Restrictification of Function Arguments

Victor Hugo Sperle Campos
UFMG
victorsc@dcc.ufmg.br

Péricles Rafael Alves
UFMG
periclesrafael@dcc.ufmg.br

Henrique Nazaré Santos
UFMG
hnsantos@dcc.ufmg.br

Fernando Magno Quintão Pereira
UFMG
fernando@dcc.ufmg.br

Abstract

Pointer aliasing still hinders compiler optimizations, in spite of years of research on pointer disambiguation. Because the automatic disambiguation of pointers is a difficult endeavor, several programming languages offer programmers mechanisms to distinguish memory references, such as the "restrict" keyword in C. However, the use of such mechanisms is prone to human mistakes. In this paper we present a suite of automatic techniques that mitigate this problem. We have designed, implemented and tested three different ways to disambiguate pointers passed as arguments of functions. Our techniques combine static analyses to infer symbolic bounds of memory regions and code versioning. We generate a clone for each function whose arguments we can disambiguate and optimize it assuming the absence of aliasing among formal parameters. At runtime, we use the results of the symbolic interval tests to decide which version of a function we should call: the original one or the "restricted" clone, whenever we can prove that no aliasing can occur at runtime. An implementation of our restrictification methods in LLVM shows that we can vectorize up to 63% more operations than what could be accomplished using the -O3 optimization level of said compiler. When applying the optimization on OpenCV benchmarks, we have observed speedups as great as 40%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — Compilers, Optimization

General Terms Languages, Experimentation, Performance

Keywords Aliasing, Static analysis, Cloning, Optimization, Compiler

1. Introduction

Compilers need to deal with pointers to carry out effective optimizations. A staple feature in imperative programming languages, pointers pose several challenges to code analyses and transformations [14]. These challenges stem from the possibility of aliasing, which increases the number of dependences among data; hence, forcing optimizations to be more conservative [3, Ch 17.5]. Aliasing has compelled the programming languages community to design, implement, and test a very large number of different techniques to disambiguate pointers. Nevertheless, the classic memory disambiguation approaches are either too costly [15, 29] or too imprecise [28] to be practical.

To mitigate the problem posed by pointers, some programming languages furnish developers with high-level constructs that they can use to tell compilers that variables do not alias each other. Examples of such techniques include the several systems of ownership types [23], which, today, find concrete use in keywords like Rust's `mut` and Scala's `@unique`. In this direction, the C programming language, since the C99 Standard, features the `restrict` keyword, which programmers use to indicate that a memory position can only be accessed through a pointer p, or through values derived from p, such as $p + 1$. Yet, the task of annotating type declarations with `restrict` or similar modifiers still belongs to the programmer, and contrary to Rust's or Scala's type system, C's does not perform any verification, be it statically or dynamically, to ensure that annotations are not misplaced.

In this paper, we call the act of disambiguating two pointers "restrictification", and we attack the aliasing problem by proposing three different and complementary techniques to restrictify pointers used as arguments of functions. Our goal, as we explain in Section 2, is to provide compilers with information that enables them to perform more extensive optimizations. Restrictification is not always possible: it depends on the calling site where a function is invoked, and it depends on the arguments passed to that function. To maximize the number of times that we can apply it, we resort to code cloning: for each function that is "restrictifiable", we create a clone of it. In this clone, the compiler is free to assume absence of aliasing between the function's arguments. Even though, in principle, we could already benefit from partial restrictification, i.e., by disambiguating only some of the function's arguments, for simplicity, we only use a restricted clone when we can prove that none of its arguments alias each other. The three restrictification approaches that we introduce in this paper can be used separately, or combined, as the application of one does not prevent the use of the other. Another advantage of our approach is that these three techniques rely on infrastructure already available in mainstream compilers. Thus, their implementation is feasible; and as we will show, very effective. They are described as follows:

- In Section 3.1, we discuss a *purely static* way to disambiguate function arguments. We use off-the-shelf static analyses to decide, for each call site, when the restricted clone can be called. As we explain in Section 3.1.1, we are able to apply this approach even when we compile the modules that constitute a program separately.

- In Section 3.2.1, we discuss a hybrid restrictification approach, which performs a *forward* analysis to infer the size of memory regions and uses this information to create statically, at each call site, checks that decide, dynamically, if it is possible to invoke the restricted function instead of its original version.
- In Section 3.2.2, we discuss a second hybrid approach that uses a *backward* analysis to determine the bounds of memory regions. With this information, we also statically produce checks to determine, dynamically, when it is safe to branch to code that is aliasing-free.

To validate the ideas discussed in this paper, we have implemented them in the LLVM compilation infrastructure [17], version 3.7. As we show in Section 4, our different pointer disambiguation techniques let this compiler produce non-trivial speedups in several different benchmarks. For instance, the extra aliasing information that we provide lets LLVM produce code 8% faster for MiBench and Stanford. The code size expansion caused by function cloning is affordable: we have observed an increase of less than 10% in most programs. Furthermore, our technique can bring benefits that would not be possible with inlining, our most direct competitor. One of such benefits is the possibility to apply it onto library code, without the need to recompile client programs. To support this last statement, we have applied our optimizations onto OpenCV, a broadly used computer vision library. In this experiment, we were able to speedup 39% of the benchmarks.

2. Overview

To explain our contributions, we shall use two functions seen in Figure 1: `sumTillZero` and `sumTillSize`. Before we proceed in our presentation, we would like the reader to consider the following question: under which circumstances can we transform these functions into the optimized versions seen in Figure 2? The optimization that we use in this example is *scalar promotion*: we are replacing the store operations on lines 12 and 18 of Figure 1 by register assignments.

We cannot always do this optimization, because, even though this transformation is safe for the three calling contexts seen on lines 27-30 of Figure 1, these functions can still be invoked from outside the program. For instance, functions `sumTillZero` (Fig. 1) and `sumTillZeroOpt` (Fig. 2) produce different results when they receive the following actual parameters: $\mathtt{r} = \{1, 2, 3, 0\}$ and $\mathtt{w} = \mathtt{r} + 1$. The former will yield $*\mathtt{w} = 9$, whereas the latter will give us $*\mathtt{w} = 8$. In other words, functions `sumTillZero` and `sumTillZeroOpt` are only equivalent if their arguments do not reference overlapping memory regions. Notice that this limitation would apply to any purely static pointer disambiguation approach, including inter-procedural points-to analysis combined with function specialization and cloning[1].

Purely Static Restrictification. Henceforth, we shall say that for each function of interest, we have its *original* and its *restricted* version. A function is deemed "restricted" if we can prove, by whatever means, that its arguments do not reference overlapping memory regions. We shall call the act of replacing a function by its restricted version *restrictification*. If we had access to both versions of each function, then we could call the restricted code in every call site in which we can prove that aliasing does not

[1] This combination of inter-procedural points-to analysis plus cloning is used, for instance, by the `-qipa` option of the XLC compiler (see `http://www-01.ibm.com/support/knowledgecenter/SSPSQF_9.0.0/com.ibm.xlcpp111.linux.doc/proguide/qipa_opts.html`).

```
 1 char* get_new_array(int size, char init) {
 2   char* A = malloc(size);
 3   for (int i = 0; i < size - 1; i++) {
 4     A[i] = init;
 5   }
 6   A[size - 1] = 0;
 7   return A;
 8 }
 9
10 void sumTillZero(char* r, char* w) {
11   while (*r) {
12     *w += *(r++);
13   }
14 }
15
16 void sumTillSize(char* r, char* w, int N) {
17   for (int i = 0; i < N; i++) {
18     *w += r[i];
19   }
20 }
21
22 int main(int argc, char** argv) {
23   char A1[argc], A2 = 0;
24   char* A3 = get_new_array(argc, 0);
25   char* A4 = get_new_array(1, 0);
26   char* A5 = get_new_array(1 + argc, 1);
27   sumTillZero(A1, &A2);
28   sumTillZero(A3, A4);
29   sumTillSize(A3, A4, argc);
30   sumTillSize(A5, A5 + argc, argc);
31 }
```

Figure 1. Our different analyses let us disambiguate all the pointer arguments passed to the two `sumTill*` functions.

happen among actual parameters. However, disambiguating pointers through standard static analysis is not always feasible. For instance, LLVM's alias analyses, currently based on five different algorithms, can only disambiguate the arguments of `sumTillZero` in line 27 of Figure 1. These purely static approaches, currently available in LLVM, are not precise enough to disambiguate the arguments of the other functions, invoked at lines 28 and 30. Thus, in this example, we would be able to restrictify only the calling site at line 27 of Figure 1 using static alias analyses. Figure 3, line 8, shows the original `main` function after we rewrite the calling site to invoke the improved version of `sumTillZero`.

Restrictification via Forward Region Analysis. Alias analyses with no context sensitivity will not be able to determine that arrays `A3` and `A4` do not overlap each other. This imprecision happens because these pointers share the same allocation site: line 2 of Figure 1. The static alias analyses available in mainstream compilers such as LLVM or gcc are context-insensitive; hence, they suffer from this limitation. However, it is still possible to invoke the optimized function `sumTillZeroOpt` at line 28 of Figure 1 – provided that we can test for aliasing at the calling site. Arrays `A3` and `A4` will not alias if the region that can be addressed from these two base pointers do not overlap. If we let the size of `A3` be S_3, and the size of `A4` be S_4, then we have an absence of aliasing if $\mathtt{A3} + S_3 \leq \mathtt{A4} \vee \mathtt{A4} + S_4 \leq \mathtt{A3}$. There are several ways to infer the values of S_3 and S_4. In this paper, we experiment with a forward region analysis which produces results similar to those produced by the techniques introduced by Bodik *et al.* [5] and Nazaré *et al.* [20]. Figure 3, lines 10-14, shows the result

```
 1 void sumTillZeroOpt(char* r, char* w) {
 2   int tmp = *w;
 3   while (*r) {
 4     tmp += *(r++);
 5   }
 6   *w = tmp;
 7 }
 8
 9 void sumTillSizeOpt(char* r,char* w,int N) {
10   int tmp = *w;
11   for (int i = 0; i < N; i++) {
12     tmp += r[i];
13   }
14   *w = tmp;
15 }
```

Figure 2. Optimized versions of the functions seen in Figure 1. Notice that these functions are only equivalent to their original selves in the absence of aliasing.

```
 1 int main(int argc, char** argv) {
 2   char A1[argc], A2 = 0;
 3   char* A3 = get_new_array(argc, 0);
 4   char* A4 = get_new_array(1, 0);
 5   char* A5 = get_new_array(1 + argc, 1);
 6  ------------------------------------ Section 3.1 ---
 7   // PSA:ok, FRI:ok, BRI:no
 8   sumTillZeroOpt(A1, &A2);
 9  ----------------------------------------------------
   ---------------------------------- Section 3.2.1 -
10   // PSA:no, FRI:ok, BRI:no
11   if (A3 + argc ≤ A4 || A4 + argc ≤ A3)
12     sumTillZeroOpt(A3, A4);
13   else
14     sumTillZero(A3, A4);
    --------------------------------------------------
15  ---------------------------------- Section 3.2.2 -
16   // PSA:no, FRI:ok, BRI:ok
17   sumTillSizeBri(A3, A4, argc);
18
19   // PSA:no, FRI:no, BRI:ok
20   sumTillSizeBri(A5, A5 + argc, argc);
    --------------------------------------------------
21 }
```

Figure 3. Invocation sites seen earlier in Figure 1 rewritten to accommodate our optimization. Our approaches are named as follows: **PSA** - purely static approach (Sec. 3.1); **FRI** - forward region inference (Sec. 3.2.1); **BRI** - backward region inference (Sec. 3.2.2). If the approach is able to restrictify function arguments at a given call site, then we mark it with *ok*, otherwise we mark it with *no*.

of our optimization, enabled by forward region analysis. As we see in said figure, a runtime check (line 11) determines when it is safe to invoke the optimized version of `sumTillZero`.

Restrictification via Backward Region Analysis. The previous technique, based on a forward region inference analysis, is not able to disambiguate the pointers passed as parameters to `sumTillSize` at line 30 of Figure 1. On the other hand, by inspecting the body of said function (Fig. 1, lines 16-20), we know that aliasing is never possible. This inspection, be it performed automatically or manually, uses a *backward analysis*: we check how the array is used and assume that its size is given by the maximum range of any variable used to index it. In our example, a backward analysis reveals that

```
void sumTillSizeBri(char* r,char* w,int N) {
  if (r + N ≤ w || w + 1 ≤ r) {
    // See Figure 2, lines 9-15
    sumTillSizeOpt(r, w, N);
  } else {
    // See Figure 1, lines 16-20
    sumTillSize(r, w, N);
  }
}
```

Figure 4. Version of `sumTillSize` optimized by our backward analysis of Section 3.2.2.

Section	WPI	Ch.Call	Ch.Fun	Dyn
3.1	Yes	Yes	No	No
3.2.1	Yes	Yes	No	Yes
3.2.2	No	No	Yes	Yes

Figure 5. A comparison between the different pointer disambiguation approaches that we use in this paper. **WPI**: Whole-program Information affects precision. **Ch.Call**: involves inserting computation at call sites. **Ch.Fun**: involves inserting computation in the target function. **Dyn**: determines, based on runtime values, when it is safe to call optimized code.

the size of `r` is `N` and the size of `w` is 1 byte. Therefore, as long as $\mathtt{r} + N \leq \mathtt{w} \vee \mathtt{w} + 1 \leq \mathtt{r}$, no aliasing is possible within the body of `sumTillSize`. The technique that we shall describe in Section 3.2.2 performs this kind of analysis and transformation. In this example, that approach produces a new version of `sumTillSize`, which we show in Figure 4. Backward analysis has one advantage: it does not require whole-program analysis. Instead, it obtains information exclusively from syntax available within the target function. Its disadvantage is the fact that it can only analyze data structures accessed by variables whose integer ranges are bounded, such as `i` in Figure 1, lines 17-18.

3. Three Pointer Disambiguation Techniques

The three different approaches that we use to restrictify function arguments provide different tradeoffs, which we summarize in Figure 5. The purely static approach of Section 3.1 yields more precise results if we are able to infer aliasing in an interprocedural fashion; however, in this work, we limit ourselves only to the alias analyses available in LLVM, all of which are intraprocedural. The two approaches discussed in Section 3.2 generate code that decides, at runtime, when it is possible to invoke the restricted version of a function. This decision is based on the arguments passed to the function. The approach of Section 3.1, on the other hand, cannot afford such flexibility: either the restricted function or its original version is always invoked at a given call site. As a consequence, the purely static approach does not involve any runtime overhead, unless we link our code against dynamic libraries, as we explain in Section 3.1.1. In the rest of this section, we discuss each approach.

3.1 Static Restrictification

As previously mentioned, the purely static approach to the restrictification of function arguments resorts to traditional points-to analysis to identify invocation sites in which the arguments passed to the callee do not alias each other. Whenever this is the case, we replace a function with its restricted clone. Compiler-related literature describes a plethora of different algorithms to solve alias analyses.

In this paper, we have used the five different analyses available in LLVM 3.7 to disambiguate arguments of pointer type. These analyses are described below:

- `basic-aa`: this points-to analysis uses several known facts to disambiguate pointers. For instance, none of the following pairs of pointers can alias each other: different fields within a structure, different global variables, different local variables, memory allocated in the stack and in the heap, etc. All these checks run in $O(1)$. In spite of being one of LLVM's simplest implementations of alias analysis, `basic-aa` is one of the most effective.
- `type-based-aa`: this analysis relies on the fact that the C99 standard forbids pointers of different types from referencing overlapping memory regions.
- `globals-aa`: this analysis uses the fact that global variables that don't have their address taken cannot alias other pointers.
- `scev-aa`: this analysis uses scalar evolution expressions [9, p.18] to solve alias queries. Its main target is the disambiguation of pair of pointer operations within a single iteration of a loop, by statically inferring their access distance. For instance, this analysis would be able to tell that the access operations `a[i]` and `a[i+3]` can never alias each other in the same iteration of a loop, as long as `i` is an induction variable whose value does not vary within a single iteration.
- `cfl-aa`: this analysis implements a context-insensitive alias analysis based on context-free grammars (CFL). This algorithm is implemented after Zheng *et al.* [32] and Zhang *et al.* [31].

The precision of all these analyses is cumulative: if any of them can prove that two pointers are non-aliases, then they are considered non-aliases. Thus, by invoking more or less analyses, the LLVM user trades precision for speed.

The main advantage of the purely static restrictification approach is the fact that it does not incur any runtime overhead. If we can prove the absence of aliasing among the arguments of a function f at a given invocation site s, then we are free to invoke the restricted version of f. In this case, we do not have to insert any dynamic check at s, as the approach of Section 3.2.1 forces us to do. Nor do we need to insert guards in the restricted function, as the technique that we shall discuss in Section 3.2.2 requires. On the other hand, the purely static restrictification of functions is also the most inflexible of our techniques: either we always use the restricted version of a function, at a given invocation site, or we never use it. In other words, we cannot check for no-aliasing based on runtime data, in hopes to call the restricted function more often. Consequently, the technique that we discussed in this section is only appliable in one of the four invocation sites of Figure 1, i.e., the call at line 27. In this example, we know that different local variables cannot alias each other; hence, we are free to invoke the restricted version of `sumTillZero` in line 27 of Figure 1, as Figure 3, line 8, shows.

3.1.1 Dynamic Linkage

We generate optimized clones at compilation time, and replace call sites at link time. This approach has one limitation: it does not work when the function to be restricted f belongs to a dynamically linked library. If this is the case, when code is generated for a caller g, the compiler will not know that f has an optimized version. Going back to our example of Figure 1, to replace the call of `sumTillZero` at line 27 by a call to `sumTillZeroOpt` (Figure 2), we need to know, when linking the executable for function `main`, that `sumTillZeroOpt` exists. This shortcoming would prevent us from applying our optimization in programs meant to be used as shared libraries, for example.

```
int main(int argc, char** argv) {
  char A1[argc], A2 = 0;
  ...
  void *opt = dlsym(RTLD_NEXT, "sumTillZeroOpt");
  if (opt) {
    void (*sumTillZeroOpt)(int*, int*) = opt;
    sumTillZeroOpt(A1, &A2);
  } else {
    sumTillZero(A1, &A2);
  }
  ...
}
```

Figure 6. Snippet of function `main`, seen in Figure 1, rewritten after the replacement phase used in purely static restrictification.

To overcome this limitation, the static restrictification of functions happens in two phases. The first phase is called *cloning*: at compilation time, we produce a restricted version of every eligible function that is not marked `static`. A function is deemed eligible if it receives two or more arguments having pointer type. Because a non-static function f can be called outside the module m where f is implemented, restrictification happens even if f is not used within m. The second phase is called *replacement*: at linking time, we replace original function calls by their optimized versions whenever their definition is available and the arguments don't alias. Furthermore, for the remaining functions whose optimized versions aren't known yet, we surround every call site with guards. These guards contain code to check, at execution time, if the optimized version of the function is available at any library loaded dynamically. The client will invoke the optimized clone whenever possible, or the original function otherwise. We have implemented this runtime verification via the `dlsym`[2] system call, which is provided by any Unix-like operating system, while similar routines also exist for Windows. This system call receives one string as argument – the name of a function – and searchers all shared libraries for this routine. Figure 6 shows the code that we produce for our running example.

These two phases – cloning and replacement – are independent. If only replacement is performed, then the program still works normally. When the optimized version of a function f does not exist, the `dlsym` call returns null. In this case, the original version of f is used. At any point in time, the optimized shared library that contains the restricted clone of f can be linked to the client binary. Once this happens, the client program will start using the restricted version of f automatically and seamlessly. This approach has one shortcoming: name collisions. The `dlsym` call searches functions by name, and the cloning procedure adds to each cloned function's name a string that is unlikely, though not impossible, to produce collisions with the remaining names in the program. Nevertheless, this possibility exists. Also, since `dlsym` performs searches only inside shared libraries, this method doesn't support the optimization of calls to functions that are defined in static libraries.

3.2 Hybrid Restrictification

We call *hybrid restrictification* the combination of static and dynamic techniques to restrictify function arguments. We use static analysis to infer symbolic expressions that denote the bounds of memory regions. These bounds let us create checks, which will determine, at runtime, when it is safe to assume the absence of aliasing among function arguments. We have tested two different static analyses to infer memory bounds. Henceforth, these approaches will be called *forward* region inference and *backward* region inference.

[2] `http://linux.die.net/man/3/dlsym`

These two analyses are bootstrapped by a *symbolic range analysis*, which produces a map R from every integer variable v to $[v_\downarrow, v_\uparrow]$. We let $v_\downarrow$ be the symbolic lower bound of v, and $v_\uparrow$ be its symbolic upper bound. Programming languages literature contains several implementations of such an analysis [4, 20, 26]. In this paper we use the algorithm of Nazaré *et al.* [20]. This analysis gives us, for instance, that $R(\texttt{i}) = [0, \texttt{size} - 1]$ at line 4 of Figure 1.

To represent ranges, we use a set S of *symbolic expressions*. A *symbol* is any name, in the program text, that cannot be reconstructed as function of other names. In our case, symbols are: (i) values returned from functions whose body is not present in the target code; or (ii) values created by load instructions that might dereference more than one memory location. A symbolic expression is defined by the grammar below, where s is a symbol and $n \in \mathbb{N}$:

$$\begin{aligned} E \quad ::= \quad & n \mid s \mid \min(E, E) \mid \max(E, E) \mid E - E \\ & \mid E + E \mid E/E \mid E \bmod E \mid E \times E \\ & \mid -\infty \mid +\infty \end{aligned}$$

We augment S with a partial order, which is defined only between symbolic expressions that are integer constants, e.g.: $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots < +\infty$. There is no ordering between expressions involving different symbols. The product of two instances of this partially ordered set defines a semi-lattice $(S^2, \subseteq, \cup, \emptyset, [-\infty, +\infty])$. The join operator "$\sqcup$" is given by:

$$[a_1, a_2] \sqcup [b_1, b_2] = [\min(a_1, b_1), \max(a_2, b_2)]$$

The least element $\emptyset$ is such that:

$$\emptyset \sqcup [l, u] = [l, u] \sqcup \emptyset = [l, u]$$

and the greatest element $[-\infty, +\infty]$ gives us that:

$$[-\infty, +\infty] \sqcup [l, u] = [l, u] \sqcup [-\infty, +\infty] = [-\infty, +\infty]$$

Neither the forward nor the backward region inference analyses are contributions of this paper: they have been described in previous work. Backward region inference has been used to check the complexity of code [25] and to disambiguate pointers in *doall* loops [26]. Variations of forward region inference have been used in previous work to remove bound checks in Java [5] and in type-safe C code [20]. Nevertheless, we claim as a contribution the use that we make of them, for, to the best of our knowledge, they have not been used to generate tests that disambiguate pointers at function level. Section 5 provides a deeper comparison of our work and the previous literature.

3.2.1 Forward Region Inference

Forward region inference discovers the sizes of arrays based on the allocation sites visible within the program code. In other words, the arguments of memory allocation routines, such as `malloc`, `valloc`, `calloc`, and `realloc` are bound to the arrays that said routines produce. In this paper we have experimented with a very simple version of forward region inference, which is implemented by the transfer functions in Figure 7. These transfer functions are standard in symbolic range analysis. Finding their fixed point can be done in time linear on the number of functions, because widening (∇) ensures quick termination. The forward region inference analysis finds, for each pointer variable v, the *maximum addressable offset* $F(v)$ that can use v as a base pointer. Thus, if we say that $F(v) = N$, then we know that any address in the range $[v + 0, v + N]$ *may* dereference a region pointed to by v. Notice that this is a fundamentally different abstract semantics than the one adopted by previous forward region inference analyses [4, 5, 20]. In that case, the authors were interested in detecting bugs, so their meet operation was *minimum*. We are interested in finding conservative memory bounds; hence, our meet operation is *maximum*.

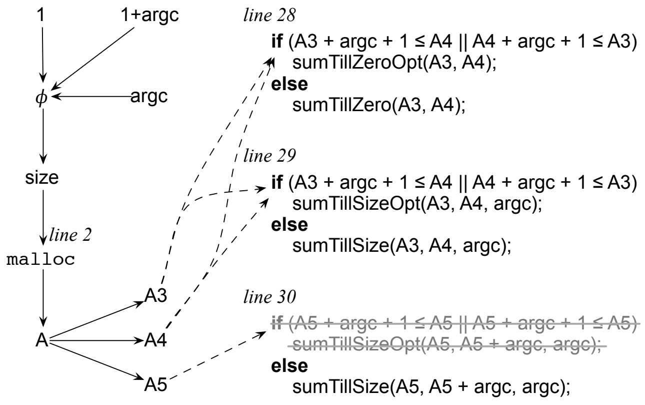
We run our analysis on programs in the Static Single Assignment (SSA) form [7]. This representation ensures that the region information associated with a variable is invariant along the entire live range of that variable. The ϕ-functions, typical of the SSA format, let us join information produced at different program points. Many arrays are defined in functions other than the procedure where they are used; thus, ideally, forward region inference should run interprocedurally. We use ϕ-functions to implement interproceduralty. To explain this last point, let's assume that the target program contains a function declaration such as $f(\ldots, v_i, \ldots)$, and n different calls such as $f(\ldots, a_i^1, \ldots)$ and $f(\ldots, a_i^n, \ldots)$, where the i^{th} argument is of pointer type. If that is the case, then we model the transfer of information from actual to formal arguments via the following n-ary ϕ-function: $v_i = \phi(a_i^1, \ldots, a_i^n)$. Similarly, we use simple copies, i.e., $v_1 = v_2$ to model the transfer of information due to the return operation.

Figure 8 shows the result of applying our forward range estimation on the program seen in Figure 1. We notice that the call to `malloc` in line 2 of Figure 1 is reached by three different definitions of `size`. Thus, our meet operation gives us that the most conservative estimate on the upper bound of array `A` is $\max(1, 1 + \texttt{argc}, \texttt{argc})$, which we can simplify to $\texttt{argc} + 1$, given the knowledge that $R(\texttt{argc})_\downarrow \geq 0$[3]. These bounds let us modify the call sites at lines 28 and 29 of Figure 1, as to check memory ranges at runtime. The rewritten code is shown at the right side of Figure 8.

$$\begin{aligned}
v_1 = \texttt{malloc}(v_2) \quad &\Rightarrow \quad F(v_1) = R(v_2)_\uparrow - 1 \\
a = b \quad &\Rightarrow \quad F(a) = F(b) \\
v_1 = v_2 + n \text{ with } n \in \mathbb{N} \quad &\Rightarrow \quad F(v_1) = F(v_2) - n \\
v_1 = v_2 + v_3 \text{ with } v_3 \text{ scalar} \quad &\Rightarrow \quad F(v_1) = F(v_2) - R(v_3)_\uparrow \\
v = \phi(v_0, v_1) \quad &\Rightarrow \quad F(v) = \max(F(v_0), F(v_1))
\end{aligned}$$

$$R_1 \nabla R_2 = [l, u], \text{ where } \begin{cases} l = R_{1\downarrow} & \text{if } R_{1\downarrow} = R_{2\downarrow} \\ l = -\infty & \text{otherwise} \\ u = R_{1\uparrow} & \text{if } R_{1\uparrow} = R_{2\uparrow} \\ u = +\infty & \text{otherwise} \end{cases}$$

Figure 7. Forward Region Inference Analysis.



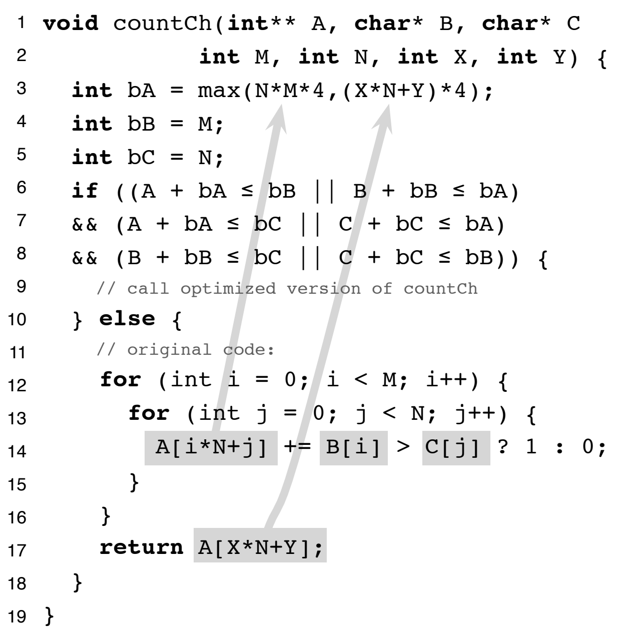
```
if (A3 + argc + 1 ≤ A4 || A4 + argc + 1 ≤ A3)
    sumTillZeroOpt(A3, A4);
else
    sumTillZero(A3, A4);

if (A3 + argc + 1 ≤ A4 || A4 + argc + 1 ≤ A3)
    sumTillSizeOpt(A3, A4, argc);
else
    sumTillSize(A3, A4, argc);

~~if (A5 + argc + 1 ≤ A5 || A5 + argc + 1 ≤ A5)~~
    ~~sumTillSizeOpt(A5, A5 + argc, argc);~~
else
    sumTillSize(A5, A5 + argc, argc);
```

Figure 8. The result of applying the forward region inference analysis on the program seen in Figure 1.

[3] In Section 5.1.2.2.1 of the ISO/IEC 9899:2011 C standard, "The value of `argc` shall be nonnegative".



```
1  void countCh(int** A, char* B, char* C
2               int M, int N, int X, int Y) {
3    int bA = max(N*M*4,(X*N+Y)*4);
4    int bB = M;
5    int bC = N;
6    if ((A + bA ≤ bB || B + bB ≤ bA)
7    && (A + bA ≤ bC || C + bC ≤ bA)
8    && (B + bB ≤ bC || C + bC ≤ bB)) {
9      // call optimized version of countCh
10   } else {
11     // original code:
12     for (int i = 0; i < M; i++) {
13       for (int j = 0; j < N; j++) {
14         A[i*N+j] += B[i] > C[j] ? 1 : 0;
15       }
16     }
17     return A[X*N+Y];
18   }
19 }
```

Figure 9. Example of code produced by our backward region inference analysis. Memory accesses are marked grey. Arrows indicate the constraints produced for matrix A.

The check inserted at the last call of `sumTillSize` is trivially false, and we do not insert it. Notice that the size of a check is quadratic on the number of pointers passed as arguments of the target function.

3.2.2 Backward Region Inference

The forward region inference analysis of Section 3.2.1 is a whole-program analysis. This interprocedurality might be a limitation if we need to analyze code that is only partially available. To mitigate this problem, we have used a backward region inference analysis, which is intraprocedural by nature. This analysis maps every variable v, of pointer type, to a symbolic expression $B(v_1)$, which has the same semantics as F (introduced in Section 3.2.1), i.e., a conservative bound on the maximum valid offset that can use v_1 as base address[4]. To infer the bounds of arrays, we create, for each occurrence of a memory access $v_1[v_2]$, a constraint $B(v_1) \geq R(v_2)_\uparrow$. Notice that this analysis is flow-insensitive, and the relative order in which different dereferences of v_1 appear within a function are immaterial.

As an example, the backward region inference, when applied to the function `get_new_array`, seen in Figure 1, would give us the following two constraints: (i) $B(\texttt{A}) \geq R(\texttt{i})_\uparrow = \texttt{size} - 2$, due to the access at line 4, and (ii) $B(\texttt{A}) \geq R(\texttt{size})_\uparrow - 1 = \texttt{size} - 1$, due to the access at line 6. The union of these two constraints gives us that the largest valid offset from pointer A, within function `get_new_array`, cannot be larger than $\texttt{A} + \texttt{size} - 1$. Similarly, the backward region inference analysis gives us two constraints for function `sumTillSize`, in Figure 1: (i) $B(\texttt{r}) \geq R(\texttt{N})_\uparrow - 1$ and (ii) $B(\texttt{w}) \geq 0$.

Figure 9 shows an example of code produced by our backward analysis. Every memory access within the target function gives us constraints, as long as we can infer the bounds of this access using a symbolic range analysis. Whenever we have two or more constraints for the same pointer p, we use the maximum of them to bound p.

[4] Even though $F(v)$ (Sec. 3.2.1) and $B(v)$ (Sec. 3.2.2) have the same semantics, we shall keep these two maps to avoid confusion when referring to the different region inference analyses.

In this example, we have four memory accesses, two of them to matrix A. The memory bounds that our analysis finds are given on lines 3-5 of the example. The tests that we generate can be seen on lines 6-8. Notice that the backward analysis can only infer bounds to memory access functions that use variables bound to known range expressions. It cannot, for instance, analyze expressions such as `a[b[i]]`. Moreover, we cannot infer bounds of a pointer p, passed as argument of a function f, if p might escape the scope of f into another function g. In practice, we use LLVM's escape analysis to identify this kind of situation. Yet, our technique is not limited to scientific codes; rather, it is widely applicable. In Section 4 we show that it can handle thousands of benchmarks present in the OpenCV's test suite.

3.3 Code Generation

Both the forward approach of Section 3.2.1 and the backward approach of Section 3.2.2 produce bound comparisons that, at runtime, check for the possibility of aliasing among pointers. Given two pointers p_1 and p_2, of type T_1 and T_2, and bounds M_1 and M_2, we generate the following guard:

$$p_1 + M_1 \times \texttt{sizeof}(T_1) \leq p_2 \text{ or } p_2 + M_2 \times \texttt{sizeof}(T_2) \leq p_1$$

Whenever the guard is true, we divert the program flow to code that can be optimized under the assumption that p_1 and p_2 cannot refer to overlapping memory regions. If a function contains n different pointers used as arguments, then we can generate up to $n \times (n-1)/2$ such guards for that function, using either the backward or the forward approach. In other words, we generate a *quadratic* number of comparisons per number of memory accesses within the optimized region. Nevertheless, the overhead of the dynamic checks is negligible, as we shall explain in Section 4.

The difference between these two techniques, in terms of code generation, is that the tests of Section 3.2.1 are placed at the function call site, whereas the tests of Section 3.2.2 are placed within the function, at its entry point. This fact is an advantage of the backward approach, because it allows us to apply it independently per program module. In other words, we can link our optimized code against non-optimized binaries without the need to recompile those binaries. Currently, we only generate tests if the symbolic bounds of a variable are expressions of variables *alive* at the program point where the guard will be placed. We say that a variable v is alive at a program point i if it is used at a program point i', there is a path from i to i', and v is not redefined along this path. Global variables are alive everywhere in the program code.

A Note on Safety. Our hybrid analyses might generate invalid checks if the target program already presents out of bounds accesses. Let's imagine, for instance, that function `countCh`, in Figure 9, receives a size M less than the actual number of lines of A. This program has undefined behavior, for an out-of-bounds access would happen at line 14 of Figure 9. This programming error might cause our checks to branch to the optimized code, when pointers indeed alias. Our analyses are used to optimize programs, and we believe that it is meaningless to apply it on incorrect code. Incorrect code is not an issue for the forward analysis. Because it keeps track of the actual expressions used to allocate memory, it only generates tests when such expressions – which denote the correct bounds of memory – are available.

4. Evaluation

To validate our techniques, we have implemented them on the LLVM compilation infrastructure [17], version 3.7. In the first part of this evaluation section, we have tested our implementation on four different benchmarks suites: (i) BitBench, composed of different Bit Stream manipulation algorithms, (ii) MiBench, containing a

Benchmark	Size	Functions	Eligible	Calls
BitBench	1,167	28	11	12
MiBench	357,159	1,395	525	1,505
Shootout-C++	25,701	2,482	720	1,082
Stanford	2,249	80	7	10
OpenCV	2,150,492	97,123	37,523	94,354

Figure 10. Benchmark descriptions: size (number of LLVM Intermediate Representation instructions), number of functions overall, number of functions eligible for cloning (take two or more pointer arguments), number of call sites where eligible functions are invoked.

Benchmark	PSA	FRI	BRI
BitBench	37%	6%	17%
MiBench	15%	1%	72%
Shootout-C++	26%	4%	44%
Stanford	60%	10%	63%

Figure 11. Reach of our different disambiguation methods. **PSA**: number of call sites optimized by the Purely Static Approach. **FRI**: number of call sites instrumented when using Forward Region Inference. **BRI**: number of functions guarded with dynamic checks by the Backward Region Inference.

number of applications for embedded systems, (iii) Shootout-C++, which encompasses several scientific and mathematical algorithms, and (iv) Stanford, with common general-purpose routines, such as sorting, permutation, and puzzle-solving. These benchmarks are bundled in the LLVM test suite[5] and details about their structure are presented in Figure 10. Later in this section, we present the evaluation of runtime improvements produced by our techniques on the OpenCV library.

All the benchmarks used in this evaluation have been compiled and linked with the `-O3` optimization level of LLVM prior to being transformed by our techniques. At this level, the compiler performs roughly 60 analysis and optimization passes on the target programs. Among these, LLVM applies function inlining, which is one of our direct competitors. Therefore, improvements seen in this section are on top of the highest degree of optimization in LLVM, which includes inlining. The numbers presented here were obtained using an Intel Xeon 2.0GHz machine, with 16GB of RAM, running Linux version 3.13.0. Run time numbers represent the average of at least 10 executions. In the following subsections we explain the effectiveness of our methods by investigating the different forms in which they can affect the compilation process.

4.1 The Reach of Restrictification

Figure 11 shows the reach of our three disambiguation techniques when analyzing and instrumenting the four suites of benchmarks described previously. The numbers shown are for each suite as a whole. For the Purely Static method (Section 3.1), we present the relative number of call sites optimized out of all points in the program that invoke functions passing pointers as arguments. For MiBench, we were able to optimize 15% of the call sites. In Stanford, the best relative result, we could disambiguate 6 out of 10 calls. When considering all four suites, the average reach of the static method was 18%. We also note that 18% of the functions that receive pointer as parameters had all their call sites optimized, i.e., their original – unoptimized – version could be removed from the program after linking.

It is known that actual aliasing is rare at runtime [11], i.e., it is uncommon for different pointers in a program to reference the same memory region during execution. For this reason, the reach of our hybrid techniques can be approximated by the number of functions for which dynamic alias checks can be generated, as such, checks tend to pass at runtime. For our forward region inference method (Section 3.2.1), we present the number of call sites instrumented with dynamic checks. As seen in the figure, the numbers for this approach were not as significant as the ones presented for the Purely Static method, with our best result being a 10% reach in Stanford. We attribute such low numbers to the fact that symbols used as argument of dynamically memory allocation functions usually are not alive at the call sites where the arrays created by these functions are passed as parameters. To generate guards, our forward analysis needs to have access to these symbols at the place where the tests must be constructed. As a further experiment, we have tried to identify pairs formed by arrays and integers that represent the size of these arrays among the arguments of functions. Only 4% of these pairs denote such size relations.

Our backward region analysis, on the other hand, instruments function entries rather than call sites and does not require whole-program information to work. To measure the reach of this technique, we present the relative number of functions that receive pointers as inputs and could be guarded with dynamic pointer tests. The best result for this method was in MiBench, where we could optimize 72% of target functions. For BitBench, our rather shy results stem from its memory access patterns, from which it isn't possible to extract suitable scalar evolution expressions that enable our backward region analysis to generate dynamic checks.

4.2 Strengthening Code Optimizations

Lack of precise knowledge about dependencies between memory regions can greatly undermine code optimizations. Such an effect is particularly observed in transformations which aim to eliminate redundant computation and exploit parallelism between independent portions of code. Notorious examples of such optimizations are Loop Invariant Code Motion (LICM), Global Value Numbering (GVN), and Automatic Loop Vectorization, all three currently available in LLVM. Therefore, more precise disambiguation techniques, such as the ones we propose in this work, are expected to considerably improve the reach of such program transformations.

Figure 12 shows how our disambiguation methods improve program optimizations. *Locations promoted to registers* represents the number of memory operations that LICM could convert into less costly scalar instructions. We see that both our purely static and backward inference approaches delivered a significant improvement in this metric for the Stanford and MiBench suites. *Redundant loads/stores deleted* represent the number of memory access instructions that GVN could completely eliminate from the program's code. In general, all three methods could improve one of these metrics to some extent, with the purely static and forward inference approaches being the most effective. *Vector instructions generated*, in turn, is the number of vector operations resulting from the combination of different independent instructions. This transformation is also known as Superword-Level Parallelization. An increase in this metric, such as the one that we observed in MiBench, also indicates an increase in the number of loops for which the compiler can prove independence between iterations, making them also suitable for more aggressive optimizations, such as automatic GPU code placement [19].

4.3 Code Size Expansion

Figure 13 shows the relative increase in the program's size when using each of the three techniques. The numbers represent the ratio between the space occupied on disk, in bytes, of the generated binaries. The increase promoted by the purely static method is relatively small, under 10% in all cases, mainly due to its lower precision when compared, for instance, to the more aggressive

[5] `http://llvm.org/docs/TestingGuide.html`

	Locations promoted to registers			Vector instructions generated			Redundant loads deleted			Redundant stores deleted		
Benchmark	PSA	FRI	BRI	PSA	FRI	BRI	PSA	FRI	BRI	PSA	FRI	BRI
BitBench	0%	0%	0%	0%	0%	0%	800%	800%	0%	0%	0%	0%
MiBench	32%	0%	34%	63%	38%	63%	4%	4%	4%	63%	77%	65%
Shootout-C++	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Stanford	200%	0%	200%	0%	0%	0%	26%	18%	0%	0%	0%	0%

Figure 12. Relative improvement of code optimizations when combined with our different disambiguation techniques. **PSA**: purely static approach. **FRI**: forward region inference. **BRI**: backward region inference.

Benchmark	**PSA**	**FRI**	**BRI**
BitBench	8%	8%	0%
MiBench	9%	0%	74%
Shootout-C++	1%	0%	3%
Stanford	4%	0%	26%

Figure 13. Relative increase in executable's size when using each disambiguation method. **PSA**: purely static. **FRI**: forward region inference. **BRI**: backward region inference.

backward inference method. The forward analysis does not increase executable size by a significant factor, except in BitBench. This result is due to the low coverage of call sites that the forward region inference achieves. Furthermore, if the original function is no longer called in the program, we can eliminate it; hence, keeping only the optimized function. The highest increase is observed when using our backward inference technique. Such a result comes from the fact that this method overcomes the major limitations faced by our other two approaches: it uses information contained only within the target function itself and it has a dynamic component that lets us evaluate pointers at runtime.

4.4 Speedup

We measured how our techniques enable runtime improvements with the compiler optimizations that exist today in LLVM. As previously noted, these runtime numbers were obtained after performing the highest level of optimization provided by LLVM, `-O3`, which also includes function inlining, our most direct competitor. Figure 14 shows the ratio between the execution time of the optimized program and the run time of the original program for the first four benchmarks suites. Each bar gives the geometric mean of the run time of all the programs in the corresponding benchmark.

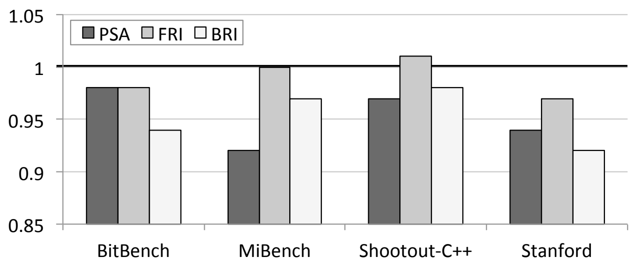


Figure 14. Execution time ratio of each benchmark suite for our three techniques (lower is better).

Even though our methods were able to improve the effectiveness of major program optimizations, the resulting runtime improvements were relatively small. In the best case we could speedup the MiBench and Stanford programs by 8% using the purely static and the backward inference approach, respectively. In the other cases, however, our gains were under 5%, with one specific slowdown in Shootout-C++ using the forward inference method. Figure 15 shows the runtime ratios for the programs in Shootout-C++ suite individually. We achieved mixed results in this benchmark: while there were some good improvements, such as in `ackermann` for the PSA (13%) and `nestedloop` for BRI (15%), we also had more than 10% slowdowns in three programs. At the vast majority of the programs, though, ratios were very close to 1. The slowdowns that we have observed were not caused by the overhead of the guards. Rather, they are the result of optimizations that still offer room for tuning. It is important to notice that our pointer disambiguation techniques enable optimizations, but they do not implement them: this task is left to the compiler.

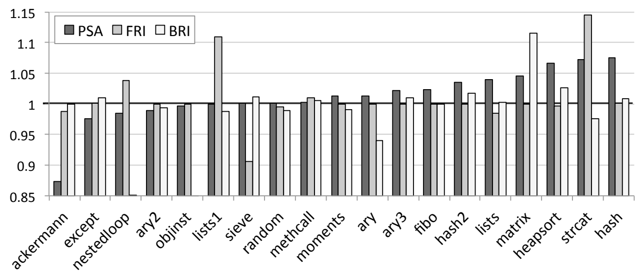


Figure 15. Execution time ratio for each benchmark in Shootout-C++, using our three methods (lower is better).

4.5 Testing Our Approach on Larger Programs

In order to investigate the effectiveness of our approach on large programs, we tested our pointer disambiguation methods on the widely used computer vision library OpenCV[6]. OpenCV is organized as a set of shared libraries to be used by client applications, which are built separately. This represents a perfect scenario for the dynamic linkage extension of our static method, presented in Section 3.1.1, which provides client code access to optimized versions of library functions. In this scenario, pointer disambiguation via function inlining is not possible, since the definitions of core functions are inside the libraries, external to the client code. Here we present results for the PSA and BRI approaches, as they produced the best results for the benchmarks in Figure 15. Additionally, we investigate the effects of these two techniques when used in a complementary fashion: PSA is initially used to replace function calls at linking time through static analysis whenever possible. Then, the remaining calls are handled by BRI, inserting dynamic disambiguation checks.

Figure 16 shows the runtime ratios between the optimized and the regular versions of OpenCV. The OpenCV Benchmark Suite contains thousands of tests, which are bundled together with a simple infrastructure to run and time them all. These tests are grouped into 13 modules. In this experiment we chose two of them: `core` and `imgproc`, because they contain the largest number of benchmarks:

[6] `http://www.opencv.org`

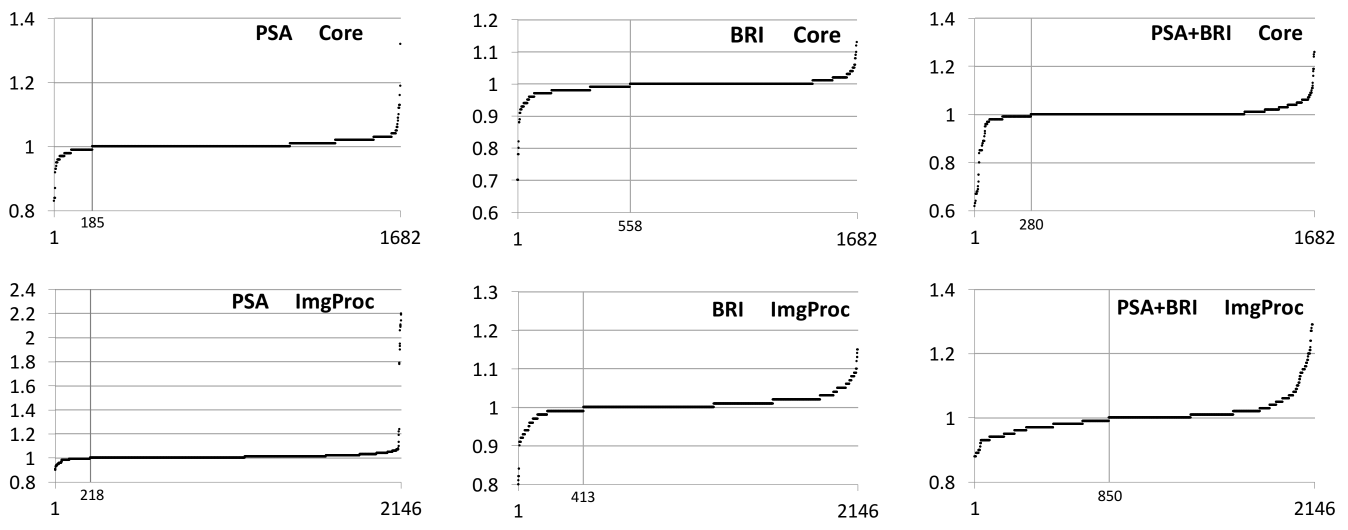


Figure 16. Execution time ratio for the `core` and `imgproc` test modules of OpenCV, using the PSA and BRI methods. Each point along the horizontal axis represents a different test. Lower is better. The vertical line in each chart marks the point where ratios become greater than or equal to 1.

1,682 and 2,146 respectively, and also because they don't require non-free inputs, such as proprietary images or videos. The former measures performance of basic data structures, including dense multi-dimensional arrays and functions that are used across the entire library. The latter is composed mostly by image processing functions, such as filtering and transformations. Each point on the X-axises of Figure 16 represents a different test case, e.g., a whole program that calls OpenCV primitives. On each case, we highlight the number of benchmarks that we have speeded up with our approach. The tests in each figure took about two hours to run.

The leftmost column corresponds to the purely static approach (PSA). In the `core` suite, 11% of the benchmarks had improvements in runtime, with individual speedups of as much as 1.2x. In the `imgproc`, 10% of the benchmarks showed speedups, with individual numbers as high as 10%. The center column of Figure 16 presents runtime ratios for the Backward Region Inference method (BRI). We notice improvements over the PSA: in the first benchmark suite of OpenCV, the BRI produced speedups as high as 42%; in the second, 19% of the benchmarks experienced speedups, with the highest decrease in runtime of 20%.

Combining Different Approaches. The third column shows ratios for the combined method, where PSA is run over programs in a first moment, and BRI is then used to catch disambiguation opportunities that weren't handled statically by PSA. On the `core` suite, 16% of the benchmarks had gains, of which the best gain gave us 61% of speedup. If we compare solely the average ratios of the benchmarks that had speedups (defined here by ratio < 0.97), we notice that for BRI there was a geomean ratio of 93%, while for PSA+BRI the geomean ratio is 81%, accounting for an improvement of 14% in the average. On the second suite, `imgproc`, 39% of the benchmarks experienced speedups, which is an evident step up compared to both PSA and BRI by themselves (10% and 19% respectively). The best ratio on this suite for PSA+BRI was 88%. Based on the much larger number of OpenCV benchmarks that met improvements overall, we conclude that combining the static method, PSA, with the Backward Region Inference, BRI, is the most promising approach to optimize programs. It is still possible to obtain further gains with restrictification, if we tune our techniques more carefully. The use of improved cost models is left as future work, as we discuss in Section 6.

Understanding Slowdowns. A number of tests yielded slower execution times when using our disambiguation approaches. After a thorough inspection, we noticed that several slowdowns were caused by poorly tuned optimizations. Our techniques widen the reach of many optimizations, as seen in Figure 12, but the new version of a program may sometimes be slower than its original version. Figure 17 (a) shows a loop extracted from one of the tests in the `core` module. In this fragment, the developer manually unrolled the array copy operation in order to increase its performance. Without knowing if `src` and `dst` can point to the same place in memory, LLVM's memory optimization passes are not able to modify this code. When the library is compiled using our disambiguation techniques, LLVM becomes aware of the absence of aliasing and can perform more aggressive transformations. In particular, the loop in the Figure is subject to an optimization called *Superword-Level Parallelism*. This transformation combines different load and store operations in single vector instructions, generating the loop seen in Figure 17 (b). When testing in a development version of LLVM 3.7, however, the optimized version of the loop is around 1.3x slower than its original, manually unrolled, counterpart.

This anecdote illustrates the well-known fact that the code generated by compiler optimizations is not always faster than its original version. Nevertheless, we also have our share of blame in the slowdowns. When looking at the results for the Purely Static method for the `imgproc` module (leftmost column, second row of Figure 16), we see that a very small subset of the tests (13 out of 2146) were subject to slowdowns of around 2x. We attribute these slowdowns to extra calls to `dlsym`, a hook used by client applications to find the alias-free version of a function. All this said, we reinforce that these benchmarks constitute a very tiny portion of test cases. Moreover, approaches like BRI and FRI can be used to avoid such slowdowns. We did not perceive large overheads with the other approaches: as a simple experiment, we have added the `restrict` modifier to the arguments of functions in OpenCV, and this new version was less than 2% faster than the version that we produce with our dynamic tests.

```
for (int i = 0; i <= len - 4; i += 4 ) {
  int s0, s1;
  s0 = dst[i] + src[i];
  s1 = dst[i+1] + src[i+1];
  dst[i] = s0; dst[i+1] = s1;
  s0 = dst[i+2] + src[i+2];
  s1 = dst[i+3] + src[i+3];
  dst[i+2] = s0; dst[i+3] = s1;
}
```
(a)

```
for (int i = 0; i <= len - 4; i += 4) {
  dst[i:i+3] = dst[i:i+3] + src[i:i+3];
}
```
(b)

Figure 17. (a) Example of loop found in OpenCV's `core` benchmarks. (b) Version of the loop that LLVM -O3 produces.

5. Related Work

This paper describes three different techniques to disambiguate pointers. Pointer disambiguation is an important and well researched problem in computer science. Since Andersen's work [2], which represented one of the most important forays in this field, much has been accomplished. Currently, points-to analyses can be implemented very efficiently [12, 24] and very precisely [13, 29]. The purely static approach that we have described in Section 3.1 can use any of these types of alias analyses to disambiguate pointers. Thus, we are a client, not a challenger, of these previous works. In the same direction, there are several works describing algorithms that are clients of alias analyses. For instance, Chen *et al.* [6] have shown that finer granularity of pointer analysis can bring up to 15% of performance improvement to a typical compiler. Nevertheless, we are not aware of previous literature that tries to use static alias analyses to disambiguate pointers used as arguments of functions, as we do.

Compiler-related literature describes several different algorithms that use forward region inference to eliminate bound checks in memory accesses. As an example, Bodik *et al.* [5] have used a variation of forward region inference to remove guards around memory accesses in Java. Logozzo *et al.* [18] have defined *Pentagons*, a lattice that they have used to implement forward region inference. Nazaré *et al.* [20] have improved Pentagons with symbolic ranges, which is yet another way to implement forward region inference. The publicly available implementation of Nazaré *et al.* has been the starting point of the algorithm that we described in Section 3.2.1. It is important to clarify that, to the best of our knowledge, no previous work has used a forward region inference analysis to disambiguate pointers passed as the arguments of functions.

The body of work most closely related to ours concerns backward region inference, because this technique has, indeed, being used to generate checks that disambiguate pointers at runtime, albeit not pointers used as function arguments. To this effect, backward region inference has been used locally, at the level of basic blocks and at the level of loops. Alexandru Nicolau [22], Gallagher *et al.* [8], and Guo *et al.* [10] have proposed ways to disambiguate addresses used in load/store operations within basic blocks. Rus *et al.* [27] and Alves *et al.* [1] have designed backward techniques that disambiguate pointers used within nests of loops. Today, compilers such as gcc and LLVM use backward analysis to disambiguate affine memory accesses within innermost loops as a way to enable vectorization. Contrary to these previous works, our backward region inference performs this disambiguation at the function level. Algorithmically speaking, we use a global data-flow analysis that propagates information backwardly, and we apply a meet operation (maximum) at branches, i.e., program points that span multiple paths in the control flow graph. Previous literature uses local approaches, either restricted to the analysis of straight-line code or to the analysis of induction variables of loops. Due to this global scope, our technique requires, for instance, the use of escape analysis to ensure that pointers are not updated outside the function to be restricted.

These different scopes enable different compiler optimizations. For instance, let's assume that we were using pointer disambiguation to enable code parallelization, like Rus *et al.* [27] have done. In this case, Gallagher's local disambiguation would allow better instruction scheduling; hence, they would improve instruction level parallelism (ILP). Rus *et al.*'s technique would help the compiler to prove that different iterations of a loop are independent, thus creating more opportunities for the compiler to find data parallelism, in addition to enabling more aggressive ILP. Finally, by disambiguating pointers in the entire scope of a function, we can, for instance, give the compiler the opportunity to say that different - non-nested - loops could be executed in parallel. Following the parallelization jargon, we say that we are helping the compiler to identify more task parallelism, in addition of giving it more opportunities to find data and instruction-level parallelism.

6. Conclusion

This paper has presented three different techniques to disambiguate pointers used as arguments of functions. This disambiguation helps compilers to carry out more extensive optimizations, as we have demonstrated experimentally. One of our three techniques relies on the static alias analysis already available in mainstream compilers to perform pointer disambiguation. The others combine static bound inference (via forward or backward analysis) with code cloning; hence, extending the reach of pointer disambiguation. These techniques can be used in tandem, as the application of one does not hinder the application of the other. Whenever possible to disambiguate pointers statically, the purely static approach should be used, due to its low overhead. However, the other techniques, mainly the backward approach, can be used more extensively. To demonstrate the advantages of our approaches over techniques such as inlining, we have applied it onto OpenCV. In our experiments, this image processing library was compiled separately from its clients. In this case, inlining would not be able to be applied.

Future Works. Our current implementation of the three different restrictification techniques is mature enough to handle large code bases, as we have seen in Section 4.5. Yet, much work is still to be done towards the craft of a industrial-quality pointer disambiguation framework. Below we list three possible directions in which our work can be improved:

- Handle name collisions. Although unlikely, our current implementation still leaves room for this possibility. There are several ways to resolve the implicit link between the original function and the cloned function (hidden map, link time replacement, name convention, etc). We intend to explore some of these venues in the short-term.
- Test different cost models for judging when optimizations are profitable. As we have seen, restrictification sometimes leads to slower code, due to poorly tuned optimizations. Identifying and discarding unprofitable functions remains a problem to be solved. We have been using a static profiler, à la Wu & Larus [30], to solve this problem, but our results, so far, have not been fully satisfactory.

- Combine the purely static approach of Section 3.1 with more powerful alias analyses, e.g., flow sensitive, context sensitive, inter-procedural, etc. So far, we have only used our restrictifier together with the five default analyses available in the LLVM framework. This omission is due to our inability to find implementations of more comprehensive alias analyses in LLVM that can handle very large programs.

Final Thoughts. We conclude our paper with a final observation on the future of techniques such as those that we have introduced. Given the progress that we have recently observed towards the automatic extraction of data parallelism from programs, be it through vectorization [16], be it through SIMT parallelization [21], we believe that approaches such as ours, which decrease the amount of memory dependencies in programs, will have an importance that transcends classic compiler optimizations.

Acknowledgement. This work is supported by LG Electronics Brazil, under grant FUNDEP-22461. We thank Michael Frank, from LG, for all the insights he has provided during this research. Fernando Pereira is supported by FAPEMIG and CNPq.

References

[1] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. Pereira. Runtime pointer disambiguation. In *OOPSLA*. ACM, 2015.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[3] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[4] W. Blume and R. Eigenmann. Symbolic range propagation. In *IPPS*, pages 357–363, 1994.

[5] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[6] T. Chen, J. Lin, W.-C. Hsu, and P.-C. Yew. An empirical study on the granularity of pointer analysis in c programs. In *LCPC*, pages 157–171. Springer, 2005.

[7] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[8] D. Gallagher, W. Chen, S. Mahlke, J. Gyllenhaal, and W.-m. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *ASPLOS*, pages 183–193. ACM, 1994.

[9] T. Grosser, A. Groesllinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 04(22):X, 2012.

[10] B. Guo, Y. Wu, C. Wang, M. J. Bridges, G. Ottoni, N. Vachharajani, J. Chang, and D. I. August. Selective runtime memory disambiguation in a dynamic binary translator. In *CC*, pages 65–79. Springer, 2006.

[11] B. Hackett and A. Aiken. How is aliasing used in systems software? In *FSE*, pages 69–80. ACM, 2006.

[12] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299. ACM, 2007.

[13] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, pages 265–280, 2011.

[14] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61. ACM, 2001.

[15] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL*, pages 66–74. ACM, 1982.

[16] R. Karrenberg and S. Hack. Whole-function vectorization. In *CGO*, pages 141–150. IEEE, 2011.

[17] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[18] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.*, 75(9):796–807, 2010.

[19] C. Margiolas and M. F. P. O'Boyle. Portable and transparent host-device communication optimization for GPGPU environments. In *CGO*, pages 1–10. ACM, 2014.

[20] H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Q. Pereira. Validation of memory accesses through symbolic analyses. In *OOPSLA*, pages 791–809. ACM, 2014.

[21] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30: 56–69, 2010.

[22] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *Transactions on Computers*, 38(5):663–678, 1989.

[23] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECCOP*, pages 158–185. Springer-Verlag, 1998.

[24] F.M.Q. Pereira, and D. Berlin. Wave Propagation and Deep Propagation for Pointer Analysis. In *CGO*, pages 126–135. ACM, 2009.

[25] G. Piccoli, H. Nazaré, R. Rodrigues, C. Pousa, E. Borin, and F. Pereira. Compiler support for selective page migration in NUMA architectures. In *PACT*, pages 369–380. ACM, 2014.

[26] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *TOPLAS*, 27(2):185–235, 2005.

[27] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: Static and dynamic memory reference analysis. In *ICS*, pages 251–283. IEEE Computer Society, 2002.

[28] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, 1996.

[29] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.

[30] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *MICRO*. IEEE, 1994.

[31] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *PLDI*, pages 435–446. ACM, 2013.

[32] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL*, pages 197–208. ACM, 2008.