Runtime Pointer Disambiguation

Péricles Alves  
UFMG – Brazil  
periclesrafael@dcc.ufmg.br

Fabian Gruber  
INRIA – France  
fabian.gruber@inria.fr

Johannes Doerfert  
Saarland University – Germany  
doerfert@cs.uni-saarland.de

Alexandros Lamprineas  
INRIA – France  
labrinea@gmail.com

Tobias Grosser  
Dep. of Computer Science  
ETH Zurich – Switzerland  
tobias.grosser@inf.ethz.ch

Fabrice Rastello  
INRIA – France  
fabrice.rastello@inria.fr

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

Abstract

To optimize code effectively, compilers must deal with memory dependencies. However, the state-of-the-art heuristics available in the literature to track memory dependencies are inherently imprecise and computationally expensive. Consequently, the most advanced code transformations that compilers have today are ineffective when applied on real-world programs. The goal of this paper is to solve this conundrum through dynamic disambiguation of pointers. We provide different ways to determine at runtime when two memory locations can overlap. We then produce two versions of a code region: one that is aliasing-free - hence, easy to optimize - and another that is not. Our checks let us safely branch to the optimizable region. We have applied these ideas on Polly-LLVM, a loop optimizer built on top of the LLVM compilation infrastructure. Our experiments indicate that our method is precise, effective and useful: we can disambiguate every pair of pointer in the loop intensive PolyBench benchmark suite. The result of this precision is code quality: the binaries we generate are 10% faster than those that Polly-LLVM produces without our optimization, at the -O3 optimization level of LLVM.

Categories and Subject Descriptors  D - Software [D.3 Programming Languages]: D.3.4 Processors - Compilers

General Terms Languages, Performance, Experimentation

Keywords Alias analysis, dynamic guards, optimization

1. Introduction

A fundamental chain in the compilation pipeline is the resolution of dependencies between memory locations. Solving such dependencies enables better instruction scheduling as dependence information gives the compiler more freedom to reorder program statements. The ability to disambiguate memory locations is also essential to enable more specific optimizations, such as code vectorization [24], automatic parallelization [29, 47], scalar promotion [43] and several loop transformations, such as fusion, fission, reversal, interchanging, skewing and tiling [46, Ch.9]. Nevertheless, as important as this problem is, the research community has not yet solved dependence analyses satisfactorily.

To solve memory dependencies effectively, compilers need an accurate alias analysis which tells, for each pair of pointers, if they must, may or may-not dereference overlapping areas. However, solving these queries precisely is undecidable in face of procedure calls [27], and is NP-hard in the absence of this feature [22]. The goal of this paper is to improve this scenario by equipping compiler writers with a new dynamic pointer disambiguation technique that lets them address two challenges – the resolution of memory dependencies and the maintenance of this solution across program optimizations – in a precise and efficient way.
We present different ways to distinguish memory regions at runtime. Section 3 introduces a totally dynamic pointer disambiguation technique: we have augmented libc’s memory allocator with machinery to associate size information with pointers. We can query this meta-data at runtime, to check if two pointers may dereference areas that overlap. In Section 4 we discuss a hybrid pointer disambiguation technique. Given a program region where memory is accessed, we generate statically – the conditions that must be met so that memory locations do not overlap. We discuss two different ways to produce such checks: either we rely on the polyhedral model [8], or we rely on a symbolic range analysis [35]. We clone the regions guarded by dynamic checks, letting the compiler optimize them with the extra knowledge that they are alias-free. At runtime we can query these guards, and the result tells us when it is safe to jump into the optimized version of the code. The purely dynamic approach of Section 3, and the hybrid techniques of Section 4 can be used independently, or combined. The former can be used more extensively: it works for any pointer. The hybrid approaches have lower overhead, but they only work for pointers whose bounds can be determined statically.

The dynamic disambiguation of pointers has been used in specific contexts before, for instance, to enable automatic code parallelization [41]. Compilers for C/C++ and other languages also commonly use guards and code versioning to enable vectorization in the presence of pointer aliasing, but their use of versioning is commonly limited to what is needed to permit classical inner-loop vectorization. As we aim to enable the effective optimization of large and possibly complex loop nests, we propose new techniques that are powerful enough to handle larger program regions and possibly imperfectly nested, multi-level loop nests.

In Section 5 we validate these points. We have implemented our ideas in the LLVM [28] compilation infrastructure, and have used this implementation to improve the effectiveness of Polly-LLVM [17], a loop-optimizer used in tandem with that compiler. We have tested our implementation on PolyBench [40]. As we show in Section 5, the alias analyses currently available in the LLVM compiler, including a sub-cubic implementation of the Dyck-CFL-Reachability algorithm [48], are unable to disambiguate most of the memory access in PolyBench. Our checks, on the contrary, can do this job. Consequently, we produce code that is 10% faster than the code that Polly-LLVM would produce originally, at the -O3 optimization level of LLVM. The time that we take to generate the tests is negligible: we analyze the 30 benchmarks available in PolyBench in milliseconds.

2. Overview

We shall use the functions in Figure 1 to motivate the ideas that we discuss in this paper. Function array_sum_1 is a simple routine that sums up all the elements in an array of integers. Figure 1 contains another version of this routine: array_sum_2. This second implementation corresponds to the code that would be produced by a traditional compiler technique: loop invariant code motion, which consists in moving invariant computations outside loops. In our example, this computation is the load of s, which happens in line 3 of array_sum_1, and the store into acc, in line 4.

At first glance, these optimizations may seem easy venture for mainstream compilers. However, that is not the case. We have fed array_sum_1 to three different compilers: clang 3.4, gcc 4.8.2 and icc 15.0, and none of them, at their highest optimization level, has been able to produce array_sum_2. The culprit for this failure is aliasing. Not knowing if acc and s alias or not, the compiler must assume such a possibility. In face of aliasing, the store at line 4 of array_sum_1 could change the result of the load of s at line 3. The compiler cannot hoist the store into acc outside the loop either. If acc overlaps any address within array src, then functions array_sum_1 and array_sum_2 are not semantically equivalent. As a testimony of these limitations, it suffices to add the restrict keyword to the declaration of the arguments of function array_sum_1, and all compilers will be able to vectorize that code.

In Figure 1, the possibility of aliasing is unfortunate, for it hinders the compiler from applying very effective optimizations: function array_sum_2 is substantially faster than its original version. On an Intel Core i5 at 2.9GHz, using clang 3.4 -O3, array_sum_2 can be over 2.5× faster than array_sum_1. Such speedup is possible due to the extensive vectorization support that exists in the Intel hardware. Static pointer analyses [4, 39, 48] cannot enable this kind of optimization, because pointer overlapping may indeed happen in Figure 1. In other words, nothing prevents function array_sum_1 from receiving the same pointer as its actual parameters.

```
1 void array_sum_1(int *src, int *s, int *acc) {
2   int i;
3   for (i = 0; i < *s; i++) {
4     *acc += src[i];
5   }
6 }
```

```
1 void array_sum_2(int *src, int *s, int *acc) {
2   int i;
3   int N0 = *s;
4   int N1 = *acc;
5   for (i = 0; i < N0; i++) {
6     N1 += src[i];
7   }
8   *acc = N1;
9 }
```

Figure 1. An example (array_sum_1) in which potential aliasing hinders compiler optimizations such as loop invariant code motion as done in array_sum_2.

\footnote{We use source code to illustrate our techniques, but all our optimizations happen at the compiler’s intermediate representation level.}
Figure 2. Modified version of `array_sum_1` in which the compiler generates checks to disambiguate pointers.

A purely dynamic approach to pointer disambiguation.

The disambiguation of pointers such as those passed to `array_sum_1` as parameters requires runtime knowledge. A possible way to make such knowledge available consists in modifying the memory allocation system used by C. In this manner, we can tag different memory regions with different identifiers to disambiguate pointers that refer to distinct allocations. Binding meta-information to pointers is not a new technique. It has been used to secure programs written in C against out-of-bounds memory errors [2, 34, 36, 42]. Nevertheless, to the best of our knowledge, these techniques have not yet been used to disambiguate pointers in order to enable optimizations. Figure 2 illustrates such a use. In this example, we assume that `T(p)` returns a unique identifier of the allocated memory block pointed by `p`. By combining these runtime checks with code versioning, we can certify to the compiler that some program regions are alias-free. Some compilers provide support for this type of interaction. The LLVM intermediate representation, for instance, contains a `noalias` type modifier and corresponding metadata to model the absence of possible pointer aliasing.

The technique that we have discussed in Figure 2 is totally dynamic: it relies on a modified memory allocator to disambiguate pointers at runtime. Its main advantage is applicability: it gives us the opportunity to distinguish any pair of pointers that refer to different allocations. On the other hand, it may impose a runtime overhead on the program: metadata must be kept for each pointer, and depending on the implementation, `T(p)` may not be $O(1)$. We shall provide more details about this technique in Section 3. For now, we present an alternative to this approach, which combines checks produced statically with runtime knowledge.

A hybrid approach to pointer disambiguation. Figure 3 shows a different way to carry out pointer disambiguation. In this example, we try to infer - statically - conservative bounds on the memory referenced by the pointers that exist within a given code region. These bounds are written as functions of variables used in that area. The value of these variables may not be known statically. However, during the execution of the program, once its flow reaches the entry of the region analyzed, we can inspect these values, and use them to obtain a concrete estimate of the size of allocated memory. When analyzing function `array_sum_1` in Figure 1 we know that pointers `acc` and `s` are constants—even though their contents may not be, due to aliasing. Moreover, we know that the induction variable `i`, used to index `src`, ranges from 0 to `*s`. In the absence of aliasing, this upper limit, e.g., `*s`, is invariant. These observations give us the subsidies to generate the checks seen in lines 2-5 of Figure 3. In Section 4 we explain in deeper detail how we produce these checks without any intervention from the user.

3. Purely Dynamic Pointer Disambiguation

We have designed and implemented a memory allocator that lets us disambiguate pointers at runtime. To achieve this end, our allocator provides an efficient data-structure for querying the starting address of the heap allocation for arbitrary pointers. Disambiguation is enabled by the fact that this starting address, the base pointer, uniquely identifies each heap allocation. The identifier for pointers that do not point to a valid heap allocation is the null pointer, since that is never a valid heap address. Our memory allocator works for programs written in C/C++; nevertheless, similar techniques can be used in other programming languages that support pointer arithmetic, including assembly languages. We have replaced libc’s `malloc` routine with an implementation of our own. Our surrogate overwrites the implementations of `malloc`, `calloc`, `realloc`, `aligned_alloc`, `posix_memalign` and `free` with versions
that updates the data-structure used to query the heap allocation for a given pointer. This is implemented as a red-black tree [6] \( T \) that will provide us with logarithmic access to this meta-information. The additional data required for the tree is embedded in a header directly before the user visible allocation. A tree lookup gives us the address of the header and thus the unique identifier associated with the block of memory that \( p \) can dereference. The actual memory allocations are performed via the wrapped allocator. The sizes of all allocations are adjusted so they can fit the required header for the search tree.

Whenever memory is allocated through libc’s \texttt{malloc}, we add an entry to \( T \). An assignment such as \( p = \texttt{malloc}(N) \) will assign \( p \) to the range \([p, p + N]\) in our red-black tree. Notice that these intervals are bounded by arbitrary integers determined at runtime, not symbols. That is possible because the values of \( p \) and \( N \) are known at runtime. Now, assuming the program is correct, a pointer \( p_1 \) derived from another pointer \( p_0 \) (e.g. \( p_0 = p_1 + c, \ c \in \mathbb{N} \)) will cause \( T(p_0) = T(p_1) \). Contrary to previous work on program correctness [2, 34, 42], our goal is to enable optimizations, not to find bugs in programs.

In this section, and also in Section 4, we shall be using the program in Figure 4 to illustrate our ideas. The program in the figure has been adapted from an example used by Chabbi and Crummey [10] to illustrate performance bugs. Function \texttt{copy} moves the contents of array \( a \) or \( b \) to buffer \( r \). The program contains two statements that store into \( r \): the first at line 4, the second at line 6. If every cell of array \( b \) is non-zero, then both store instructions will be always performed during the program’s execution. The double occurrence of the store instruction cannot be optimized away, because if arrays \( b \) and \( r \) overlap, then the assignment at line 4 may change the outcome of the test that happens in line 5. In other words, due to aliasing, we cannot convert that code into \( \text{tmp} = b[i] \); \( r[i-1] = \text{tmp} != 0 \ ? \text{tmp} : a[i] \). Instead, we need to store \( r[i-1] \) before loading \( b[i] \), as the value of \( b[i] \) may be changed by this store.

We insert runtime checks at the beginning of the subparts of a program called Single-Entry-Single-Exit (SESE) regions [16]. SESE regions are sets of basic blocks with a single incoming edge and a single outgoing edge to the rest of the program’s CFG. We only insert tests at the beginning of SESE areas because, in this way, we are certain that any program flow entering that region will execute our tests. Figure 5 highlights the two regions that exist in our original example. Whenever we have a nested set of SESE blocks, we try to insert checks at the outermost sector in which all the pointers to be tested are \textit{alive}. We say that a variable \( v \) is \textit{live} at a program point \( p \) if there exists a path from \( p \) to a use of \( v \) that does not go across any redefinition of \( v \). In Figure 4 we have three pointers to disambiguate. All these pointers are alive at the entry of the outermost SESE sector within the function. Our checks will be inserted at that point. Our approach is actually not limited to SESE regions but can also support regions with multiple entries and exits, though this complicates code versioning. The limitation to SESE regions in our implementation stems from the fact that optimizations performed by Polly, the optimizer used in our experiments, are restricted to such regions.

Figure 6 shows our example program, after it has been instrumented with runtime checks. The original program con-
1 char* copy_message(char* src, char* dst) {
2   for (int i = 0; i < 4; i++) {
3      dst[i] = src[i];
4   }
5 }
6
7 struct message *msg = malloc(sizeof(struct message));
8 copy_message(&msg.s, &msg.d);

Figure 7. Program that illustrates over-approximation of our purely dynamic approach.

contains uses for three different pointers at the SESE region that we are instrumenting. Thus, we had to insert checks to disambigu$e$ three pairs of memory regions. For the sake of simplicity, we insert $O(N^2)$ checks at the beginning of a region that contains $N$ pointers. We avoid inserting some checks whenever static pointer analysis is able to disambiguate pairs of pointers at compilation time. As we show in Section 5, in practice our technique creates a small number of guards per region.

At runtime we can query $T$ for the unique identifier associated with $p$. The cost of such a query is logarithmic in the number of allocations that are live in the program at the time of the query. We know that two pointers, $p_0$ and $p_1$ cannot dereference overlapping memory regions if $T(p_0) \neq T(p_1)$. Notice that in the opposite case, e.g., $T(p_0) = T(p_1)$, it is still possible that $p_0$ and $p_1$ do not overlap, as the function copy_message, in Figure 7, illustrates. In this figure, the store and load at line 3 cannot dereference the same memory address. Nevertheless, we have that the base pointers of these stores will be associated with the same block of memory, which is passed to copy_message in line 9. We adopt this conservative approach for three reasons. First, it makes our implementation simpler. Second, situations such as that seen in Figure 7 are likely to be uncommon. Finally, even when they happen, it may still be possible that we use static approaches to avoid inserting dynamic guards. As an example, LLVM’s basic alias analysis is able to disambiguate the two pointers in line 9 of Figure 7.

Implementation details. Figure 8 shows a few implementation details of our memory allocator. The structure of each node can be seen in Figure 8 (a). The part (b) of the figure shows a snapshot of the tree. Each call to memory allocation routines such as malloc, calloc, realloc and aligned_alloc is intercepted and the requested size is augmented to make room for meta-data. The extra memory region where we store this meta information shall be called the node’s header. The actual pointer returned to the user references the next address after this header. Besides the pointers used by the tree itself, e.g., left and right, a node’s header also stores the size of the allocation and its alignment, as we show in Figure 8 (a). Hence, on a 64 bit system, by default, our implementation uses 32 bytes of data per alloc-
4. Hybrid Pointer Disambiguation

The purely dynamic approach that we have discussed in Section 3 is very precise; however, it has one shortcoming: queries such as \( T(a), T(b) \) or \( T(c) \) in lines 3-5 of Figure 6 may be expensive. In our implementation, each such query is logarithmic in the number of currently alive memory allocations. To avoid this cost, we have designed a hybrid pointer disambiguation technique, which does not require us to change the memory allocation library, and whose queries are \( O(1) \). In this section we describe such approach.

Figure 9 presents a general overview of the approach that we advocate in this paper. For each SESE region in a program we run the steps seen in Figure 9. We combine code versioning with runtime checks to give the compiler the opportunity to optimize certain program regions assuming the absence of aliasing. To reduce the overhead of the execution of the runtime checks, we may optionally prune some of these tests away with a profiler. In other words, if a pair of pointers is found to alias with some probability, we do not try to disambiguate those pointers dynamically.

In this section we explain how we generate the checks that we use to disambiguate pointers at runtime. To this end, we define, in Section 4.1 a core language, on top of which we shall work. We explain the actual computation of the disambiguation checks in Section 4.2. We answer more practical questions, such as where to insert the checks, in Section 4.3. We use the example first seen in Figure 4 to illustrate the notions introduced along these developments.

4.1 The Core Language

The technique we present in this paper requires us to compute symbolic bounds on expressions. We handle a small subset of the expressions typically found in programming languages, which is enough to describe the conditions that control most of the loops found in actual programs. Figure 10 defines the syntax of the programs that we handle. This language has only loops, assignments, dereferences and sequences of commands. In addition, we assume a few facts about loops. These assumptions simplify the analyses that we will describe further. Firstly, every loop in our language is controlled by a comparison such as \( v_i < v_n \), in which \( v_n \) is a loop invariant value. The generalization of the techniques that we discuss in this Section to the other three relational disjunctions, e.g., \( \leq, \geq \), is trivial. Such loops are called interval loops by Alves et al. [3] and although simple, they are very common. According to Alves et al., 67% of all the loops in SPEC CPU 2006 have this format. Secondly, we assume that \( v_i \) is a canonical induction variable. An induction variable is canonical if it is always incremented by one at each loop iteration. There are standard techniques to canonicalize affine induction variables used in interval loops [5]. An affine induction variable’s value is given by \( b + i \times s \), where \( b \) is its initial value, \( s \) is its increment, and \( i \) counts the number of iterations of the loop. Therefore, even assuming canonical induction variables, we are still able to handle all the 67% of interval loops found in SPEC CPU 2006.

Example 1 Figure 11 shows the program first seen in Figure 4 rewritten in our core language. We have eliminated the control flow and the negation inside the loop, as these constructs play no role in our analyses.

Because our core language is so uninvolved, we shall not provide a formal semantics to it; instead, we shall describe a few of its constructs informally. The command “\( \text{deref}(v_p, v_i, s_i) \)” denotes an array dereference: we are accessing the memory location of size \( s_i \) at address \( v_p + v_i \), where \( v_p \) is a base pointer, and \( v_i \) is an indexing expression, as defined in Figure 10. This access can denote either a load or a store operation. We distinguish sign and unsigned (\( u \)) operations. We are able to determine bounds of expressions involving signed and unsigned addition and multiplication, but we only deal with unsigned division. We shall use the
subscript \( u \) next to the division bar, to indicate that it is unsigned. We only allow multiplications and divisions of expressions by a constant. The operation “\( \text{trunc}(n, v) \)” converts the value carried by \( v \) into an \( n \)-bits integer by removing \( v \)'s most significant bits. Nothing happens if \( v \) already uses less than \( n \) bits. The operation “\( \text{sign}_{\text{ext}}(n, v) \)” does the opposite: it extends the binary representation of \( v \)'s value to an \( n \)-bits integer. This extension preserves \( v \)'s arithmetic sign. Similarly, “\( \text{zero}_{\text{ext}}(n, v) \)” does type extension, but fills the extra most-significant bits with zeros. For reasons that we clarify in Section 4.2, we work on programs in the Static Single Assignment (SSA) format \[13\]; thus, our core language defines \( \phi \)-functions. Induction variables are defined by \( \phi \)-functions.

4.2 Computing Symbolic Variable Bounds

If the same base pointer \( v_p \) is dereferenced \( n \) times within a loop nest, then let \( \text{deref}(v_p, v_i, s_i) \), \( 1 \leq i \leq n \) be each one of these accesses. To implement a bound check, we generate code to compute, for each \( v_i \), its lower and upper limits. We store the lower limit into a fresh variable \( v_{il} \) and the upper limit in another fresh variable \( v_{iu} \). We estimate the largest region \( M \) covered by the array via Equation 1. A comparison of two ranges will later allow us to prove non-aliasing inside the region.

\[
M = \max(v_{1u} + s_1, \ldots, v_{nu} + s_n) - \min(v_{1l}, \ldots, v_{nl}) - 1
\tag{1}
\]

If \( M_1 \) and \( M_2 \) are estimations for the regions dereferenced by two different pointers \( p_1 \) and \( p_2 \), then we say that they will not overlap if:

\[
p_1 + M_1 \leq p_2 \text{ or } p_2 + M_2 \leq p_1
\tag{2}
\]

To compute \( M_p \), we implemented and evaluated two approaches: (1) bound computation based on the polyhedral model, and (2) bound computation bootstrapped by symbolic range analysis. The latter can be used to disambiguate pointers for arbitrary regions without the need of a polyhedral analysis while the former requires the polyhedral description of the SESE region. One advantage of (2) is the fact that it can handle non-affine patterns of access while (1) cannot; one advantage of (1) (when relations are affine) is the extra precision: it relies on a relational analysis; hence, it is able to derive relations between variables, even if these variables are not linked syntactically in the program text. The program below illustrates this point:

```c
void foo(int *u, int *v, int N, int S) {
    int i;
    int j = S;
    for (i = 0; i < N; i++) { u[j] = v[i]; j++; }
}
```

Our polyhedral-based approach (Section 4.2.1) relies on the relations between integer points bounded by linear constraints. It can infer that index \( j \) is such that \( S \leq j \leq S + N \). On the other hand, the method based on range analysis, which we discuss in Section 4.2.2, tries to infer symbolic bounds to variables. This method assigns \([0, N - 1]\) to \( i \), and \([0, +\infty]\) to \( j \), because \( j \) is not bounded by the loop. Therefore, whereas the polyhedron-based approach of Section 4.2.1 can deal with the program above, the technique based on symbolic range analysis of Section 4.2.2 would not disambiguate pointers in this case.\(^2\)

4.2.1 Polyhedral Access Range Analysis

For the polyhedral access range analysis we require all \( v_i \) to be (piecewise) multidimensional affine functions \( f_{v_i} \) in values invariant in the region (parameters) and induction variables of loops surrounding deref \( [v_p, v_i, s_i] \). This excludes non-affine accesses, however the computation of access ranges is a compositional task and a general range analysis can be applied to non-affine expressions.

To get a symbolic range for all dynamic values of the access index \( v_i \) we first apply the access function \( f_{v_i} \) to the iteration domain of the access. Hence, we symbolically compute the access index for each dynamic instance of deref \( [v_p, v_i, s_i] \) in the multidimensional access space. The linearized minimal and maximal access indices of any access to a base pointer \( v_p \) are the bounds of the symbolic access range \( M_p \). To represent the access functions and the iteration domain we utilize the integer set library isl \[44\]. Using isl we also compute the lexicographical minimum and maximum of the multidimensional access space, hence the respective minimal and maximal access, to each base pointer. Note that for non-perfect loops, these bounds can be produced around regions where the accesses are defined by affine functions.

\(^2\)Our implementation of the symbolic range analysis presented in Section 4.2.2 still works precisely for this example, because it will use LLVM’s scalar-evolution analysis to infer bounds to variable \( j \). Scalar evolution is described in \[17, p.18\].
The bounds we obtain are again modeled as piecewise affine expressions. To translate them into actual program code, we pass these bounds to a polyhedral AST generator [19] which derives program code that can compute the array bounds at run-time. During program execution, the parameter values in the bounds are known and constant for one execution of the region. As a result, we obtain for each execution of a region precise minimal and maximal access bounds. The parametric bounds we obtain also let us identify accesses that can never be executed under the same parameter evaluation. Hence, if only one of two possible aliasing base pointers will be accessed for fixed but unknown parameter values, then there is no need to generate runtime alias checks.

As the polyhedral access range analysis is part of Polly it can cope with all inputs Polly can, except non-affine accesses. It is worth to note that not only statically sized multi-dimensional arrays, but also multi-dimensional arrays of parametric size [18] are fully supported.

### 4.2.2 Symbolic Range Analysis

The computation of $M$, in Equation 1, reads a few variables $v_{il}$. We can estimate the bounds of these variables using expressions produced by the code generator in Figure 12. The rules in Figure 12 create a mapping $B$. $B$ maps every variable $v_i$ used as a dereference index into a tuple $(I, v_{il}, v_{iu})$, such that $I$ is a set of assembly instructions, $v_{il}$ is the name of the variable that will hold $v_i$’s lower bound and $v_{iu}$ is the name of the variable that holds $v_i$’s upper bound. The assembly instructions $I$ will be used later to produce the runtime checks that we need to disambiguate pointers, as we shall explain in Section 4.3. Because our mapping $B$ is a function on variable names, we require every variable name to be unique in the program. The SSA representation [13], so common in modern compilers, gives us this property.

**Bootstrap the code generator with integer ranges:** When generating code, we must ensure that each instruction visited has all its operands already mapped by $B$. To meet this requirement we must (i) define mappings for loop invariant variables plus canonical induction variables, and (ii) visit instructions within the loop in a pre-order of the program’s dominance tree. A loop invariant variable is used within the loop, but defined outside it. To ensure (i), we “bootstrap” the code generator with range information. We assume the existence of an environment $R$ which maps variables to their symbolic range intervals. Constructing $R$ for programs in our core language is a standard procedure in the compiler literature. A more formal treatment on this subject is given by Nazaré et al. [35]. If $R(v_i) = [l, u]$, then before starting the traversal of the loop instructions, we perform the binding below for each variable $v_i$ that we need to bootstrap:

$$ B(v_i) \rightarrow (l, v_{il}, v_{iu}) $$

**Example 2** The loop in Figure 11 contains one invariant variable: $N$, and one canonical induction variable: $t_1$. A symbolic range analysis determines that $R(N) = [N, N]$, and that $B(t_1) = [0, N]$. Thus, we start the mapping $B$ with the following bindings: $B(v_i) = (v_{N1} = N; v_{N2} = N; v_{N1}, v_{N2})$ and $B(t_1) = (v_{t1} = 0; v_{t1} = N; v_{t1}, v_{t1})$.

As mentioned before, we visit the instructions in the loop body in a pre-order traversal of that body’s dominance tree. During this traversal, $G$ updates the mapping $B$. We let $B(v) \leftarrow t$ denote a new function $B'$, such that $B' = \lambda x. \text{if } x = v \text{ then } t \text{ else } B(x)$.

**Example 3** When analyzing the body of the loop seen in Figure 11, we visit instructions in the order $e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}$. Only variables $i_1$ and $t$ are used to index arrays. $B(i_1)$ has been initialized in the bootstrapping phase. Upon visiting
Algorithm 13: Let \( p_1 \) and \( p_2 \) be two pointers deregereferenced within the same Single-Entry-Single-Exit region, such that:

- \( p_1 \) is deregereferenced by a set of \( n \) instructions \( \text{deref}(p_1, v_{11}^1, v_{12}^1, \ldots, v_{1n}^1) \)
- \( p_2 \) is deregereferenced by a set of \( m \) instructions \( \text{deref}(p_1, v_{21}^1, v_{22}^1, \ldots, v_{2m}^1) \)
- \( B(v_{11}^j) = (I_{11}, v_{11}^j, v_{12}^j), 1 \leq j \leq n \)
- \( B(v_{21}^j) = (I_{21}, v_{21}^j, v_{22}^j), 1 \leq j \leq m \)

The following code sequence disambiguates \( p_1 \) and \( p_2 \) within the same Single-Entry-Single-Exit region, such that:

- \( v_{M1} = \max(v_{1u}^1 + s_1^1, \ldots, v_{nu}^1 + s_n^1) - \min(v_{hi}^1, \ldots, v_{ni}^1) - 1 \)
- \( v_{M2} = \max(v_{1u}^2 + s_1^2, \ldots, v_{nu}^2 + s_n^2) - \min(v_{hi}^2, \ldots, v_{ni}^2) - 1 \)
- if \( p_1 + v_{M1} \leq p_2 \) or \( p_2 + v_{M2} \leq p_1 \)
- then goto “\( p_1 \) and \( p_2 \) do not overlap”
- else goto “\( p_1 \) and \( p_2 \) overlap”

Figure 13. Generation of dynamic pointer checks.

\( \ell_5 : t = i_3 - 1, \text{we find that } B(t) \rightarrow ("v_{tl} = v_{1l} - 1; v_{tu} = v_{N u} - 1", v_{tl}, v_{tu}). \)

4.3 Code Generation

Algorithm 13 inserts runtime checks in a program to disambiguaitate pointers. The “goto” in line 5 executes whenever we can prove that two pointers, e.g., \( p_1 \) and \( p_2 \) do not overlap. If we cannot offer such a proof, then we use the jump in line 6. These checks are created for every pair of pointer used within a SESE region. Notice that they are generated statically, but their execution happens dynamically. Thus, by filling up the values in the tests with runtime values we can solve very complex “less-than” checks, even those involving max and min expressions.

Finding a place to insert checks. The loops in any program written in our core language (Figure 10) have the SESE property. In other words, these loops have a header block \( H \), and an exit block \( E \), such that any node outside the loop can reach any node inside only through \( H \). Similarly, nodes within the loop can reach nodes outside it only through \( E \). \( H \) dominates all the nodes in the loop, and \( E \) dominates all the nodes outside. In practice, not every loop in an actual assembly program has the SESE property; yet the proportion is high: we have found that all but one of all the loops in PolyBench are SESE. Thus, for practical reasons, we restrict our transformations to SESE loops.

We insert pointer checks in the pre-header of an outermost loop in a nest of loops. The pre-header of a loop is the single predecessor of its header. If a pre-header does not exist, i.e., the header has multiple predecessors, then we create one. We only create a check to disambiguate a pair of pointers \( p_1 \) and \( p_2 \) if all the symbolic expressions that the algorithm in Figure 13 needs for this check are available at the loop header. Available expressions, a classic compiler analysis [1, Ch.9], gives us this information.

Example 4 Continuing with our example, the disambiguation checks that we want to insert in Figure 11’s program require only the value of variable \( N \). The pre-header of the loop that we analyze contains labels \( \ell_1 \) and \( \ell_2 \). We insert the check after \( \ell_2 \). The code sequence that constitutes our check is given by:

\[
\begin{align*}
\text{if } & v_{N1} = N; v_{Nu} = N; v_{t1} = 1; v_{u1} = N; v_{tl} = v_{tl} - 1; v_{tu} = v_{tu} - 1; v_{t1} = v_{t1} - v_{tl}; v_{M2} = v_{tu} - v_{tl}; \text{if } r + v_{M1} > v_{tu} \text{ or } b + v_{M2} > v_{tl} \text{ goto } p_1, p_2 \text{ do not overlap else goto } p_1, p_2 \text{ overlap.}
\end{align*}
\]

The check that we insert for our running example contains many instructions that can be trivially optimized away by classic compiler transformations. For instance, the sequence in Example 4 computes \( v_{tu} - v_{tl} \) twice. One of these operations will be removed by Kennedy’s redundancy elimination [25], a technology readily available in modern compilers.

5. Experiments

To validate the experiments that we discuss in this paper, we have implemented them on top of the LLVM [28] compilation infrastructure and have used it together with Polly. Polly [17] is a loop optimizer built on top of LLVM. It implements typical transformations, such as tiling and loop fusion to improve the target program’s data locality. Several of these transformations are hindered by the lack of aliasing information. The techniques that we introduce in this paper provide such information.

The polyhedral-based approach discussed in Section 4.2.1 was implemented in Polly revision r236395, from May 3rd, 2015. The other two approaches, from Section 3 and Section 4.2.2 were implemented in Polly revision r216844, from August 31st, 2015. The reason for these two versions is pragmatic: the techniques were implemented by different people at different times, and the variation between them was too small to justify the work of recoding one of the implementations. Nevertheless, these versions of Polly apply different compiler optimizations on the code that they produce. Thus, our experiments let us know the speedup enabled by a pointer disambiguation technique on top of the original compiler, and the capacity of each technique to identify more optimizable regions. However, they should not be used to compare the runtime of the code produced by the three pointer disambiguation approaches, given that the set of optimizations applied on these programs differs.

We have chosen to test our approach on PolyBench 4.0 [40]. In addition to being widely adopted by the polyhedral community, PolyBench provides a number of options...
that can be enabled or disabled at the tester's will, such as:
different dataset sizes, automatic insertion of the restrict
keyword in pointer parameters, use of scalar values in loop
bounds, and a number of cache-related options. These options
give us a more controlled environment in which to test our runtime pointer disambiguation strategies. All the
numbers that we show have been obtained on an second
test our runtime pointer disambiguation strategies. All the
tions give us a more controlled environment in which to
test our runtime pointer disambiguation strategies. All the
numbers that we report are obtained on top of LLVM-O3, and are the average of six executions of
each benchmark.

Our goal, in this section, is to show that: (i) our tech-
nique is effective, i.e., it delivers runtime improvement on
top of LLVM + Polly-O3; (ii) the amount of code replication
that we cause is affordable, given the runtime gains that we
bring; and (iii) purely static techniques cannot do better than
we do. We address this third point in Section 5.1, and leave
the other two for Sections 5.2 and 5.3. A brief discussion
comparing our three different pointer disambiguation tech-
niques is presented in Section 5.4.

5.1 What Can LLVM’s Purely Static Approaches Do?
Traditionally, compilers use static alias analyses to disam-
biguate pointers. It is well-known in the literature that scal-
able implementations of such analyses are imprecise [33].
In this section we support this knowledge with data of our
own. We have performed an alias query for every pair of
pointers in our benchmarks, using the LLVM implementa-
tions of points-to analysis. LLVM uses five different tech-
niques to disambiguate pointers statically:

- **type-based**: C and C++ forbid aliasing between pointers
different types since C89/C++98. Thus, this analysis
flags pointers of different types as no-alias;
- **global-ref-based**: relies on the fact that globals that do
not have their address taken cannot alias anything;
- **basic**: uses a suite of simple rules, i.e. the stack does not
alias the heap or globals, for instance;
- **scalar-evolution-based**: which tries to place bounds on
arrays, and based on these bounds determines if they may
overlap each other or not.
- **Dyck-CFL-based**: implements a context-free language
(CFL) based context-insensitive alias analysis. This algo-

The precision of queries is cumulative: if any of these five
implementations is able to disambiguate two pointers, than
they are marked as no aliases. In our experiments, we use all
of these analysis in combination.

After applying these static analyses in PolyBench, we
found out that each kind of aliasing appears in the follow-
ing percentages for pairs of pointers: may alias = 62.78%,
must alias = 9.92%, no alias = 8.73% and partial alias =
18.57%. For three benchmarks: lu, floyd-warshall and
seidel=2d, the may-alias rate is lower than 20%. For six
others, it is above 70% and for the rest it is above 50%. The
percentages for no-alias and must-alias is remarkably low,
below 18% across all the benchmarks. Only 8.7% of queries
are no-alias. Hence, if we were to use a purely static ap-
proach to disambiguate pointers, then there would be still
over 62% of pairs whose relation we would not know.

Notice that the PolyBench kernels do not contain over-
lapping arrays. Yet, as discussed in the previous paragraphs,
typical implementations of static analyses – even in an in-
dustrial compiler – are not able to prove this fact. For Poly-
Bench, fully inlining all function calls would be sufficient
to give the missing context, but as PolyBench uses an allo-
action strategy that is defined in a different translation
unit, full inlining is only possible when running link-time
optimizations. For libraries, even optimizing at library link-
time is insufficient, as library functions can be called from
outside the library with parameters that may indeed alias.

Hence, optimizing under the assumption of possible aliasing
is often necessary to maintain correctness.Inlining function
calls or cloning and specializing functions to handle these
kind of problems has already been proposed by Metzger and
Stroud [32] or by Hall [21, Cp.5]. For cases with insuffi-
cient static information (or limited analysis power) main-
stream compilers such as LLVM, gcc and icc use code ver-
sioning with run-time alias checks for specific use cases,
e.g. to prove correctness of inner-loop vectorization, but they
commonly do not use function specialization.

Going beyond versioning of simple innermost loops, our
technique increases the number and the size of regions that
can be safely optimized by the compiler. These regions are
known, within Polly, as Static Control Parts (SCoP). A SCoP
is a Single-Entry-Single-Exit region, containing structured
control flow (a combination of for-loops and if-then-else
blocks) where control flow conditions, loop bounds and
memory offsets can be statically modelled as piecewise-
affine expressions in loop induction variables and SCoP-
irrelevant variables (parameters). The address of a memory
access is modelled as a combination of a base pointer and
a possibly multi-dimensional offset. For a region R to be a
SCoP, Polly requires that within R accesses with distinct
base pointers never point to the same memory location.

By proving the last condition more often, a more powerful
pointer analysis can: (i) increase the number of SCoPs,
by marking additional program regions as SCoPs, and (ii)
increase the size of SCoPs by, for instance, being able to
analyze all loops in a nest, instead of just the innermost one;
 hence, recognizing the whole region as a unique SCoP.

Figure 14 shows the relative increase in the number of
SCoPs triggered by our techniques. The purely dynamic
approach recognizes 1.31x more SCoPs than the baseline,
e.g., Polly + the suite of static analyses available in LLVM.
The hybrid approach of Section 4.2.2 recognizes 1.25x more
SCoPs than the baseline. Finally, the hybrid approach of Sec-
ction 4.2.1 recognizes 1.12x more SCoPs than the baseline.
Notice, however, that these numbers are not directly compa-
are the number of SCoPs optimized when using only LLVM's static alias analyses. We see that the polyhedral approach detects fewer SCoPs for some benchmarks, but these SCoPs are generally larger, as we further discuss in Figure 15.

To demonstrate this last fact, Figure 15 shows the increase in the number of loops within SCoPs that our three different pointer disambiguation approaches provide. The purely dynamic approach of Section 3, and the hybrid approach of Section 4.2.2, which is based on symbolic range analysis, identify 2.21x more loops within SCoPs than the baseline compiler. Again, the baseline used in this experiment is Polly plus the five static analyses available in LLVM. The polyhedron-based approach of Section 4.2.1 recognized 2.19x more loops within SCoPs than the baseline approach.

This increase is even more substantial when considering only kernel functions. A kernel, in the PolyBench jargon, is the function that contains the bulk of the computation that will be performed by a benchmark, thus being our main optimization target. The numbers that we report in Figure 15 include loops that are not inside a kernel, e.g., code in charge of initializing arrays or checking results. If we consider only kernels, both the symbolic and purely dynamic approaches increase the number of loops being optimized by 5.88x. The polyhedral-based technique, by its turn, recognizes 5.56x more loops. This small difference between approaches is due to the use of different compiler versions, as explained in the beginning of this section. We note that all three approaches were able to correctly recognize all analyzable loops in the kernels of the PolyBench suite. By “analyzable” we mean the loops that Polly could handle if it had access to perfect aliasing information.

**Conclusion:** to disambiguate pointers effectively, compilers must either use more expensive analyses, which are usually not available in their tool box, or must use the more dynamic approaches that we advocate in this paper.

5.2 Hybrid Approaches

In this section, we present data that shows that the two hybrid approaches of Section 4 are effective and useful. Our goal is to demonstrate that these techniques can bring non-trivial speedup on top of highly optimized code, at the expense of an affordable increase in code size.

5.2.1 Overhead of Dynamic Checks

Our dynamic checks incur negligible overhead on the PolyBench programs, as Figure 16 reports. To measure this overhead, we have modified Polly to assume absence of aliasing in the source programs. We do it by adding the “restrict” keyword to the arguments of the functions. This keyword, available since C99, tells the compiler that a pointer does not share memory accesses with aliases in the scope of the function in which that pointer is declared. If a function argument $p$ is marked as “restrict”, it is still possible to derive new pointers out of it, such as $p_0 = p + 1$. Thus, by using this keyword, the programmer signs a contract with the compiler, specifying that the memory pointed by $p$ can only be accessed through it, or via one of its derived pointers.

PolyBench comes with an option to enable the “restrict” modifier for parameters of pointer type. In this case, Polly + LLVM-03 optimize the same regions that we do, but without having to resort to dynamic bound checks and code duplication. Hence, this experiment lets us check the overhead of our dynamic checks. For a few benchmarks, our hybrid techniques are not as efficient as the “restrict” keyword. In cases like gesummv and ludcmp, we need to disambiguate a large number of pairs of pointers in the entry of regions that the compiler is not able to aggressively optimize, even though our dynamic checks succeed at runtime. For instance, the polyhedron-based approach of Section 4.2.1 inserts for the kernel function of ludcmp a runtime check to disambiguate four base pointers (two read-only, two read-write). This check performs for each of the twelve pointer pairs with least one read-write base pointer two comparisons as well as a boolean or and also requires eleven boolean and operations to combine the conditions between pointer-pairs. For gramschmidt and 3mm, the symbolic approach can disambiguate more pointers than its polyhedral-based counterpart. In doitgen, the opposite happens, i.e., the polyhedral version can analyze more pointer pairs. This observation jus-
Figure 15. Relative increase in the number of loops within Static Control Parts when using our disambiguation approaches. The base values are the number of loops optimized when using only LLVM’s static alias analyses. The three approaches provide the same increase in almost all benchmarks.

Figure 16. Execution time overhead of our hybrid approaches when compared to the use of “restrict” in Polly. Our biggest overhead was in granschmidt, which executes in 5.19 seconds in its “restrict” version. When running on Polly augmented with our polyhedral approach, this number grows to 7.86 seconds, being 1.5x slower, as the chart shows.

Figure 17. Execution time improvement of our hybrid techniques (section 4) when applied to LLVM-O3. Our biggest speedup was in bicg, which is executed in 150 milliseconds by LLVM-O3. Both of our hybrid approaches reduce this number to 56 milliseconds, making it 2.7x faster, as the Figure shows.

On average we see that the runtime of the polyhedral-based approach (Section 4.2.1) is 2.8% slower and the runtime of the symbolic range analysis based approach (Section 4.2.2) is 1.6% slower than using “restrict”.

Conclusion:  
(i) our dynamic tests produce results that are very close to perfect alias information.  
(ii) these tests impose a very low overhead on the running program, and do not impact its runtime, even when never taken.

5.2.2 Speedup

Our ultimate goal is to speed up programs. We achieve this goal by giving the compiler the opportunity to run more aggressive optimizations on said programs. There are many different ways to test the benefits of our analysis. Compilers commonly provide many options to select the transformations that are run on a given program and LLVM is no exception. We have performed an extensive search in this space, and report findings in this section. First, Figure 17 shows...
how each of our hybrid approaches improves the runtime of LLVM-O3. On average, the polyhedral approach (Section 4.2.1) improves LLVM-O3 by 8.7%. The range-based approach (Section 4.2.2) gives us a speedup of 6.5%. It is fundamental to emphasize that these numbers have been obtained on top of the highest optimization level of LLVM, one of the most well-engineered open source compilers available nowadays.

It is well known that the order in which optimizations are applied on a program can influence the final runtime of its binary code [26]. The numbers reported in Figure 17 use the default optimization order of LLVM -O3. Thus, we can go beyond the speedups seen in that figure by changing the order in which some of the LLVM optimizations run. For instance, if we follow the insertion of our checks with a round of loop invariant code motion, then we can hoist more load and store operations outside loops. This strategy lets us convert array_sum_1 into array_sum_3 in Figure 1, for instance. Figure 18 shows the runtime numbers that we obtain in this way, again, comparing against LLVM-O3. The only change that we performed in this case was to run loop invariant code motion right after inserting the disambiguation checks. This order is the same for all the benchmarks. We have been able to achieve a speed up of 11.3% across all the benchmarks. We could use only the approach of Section 4.2.2 in this experiment, because the polyhedral based approach (Section 4.2.1) cannot analyze loops that are bound by values stored in memory – these bounds are not considered affine by the Polly-LLVM framework.

We have also evaluated the runtime benefit of our approaches when applied on top of the combined optimization sequence of LLVM-O3 and Polly [17]. The optimizations obtained by Polly’s scheduling optimizer and possibly enabled by our run-time checks can include combinations of classical loop transformations, such as strip mining, tiling, fission, fusion and interchange, but also transformations that are difficult to express as a set of classical loop transformations. Figure 19 shows these results. The range-based approach of Section 4.2.2 gave us a speedup of 7.2%, and the polyhedral-based approach of Section 4.2.1 gave us a speedup of 10.6%. The speedups we report are for the default compile-time options of PolyBench and unnecessarily low. PolyBench uses by default parametric loop bounds and fixed-sizes arrays. This mismatch requires Polly to respect data-dependences that are only relevant when the parametric loop bounds have a value that is larger than the corresponding fixed size array dimensions. In practice, the loop bounds and the array dimensions always have identical (or clearly related) sizes. When compiling PolyBench with scalar loop bounds (~POLYBENCH_USE_SCALAR_LB) no spurious data-dependences hinder Polly optimizations and we see for the polyhedral-based approach a speedup of 18.5% comparing LLVM-O3 and Polly with run-time alias checks against LLVM-O3 and Polly without run-time alias checks.¹

As Figure 19 shows, we have observed slowdowns in a few benchmarks: trisolv, gemm, atax, etc. Several of these slowdowns are caused by Polly “optimizing” additional or larger code regions that only become amenable for optimizations when using our new alias-checking techniques. The transformations Polly applies are obtained through (a slightly modified) reimplementation of the Pluto scheduling optimizer [8]. Pluto optimizes the schedule of a SCoP by constructing an ILP problem that minimizes data-dependences while maximizing tilability and parallelism. In this process Pluto only considers data-dependences, but does not consider spatial locality. As a result, the Pluto scheduling optimizer may choose schedules that are better according to the criteria Pluto optimizes for, but which reduce spatial locality and consequently performance. The original implementation of Pluto addresses this problem by applying a set of post-scheduling optimizations that focus on spatial locality. The (current) lack of these optimizations in Polly seems to be the main reason for non-optimal code transformations. The choice of optimal transformations is left to the compiler. Even if we only use LLVM optimizations, but exclude Polly’s, it is still possible to observe slowdowns in some benchmarks. For instance, Figure 18 shows slowdowns in two benchmarks - jacobi-2d and trisolv. We account this behavior to the choice and ordering of optimizations. Finding an optimal spot in this space is still an open problem [26].

**Conclusion:** from the experiments in this section we draw three conclusions. First, our hybrid pointer disambiguation approach is able to bring a non-negligible improvement on the runtime of highly optimized code. Second, our hybrid approach based on symbolic range analysis enables extensive hoisting of loop invariant code. Third, the hybrid disambiguation checks have been able to reveal new opportunities for code optimization and tuning, because they allowed Polly to optimize more code regions than before.

5.2.3 Code Size Expansion

Figure 20 shows how much our technique increases the size of binaries. On average, the range-based hybrid approach of Section 4.2.2 caused a code size expansion of 35.3%. The polyhedral-based approach of Section 4.2.1 increased code size by 32.1%. This expansion is due to (i) the duplication of code, and (ii) the checks that we create to disambiguate pointers at runtime. We have not created tests for nussinov,
Figure 18. Runtime improvement of our symbolic approach (Sec. 4.2.2) over LLVM-O3, when followed by a round of loop invariant code motion. As in Figure 17, our biggest speedup was in bicg, being 2.7x faster.

Figure 19. Execution time improvement of our hybrid methods over Polly, running on LLVM-O3. Our best result can be observed in covariance, which is executed in 4.29 seconds by Polly. Our polyhedral approach lowers this number to 1.29 seconds, being 3.3x faster.

Figure 20. Code size expansion due to our optimization. In total, our executables have 10,327 instructions, against 6,570 without our optimization, an increase of 57%.

and seidel-2d, because these benchmarks manipulate only one vector. Our checks are necessary only if the optimizable region contains two or more different array accesses. Thus, for these two benchmarks there was no code-size expansion.

Conclusion: our technique increase code size by a factor of about 30-35%.

5.2.4 On the Precision of Memory Access Estimates

The disambiguation checks of our symbolic hybrid approach (Section 4.2.2) are not always precise, but in some cases over-estimate the size of the accessed memory. We are interested in knowing how precise this over-estimation is. To this end, we have compared our estimates of the lower and upper bounds of the range of accessed array addresses against the results that we obtain using the dynamic approach of Section 3. We use our memory allocator in the instrumented programs to check size estimates determined statically against actual memory size. In PolyBench the estimation is perfectly accurate as the accessed regions can be described lossless by both our static analyses.

Conclusion: our technique estimates the accessed memory regions with high accuracy in regular code.
5.3 The Purely Dynamic Approach

Figure 21 shows a runtime comparison between the purely dynamic approach, discussed in Section 3, and the hybrid approach with the symbolic range tests seen in Section 4.2.2. We show results for PolyBench. Overall, the runtime of the benchmarks was very similar. We have observed that the hybrid approach is 6% faster. This number is the geometric mean of the execution times. The hybrid approach is faster because, in general, its dynamic checks have a faster runtime, and it does not impose memory allocation and deallocation overheads. Whereas the purely dynamic checks are \(O(\ln n)\), \(n\) being the number of memory allocations currently live in the program, the checks inserted by the hybrid approach execute in \(O(1)\). Nevertheless, this constant can be high, because these guards may be formed by the combination of several complex arithmetic expressions. This observation explains why the purely dynamic methodology yields faster runtimes in some benchmarks. Additionally, the hybrid approach has been used to disambiguate more pointers pairs. This happened whenever we could not infer - statically - symbolic bounds for some pointers.

We have tried also to combine the purely dynamic approach and the hybrid approach of Section 4.2.2. In this case, whenever the hybrid approach could not generate a test to disambiguate a pointer pair, we would resort to the purely dynamic disambiguation. Figure 21 also shows these results. This combination has produced slight improvement on the runtime of the purely dynamic approach. We believe that we could not observe a larger speedup because a substantial part of the cost of this technique is due to the memory allocation overhead. For instance, we tried the purely dynamic approach in SPEC CPU 2006’s 401.bzip2, a benchmark that performs several calls to the memory allocation routine. In this allocation-heavy benchmark, the purely dynamic approach slowed the program down by 29%. Notice, however, that the purely dynamic technique does not depend on a particular implementation of a memory allocator. Thus, we believe that an interesting line of future research is to check if, and by how much, other implementations, such as SoftBounds [34], for instance, could reduce this overhead.

5.4 Discussion

In this paper, we have discussed very different techniques to disambiguate pointers at runtime: the purely dynamic approach of Section 3, and the hybrid approach of Section 4. Additionally, we have used two different methods to generate the checks used in the hybrid approach, the polyhedrons of Section 4.2.1, and the symbolic range analysis of Section 4.2.2. All these techniques have advantages and shortcomings, and, to illustrate them, we shall rely on the programs seen in Figure 22.

The purely dynamic approach is more applicable, handling sparse data-structures such as linked-lists, for instance. Furthermore, it does not depend on the ability of a static analyzer to infer bounds for arrays. For example, only the purely dynamic approach lets us produce tests for function \(f0\) in Figure 22. The hybrid approaches would fail in this case because the limits of the array \(src\) are not explicit in the loop. The purely dynamic technique can also handle loops containing function calls and non-affine array accesses, which are not addressed by the other strategies.

The main drawback of the purely dynamic method is the query time. Our implementation of it uses a balanced tree to store meta-information associated with each chunk of memory allocated during the execution of a program. In principle, we could use other data-structures to retrieve the meta-data associated with the blocks of allocated memory. We chose to implement our memory allocator with a red-black tree because it is relatively easy to implement and it offers us reliable access times. Furthermore, this data-structure has been already used for similar purposes in the literature [31]. Nevertheless, for memory intensive programs, the red-black tree’s search time might bring in a non-negligible overhead. In its current form the purely dynamic approach is completely ignorant of the underlying memory it wraps, as long as it provides a \(libc\) like interface. While this makes our implementation trivially portable between different platforms and allocators, it also means that we have to duplicate some metadata that most modern allocators already keep. In the future we plan to modify a high performance allocator to
Figure 22. Examples of programs that illustrate advantages and disadvantages of our different runtime pointer disambiguation techniques.

remove the overhead of tree queries for a large part of all allocations and significantly reduce the space overhead.

The hybrid approaches of Section 4 have lower overhead and work independently of the program's execution environment. Our experiments indicate that inferring bounds for pointers statically should be preferred whenever possible. There are also programs that the hybrid approaches can disambiguate, but the dynamic technique cannot. For instance, the latter method will not be able to disambiguate the accesses to u[i] and v[j] in function f1 in Figure 22 (line 6), because these pointers dereference addresses within the same memory region.

In our experiments, we have not found programs that could be analyzed differently by one of the two hybrid approaches of Section 4, but not for the other. Nevertheless, we can create such examples by hand. For instance, both methods are able to analyze function f2 in Figure 22; however, the polyhedron-based one, from Section 4.2.1, will be able to infer – statically – that the accesses u[i] and v[i] are independent, because for a given set of parameters either line 5 or line 7, but never both, can be run during the execution of the kernel. On the other hand, this technique is not able to produce tests for the program in function f3 in Figure 22. The culprit, in this case, is the fact that the expression i * j is non-affine. The approach based on symbolic range analysis can handle this example, although this feature is not yet available in our implementation.

6. Related Work

This paper touches different facets of the pointer disambiguation problem; hence, it is related to several previous work. Nevertheless, we are not aware of work that incorporates all the elements that we have introduced.

Profiling based alias analysis. One of the inspirations to this work is the study performed by Mock et al. [33], in which they have shown that the pointer information produced by state-of-the-art static alias analyses is markedly worse than the actual behavior observed at runtime. This work has also inspired other groups [14, 20], which, differently from us, use profiling to determine the probability that aliasing happens in practice. In this case, a recovery mechanism is necessary to preserve the semantics of the program in face of actual aliasing. Lin et al. [30] provide some examples of different ways to recover from wrong speculation.

This kind of speculative modus operandi has seen use in several other works [9, 14, 15, 23], which have in common the fact that a profiler is used to derive alias information. This information is then made available to the program’s runtime environment. For instance, Huang et al. [23] have applied speculative pointer disambiguation to solve dependencies in the context of a VLIW machine. They assume that there is no dependence between some memory references that overlap with low probability. If this dependence is observed at runtime, then they rollback the execution. They also mention that an alternative would be to branch to a different version of the code, as we do. In Huang’s case, dependencies are resolved in hardware. Similar approaches, implemented at the software level, have been proposed by Fernandez et al. [15] and Silva et al. [14]. Chen et al. [11] have designed several enhancements on the basic speculative methodology to make it faster and more accurate.

All these previous efforts apply optimizations assuming that pointers do not alias at runtime: profiling enables code optimizations whenever it indicates that aliasing happens with low probability. However, our approach is not speculative. Instead of speculating, we replicate code, and use the runtime checks to decide where to branch. Furthermore, contrary to us, none of these previous works tries to generate disambiguation checks statically.
Hybrid Pointer Disambiguation for Parallelism. Rus et al. [41] generate statically checks that disambiguate pointers at runtime. This technique to disambiguate pointers differs from our work because the test that Rus et al. generate is a necessary condition to enable loop parallelization. This constraint may lead to complex dynamic checks that need to be simplified. Oancea and Rauchwerger [37] have proposed several ways to perform this simplification within a framework of logical inference rules. They use, for instance, variable elimination in a Fourier-Motzkin style. Posterior work [38] has adopted similar ideas to disambiguate induction variables that can be converted to a closed form. The method that we advocate in this paper has the primary goal of overcoming the limitations of the static analyses commonly applied on low-level intermediate program representations. We use a lightweight approach that allows us to: (i) take advantage of existing compiler infrastructure (e.g. relational analysis); (ii) let the compiler make the decision of which optimization to perform (e.g. among all possible polyhedral transformations); (iii) generate simpler run-time checks; and (iv) improve the practicality of the static alias analyses approaches which mainstream compilers use.

Memory Aliasing in research compilers. Research compilers such as Pluto [8] or PPCG [45] ignore memory aliasing when applying state-of-the-art loop optimizations for cache locality, parallelism, vectorization or accelerator usage. As a result, their transformations often change the program behavior in case aliasing actually happens. Thus, in practice they may cause general code transformation to be unsound for the vast majority of existing C/C++ codes. Outside of a user controlled benchmark environment, pointer disambiguation techniques such as the ones presented in this paper are required to maintain program correctness and enable the use of advanced loop optimizations.

Memory aliasing in static compilers. Static compilers such as icc, IBM XL, gcc or clang all support some kind of loop versioning to enable optimizations such as vectorization in the presence of pointer aliasing. For closed source compilers it is difficult to understand how general their code versioning support indeed is, but the experiment in Figure 1 suggest that at least icc, even for simple examples, does not always apply loop versioning. Looking at the source code of gcc and clang, we can confirm that even their latest development versions only allow versioning of simple, innermost loops, as it is required for loop vectorization and other transformations that focus on innermost loops. Pointer disambiguation for complex loop nest or fully dynamic pointer disambiguation approaches, as we presented them, have not been used by the static compilers that we analyzed.

7. Conclusion
This paper has presented a suite of techniques to produce checks that let us disambiguate pointers at runtime. Unimpied by aliasing constraints, the compiler has more freedom to perform aggressive optimizations in the cloned region. We have tested our prototype in PolyBench, and have been able to observe an average speedup of 10% on top of LLVM-O3. This performance boost is non-negligible, as it has been obtained on top of one of the most well-engineered compilers in the world. Nevertheless, we are not proposing new compiler optimizations; instead, we are proposing a way to enable them. Our techniques are more general than previous work, and can be applied onto complex data-structures, in addition to arrays. The present work let us conclude that this combination of cloning plus dynamic disambiguation of pointers is an effective way to make compiler optimizations more practical.

Software The hybrid approach discussed in Section 4.2.1 has been available in Polly since June of 2014.

Acknowledgements
We thank the OOPSLA referees for many constructive comments and suggestions. The Brazilian side of this cooperation has been sponsored by CNPq, FAPEMIG, CAPES and LG Electronics. Tobias Grosser got partial funding provided by Swissuniversities through the Platform for Advanced Computing Initiative (PASC). Johannes Doerfert is funded by the International Max Planck Research School for Computer Science (IMPRS-CS).

References


