et al. 2007; Pearce et al. 2007; Shapiro and Horwitz 1997; Steensgaard 1996].

Alias analysis is undecidable in the presence of conditional statements, loops, dynamic storage, and recursive data structure [LANDI 1992; Ramalingam 1994]. The efficiency of an alias analysis is measured in terms of scalability and precision. Intra-procedural alias analyses [Hardekopf and Lin 2011; Zheng and Rugina 2008] tend to have low precision but have high scalability. On the other hand, context-sensitive inter-procedural alias analyses [Berndl et al. 2003; Whaley and Lam 2004; Zhu 2005] have high precision but are not scalable. Many production compilers sacrifice precision for scalability and rely on intra-procedural alias analysis to resolve data dependency.

Due to these limitations of static alias analysis, existing techniques [Alves et al. 2015; Doerfert et al. 2017; Sampaio et al. 2017] and production compilers [lvR 2022; Naishlos 2004] implement loop-versioning with dynamic checks for overlapping memory accesses within a loop. The loop version that does not have overlapping accesses can be optimized further by the loop transformation passes.

Polyhedral model [Bondhugula et al. 2008; Feautrier 1992] and symbolic range analysis [Nazaré et al. 2014; Paisante et al. 2016; Rugina and Rinard 2000] can compute an $O(1)$ dynamic range check to disambiguate the memory regions accessed within a loop. These checks are placed outside the loop and used as a condition for loop-versioning. These techniques can disambiguate memory regions even if they belong to the same array. The polyhedral analysis requires loop bounds and all data access functions to be an affine combination of loop induction variables and loop invariants [Alves et al. 2015; Bondhugula et al. 2008]. Symbolic range analysis can additionally handle non-affine accesses within a loop; however, polyhedral analysis is more precise when both of these techniques can be applied to a loop. Both polyhedral analysis and symbolic range analyses require:

(1) Loop bounds to be loop invariants, and
(2) Loop to be iterated using affine induction variable.

The existing implementation of the LLVM compiler also uses loop-versioning and $O(1)$ dynamic checks to improve the program's performance by disambiguating pointers at runtime. However, it also requires loops to satisfy the conditions mentioned above. The LLVM compiler uses scalar evolution analysis [Van Engelen 2000, 2001] to compute the loop bounds and affine memory accesses within a loop.

Consider the following example:

```
void foo(int *a, int *b, int *c, int *size) {
  for (int i = 0; i < *size; i++)
      a[i] = b[i] + c[i];
}
```

In function `foo`, the upper bound of the loop (i.e., `*size`) is not a loop invariant, because it can overlap with the array `a` which is being updated in the loop. Thus, neither polyhedral analysis

nor symbolic range analysis can insert a dynamic check to disambiguate the array accesses within the loop. For such loops, Alves et al. [Alves et al. 2015] propose another technique named purely dynamic analysis that uses dynamic checks to infer if the memory accesses are performed on different memory allocations. If so, they safely conclude that the memory accesses do not overlap, assuming that the behavior of out-of-bounds object accesses is undefined. Thus, this technique does not add any constraint on the array indices used to access the memory inside the loop. The key idea in this scheme is to attach a unique tag to every object during allocation. At runtime, a red-black tree lookup is used to compute the starting address of the object from an internal address of an object. The starting address also serves as a unique tag for the object. Two pointer addresses never overlap if the starting addresses of the corresponding objects (or tags) are different. The cost of these checks is logarithmic in terms of the number of live memory allocations ($O(log\ n)$ where n represents the number of live memory allocations). Such high overheads for dynamic checks make it unsuitable for large applications. Our work is motivated by the purely dynamic approach suggested by Alves et al. [Alves et al. 2015]. However, our focus is on minimizing the overheads introduced by the dynamic checks that would make the approach practical for real-world applications.

We would also like to argue that such code patterns are not uncommon, in which the loop bounds are not loop invariants. In Figure 6, we have listed some code patterns from the CPU SPEC 2017 benchmark suite that show more than 20% improvement with our technique. Thus, optimizing these loops is a significant problem that needs to be addressed.

This paper focuses on reducing the overhead of dynamic checks for loops on which the polyhedral analysis or symbolic range analysis cannot be applied. Our approach, encapsulated in a tool called Scout[1], performs dynamic checks for these loops in $O(1)$. The key idea is to constrain the size and alignment during memory allocations that enable Scout to disambiguate two pointers in just one memory access. The overheads of our checks are considerably lower than the $O(log\ n)$ checks required by Alves et al. [Alves et al. 2015].

Scout implements loop-versioning for an already optimized loop with runtime checks for disambiguation of arrays that are accessed within the loop. The loop transformations are performed on the loop version in which the array accesses do not overlap. If the additional non-overlapping information actually enables some optimization, Scout keeps both versions of the loop; otherwise, it rollbacks to the original code. Furthermore, the dynamic checks for disambiguation are added for only those array accesses that are needed to enable the loop transformation. Scout statically analyzes the transformed loop and the original loop to identify checks that are needed for the transformation.

The potential benefits of our loop-versioning algorithm cannot be inferred solely at compile time because it further depends on the path taken at runtime or how many times the loop executes. Additionally, Scout uses a runtime profiler to identify loops for which the cost of dynamic checks is higher than the benefits of loop transformation. Scout disables loop-versioning for these loops.

We integrated Scout with LLVM by implementing a new pass in the LLVM compilation infrastructure. We evaluated our technique for Polybench and CPU SPEC 2017 benchmark suites.

This paper makes the following contributions:

(1) An efficient technique that allows the compiler to explore more optimization opportunities based on alias analysis. Our novel technique combines loop-versioning with constant time dynamic check, explained in Section 3, to disambiguate memory accesses. The dynamic checks

[1]The source code and other artifacts are available at https://doi.org/10.5281/zenodo.7089827 and https://github.com/khushboochitre/Scout-Artifact.git.

are performed using a custom allocator, which is presented in Section 3.3. The technique further uses a feedback mechanism to reduce the number of dynamic checks, explained in Section 3.
(2) Implementation and integration of our technique with LLVM.
(3) Estimation of the performance benefits for the Polybench and CPU SPEC 2017 benchmark suites.

The rest of the paper is organized as follows. Section 2 provides the relevant background knowledge. Section 3 discusses in detail the approach encapsulated by Scout. The implementation details are discussed in Section 4. Section 5 presents and discusses the results. Some of the prior work done in this area has been discussed in Section 6. Section 7 provides the concluding remarks.

## 2 BACKGROUND

### 2.1 Alias Analysis

Alias analysis determines whether a given pair of pointers point to the same memory locations or not. Generally, there can be three types of relationships between a pair of pointers: *must-alias*, *no-alias* and *may-alias*. The *must-alias* relationship represents that the pointers point to the same memory locations during execution at given program locations. The *no-alias* relationship means that the pointers never overlap during execution. The *may-alias* relationship represents that the pointers sometimes have overlapping memory addresses and sometimes do not. Out of these three relations, the *must-alias* and *no-alias* relationships are useful since these relationships provide deterministic information to the compiler allowing it to explore optimization opportunities. These optimizations are described in the following section.

### 2.2 Optimizations Leveraging Alias Information

The optimizations use alias information to generate a more efficient program version. This section describes the optimizations used in our study that rely on alias information.

(1) **Loop Invariant Code Motion (LICM):** Code that can be moved outside the loop body without affecting the semantics of the program is known as loop invariant code. This optimization focuses on moving such code outside the loop body. Loop invariant code can be identified using the *no-alias* information provided by alias analysis. The hoisted code executes less often, improving the performance of the program.
(2) **Global Value Numbering (GVN):** This optimization removes partially or fully redundant code without affecting the semantics of the program. Every variable or expression is assigned a symbolic value. A pair of expressions or variables are assigned the same symbolic values if proved equivalent by the alias analysis, and one can be eliminated. This optimization is performed across basic blocks. Thus, it is known as global value numbering.
(3) **Dead Store Elimination (DSE):** A store that assigns value to a variable that is never used is known as a dead store. These dead stores waste processor's cycles. This optimization focuses on removing such stores based on the information obtained by alias analysis.
(4) **Loop Vectorization (LV):** This optimization allows one operation to be performed on multiple pairs of operands in parallel, which can be achieved using a special type of instructions known as vector instructions. This kind of optimization can help in improving the performance of the program significantly since it allows computations to be performed in parallel. The loop can only be vectorized when there are no memory dependencies inside the loop body. This information can be obtained from alias analysis.
(5) **Superword-Level Parallelism Vectorizer (SLP):** SLP vectorization identifies similar types of independent instructions in a basic block. It converts the identified instructions into vector

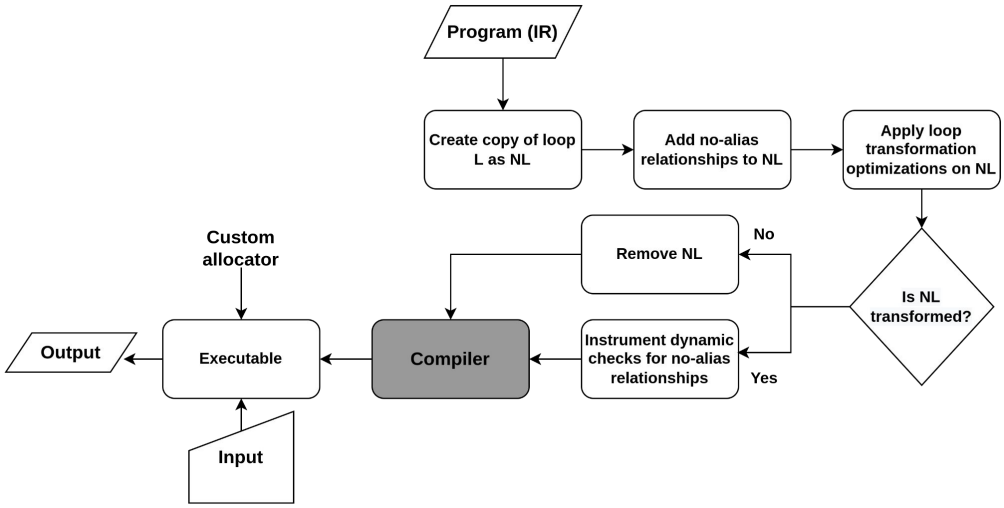


Fig. 1. Architecture of Scout.

instructions. This conversion allows operations to be performed in parallel instead of one at a time.

## 3 SCOUT

### 3.1 Design Overview

Figure 1 shows the architecture of Scout. Scout works on the input program's intermediate representation (IR). It performs versioning for an already optimized loop with checks for non-aliasing (non-overlapping) memory accesses within the loop. Loop transformation optimizations are performed on the non-aliasing version of the loop. If the loop is not transformed, it is discarded; otherwise, dynamic checks to detect non-aliasing memory accesses are inserted into the program. The transformed program is sent to the compiler to generate the final executable. At runtime, a custom memory allocator is linked to the executable generated by Scout. Now, we will discuss the overview of our approach using the example in Figure 2a.

As discussed earlier, the loop at line-1 cannot be transformed using polyhedral or symbolic range analysis because the upper bound of the loop is not a loop invariant. Scout inserts dynamic checks to rule out the possibility of overlapping between `a`, `b`, `c` and `size`. The dynamic checks ensure that memory accesses using `a, b, c` and `size` belong to different objects, and thus they cannot overlap no matter what the value of `i` is at runtime. For fast dynamic checks, Scout sets the allocation size and the alignment of an object during the allocation to the nearest $2^K$, where $2^K$ `>= allocation size`. The objects are allocated from segments. All objects on a segment are of the same size, which is a power of two. The starting address of a segment is always aligned to 4GB. The object size for a segment is stored on the top of the segment. We discuss the structure of a segment in more detail in Section 3.3 using Figure 3. The starting address of a segment can be computed using just an "and" operation from any internal address of the segment using the alignment property of the segment. Two objects can overlap if and only if they belong to the same segment. We can

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

(a) The original code.

```
void foo(int *a, int *b, int *c, int *size) {
3.  size_t s = GetObjectSize(a);
4.  if(IsNoAlias(a, b, s) &&
5.     IsNoAlias(a, c, s) &&
6.     IsNoAlias(a, size, s)) {
7.    int t = *size;
8.    int i;
9.    for (i = 0; i+3 < t; i = i+4)
10.      a[i:i+3] = b[i:i+3] + c[i:i+3];
11.
12.   for (; i < t; i++)
13.     a[i] = b[i] + c[i];
14. }
15. else {
16.   for(int i = 0; i < *size; i++)
17.     a[i] = b[i] + c[i];
18. }
}
```

(b) The transformed code using Scout.

Fig. 2. An example to demonstrate the overview of Scout. Scout inserts dynamic checks to rule out the possibility of overlapping between a, b, c and size. The compiler could vectorize the loop in the scope in which memory accesses do not overlap.

check if two objects on a segment overlap using an "xor" operation and compare it with the object size of the segment. To check non-overlapping (non-aliasing) of two pointer addresses, we first need to compute the object size of the first pointer (at line-3). The GetObjectSize routine uses the alignment property of the segment to compute the size in just one memory access. Once we know the size, it is passed to the IsNoAlias routine, which takes two pointers and the object size of first pointer as input and returns true if the pointers point to different objects. This routine does not access any memory. In this example, Scout adds dynamic alias checks for every read-write and write-write pointer pair because the static alias analysis returns *may-alias* for each of these queries. Scout inserts additional metadata in the if-block such that the compiler can treat each of these pointer pairs (in the if-block) as *no-alias* during the transformation passes. After this, Scout performs loop transformation optimizations for the loop in the if-block. With the additional non-aliasing information, the compiler can now vectorize the loop in the if-block, as shown in Figure 2b. At line-10, the syntax "i:i+3" is used to denote that four parallel writes at indices i, i+1, i+2, i+3 are performed using a single instruction. If the compiler cannot transform the loop with the additional non-aliasing information in the if-block, Scout restores the original code.

Unlike LV, other optimizations, i.e., LICM, DSE, and GVN, do not require checks for every read-write and write-write pair. For example, if a load LD is moved outside the loop, we need to check that LD does not overlap with the writes present in the loop. Let us consider that instead of vectorizing the loop, the loop transformation pass in Figure 2 only moves *size outside the loop (at line-7). In this case, we need to check that "size" and "a" do not overlap instead of adding three checks as in the case of vectorization. Scout statically analyzes the original and transformed loop in the if-block and adds only those checks needed for the transformation.

Finally, it is possible that the optimized path is rarely taken at runtime, or the overhead of the runtime checks is more than the benefit of the loop transformation. Scout uses a profiler to disable loop-versioning for such loops.

We will now discuss our approach in detail in the rest of this section. In Section 3.2, we discuss the loop-versioning algorithm. Section 3.3 discusses the structure of our custom allocator. In Section 3.4, we describe the mechanism to perform the dynamic checks in $O(1)$ memory access. Finally, in Section 3.5, we describe the profiler.

### 3.2 Loop-versioning

Algorithm 1 shows the steps followed by Scout to perform loop-versioning. The algorithm takes a function IR `F` and a loop `L` as arguments. This algorithm is performed after all loop transformations except vectorization have been performed on L. The reason is that it is hard to optimize an already vectorized loop with additional non-overlapping information. However, the additional non-aliasing information enables further transformations in a non-vectorized loop. Vectorization is later performed on the transformed loop.

```
   Input: Function F, Loop L
   Result: Transformed/Untransformed Loop
1  begin
2      IsVectorizableBefore ← isVectorizable(L);
3      NL ← cloneLoopWithDummyCheck(L);
4      insertNoAliasRelation(NL);
5      RunLICM(NL);
6      RunGVN(F);
7      RunDSE(F);
8      IsVectorizableAfter ← isVectorizable(NL);
9      if isLoopTransformed(L, NL) or
         (not IsVectorizableBefore and IsVectorizableAfter) then
10         identifyDynamicChecks(L, NL);
11         insertDynamicChecks(L, NL);
12         removeNoAliasRelation(NL);
13     else
14         removeLoopAndDummyCheck(NL);
15     end
16 end
```

Algorithm 1: `Loop-versioning` algorithm of Scout.

At line-3, the algorithm creates a copy of the original loop and inserts a dummy check for non-aliasing. We refer to the versioned loop as `NL` in this section. At line-4, it inserts *no-alias* metadata in the non-overlapping version of the loop. The compiler can use *no-alias* metadata to infer that two memory accesses do not overlap. The *no-alias* metadata is added for only those pointer pairs whose bases are loop invariants, and it is safe to access at least one of them inside the loop's preheader. It then performs LICM, GVN, and DSE optimizations (from line-5 to line-7) on the non-overlapping version of the loop (i.e., `NL`). The `isVectorizable` routine at line-2 and line-8 does not vectorize the loop. It only checks that, at this point, the compiler can vectorize this loop if needed.

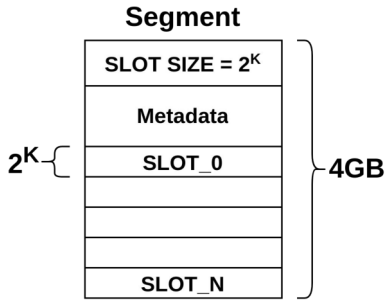


Fig. 3. Structure of a Segment.

Scout takes feedback from static optimizations and decides whether the loop was optimized or not. The versioned loop (NL) is considered to be optimized if:

(1) Any of the loop transformation optimizations has transformed NL or,
(2) L cannot be vectorized, but NL can be vectorized.

If the loop transformation optimizations do not optimize NL, then NL and the dummy checks are removed at line-14. Otherwise, actual dynamic checks are inserted to check the non-overlapping of memory accesses within the loop. Based on the feedback from static optimizations, Scout identifies the dynamic checks that need to be inserted to check the non-overlapping of memory accesses within the loop. These dynamic checks are chosen based on the following cases:

**Case 1:** If L is not vectorizable, but NL can be vectorized, insert dynamic checks for the base pairs of all possible pairs of the read-write and write-write memory accesses present in the loop body.

**Case 2:** If case-1 is not true, for every read memory access R, which is either moved outside of the loop body or removed from the loop body, insert dynamic checks for the base pairs of R and all write memory accesses present in the loop body.

**Case 3:** If case-1 is not true, for every write memory access W, which is either moved outside of the loop body or removed from the loop body, insert dynamic checks for the base pairs of W and all other (read/write) memory accesses present in the loop body.

For the identified base pointer pairs, the dynamic checks (Section 3.4) are instrumented (at line-11). At line-12, the `loop-versioning` algorithm removes *no-alias* metadata for pointer pairs for which the dynamic checks were not added in the previous step.

### 3.3 Custom Allocator

The allocator maintains a list of segments. A segment is a 4GB (configurable at compile time) contiguous virtual address space as shown in Figure 3. The starting address of a segment is a 4GB aligned address. The segment is further divided into fixed-size slots. The size of the slot is $2^K$, where K is fixed for a given segment. The starting address of the slot is aligned to $2^K$. Objects of different sizes are allocated from different segments. A slot is returned to the caller for each allocation from a segment. The first few slots of a segment are reserved for the metadata. Metadata includes a bitmap to keep track of free slots—the first eight bytes of a segment store the size of the slots of that segment. At runtime, Scout identifies an object's size by reading the first eight bytes value from the segment.

### 3.4 Dynamic Checks for Non-overlapping

The dynamic check for the non-overlapping for a pointer pair (P1, P2) is performed in two steps. In the first step, Scout computes the size of pointer P1, which involves memory access. In the second step, Scout checks the non-overlapping of P1 and P2 without memory access.

```
#define SEGMENT_SIZE (1ULL << 32)
size_t GetObjectSize(void *Ptr) {
    Segment *S = (Segment*)((size_t)Ptr & ~(SEGMENT_SIZE-1));
    if (S == DATA_SECTION_SEGMENT)
        return MAX_VIRTUAL_ADDR;
    return S->SlotSize;
}

bool IsNoAlias(void *P1, void *P2, size_t Size) {
    return xor((size_t)P1, (size_t)P2) >= Size;
}
```

Fig. 4. Dynamic check to determine if a pair of pointers do not overlap. The `GetObjectSize` routine returns the object size from a pointer address. It uses the alignment property of the segment to read the size of the slot that is stored on the first eight bytes of a segment. The `IsNoAlias` routine takes two pointer arguments and the object size of one of the arguments and returns true if the input pointers point to different objects.

*Size computation.* The size computation is done using `GetObjectSize` in Figure 4. Scout computes the address of the segment by resetting the last 32-bits of the address (note that the starting address of a segment is aligned to 4GB). If the segment address is equal to the segment corresponding to the data section, then the pointer is a global variable. In this case, `GetObjectSize` returns the maximum virtual address available on the host platform. Notice that the global variables are not allocated from the segment, and thus we cannot compute their sizes at runtime. The size computation logic for stack allocation is discussed in Section 4. The first eight bytes on the segment store the size of the slot (Section 3.3), represented using the `SlotSize` field in the pseudo-code. The size of the slot is the size of the object, which is returned to the caller.

*Check for non-overlapping.* The `IsNoAlias` routine in Figure 4 takes two pointers and the size of one of these pointers as an argument. It returns true if the pointer arguments belong to different objects. Because of the property of a segment, if two objects are of a different size, they belong to different segments. In this case, the "`xor`" of two addresses would be more than or equal to the size of the segment. Two objects can only overlap if they belong to the same segment; in that case, if the "`xor`" of two addresses is equal to or greater than the size of the slot, they cannot overlap (because of the alignment property of the slots). Notice that in the case of global variables, the object size is equal to the maximum virtual address on a 64-bit platform (as returned by the `GetObjectSize` routine). Due to this, `IsNoAlias` considers two global variables as may-aliases.

### 3.5 Profiler

The performance benefits of our approach can only be determined at the time of execution as they depend on the number of times the optimized loops are executed. One way of obtaining this information is by executing the program with a profiler. This leads us to the implementation of a profiler for Scout to maximize the benefits. This profiler identifies loops that improve the program's overall performance by executing them.

In the presence of the profiler, Scout works in two phases. In the first phase, the program is instrumented to collect the execution times of loops using the *rdtsc* instruction [rdt 2022] by

executing it on a given set of inputs for both versions of the loop. In the second phase, Scout uses the generated profile files to identify the transformed loops that improved the program's overall performance. These loops are identified based on different thresholds such as 10, 20, and 30, representing the percentage improvement in the execution time of the loop. Scout then performs versioning only for these loops.

## 4 IMPLEMENTATION

We implemented Scout as a part of the LLVM infrastructure (v10.0.0). We have added a pass to LLVM that follows the approach discussed in Section 3. We have integrated our pass with the Clang `O3` optimization level, and it runs by default with the `O3` option. Our pass works at the LLVM IR level before executing the loop vectorization pass. The pass can be disabled using the option *disable-additional-vectorize*. We extended the widely used production allocator `JEMALLOC-5.2.1` to implement our segment-based allocator.

*Allocator.* `JEMALLOC` is essentially a buddy allocator, which maintains buckets of different sizes. For each bucket size, `JEMALLOC` allocates a large contiguous memory area called extent and divides the extent into fixed-size slots. These slots are cached and served during the allocation requests. `JEMALLOC` uses per-thread caches. The extents are allocated using the `mmap` API. We leveraged the caching framework of `JEMALLOC` and modified the extent allocation logic to use the segments (discussed in Section 3.3) instead of `mmap`. For an extent that is used to serve the objects of size $2^K$, we allocate the extent from a segment that is used to allocate objects of size $2^K$. We also ensure that an extent is not recycled for a different bucket size.

Extending the size to $2^K$ may create fragmentation issues, especially for large objects. `JEMALLOC` uses a bucket-based allocation strategy for small objects; because of that, the additional fragmentation caused by our technique is reduced. We found that the bucket (or class) sizes used by the `JEMALLOC` are not always $2^K$. The bucket sizes used by `JEMALLOC` in bytes are 8, 16, 32, 48, 64, 80, 96, 112, 128, 160, and so on. For large objects, our fragmentation issue could be severe. For example, imagine a scenario where we need to allocate 2GB of memory for an allocation request for the size "1GB + 1Byte". To mitigate this issue, for object size larger than $2^{14}$, we map physical pages that are actually needed for the allocation. For example, in the case of "1GB + 1Byte", we map the physical pages corresponding to "1GB + 4096Bytes", where 4096 is the page size. The virtual address reserved for a memory allocation is always $2^K$. The average memory overhead of our scheme for CPU SPEC 2017 benchmarks is 7.47%, as discussed in Section 5.

An alias query can also be made for the stack objects at runtime. In most cases, the compiler can statically tell whether a stack object does not alias with another object. However, if a stack address escapes from the static scope, i.e., 1) stored in memory, 2) passed to a routine, or 3) cast to an integer, the compiler may not identify the stack object uniquely. For these cases, we tried replacing the stack allocations with `malloc` and `free`. However, the overhead of this approach is very high. To reduce the overheads, we used custom per-thread bump allocators for bucket sizes (in bytes) 8, 16, 32, ..., and 1024. In this setting, for objects larger than 1024, `malloc` and `free` are used. The bump allocators use segments with slot sizes equal to the bucket sizes of the bump allocators. Even with the bump allocators, the overheads are high for some benchmarks. To reduce the overheads further, we switch to a new stack during the main routine. The new stack is allocated from a segment for which the slot size is 64. If the size of an escaping stack object `X` is less or equal to 64, we set the alignment of `X` to 64. Due to this, at runtime, if the size of `X` is queried using the `GetObjectSize` (see Section 3.4) API, it will correctly return 64. If the object size is more than 64, we use per-thread bump allocators as discussed before. For multi-threading, Scout inserts a custom wrapper around calls to `pthread_create`. In the wrapper code, it allocates a 64-bit aligned stack

from the segment and uses `pthread_attr_setstack` to set the new stack for the target thread. We discuss the overheads of our allocator in detail in Section 5.

Some library functions, such as `__ctype_b_loc`, `__ctype_toupper_loc`, etc., return its internal addresses that may not belong to a valid segment. Scout adds wrappers for these routines. The wrappers allocate a new object, copy contents from the library's object to the new object, and return the allocated address to the caller. If the contents of libraries internal object never change (e.g., an internal array is used by the library for character classification functions such as `isalpha`, `isspace`, etc. that are not updated post-initialization), the new object is cached and reused during the subsequent use of the library function.

*Embedding non-aliasing information.* Scout inserts the non-aliasing information to the loop body with non-overlapping memory accesses. The compiler uses the non-aliasing information to check if two memory accesses do not alias statically. This information is incorporated using the `alias.scope` and `noalias` metadata of LLVM [Ali 2022]. It specifies that the two pointers do not alias each other. This metadata can only be attached to memory instructions like load and store.

Because the array bases corresponding to memory accesses in the loop may not necessarily be memory instructions, Scout inserts a custom intrinsic [llv 2022] to the program to specify the *no-alias* relationships between the base pairs of the pointers. Our custom LLVM intrinsic is known as `llvm.custom.noalias`. This intrinsic holds a pair of pointers as arguments. The pair of pointers present in the custom intrinsic are treated as *no-alias* with each other. These pointer pairs are wrapped in the form of `LLVM's metadata nodes`. Sample usage of the intrinsic is shown below:
`call void @llvm.custom.noalias(metadata DATA_TYPE %a, metadata DATA_TYPE %b)`
where `metadata` represents the LLVM's metadata node, `DATA_TYPE` represents the type of the pointer, %a and %b represent base pointers.
We also modified the static alias analysis algorithm to use our intrinsic. Suppose the original static alias algorithm cannot find the alias relationship between a pointer pair (P1, P2) at a given point. In that case, our modified algorithm additionally checks the presence of our custom intrinsic in the current or an outer scope with bases of `P1` and `P2` as operands. If it exists, the alias analysis algorithm treats the pointer pair as *no-alias*.

*Profiler.* The profiler implemented in Scout generates two types of profile files in the first phase. The first file is generated to obtain the time taken statistics for loops when the program is executed, disabling our approach using the options *execute-unoptimized-path* and *get-time-stats* together. The second file is generated to obtain the time taken statistics for loops when the program is executed with our approach using the option *get-time-stats*. The profiler uses the generated profile files in the second phase to identify loops that benefited the program's overall performance for different thresholds. This can be done by specifying the options *allow-ben-loops* and *ben-loop-threshold* together. The user can specify different thresholds (in percentage) using the option *ben-loop-threshold*. The default value of the threshold is set to 0%.

*Optimizing dynamic checks.* Scout implements loop-versioning algorithm (Algorithm 1) for the innermost loops, which is consistent with the LLVM policy of vectorizing only the innermost loops. Scout moves the logic to generate the condition for the dynamic check to the preheader of an outer loop if it is safe to access the base address (for obtaining the size) in the preheader of the outer loop. To identify if a base pointer is safe to access at point P, Scout statically checks if a pointer derived from the base pointer is guaranteed to be accessed if the execution reaches P.

## 5 EVALUATION

We evaluated Scout using 30 Polybench (version 4.2.1) and 16 C and C++ benchmarks available as a part of Intrate and FPrate suites CPU SPEC 2017 [spe 2022; Bucek et al. 2018] benchmarks. For Polybench, we used the default input set, and for CPU SPEC 2017, we used the reference input set.

Table 1. Memory and CPU overhead for CPU SPEC 2017 benchmarks. (MC = Memory overhead of using custom allocator w.r.t. to native JEMALLOC allocator, MB = Memory overhead of using bump allocator w.r.t. to native JEMALLOC allocator, MM = Memory overhead of replacing stack alloactions with calls to `malloc` and `free` when the stack objects go out of scope w.r.t. to native JEMALLOC allocator, CC = CPU overhead of using custom allocator w.r.t. to native JEMALLOC allocator, CB = CPU overhead of using bump allocator w.r.t. to native JEMALLOC allocator, CM = CPU overhead replacing stack alloactions with calls to `malloc` and `free` when the stack objects go out of scope w.r.t. to native JEMALLOC allocator. The values indicate percentages.)

| | **MO** | | | **CO** | | |
|---|---|---|---|---|---|---|
| **Benchmark** | **MC** | **MB** | **MM** | **CC** | **CB** | **CM** |
| namd_r | -2.29 | -2.22 | -2.09 | 0 | 0.51 | 0.51 |
| parest_r | 11.72 | 11.79 | 8.06 | 0.43 | 1.15 | 1.51 |
| lbm_r | 0.02 | 0.03 | 0.02 | -0.61 | -0.25 | -0.25 |
| imagick_r | -0.33 | -0.32 | -0.28 | 1.4 | 0.37 | 29.98 |
| nab_r | 17.58 | 17.6 | 17.56 | 1.54 | 0.73 | 0.55 |
| perlbench_r | 6.53 | 6.46 | 5.98 | 0.33 | 8.55 | 594.75 |
| gcc_r | 6.15 | 5.67 | 3.72 | -0.98 | 1.22 | 168.86 |
| mcf_r | 0.01 | 0.01 | 0.03 | -2.15 | -2.07 | -1.61 |
| omnetpp_r | 16.72 | 16.81 | 16.79 | 8.36 | 14.08 | 212.29 |
| xalancbmk_r | 19.97 | 19.99 | 19.99 | 3.05 | 3.36 | 189.64 |
| x264_r | 0.06 | 0.08 | 0.04 | -2.36 | -1.93 | 22.46 |
| deepsjeng_r | 0.08 | 0.06 | 0.08 | 1.64 | 2.41 | 77.89 |
| leela_r | 10.02 | 9.89 | 10.59 | 1.31 | 3.56 | 67.86 |
| xz_r | 0.05 | 0.06 | 0.06 | 0.61 | 4.43 | 4.36 |
| povray_r | 34.89 | 36.79 | 33.18 | 7.5 | 11.87 | 378.53 |
| blender_r | 5.2 | 5.25 | 5.21 | -2.57 | 1.78 | 55.82 |
| **GM** | **7.47** | **7.54** | **7.03** | **1.05** | **3.01** | **72.75** |

We compiled all the benchmarks with the `O3` optimization level. We performed the experiments on a 3.60GHz Intel(R) Core(TM) i9-9900K CPU 8 core machine with an x86_64 architecture and 32 GB primary memory, which uses a 64-bit Ubuntu 20.04.2 LTS operating system. We disabled hyper-threading during our experiments. We considered the arithmetic means of the execution time for five runs of each benchmark.

### 5.1 Memory and CPU Time Overhead of the Custom Allocator

Table 1 shows the memory and CPU time overheads of the CPU SPEC 2017 benchmarks. Column-2 and Column-5 show the memory and CPU overhead of the custom allocator with respect to the native JEMALLOC allocator, respectively. Column-3 and Column-6 show the memory and CPU overhead of the bump allocator with respect to the native JEMALLOC allocator. Column-4 and Column-7 show the memory and CPU overheads of replacing stack allocations with calls to `malloc` and `free` when the stack objects go out of scope with respect to the native JEMALLOC allocator.

We used the "Maximum resident set size" reported by the "/usr/bin/time -v" command for memory overheads. The memory overhead of our custom allocator is in the range -2.29% to 34.89% for CPU SPEC 2017 benchmarks (Table 1, column-2). The memory overhead of `povray_r` is high, as the peak memory consumption for this benchmark is only 8.4 MB. The major memory overhead comes from our custom stack and the page-table pages corresponding to segments. For other benchmarks, the memory overhead is always less than 20%. The geometric mean of memory overhead is 7.47%.

Table 2. Memory and CPU time overhead for Polybench benchmarks. We report the benchmarks that resulted in the absolute CPU overhead of more than 1%. (MD = Memory overhead of custom allocator w.r.t. native JEMALLOC allocator when compiled using the default size of memory allocations, MA = Memory overhead of custom allocator w.r.t. native JEMALLOC allocator when the size of the memory allocations is aligned to $2^K$, CD = CPU overhead of custom allocator w.r.t. native JEMALLOC allocator when compiled using the default size of memory allocations, CA = CPU overhead of custom allocator w.r.t. native JEMALLOC allocator when the size of the memory allocations is aligned to $2^K$. The values indicate percentages.)

| | MO | | CO | |
|---|---|---|---|---|
| **Benchmark** | **MD** | **MA** | **CD** | **CA** |
| gemm | 0.1 | 0.4 | 5.3 | 0.02 |
| symm | -0.04 | 0.33 | 2.01 | 1.22 |
| doitgen | 0.26 | 0.41 | -2.55 | 0.04 |
| jacobi-2d | 0.17 | 0.7 | -1.45 | 0.01 |

The CPU time overhead of our modified allocator is in the range of -2.57% to 8.36% for CPU SPEC 2017 benchmarks (Table 1, column-5). The percentage change in the execution time of the custom allocator w.r.t. the native allocator represents the CPU time overhead. In our design, the objects (an extent in the case of JEMALLOC) from buckets of large sizes cannot be recycled for buckets of smaller sizes. But this is not true for the unmodified allocator. Because of this, the behavior of the per-thread caches is different in both runs. Using better allocation tuning can minimize this problem. We plan to investigate this in the future. Nevertheless, only three out of the 16 benchmarks incur CPU time overheads above 3%. With the usage of custom per-thread bump allocators (discussed in Section 4), the CPU overhead for four benchmarks (LoC > 300K), namely, `perlbench_r`, `gcc_r`, `xalancbmk_r` and `parest_r`, is either negative or better than our previous implementation. The geometric mean of CPU time overhead is 1.05%. The CPU overhead lies in the range of -2.07% to 14.08%, when we use bump allocator instead of custom stack (Table 1, column-6), as discussed in Section 4. However, the CPU overhead of replacing stack allocations with calls to `malloc` and `free` is considerably high for these benchmarks (Table 1, column-7). To reduce this overhead, we use the bump allocator and the custom stack as discussed in Section 4.

Table 2 presents the memory and CPU time overheads for the Polybench benchmarks for which the absolute CPU overhead is more than 1%. The memory overhead of our custom allocator lies in the range of -0.41% to 0.72% with a geometric mean of 0.13%. The CPU overhead of our custom allocator lies in the range of -2.55% to 5.3%, with a geometric mean of 0.21%.

In the case of `gemm`, `doitgen` and `jacobi-2d`, our custom allocator either performs better or worsens the performs as compared to the native allocator. This is due to the fact that our custom allocator aligns the memory allocations to $2^K$. To verify this, we performed another experiment, in which we align the sizes of the memory allocations to $2^K$ in the source code itself. We observed negligible overheads in that case for these three benchmarks. However, we observed a CPU time overhead of 1.22% (Table 2, column-5) for `symm` even when the sizes are aligned to $2^K$. We found that this benchmark is doing three large allocations. We believe the overhead is mainly due to the different allocation strategies used for the large objects. The memory overhead for this benchmark is -0.04% (Table 2, column-2). If we use $2^K$ aligned sizes, the memory overhead is 0.33% (Table 2, column-3).

### 5.2 Performance Benefits without Profiler

We first discuss the performance benefits obtained by applying Scout on the programs from Polybench and CPU SPEC 2017 benchmarks in the absence of the profiler. To obtain the performance

Table 3. Performance benefits for Polybench without profiler. We report six benchmarks that resulted in performance benefits of more than 3%. ($\#L_a$ = Total number of versioned loops by Scout, $P_a$ = Performance benefits when compiled with Scout, $P_r$ = Performance benefits when compiled with the `restrict` keyword, $V_c$ = Variance of the execution time when compiled with the custom allocator, $V_s$ = Variance of the execution time when compiled with the custom allocator and Scout. Performance-benefit numbers represent percentages.)

| **Benchmark** | $\#L_a$ | $P_a$ | $P_r$ | $V_c$ | $V_s$ |
|---|---|---|---|---|---|
| gesummv | 1 | 49.37 | 48.42 | 0.00000001 | 0.00000001 |
| 2mm | 2 | 4.22 | 4.1 | 0.00025563 | 0.00015497 |
| 3mm | 3 | 4.28 | 4.03 | 0.00017649 | 0.00009542 |
| bicg | 1 | 51.11 | 51.01 | 0.00000001 | 0.00000001 |
| doitgen | 2 | 10.2 | 11.11 | 0.00000132 | 0.00000334 |
| jacobi-1d | 2 | 11.8 | -1.11 | 0.00000001 | 0.00000001 |

benefits, we computed the percentage change in the execution time of the benchmark when compiled with (optimized) and without (native) our pass, using the custom memory allocator in both cases.

*Polybench benchmarks.* For Polybench, we report the performance benefits for two different methods of benchmark compilation. Firstly, we report the performance benefits obtained when compiled with and without our pass along with the custom allocator. Table 3 shows the number of versioned loops by Scout (column-2 $\#L_a$), and the performance benefits when compiled with and without Scout for the six benchmarks showing performance benefits of more than 3% (column-3 $P_a$). Secondly, we report the performance benefits when compiled with and without the `DPOLYBENCH_USE_RESTRICT` option of Polybench. This option inserts the `restrict` keyword to allow the compiler to assume absence of aliasing for function arguments. Column-4 ($P_r$) of Table 3 shows the performance benefits for the same six benchmarks using the `restrict` keyword. Additionally, this table shows the variance of the execution times for the benchmarks' five runs. Column-5 shows the variance of the execution times for five runs when the benchmarks are compiled with the custom allocator. Column-6 shows the variance of the execution times for five runs when the benchmarks are compiled with the custom allocator and Scout. The variance is always less than 0.001.

For Polybench, Scout shows similar performance benefits as that of using `restrict` keyword except for two benchmarks, out of 30. We observed that in the case of Scout, non-aliasing relationships in the benefited versioned loops involve the function arguments. Therefore, the triggered optimizations in both cases were identical, resulting in similar performance benefits.

We observed a substantial performance degradation when `jacobi-1d` (-1.11%) and `adi` (-5.16%) benchmarks were compiled with the `restrict` keyword. In this case, LLVM failed to preserve non-aliasing information throughout the transformation passes [res 2022]. The loss of non-aliasing information resulted in introducing some extra instructions by loop strength reduction and SLP optimizations, slowing down the performance. Therefore, usage of the `restrict` keyword degraded the performance of these benchmarks. However, Scout preserved non-aliasing information throughout the transformation passes with the help of custom intrinsic and *no-alias* metadata, resulting in performance benefits of 11.8% and 0.42% for `jacobi-1d` and `adi` benchmarks, respectively. Scout enabled more optimization opportunities (i.e., LICM and SLP) for these benchmarks.

Out of 30 benchmarks, six benchmarks show performance benefits of more than 3% when compiled with Scout, as shown in Table 3 (column-3). Out of these six benchmarks, four benchmarks have high performance benefits of more than 10%. We have listed code snippets from these four benchmarks in Figure 5, referred to in the discussion below.

```
static void kernel_bicg(int m, int n,
  double A[], double B[], double q[],
  double p[], double r[]) {
  ...
  for (i = 0; i < _PB_N; i++) {
    q[i] = SCALAR_VAL(0.0);
    for (j = 0; j < _PB_M; j++) {
      s[j] = s[j] + r[i] * A[i][j];
      q[i] = q[i] + A[i][j] * p[j];
    }
  }
}
```

```
static void kernel_gesummv(int n,
  double alpha, double beta,
  double A[], double B[],
  double tmp[], double x[],
  double y[]) {
  ...
  tmp[i] = SCALAR_VAL(0.0);
  y[i] = SCALAR_VAL(0.0);
  for (j = 0; j < _PB_N; j++) {
    tmp[i] = A[i][j] * x[j] + tmp[i];
    y[i] = B[i][j] * x[j] + y[i];
  }
  y[i] = alpha * tmp[i] + beta * y[i];
}
```

```
static void kernel_jacobi_1d(int tsteps,
  int n, double A[], double B[]) {
  ...
  for (i = 1; i < _PB_N - 1; i++)
   B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
  for (i = 1; i < _PB_N - 1; i++)
   A[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1]);
}
```

```
static void kernel_doitgen(int nr,
  int nq, int np, double A[][][],
  double C4[][], double sum[]) {
  ...
  sum[p] = SCALAR_VAL(0.0);
  for (s = 0; s < _PB_NP; s++)
    sum[p] += A[r][q][s] * C4[s][p];
  ...
}
```

Fig. 5. Code snippets from the Polybench benchmarks showing performance benefits of more than 10%.

(1) In `bicg` benchmark, for the loop in function `kernel_bicg` the compiler could move the read-/write access to `q[i]`, and `r[i]` outside the inner loop body.
(2) In `gesummv` benchmark, for the loop in function `kernel_gesummv` the compiler could move the read/write access to `tmp[i]`, and `y[i]` outside the loop body.
(3) In `jacobi-1d` benchmark, for the loop in function `kernel_jacobi_1d`, the compiler could move `A[i-1]` and `A[i]` from the first loop outside the loop body and `B[i-1]` and `B[i]` from the second loop outside the loop body. This optimization further allowed the compiler to vectorize both loops more efficiently by reducing the number of read memory accesses from 6 to 2 for a vector width of 4 for each loop.
(4) In `doitgen` benchmark, for the loop in function `kernel_doitgen`, the compiler was able to move `sum[p]` outside the loop body.

Out of 30 benchmarks, five benchmarks named, `floyd-warshall`, `nussinov`, `seidal-2d`, `trisolv` and `trmm` did not show any improvement as none of the loops were versioned. These benchmarks consisted of read/write memory accesses with the same base pointers for different array elements. Alves et al. [Alves et al. 2015] did not report results for the `floyd-warshall` benchmark. However, for the other four benchmarks, Alves et al. [Alves et al. 2015] did not report any substantial improvements.

For the remaining 19 benchmarks, the performance benefits varied from 0% to 3%. The transformations applied by Scout allowed the compiler to trigger optimizations like LICM and GVN. In some cases, these transformations allowed the compiler to generate a better vectorized code. The efficient vectorization improved the overall performance of the benchmarks. Based on the

Table 4. Speedups for Polybench. We report speedups for three benchmarks that resulted in substantial speedups for the hybrid approach [Alves et al. 2015]. ($S_h$ = Speedup with the hybrid approach [Alves et al. 2015], $S_{ra}$ = Speedup when compiled with `restrict` keyword using LLVM-3.6.0, $S_s$ = Speedup with Scout, $S_{rb}$ = Speedup when compiled with `restrict` keyword using LLVM-10. )

| **Benchmark** | $S_h$ | $S_{ra}$ | $S_s$ | $S_{rb}$ |
|---|---|---|---|---|
| gesummv | 2.5 | 1.92 | 1.98 | 1.94 |
| bicg | 2.7 | 2.06 | 2.05 | 2.05 |
| gramschmidt | 1.4 | 1.01 | 1.01 | 1.01 |

results obtained, we can conclude that for Polybench, the performance benefits obtained using the `restrict` keyword and Scout are similar in most cases.

Table 4 shows the results for the three benchmarks that led to a substantial speedup using the hybrid approach [Alves et al. 2015]. In this table, $S_h$ (column-2) represents the speedups reported for the hybrid approach, $S_{ra}$ (column-3) represents the speedup when we compiled the benchmarks with LLVM-3.6.0 (used in the experiments performed by Alves et al. [Alves et al. 2015]) using the `restrict` keyword, $S_s$ (column-4) represents the speedups obtained with Scout and $S_{rb}$ (column-5) represents the speedups when we compiled the benchmarks with LLVM-10 (used in our experiments) using the `restrict` keyword.

The hybrid (polyhedral and symbolic range analysis) approach [Alves et al. 2015] resulted in a substantial speedup for the three benchmarks as shown in Table 4 (column-2). The speedups using Scout ($S_s$) were lesser than the speedups with the hybrid approach ($S_h$). To investigate it further, we compiled these benchmarks with the LLVM-3.6.0 version (used by Alves et al. [Alves et al. 2015]) using the `restrict` keyword. The resulted speedups ($S_{ra}$) were similar to those of Scout and with the `restrict` keyword ($S_{rb}$) using LLVM-10 (used by Scout). For all these benchmarks, Scout could resolve memory dependencies in loops using dynamic checks. In all cases, Scout added runtime checks for the pointer arguments, and thus, we observed a similar set of optimizations using the `restrict` keyword. We expect similar behavior in Alves et al. [Alves et al. 2015] hybrid approach. Therefore, we attribute the better speedups reported in Alves et al. [Alves et al. 2015] to the different versions of processors used in both experiments.

We recommend using the hybrid approach [Alves et al. 2015] for Polybench benchmarks. For these benchmarks, Scout did not find any loop that cannot be versioned using the hybrid approach. The hybrid analysis can disambiguate pointers that involve read/write memory accesses with the same base pointers. Scout and the purely dynamic approach [Alves et al. 2015] fail to disambiguate such memory accesses. Therefore, these techniques fail to benefit the performance in such cases.

Real-world benchmarks such as CPU SPEC 2017 have many loops for which the hybrid approach cannot be applied. Due to the high cost of the purely dynamic approach proposed by Alves et al. [Alves et al. 2015], these loops cannot be further optimized. Our fast dynamic checks enabled some interesting optimizations in CPU SPEC 2017 that we will discuss next.

*CPU SPEC 2017 benchmarks.* We now discuss the results for CPU SPEC 2017 benchmarks without using feedback from the profiler. Table 5 shows loop-related statistics and performance benefits obtained for CPU SPEC 2017 benchmarks. In this table, $\#L_a$ represents the total number of versioned loops by Scout (column-2), $\#L_b$ represents the total number of versioned loops that were not vectorizable before but can be vectorized after the transformations applied by Scout (column-3), $\#L_e$ represents the number of versioned loops executed out of the total versioned loops by Scout (column-4), $P_a$ represents the performance benefits obtained without using the profiler (column-5), $V_c$ represents the variance of the execution time for the five runs when the benchmarks

Table 5. Performance benefits for CPU SPEC 2017 without profiler. ($\#L_a$ = Total number of versioned loops by Scout, $\#L_b$ = Total number of versioned loops that are vectorizable, $\#L_e$ = Total number of versioned loops executed, $P_a$ = Performance benefits when compiled with Scout, $V_c$ = Variance of the execution time when compiled with the custom allocator, $V_s$ = Variance of the execution time when compiled with the custom allocator and Scout. Performance-benefit numbers represent percentages.)

| **Benchmark** | $\#L_a$ | $\#L_b$ | $\#L_e$ | $P_a$ | $V_c$ | $V_s$ |
|---|---|---|---|---|---|---|
| namd_r | 326 | 249 | 9 | -0.38 | 0.7 | 0.3 |
| parest_r | 1020 | 568 | 69 | -0.3 | 0.3 | 0.5 |
| lbm_r | 0 | 0 | 0 | 0 | 0 | 0 |
| imagick_r | 91 | 24 | 4 | 0 | 0.8 | 0.3 |
| nab_r | 35 | 15 | 21 | -0.09 | 0.2 | 0 |
| perlbench_r | 50 | 18 | 12 | -0.4 | 1.2 | 0.7 |
| gcc_r | 175 | 48 | 35 | 0.13 | 1 | 1.7 |
| mcf_r | 0 | 0 | 0 | 0 | 0 | 0 |
| omnetpp_r | 39 | 1 | 6 | -1.79 | 0.2 | 0.7 |
| xalancbmk_r | 286 | 268 | 34 | 0.88 | 1.3 | 0.7 |
| x264_r | 124 | 25 | 32 | 0 | 0 | 0 |
| deepsjeng_r | 6 | 0 | 6 | -0.1 | 0 | 0.2 |
| leela_r | 1 | 0 | 0 | 0 | 0 | 0 |
| xz_r | 2 | 1 | 0 | 0 | 0 | 0 |
| povray_r | 25 | 7 | 2 | -2.11 | 0.8 | 1.6 |
| blender_r | 409 | 89 | 31 | -3.37 | 0 | 0.8 |

are compiled with the custom allocator (column-6), and $V_s$ represents the variance of the execution time for the five runs when the benchmarks are compiled with the custom allocator and Scout (column-7).

To obtain the performance benefits, we compiled all the benchmarks with and without our pass using the custom allocator. We then computed the percentage change in the execution time of the benchmarks. Out of 16 benchmarks, six benchmarks named `lbm_r`, `imagick_r`, `mcf_r`, `x264_r`, `leela_r` and `xz_r` reported no improvement in the execution time. For benchmarks `lbm_r` and `mcf_r`, none of the loops were versioned by Scout. For benchmarks `leela_r` and `xz_r`, some loops were versioned, but those loops were never executed.

The percentage change in the CPU time for the rest of the benchmarks lie in the range of -3.37% (`blender_r`) to 0.88% (`xalancbmk_r`), as shown in Table 5, column-5. Except for the `xalancbmk_r` and `gcc_r` benchmarks, the other benchmarks incurred negative or no improvement. Scout versioned a significant number of loops for these benchmarks; however, the following reasons led to such performance degradation:

(1) The code snippet containing the versioned loop never executes (e.g., in the case of `blender_r` only 7% of the versioned loops were executed).
(2) The dynamic checks never hold. In such a case, the original version of the loop executes, and the dynamic checks ends up adding an extra overhead (e.g., we found four such loops in case of `gcc_r` and five loops in case of `x264_r`).
(3) The overhead of dynamic checks is more than the benefits of the enabled optimizations.

Our observations showed that even though most of the benchmarks did not show a substantial performance improvement, these benchmarks showed the potential to vectorize more loops over the existing implementation. The range of additional vectorized loops lies between 2% to 93% w.r.t. the

Table 6. Performance benefits for CPU SPEC 2017 with profiler. ($\#L_p$ = Total number of versioned loops based on the profiler's feedback, $\#L_b$ = Total number of versioned loops that are vectorizable, $P_a$ = Performance benefits when compiled with Scout, $\tau$ = Threshold for improvement in the execution time of the loop, $S_{rec}$ = Scout recommended (Y = Yes, N = No). Performance-benefit numbers represent percentages.)

| | $\tau$ = 0 | | | $\tau$ = 10 | | | $\tau$ = 20 | | | $\tau$ = 30 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | $\#L_p$ | $\#L_b$ | $P_a$ | $\#L_p$ | $\#L_b$ | $P_a$ | $\#L_p$ | $\#L_b$ | $P_a$ | $\#L_p$ | $\#L_b$ | $P_a$ | $S_{rec}$ |
| namd_r | 9 | 1 | 0.51 | 5 | 1 | 0.13 | 0 | 0 | 0 | 0 | 0 | 0 | Y |
| parest_r | 24 | 17 | 0.31 | 18 | 13 | 0.73 | 15 | 12 | 0.43 | 12 | 10 | 0.31 | N |
| lbm_r | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N |
| imagick_r | 1 | 0 | 0.23 | 1 | 0 | 0.23 | 1 | 0 | 0.23 | 1 | 0 | 0.23 | N |
| nab_r | 12 | 9 | 0.27 | 11 | 9 | 0 | 11 | 9 | 0 | 8 | 7 | 0 | N |
| perlbench_r | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N |
| gcc_r | 5 | 4 | 0.62 | 2 | 2 | 0.73 | 2 | 2 | 0.73 | 0 | 0 | 0 | Y |
| mcf_r | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N |
| omnetpp_r | 3 | 0 | 0.5 | 1 | 0 | 0.25 | 0 | 0 | 0 | 0 | 0 | 0 | N |
| xalancbmk_r | 10 | 9 | 1.32 | 4 | 4 | 1.47 | 2 | 2 | 0.9 | 2 | 2 | 0.9 | N |
| x264_r | 19 | 1 | 0.89 | 12 | 1 | 0.45 | 9 | 1 | 0.34 | 3 | 1 | 0.56 | Y |
| deepsjeng_r | 2 | 0 | 0.49 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N |
| leela_r | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N |
| xz_r | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N |
| povray_r | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | N |
| blender_r | 11 | 0 | 0.52 | 3 | 0 | 0.31 | 0 | 0 | 0 | 0 | 0 | 0 | Y |

total versioned loop by Scout. Out of 16 benchmarks, for 11 benchmarks, this percentage is more than 10%. For `xalancbmk_r`, `namd_r` and `parset_r` benchmarks 93%, 76% and 55% of the additional versioned loops are vectorizable. These statistics showed that the compiler could vectorize more loops, but the existing approaches failed on such loops.

The purely dynamic approach in Alves et al. [Alves et al. 2015] slowed down the allocation-heavy `401.bzip2` benchmark from SPEC CPU 2006 by 29%. On the other hand, the maximum slow down with Scout is 10.15% for the `omnetpp_r` benchmark (including allocator overhead) and 3.37% for the `blender_r` benchmark (excluding allocator overhead), as shown in Tables 1 (column-5) and 5 (column-5). Moreover, it would be unfair to compare these results directly because unlike the purely dynamic approach, Scout takes feedback from the static optimizations to reduce the number of dynamic checks and decide which loops can be benefited from versioning. Alves et al. [Alves et al. 2015] did not report results for other SPEC CPU 2006 benchmarks (apart from `401.bzip2`).

### 5.3 Performance Benefits with Profiler

We estimated performance benefits of the CPU SPEC 2017 benchmarks based on the profiler's feedback. To obtain the performance benefits with the profiler, we computed the percentage change in the execution time of the benchmark when compiled with only those versioned loops that showed some improvement in the execution time w.r.t. the native compilation. Firstly, the benchmarks were compiled and executed with the optimized (i.e., versioned loops) and then with the unoptimized versions of the loops to obtain the profile files. Scout identified loops that showed some improvement using the generated profile files. The different thresholds for performance improvement determined the benefited loops.

Table 7. Variance of the execution time for five runs of the CPU SPEC 2017 benchmarks. $\tau$ = Threshold for improvement in the execution time of the loop, $V_c$ = Variance of the execution time when compiled with the custom allocator, $V_s$ = Variance of the execution time when compiled with the custom allocator and Scout.)

| | | $\tau$ = 0 | $\tau$ = 10 | $\tau$ = 20 | $\tau$ = 30 |
|---|---|---|---|---|---|
| **Benchmark** | $V_c$ | $V_s$ | $V_s$ | $V_s$ | $V_s$ |
| namd_r | 0.7 | 0 | 0.8 | 0.7 | 0.7 |
| parest_r | 0.3 | 0.5 | 0.8 | 0.8 | 0.3 |
| lbm_r | 0 | 0 | 0 | 0 | 0 |
| imagick_r | 0.8 | 0.7 | 0.7 | 0.7 | 0.7 |
| nab_r | 0.2 | 0.2 | 0.7 | 0.7 | 0.7 |
| perlbench_r | 1.2 | 1.2 | 1.2 | 1.2 | 1.2 |
| gcc_r | 1 | 1 | 0.2 | 0.2 | 1 |
| mcf_r | 0 | 0 | 0 | 0 | 0 |
| omnetpp_r | 0.2 | 0.7 | 0.2 | 0.2 | 0.2 |
| xalancbmk_r | 1.3 | 1.3 | 0.3 | 0.7 | 0.7 |
| x264_r | 0 | 0.3 | 0.2 | 0.3 | 0 |
| deepsjeng_r | 0 | 0 | 0 | 0 | 0 |
| leela_r | 0 | 0 | 0 | 0 | 0 |
| xz_r | 0 | 0 | 0 | 0 | 0 |
| povray_r | 0.8 | 0.8 | 0.8 | 0.8 | 0.8 |
| blender_r | 0 | 0 | 0.3 | 0 | 0 |

Table 6 shows the loop-related statistics and performance benefits for thresholds ($\tau$) 0%, 10%, 20% and 30%. In this table, $\#L_p$ represents the total number of versioned loops based on feedback from the profiler, $\#L_b$ represents the number of versioned loops not vectorizable before but vectorized after the transformations applied by Scout, $P_a$ represents the performance benefits obtained using feedback from the profiler and $S_{rec}$ represents whether we recommend using Scout for the corresponding benchmark (Y represents we recommend using Scout and N represents we do not recommend using Scout). We recommend using Scout for those benchmarks where the CPU time overhead of the custom allocator (Table 1, column-5) is lesser than the performance benefits obtained with the profiler for at least one of the thresholds. Table 7 shows the variance of the execution time for five runs of the benchmarks for different thresholds. $V_c$ represents the variance when compiled with the custom allocator, $V_s$ represents the variance when compiled with the custom allocator, and Scout.

For the benchmark `xalancbmk_r`, out of 286 versioned loops by Scout, only ten loops resulted in some performance improvement based on the statistics shown in Table 6 for threshold 0%. After discarding non-benefited loops, the performance benefits increased to 1.32% from 0.88% for threshold 0%. The performance further increased to 1.47% when the four loops with an improvement of more than 10% were versioned. However, the performance drops to 0.9% for the 20%, and 30% threshold as Scout only versioned two loops based on the feedback. Out of the ten loops that benefited, the compiler vectorized nine loops based on the transformations applied by Scout. This number was further reduced to four for threshold 10% and two for thresholds 20% and 30%, affecting the benchmark's performance. Even though the `xalancbmk_r` benchmark shows the maximum performance benefits, we do not recommend using Scout for this benchmark as the allocator's overhead for this benchmark is higher (3.05%, Table 1, column-5).

For benchmark `x264_r`, out of 124 versioned loops by Scout, only 19 loops benefited based on the statistics obtained by the profiler. This benchmark led to a performance improvement of 0.89% when non-benefited loops were not versioned. For this benchmark, the performance benefits vary with the threshold increase. Moreover, the versioning of non-benefited loops due to the absence of the profiler hampered the performance of this benchmark. The obtained results showed that this benchmark could lead to a performance improvement of around 0.89% using the feedback. For this benchmark, the allocator is also showing an improvement of 2.36% (Table 1, column-5). Therefore, we recommend using Scout for this benchmark.

The performance benefits for the `gcc_r` benchmark increased from 0.13% to 0.62% when the benchmark was compiled with versioning only those loops that showed some improvement. There were only five such loops out of 175 loops versioned by Scout. The performance benefits increased to 0.73% for thresholds 10% and 20%. For the threshold 30%, none of the loops got benefited. We recommend using Scout for this benchmark as the maximum performance benefit is 0.73%, and the allocator shows an improvement of 0.98% (Table 1, column-5).

For some benchmarks, the performance benefits decreased with the increase in threshold, such as `namd_r`, `omnetpp_r` and `deepsjeng_r` as the number of versioned loops reduces to 0. However, for benchmarks such as `parest_r`, `gcc_r` and `xalancbmk_r`, the performance improved for threshold 10% compared to threshold 0%. The possible reason behind this could be that the instrumentation of *rdtsc* instruction for collecting timing statistics during the profile phase changed the behavior of some loops. The threshold 0% setup also includes loops that show minor improvements over the un-optimized version during the profile phase. However, the benchmarks showed higher performance benefits when such loops were not versioned for the threshold 10%.

We observed the benchmarks were benefited from the profiler because it helped filter out some non-benefited loops. The user can choose the threshold according to their requirements to obtain maximum performance benefits using Scout. We recommend using Scout for four of the CPU SPEC 2017 benchmarks named, `namd_r`, `gcc_r`, `x264_r`, and `blender_r`. We conclude that with the help of the profiler, Scout can identify the loops more effectively that were ultimately benefited and version only those loops to get maximum performance benefits.

### 5.4 Code Patterns Optimized using Scout for CPU SPEC 2017

Figure 6 shows some of code snippets from CPU SPEC 2017 benchmarks for which loop bounds are not loop invariants. These loops were executed during runtime and yielded more than 20% improvement. However, Scout could optimize many such loops during compile time. In most cases, we found that structure fields or class elements are accessed in the loop condition, involving memory access. This looks like a common coding pattern that exists in real-world workloads. The compiler could not compute the static bounds of these loops due to unresolved memory dependencies. Note that none of the existing techniques can be applied to this loop except the purely dynamic approach proposed in Alves et al. [Alves et al. 2015] that would require $O(log\ n)$ checks.

## 6 RELATED WORK

Many static alias analyses have been proposed in the past [Andersen and Lee 2005; Cooper and Kennedy 1989; Hardekopf and Lin 2009, 2011; Hind et al. 1999; Lattner et al. 2007; Pearce et al. 2007; Steensgaard 1996]. These analyses differ in properties such as field-sensitivity, inter/intra-procedural nature, flow-sensitivity, and context-sensitivity. Inter-procedural static alias analysis is more precise than intra-procedural analysis. However, inter-procedural analyses are not scalable. Some attempts have been made to develop precise intra-procedural analysis [Hardekopf and Lin 2007; Zheng and Rugina 2008]. However, the intra-procedural alias analysis fails to provide deterministic information missing out on optimization opportunities.

```
//544.nab_r mme34()
//line 1978
for (i = -1; i < prm->Natom;
  i++) {
  iexw[eoff + i] = -1;
}
//Improvement: 60%
```

```
//544.nab_r nbond()
//line 866
for (i = -1; i < prm->Natom;
  i++) {
  iexw[i] = -1;
}
//Improvement: 43%
```

```
//544.nab_r set_belly_mask()
//line 1980
for( i=0; i<prm->Natom; i++ )
  prm->N14pairs[i] = 0;
//Improvement: 38%
```

```
//544.nab_r readparm()
//line 1494
for (i = 0; i < prm->Natom;
  i++)
  prm->N14pairs[i] = 0;
//Improvement: 37%
```

```
//544.nab_r mme_init()
//line 1292
for (i = 0; i < prm->Natom;
  i++) {
  pairlist[i] = NULL;
  lpairs[i] = upairs[i] = 0;
}
//Improvement: 33%
```

```
//544.nab_r mme_init()
//line 1224
for (i = 0; i < prm->Natom;
i++) {
  pairlist2np[i] = NULL;
  lpairs2np[i]
    = upairs2np[i] = 0;
}
//Improvement: 32%
```

```
//523.xalancbmk_r expand()
//line 620
//fElemCount is a
//class member.
for (unsigned int index = 0;
  index < fElemCount;
  index++)
  newList[index]
  = fRanges[index];
//Improvement: 51%
```

```
//525.x264_r
//FmoGenerateMb-
//ToSliceGroupMap()
//line 149
for (i=0; i<p_Vid->
  PicSizeInMbs; i++) {
  MbToSliceGroupMap++ =
    *MapUnitTo-
    SliceGroupMap++;
}
//Improvement: 52%
```

```
//510.parest_r
//get_dof_indices()
//line 2043
for (unsigned int i=0; i<
  accessor.get_fe()
  .dofs_per_cell;
  ++i, ++cache)
  dof_indices[i] = *cache;
//Improvement: 77%
```

```
//502.gcc_r
//df_worklist_dataflow()
//line 1023
for (i = 0; i < cfun->cfg->
  x_last_basic_block; i++)
  bbindex_to_postorder[i] =
  cfun->cfg
  ->x_last_basic_block;
//Improvement: 29%
```

```
//502.gcc_r init_graph()
//line 1116
for (j = 0; j < graph->size;
  j++) {
  graph->rep[j] = j;
  graph->pe_rep[j] = -1;
  graph->indirect_cycles[j]
    = -1;
}
//Improvement: 22%
```

```
//538.imagick_r
//ParseGeometry()
//line 967
while (isspace((int)
  ((unsigned char) *p)) != 0)
  p++;
//Improvement: 51%
```

Fig. 6. Code snippets from the CPU SPEC 2017 benchmarks that show substantialimprovement.

Our work belongs to another class of approaches that disambiguate pointers at runtime to improve the precision of alias analysis. Alves et al. [Alves et al. 2015] propose a technique based on code-versioning and dynamic disambiguation of pointers. To disambiguate pointer pairs with different bases at runtime, it uses red-black tree lookups to find the starting addresses of objects referred to by the pointers. If the starting addresses are different, they point to different objects

and thus cannot be aliases. The overhead of this scheme is $O(log\ n)$, which is substantially high. To reduce the overheadof dynamic checks, they also use polyhedral access range analysis and symbolic range analysis to place $O(1)$ checks for loops obey certain constraints. On the contrary, our tool can check non-aliasing of pointer pairs with different bases in just one memory access irrespective of the program point.

Other approaches [Chen et al. 2004; Da Silva and Steffan 2006; Fernández and Espasa 2002; Huang et al. 1994; Lin et al. 2003] to dynamic alias analysis are based on speculative execution. The key idea of these works is to generate efficient code assuming the non-aliasing of pointers. The generated code also contains dynamic checks for the assumptions that are made for the optimizations. If the assumptions are not true during runtime, the execution is transferred to a recovery code that ensures correct behavior. These works were designed when the memory latency was high. Due to this, in these works, the overheads of dynamic checks are significantly low compared to memory access, and checks are present even inside loops. However, these assumptions might not hold for modern processors with large caches and multiple layers of cache hierarchies. These schemes work well when the frequency of execution of recovery code is low, and the overheads of dynamic checks are not high. Thus, most of these works use a profiler to identify code regions in which the assumptions for non-aliasing mostly hold. We also use a profiler to remove dynamic checks for loops that do not show improvement at runtime.

In intra-procedural alias analysis, local static information is not available to the target function. To eliminate the precision loss due to function calls, Hugo et al. [Sperle Campos et al. 2016] implement code-versioning at the granularity of functions. The cloned function is optimized, assuming non-overlapping of function arguments. The optimized version is called if the function arguments are statically known to not alias at a call site. Otherwise, dynamic checks for disambiguation of function arguments are added, and depending on the result, the optimized or original version of the target function is called. This technique can be integrated into our tool to enable other optimization opportunities.

Archipelago[Lvin et al. 2008] allocates each object on a unique page. In the presence of this allocator, it is possible to implement fast dynamic checks for pointer disambiguation. However, the memory overhead of this allocator could be high because of the fragmentation issue for small objects.

## 7 CONCLUSION

We presented a novel allocator design that enabled runtime disambiguation of two pointers using a single memory access. Despite the loop-versioning algorithm based on scalar evolution analysis, LLVM cannot vectorize many loops that can be vectorized using our technique. In addition to loop vectorization (the potential benefits are generally high), our lightweight dynamic checks enable relatively simpler optimizations, e.g., loop-invariant code motion, dead store elimination, and load elimination. Unlike vectorization, LLVM does not attempt these optimizations using the existing loop-versioning mechanisms. Our technique uses the feedback from static optimizations to identify benefited loops and the minimum number of dynamic checks required to optimize the loop. Our allocator does not add significant CPU time overheads for many benchmarks. The memory overhead of our allocator is also reasonable.

## 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 [Sco 2022a,b] contains Scout (LLVM), custom allocator (jemalloc2k), and the build scripts and is available at URLs https://doi.org/10.5281/zenodo.7089827 and https://github.com/khushboochitre/Scout-Artifact.git.

## REFERENCES


2016 (accessed Apr 14, 2022). Intel 64 and ia-32 architectures software developer's manual volume 2b. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf

2019 (accessed Apr 13, 2022). Restrict Keyword in LLVM. https://lists.llvm.org/pipermail/llvm-dev/2019-March/131127.html

2021 (accessed Apr 13, 2022). CPU SPEC 2017 benchmark suite. https://www.spec.org/cpu2017/Docs/overview.html#benchmarks

2022 (accessed Apr 13, 2022). Intrinsic Functions. https://llvm.org/docs/LangRef.html#intrinsic-functions

2022 (accessed Apr 13, 2022). Runtime Checks of Pointers. https://llvm.org/docs/Vectorizers.html#runtime-checks-of-pointers

2022 (accessed Apr 13, 2022). 'noalias' and 'alias.scope' Metadata. https://llvm.org/docs/LangRef.html#noalias-and-alias-scope-metadata

2022 (accessed Sep 9, 2022)a. Scout Artifact DOI. https://doi.org/10.5281/zenodo.7089827

2022 (accessed Sep 9, 2022)b. Scout Artifact Github Repository. https://github.com/khushboochitre/Scout-Artifact.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.

Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. 2003. Points-to analysis using BDDs. *ACM SIGPLAN Notices* 38, 5 (2003), 103–114.

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

James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42. https://doi.org/10.1145/3185768.3185771

Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. 2004. Data dependence profiling for speculative optimizations. In *International Conference on Compiler Construction*. Springer, 57–72. https://doi.org/10.1007/978-3-540-24723-4_5

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. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. Association for Computing Machinery, New York, NY, USA, 273–286. https://doi.org/10.1145/258915.258940

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

Jeff Da Silva and J. Gregory Steffan. 2006. A Probabilistic Pointer Analysis for Speculative Optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 416–425. https://doi.org/10.1145/1168857.1168908

Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-Based Alias Analysis. 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, 106–117. https://doi.org/10.1145/277650.277670

Johannes Doerfert, Tobias Grosser, and Sebastian Hack. 2017. Optimistic loop optimization. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 292–304.

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

Manel Fernández and Roger Espasa. 2002. Speculative Alias Analysis for Executable Code. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*. IEEE Computer Society, USA, 222–231.

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 *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 289–298. https://doi.org/10.1109/CGO.2011.5764696

Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. 1999. Interprocedural Pointer Alias Analysis. *ACM Trans. Program. Lang. Syst.* 21, 4 (July 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

S. Jaiswal, Uday P. Khedker, and Supratik Chakraborty. 2018. Demand-driven Alias Analysis : Formalizing Bidirectional Analyses for Soundness and Precision. *ArXiv* abs/1802.00932 (2018).

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.

W LANDI. 1992. Undecidahility of static analysis. 1992. *Lett. Program. Lang. Syst* 1, 4 (1992). https://doi.org/10.1145/161494.161501

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

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

Vitaliy B Lvin, Gene Novark, Emery D Berger, and Benjamin G Zorn. 2008. Archipelago: trading address space for reliability and security. *ACM SIGARCH Computer Architecture News* 36, 1 (2008), 115–124.

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

Ganesan Ramalingam. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471. https://doi.org/10.1145/186025.186041

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

Diogo N. Sampaio, Louis-Noël Pouchet, and Fabrice Rastello. 2017. Simplification and Runtime Resolution of Data Dependence Constraints for Loop Transformations. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) *(ICS '17)*. Association for Computing Machinery, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3079079.3079098

Marc Shapiro and Susan Horwitz. 1997. The Effects of the Precision of Pointer Analysis. In *Proceedings of the 4th International Symposium on Static Analysis (SAS '97)*. Springer-Verlag, Berlin, Heidelberg, 16–34.

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

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

R Van Engelen. 2000. *Symbolic evaluation of chains of recurrences for loop optimization.* Technical Report. Citeseer.

Robert A Van Engelen. 2001. Efficient symbolic analysis for optimizing compilers. In *International Conference on Compiler Construction*. Springer, 118–132.

John Whaley and Monica S Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*. 131–144. https://doi.org/10.1145/996841.996859

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*. 197–208. https://doi.org/10.1145/1328897.1328464

Jianwen Zhu. 2005. Towards scalable flow and context sensitive pointer analysis. In *Proceedings. 42nd Design Automation Conference, 2005.* IEEE, 831–836. https://doi.org/10.1145/1065579.1065798