View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by HAL-ENS-LYON

Do we still need new Alias Analyses?

Maroua Maalej, Laure Gonnord

▶ **To cite this version:**

Maroua Maalej, Laure Gonnord. Do we still need new Alias Analyses?. [Research Report] RR-8812, Université Lyon Claude Bernard / Laboratoire d'Informatique du Parallélisme. 2015. <hal-01228581>

HAL Id: hal-01228581

https://hal.inria.fr/hal-01228581

Submitted on 13 Nov 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Do we still need new Alias Analyses?

Maroua MAALEJ, Laure GONNORD

RESEARCH REPORT
N° 8812
November 2015
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8812--FR+ENG

Do we still need new Alias Analyses?

Maroua MAALEJ[*†‡], Laure GONNORD [† ‡]

Project-Team ROMA

Research Report n° 8812 — November 2015 — 18 pages

Abstract: Alias analysis is one of the most used techniques that aim to optimize languages with pointers. It is no surprise that this topic has received much attention in many fields including compilation and program verification. However, despite all this attention, a deep study of its state-of-the art shows that it is not handled satisfactorily, by far. In this paper we demonstrate that, although many approaches have been proposed, we still cannot rely on alias analysis inside compilers. We illustrate our statement with an overview of the capabilities of state-of-the-art compilers.

Key-words: Static Analysis, Alias Analysis, Optimization, Compilers

[*] This work was supported by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

[†] Laboratoire de l'Informatique du Parallélisme, 46 Allée d'Italie, Lyon, France

[‡] Université Lyon 1, 43 Boulevard du 11 Novembre, Lyon, France

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

A-t-on besoin de nouvelles analyses d'alias ?

Résumé : L'analyse d'alias est l'une des techniques les plus utilisées pour optimiser les langages avec pointeurs. C'est donc sans surprise que beaucoup d'approches et d'analyses ont été proposées, à des fins d'optimisation et de vérification de programmes. Cependant, une étude approfondie de l'état de l'art montre que les résultats établis dans le domaine de l'analyse de pointeur sont loin d'être satisfaisants. Dans ce rapport nous montrons qu'en dépit de toutes les approches proposées, nous ne pouvons toujours pas compter sur les analyses d'alias dans les compilateurs. Nous illustrons notre propos par une vue d'ensemble des analyses de pointeurs dans les compilateurs actuels.

Mots-clés : Analyses statiques, Analyses d'alias, Optimisation, Compilateurs

1 Introduction

Low-level languages like C are widely used to write embedded software and manage hardware in order to make smaller or more efficient programs. They enable the programmer to finely adapt their code to their hardware, so that it runs faster with minimum memory usage. Today variants of the C language are still used to program on multicore or GPU machines.

However, since these languages lack abstractions and give the programmer ability to manipulate memory, they are hard to program and are vulnerable to errors. To overcome this problem and to help the user produce correct and optimized code, analysis techniques have been widely investigated in both verification [20] and compilation fields [9].

In this paper we will mainly discuss the field of *static* alias analyses also known as pointer analyses or points-to analyses that have been shown to be undecidable [32]. This kind of analysis attempts to discover, statically (at compile-time), the possible values of a pointer at runtime. Typically, this information is used to optimize programs or prove their correctness, which can be done by eliminating dead code and null pointers, parallelizing, rescheduling, etc.

Many approaches that trade precision for speed, speed for precision, or even attempt to balance between both, have been proposed and improved in time and space. However, we shall demonstrate that despite all the attention this topic has received, today's compilers still rely on rudimentary and imprecise alias analyses, which provide conservative information insufficient to generate efficient code.

The contributions of this paper are the following:

- an extensive study of the state-of-the-art approaches for alias analyses (Section 2) and their use for compiler optimizations (Section 3);
- an experimental evaluation of strengths and weaknesses of current production C compilers (Section 4);
- a discussion of the kind of information that would be useful for designing new pointer analyses for compilers (Section 5).

2 Alias analysis: context and related work

In this section, we introduce the problem of alias analyses and make an overview of existing approaches. We show that the trade-off between precision and scalability is difficult to find, especially when it comes to compiler optimization.

2.1 Problem statement

Alias analysis (or "pointer analysis") is a technique widely used to perform, on languages with pointers, transformations and optimizations such as parallelization [17] and code motion. It is also useful for program verification such as detecting bugs due to array out of bounds accesses [26, 37] and integer overflow [15, 38]. Some of alias analyses are performed for security and performance enforcement [5, 28], deadlock or race detection [33, 14], memory errors detection [7], etc. Such an analysis aims to disambiguate memory locations pointed by pointer variables to find out whether they dereference overlapping regions or not.

At runtime, a pair of pointers may have the following behaviors, depicted in Figure 1:

- No Alias: p_1 and p_2 are disjoint. They do not reference overlapping memory regions. This is the favorable case toward optimizations such as rescheduling and code motion, as we will see in Section 3.
- Alias : p_1 and p_2 reference regions that overlap. In this case, we can distinguish between *partial alias* where p_1 and p_2 overlap in some way but do not start at the same address and *must alias* where p_1 and p_2 alias and start at the same address.

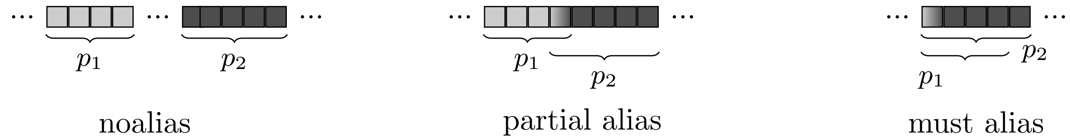

Figure 1: Concrete representation of memory regions in case of no, partial and must aliasing. Each □ represents a memory slot.

The literature of alias analysis is abundant and much work have been done in the last few decades. For a quite exhaustive bibliography, see for instance [12] and [21]. In this paper, we shall be limited to related work that are the most relevant for our study of interest.

2.2 Static analyses of pointers

Historically, starting from Andersen's work [1], *static analysis* for pointers has received much attention [12, 9, 31, 37]. The current state-of-the-art shows numerous analysis approaches.

Constraint-based pointer analyses. Andersen [1] and Steensgaard [36] propose to compute *points-to* sets (the set of variables that a given pointer can reference) by means of solving constraint systems. In these two algorithms, a set of constraints is derived from program statements involving assignments, loads and stores. For instance, for any assignment `a=b`, the points-to set of b is included in the points-to set of a (for Andersen), or the two points-to sets are equal (for Steensgaard). Solving this set of constraints is done with efficient graph algorithms. The output of these algorithms is not a comparison between pointers but a points-to set for each pointer which would enable further use by means of *alias queries.* The main drawback of these

algorithms is that they are not precise enough to properly handle pointer arithmetic: after the assignment `py=px+4`, *px* and *py* are inside the same points-to set.

Pointer analyses by Model Checking. In [20], alias analysis information is collected by means of model checking techniques. To do so, the relevant features of the C code is represented as a finite state automaton and the alias query is translated into a property verified as a reachability problem. Although the performance in terms of verification time and in scalability seems promising, we believe that such kinds of program transformation cannot be embedded in compilers, in particular because tracking back information from results obtained after transformation is tedious and error prone.

Shape Analyses. *Shape analysis* [35, 4], also known as *storage analysis*, is a kind of static alias analysis mostly used to verify programs that perform destructive updating on dynamically allocated storage. Its goal is to give, for each program point, a finite and conservative characterization of the possible *shapes* of program data structures allocated in the heap. A Static Shape Graph (SSG), which is a finite, directed and labeled graph, is computed at each program point to approximate stores done during execution until this point. The SSG, where nodes are structures and edges are pointers, is used to check memory safety (looks for memory leaks), and to find out of bounds accesses and discover properties of linked data structures dynamically allocated. Graphs are used to model program stores. Much attention has been paid to this technique in the last decade. However, the effort has mainly been done on precision issues. The state-of-the-art [18] is able to express complex properties about graph structures, but the current version of the analyzer only deals with tiny programs (20 to 30 lines of code). However, this kind of analyses are too greedy in terms of memory resources to be embedded in state-of-the-art compilers.

An example of application to verification: array out-of-bounds check. A violation of memory safety in C leads to undefined behavior that may lead to incorrect program or even a non-termination. One form of memory violation is array out of bounds access. To avoid this bug, many alias analyses [26, 37] have been proposed. In [26] the analysis tracks valid references from array base pointers to avoid out-of-bounds accesses. Example 1 illustrates this approach.

Example 1 *Consider the program shown in Figure 2. After renaming variables, we let $p_i = p + i$ at line 5 and $p_j = p + j$ at line 6. W designs the function that maps a pointer to its addressable offsets. The static analysis detailed in [26], proves that $W(p) = [0, N-1]$, $W(p_i) = [0, 1]$ and $W(p_j) = [-1, 0]$. In other words, this analysis tells us that p_i for instance can **safely** dereference addresses from 0 to 1 and that the largest addressable offset from p is N-1. Therefore, we no more need to add runtime checks [22] to guarantee the safety of memory accesses.*

2.3 Complexity and scalability of pointer analyses

As in other fields of static analyses, the different pointer analyses that have been proposed can be classified depending on the kind of information they track and the type of abstraction they do.

Historically, the two success stories of pointer analyses, namely Andersen's and Steensgaard's algorithms [1, 36], are flow-insensitive. Information concerning pointer aliases are gathered and merged at join nodes. Clearly, these kinds of analyses are cheaper than analyses that take the control flow into account, such as [41, 25, 11]. However, in programs with arbitrary pointer dereferencing, precise flow-sensitive as well as flow-insensitive analyses algorithms have been shown to be NP-hard [16, 13].

```
1: unsigned N = read()
2: int* p = malloc(N)
3: int i = 0 ; int m = 0 ; int j = N-1
4: while i < j do
5:     p[i] = -1
6:     p[j] = 1
7:     i++ ; j- - ; m++
8: end while
9: p[m] = 0
```

Figure 2: Example for out-of-bounds memory analysis.

As far as precision is concerned, the analyses may also consider the different calling contexts of a given procedure. This again has a cost [27], depending on the length of the calling chain taken into account.

When a pair of pointers is within an alias set but pointers are disjoint and don't dereference overlapping memory regions, the algorithm is said to be imprecise. Therefore, many optimizations might be disabled while they are possible and beneficial. However, the cost of a given analysis is growing with its precision.

The efficiency of alias approaches do not generally refer to the quality of their output but with to speed and to the number of instructions and lines of code that they can handle. Depending on which criterion is privileged, different algorithms have been proposed to be precise or scalable, or claim to handle this trade-off [11].

As far as we know, apart from the classic Andersen and Steensgaard Algorithms for *points-to* sets, none of the algorithms we cited before has been implemented in production compilers. The recent advances in static analyses for compilation [26, 3] have shown that *sparse* analyses, which assign information only on variables, scale better [6] than *dense* analyses. These later also assign information on variables in addition to program points, hence the complexity is higher.

2.4 Static analyses with runtime checks

Hybrid analysis [34] gathers information during compile time and use them at runtime to decide whether an optimized version can be used. The goal is to reduce time and memory overheads. An example of a hybrid analysis is given by Example 2.

Example 2 *Recall that the information needed to optimize the program in Figure 3a is that there is no aliasing between pointers at lines 5 and 6. Given this information, we can parallelize the code, which gives the program in Figure 3b. Since in this case, the value of N cannot be known at compile time, a decision could not be made based on only static analysis which would conservatively answer "may alias". The hybrid analysis technique given by [34] applied to the original program generates the test* $N < 84$ *at compile time and asserts it at runtime.*

2.5 Dynamic analyses

Incomplete control-flow and input information or high complexity of the analysis decreases the efficiency and accuracy of static analyses. Consequently, compilers are less able to perform optimizations. To improve the quality of the results of alias analysis and to increase its efficiency while avoiding memory overheads, *dynamic analysis* [23, 10] techniques have been proposed. These techniques attempt to gather pointer values at run time and check their aliasing during

```
1: int i = 0
2: int N = atoi (argv[1])
3: int* p = malloc (N/2 + 42)
4: for i = 0; i < N/2; i++ do
5:     p[i] = i
6:     p[i+42] = N − i
7: end for
```

(a) Example that illustrates the need to hybrid analysis.

```
1: int i = 0
2: int N = atoi (argv[1])
3: int* p = malloc (N/2 + 42)
4: if N < 84 then
5:     for all i < N/2 do
6:         p[i] = i
7:         p[i+42] = N − i
8:     end for
9: else
10:    for i = 0; i < N/2; i++ do
11:        p[i] = i
12:        p[i+42] = N − i
13:    end for
14: end if
```

(b) Optimizing with hybrid analysis.

Figure 3: Hybrid Analysis.

the program execution. The information gathered during executions, namely, the taken paths and the memory actually accessed, contributes to make the analysis more precise *on the tested inputs*.

Comparing two pointer addresses takes only one cycle and thus, for *one* execution, dynamic alias analyses should be faster than static ones. However, results of dynamic analyses are specific to that execution since it is input-dependent. It can not be generalized to other inputs and analyses and optimizations should be done for each input.

Compared to static analysis, which is done once (at compile time), the overall overhead of dynamic analyses can be more significant. The reader may refer to [8, 24] for further comparison between static and dynamic analyses.

2.6 Our setting: scalable correct static analyses

Since we are interested in safety analysis and code optimization, we are interested in *conservative analyses*: a positive answer “no, they do not alias” will guarantee that the two pointers never overlap during execution. On the contrary, these analyses may fail to disambiguate some pairs of pointers (returning a “may alias” answer), even if there is no execution where the two pointers actually access the same block. Transformations and optimizations will not be enabled with “may alias” to stay safe.

Generally static analysis algorithms provide an imprecise analysis compared to the dynamic ones since they aim to be conservative thus give a safe estimation of pointed locations. In this study, we will only focus on static analyses.

Correctness is also considered as a mandatory condition. Correct alias algorithms are those without false negative answers, which means two pointers are said “no alias” when they do. As demonstrated by [39], this condition is often compromised but not usually detected.

We are also concerned by scalability issues. As we are dealing with compilation optimizations, we cannot pay too much for an alias analysis. Some of the previous approaches, designed for several verification contexts, have been shown not to be scalable. We expect that the algorithmic behind alias analyses for compilers should not be too costly in order to scale well. Memory is also an issue.

3 Applications of alias analyses inside compilers

As we have seen in Section 2, alias analyses are proposed and used for verification and optimization. Inside compilers, they are mostly used to perform transformations on source code, leading to many optimizations. In this section, we make an assessment of some optimizations that are *clients* of alias analyses. We provide examples in which we illustrate the need for alias analysis for more efficient program optimizations.

3.1 An application domain: program optimization.

Alias analysis for loop code motion Code motion consists in interchanging some statements or moving others. This optimization is beneficial especially in loops where loop invariant code motion (LICM) allows removing unchanged statements from the loop by hoisting or sinking them and therefore to avoid useless executions. Code motion is also used to improve data locality and optimize memory and cache accesses.

```
1:  int* p = malloc (2*N*sizeof(int))
2:  int *p1, *p2, *a
3:  *p = 8; *a = 10
4:  p1 = p + N
5:  p2 = p + 2 × N
6:  while p2 > p1 do
7:      a = *p
8:      *p2 = 4
9:      p2- -
10: end while
```

(a) Example for possible loop invariant code motion.

```
1:  int* p = malloc (2*N*sizeof(int))
2:  int *p1, *p2, *a
3:  *p = 8; *a = 10
4:  p1 = p + N
5:  p2 = p + 2 × N
6:  a = *p
7:  while p2 > p1 do
8:      *p2 = 4
9:      p2- -
10: end while
```

(b) Example 4a with hoisted load

Figure 4: Optimizing a program with code motion given a no-alias information.

Program snippet in Figure 4a shows a loop with a read from p and a write in p_2. Clearly $^*p$ is never changed inside the loop since we don't write in p, and p and p_2 access disjoint pieces of memory. Thus, an analysis that provides these information would enable optimizing the loop by hoisting the load from the loop. Such an optimization (Figure 4b) saves a variable write and a store at each loop iteration.

Alias analysis for automatic parallelization Parallelization can be done within loops or between procedures using threads but in both cases we need alias analysis information to detect data race freedom.

Figure 5a shows a procedure PARALLEL that fills, in each loop iteration, two values in p, an integer array. Let $p_i = p + i$ and $p_N = p + N - i$ denote the array accesses at lines 4 and 5 respectively. Since $i < \frac{N}{2}$, writes in p_i and p_N are always independent and can be done in parallel. Provided this "no alias" information, one can optimize this code to parallelize the loop which would give program in 5b. In some cases, for instance for the program depicted in Figure 7, an inter-procedural alias analysis is required to detect the absence of data-races between the two FILL_TABLE calls at line 12 and 13. An automatic parallelization is then possible.

Instruction rescheduling. Rescheduling code means interchanging some of its statements. This option can be profitable to improve data locality, or to enable further optimizations.

```
1: procedure PARALLEL(int* p, int N)
2:     int i = 0
3:     for i = 0; i < N/2; i++ do
4:         p[i] = i
5:         p[N − i] = N − i
6:     end for
7: end procedure
```

(a) Example for possible parallelization.

```
1: procedure PARALLEL_THREAD(int* p, int N)
2:     int i = 0
3:     for all i < N/2 do
4:         p[i] = i
5:         p[N − i] = N − i
6:     end for
7: end procedure
```

(b) PARALLEL procedure parallelized after no alias analysis results.

Figure 5: Use of no alias result for parallelization.

```
1: procedure RESCHEDULE(int* A, int* B, int* C, int L, ...)
2:     int i; int j; int k
3:     for i = 0; i < L; i++ do
4:         B[i] = var
5:     end for
6:     for j = 0; j < 2L; j++ do
7:         A[j] = ...
8:     end for
9:     for k = 0; k < L; k++ do
10:        C[k] = B[k]
11:    end for
12:    ...
13: end procedure
```

(a) Initial RESCHEDULE procedure.

```
1: procedure RESCHEDULE(int* A, int* B, int* C, int L, ...)
2:     int i; int j; int k
3:     for j = 0; j < 2L; j++ do
4:         A[j] = ...
5:     end for
6:     for i = 0; i < L; i++ do
7:         B[i] = var
8:     end for
9:     for k = 0; k < L; k++ do
10:        C[k] = B[k]
11:    end for
12:    ...
13: end procedure
```

(b) Procedure RESCHEDULE after rescheduling.

```
1: procedure RESCHEDULE(int* A, int* B, int* C, int L, ...)
2:     int i; int j; int k
3:     for j = 0; j < 2L; j++ do
4:         A[j] = ...
5:     end for
6:     for i = 0; i < L; i++ do
7:         B[i] = var
8:         C[i] = var
9:     end for
10:    ...
11: end procedure
```

(c) Optimized RESCHEDULE procedure.

Figure 6: Using alias analysis for rescheduling.

Figure 6 shows a procedure RESCHEDULE that fills data in two tables A and B, then copy B into table C. Since B is written in the first loop and then read in the third loop, interchanging the first and second loop would improve data locality. Such an optimization is only valid if A and B are disjoint tables and values of B are never rewritten by those of A, which is implied by a non-alias property between A and B. If this is the case, the program can be rescheduled into 6b and even further optimized by a loop fusion in 6c.

```
1: procedure FILL_TABLE(int* p, int n1, int n2)
2:     int i; int* p1
3:     for i = n1; i < n2; i++ do
4:         p1 = p+i
5:         *p1 = i
6:     end for
7: end procedure
8: procedure MAIN(int argc, int** argv)
9:     assert atoi(argv[1])<atoi(argv[2])
10:    int* p = malloc (2 × atoi(argv[2]))
11:    int* q = p + atoi(argv[1])
12:    spawn FILL_TABLE (p, atoi(argv[1]), atoi(argv[2]))
13:    spawn FILL_TABLE (q, atoi(argv[2]), 2 × atoi(argv[2]))
14: end procedure
```

Figure 7: Example for data race freedom detection

```
1: procedure SORT(int* v, int N)
2:     int i = 0
3:     int j = N - 1
4:     while i < j do
5:         if v[i] > v[j] then
6:             int tmp = v[i]
7:             v[i] = v[j]
8:             v[j] = tmp
9:         end if
10:        i ++
11:        j - -
12:    end while
13: end procedure
```

Figure 8: PENTAGONS analysis for parallelization.

Dead code elimination. Dead code elimination optimization consists in removing useless statements (generally assignments) such as overwritten variables or writes that are never read.

Let us consider again the program of Figure 6a. We now suppose that B and C must "alias". Recall that must alias pointers start at the same location. Hence, these is no need to fill in table C in the third loop at line 9 because it is already done in loop at line 3 which fills table B. Loop at line 9 could be removed (the number of iterations is the same in both loops so the same fields are filled in).

3.2 Conclusion: a need for specialized information about pointers

As we have seen, alias analysis could be used to perform numerous optimizations on C code such as code motion and parallelization. Inside compilers, roles are divided into optimization passes that transform the program, and analyzing passes that perform analyses, including alias analysis.

However, optimizing passes are just clients of analysis passes that should provide them with the necessary information. Otherwise, transformations could not be done even though the analysis is correct and precise. In other words, analysis and transformation passes should speak the same language. In addition to the four types (at most) of alias response (may, must, partial alias, and alias), the compiler might need extra facts such as relations between variables and variable offsets (lower and upper bound).

As an example, in Figure 8, an analysis which do not track relationships between i and j may fail to infer that all `v[i]` and `v[j]` never refer to the same array accesses inside the loop. PENTAGONS [19] is an abstract interpretation based algorithm that tracks "less than" relations between numerical variables (here, i and j, in addition to their value ranges). We think that such "semi-relational" analyses designed to fit particular optimization needs are promising and deserve to be further studied.

4 Experimental evaluation of pointer analyses inside state-of-the art compilers

Alias analysis output does not depend on whether its results would concretely be used for verification or optimization. However, what matters most is the efficiency and scalability of the approach. In this section, we evaluate the precision of the current analyses inside production compilers, and show that there is a huge gap between theory and practice.

4.1 C compilers: GCC, LLVM, ICC

Compilers are responsible for transforming abstracted programming languages into machine code. A compiler includes three main components that process the input source file. These components are: the front-end, the middle-end, and the back-end. The front end parses the source code and generates an **I**ntermediate **R**epresentation (IR), which will be analyzed and optimized in the middle-end. Depending on the machine architecture, the back-end generates the machine code based on the optimized representation.

In this section, we will study some of the production C compilers (CGG, LLVM, and ICC) to show that the alias analyses inside compilers are far from being satisfactory. We show how these three compilers handle the simple example of Figure 4a with the maximum level of optimization (O3).

4.2 GCC

The **G**NU **C**ompiler **C**ollection (GCC) is a collection of front-ends of C, C++, Java among others and their libraries. We are particularly interested in GCC, the C compiler.

In this section, we study assembly code generated by GCC 4.9.2 (`gcc -S`) when applied to code snippet in Figure 4a with O3 optimizations. Simplified results are depicted in Figure 9.

The C compiler GCC at this level of optimization is able to know that the loop will be executed at least once. Thus, it removes the assignment that associates 10 to a and instead assigns to a variable stored in p, which is 8 at this stage of program. It then executes the loop and, at each iteration, loads value of p and assigns it to a. As a conclusion GCC, even with -O3, fails to disambiguate pointers p and p_2 and does not hoist or sink the instruction $a = {}^*p$ outside the loop. Optimization level `O3` relies on the default alias analysis, which is type-based, that just marks all variables of compatible types as aliasing.

4.3 LLVM

LLVM is the name of a research project at the *University of Illinois*. This project has contributed to a collection of modular and reusable compiler and tool-chain technologies. LLVM has *clang* as a front end which is a native C/C++/Objective-C compiler.

We now try to optimize the loop in program of Figure 4a using "*opt*" to activate LLVM optimizer (version 3.8). A simplified Intermediate Representation (IR) of optimized byte code is depicted in Figure 10. Similarly to GCC, LLVM which relies on the default optimization *basicaa* (for basic alias analysis) fails to disambiguate pointers p and p_2 and to provide the non-aliasing information to the *Loop Invariant Code Motion* (Licm) pass to optimize the code and remove the load from p_2 from the while loop.

```
.file    "code_motion.c"

     ...                    # %edx -> a; %rcx -> p2, %rax -> p
     movl     $8, %edx      # a = 8
     movl     $8, (%rax)    # *p = 8
     jmp .L3
.L6:
     movl (%rax), %edx      # a = *p
.L3:
     movl     $4, (%rcx)    # *p2 = 4
     subq     $4, %rcx      # p2 --
     cmpq     %rsi, %rcx    # p2 >? p1
     jne .L6                # if p2 <= p1 jump to .L6
     ...
```

Figure 9: Simplified assembly code given by `gcc -O3` applied to `code_motion.c`. The boxed instruction stays inside the L6 loop.

```
while.body.lr.ph:                   ; preds = %entry
  %add.ptr5 = getelementptr inbounds i32* %1, i64 %conv
  br label %while.body

while.body:                         ; preds = %while.body, %while.body.lr.ph
  %2 = phi i32 [ 8, %while.body.lr.ph ], [ %3, %while.body ]
  %p2.010 = phi i32* [ %add.ptr5, %while.body.lr.ph ], [ %incdec.ptr, %while.body ]
  store i32 4, i32* %p2.010, align 4                                ; *p2 = 4
  %incdec.ptr = getelementptr inbounds i32* %p2.010, i64 -1          ; p2 --
  %cmp = icmp ugt i32* %incdec.ptr, %add.ptr                        ; p2 >? p1
  %3 = load i32* %1, align 4                                        ; a = *p
  br i1 %cmp, label %while.body, label %while.end.loopexit
```

Figure 10: Simplified IR given by `clang -O3` applied to `code_motion.c`. %1 designs pointer p and %3 variable a in the while body. Here again, the boxed instruction is not hoisted out of the loop.

4.4 ICC

ICC is the **I**ntel **C** **C**ompiler. The main purpose of ICC is to make C programs run faster on Intel architectures. We have tested code in Figure 4a using ICC 15 released in 2014. Simplified assembly generated with -O3 optimizations is depicted in Figure 11. Highlighted code shows the load from p which is kept inside the while loop. We conclude that Intel compiler is unable to disambiguate pointers p and p_2 and to hoist the load.

4.5 Restrict keyword

The *restrict* keyword is a C99 standard keyword that can be specified by the programmer to give the compiler information about aliasing. In fact, it is applied to a pointer p to say that only p or a pointer derived from it can access that memory region during its lifetime. Hence, if p is a restricted pointer then, any access to p's block must occur through p while p is alive. Otherwise, we have an undefined behavior.

Let p and q be two pointers arguments of procedure F in Figure 12a. If F is not inlined, the compiler is not able to optimize it even using an interprocedural context-sensitive analysis. In fact, since linking is done after optimizations, optimizing a standalone function without keeping

```
..B1.12:                              # Preds ..B1.12 ..B1.11
        lea        (,%rdi,8), %r9
        incq       %rdi                   # p1 ++
        negq       %r9
        addq       %r8, %r9
        cmpq       %rbx, %rdi
        movl       %edx, (%r9)            # *p2 = 4
        movl       (%rax), %esi           # a = *p
        movl       %edx, -4(%r9)          # *(p2 - 1) = 4
        jb         ..B1.12                # if p1 < p2 jump ..B1.12
```

Figure 11: Simplified assembly code given by `icc -O3` applied to `code_motion.c`. The loop has been unrolled by a factor of two. Another loop is also generated by ICC where the load is removed but replaced by a = 8. The flow gets into this loop as soon as $p > p_2$ (which implies that p and p_2 will never alias any more, since p_2 is decreasing).

a trace of its unoptimized version would be incorrect in some other call contexts. Adding the restrict keyword to procedure F is a declaration that programmers intent p and q to never alias. No further alias analysis will be done by the compiler. Example 3 illustrates the benefit of using the restrict keyword by showing simplified assembly codes of F and F_RES using LLVM and -O3.

```
1: procedure F(int* p, int* q, int a, int b)
2:     p[a] = p[a] + q[b]
3:     p[a] = p[a] + q[b]
4: end procedure
```

(a) Procedure without restrict keyword.

```
1: procedure F_RES(int* restrict p, int* restrict q, int a, int b)
2:     p[a] = p[a] + q[b]
3:     p[a] = p[a] + q[b]
4: end procedure
```

(b) Procedure with restrict keyword.

Figure 12: Syntax of restrict keyword.

Example 3 *Figure 13 gives a simplified assembly code generated after LLVM O3 optimizations applied to* F *and* F_RES *in Figure 12. The compiler here associates %rdi to pointer p and %rsi to pointer q. The main difference between 13a and 13b is two extra memory accesses (a load and a store) done in 13a. In fact, with the restrict keyword, p and q are supposed not to alias. Addresses* $(p+a)$ *and* $(q+b)$ *are then disjoint and any write in* p[a] *does not affect any read from* q[b]. *Therefore, the compiler can optimize this code by loading only once the value stored in* q[b] *and not storing the first sum* (p[a] + q[b]) *in* q[a] *since we have a write-write dependence. Extra store and load are depicted respectively line 5 and line 6 in Figure 13a. Without such aliasing information, the compiler is unable to perform this optimization. For the readability of Figure 13, the reader may note that lines 2, 4 and 7 in Figure 13a are equivalent to lines 2, 5 and 7 in Figure 13b and correspond respectively to* p[a].

Experimental evaluation of pointer analyses inside state-of-the art compilers GCC, LLVM and ICC shows that these compilers fails in carrying out optimizations based on pointer analyses for simple benchmarks. Some published techniques have probably tried to improve these results and might theoretically succeed in optimizing them, but the fact that they are not added to compilers and not available in the distributed version of compilers still shows a certain amount of inefficiency.

```
    .text
    .file    "F.bc"
    .globl   f
    .align   16, 0x90
    .type    f,@function
f:
  ....

.Ltmp2:
    .cfi_def_cfa_register %rbp
1   movslq   %ecx, %rax
2   movl     (%rsi,%rax,4), %ecx
3   movslq   %edx, %r8
4   addl     (%rdi,%r8,4), %ecx
5   movl     %ecx, (%rdi,%r8,4)
6   addl     (%rsi,%rax,4), %ecx
7   movl     %ecx, (%rdi,%r8,4)
8   popq     %rbp
9   retq

  ....
```

(a) Procedure without restrict keyword.

```
    .text
    .file    "F_RES.bc"
    .globl   f_res
    .align   16, 0x90
    .type    f,@function
f_res:
  ....

.Ltmp2:
    .cfi_def_cfa_register %rbp
1   movslq   %ecx, %rax
2   movl     (%rsi,%rax,4), %ecx
3   movslq   %edx, %rax
4   movl     %ecx, %edx
5   addl     (%rdi,%rax,4), %edx
6   addl     %ecx, %edx
7   movl     %edx, (%rdi,%rax,4)
8   popq     %rbp
9   retq

  ....
```

(b) Procedure with restrict keyword.

Figure 13: Impact of using the restrict keyword on F. The highlighted code on the left is unnecessary if we know that p and q alias.

5 Discussion

Toward new pointer analyses in compilers. As we saw, mainstream compilers still struggle to distinguish intervals within the same array. Our study confirms that pointer analyses often fail to disambiguate regions addressed from a common base pointer via different offsets, as stated by Yong and Horwitz [42]. Field-sensitive pointer analysis could provide a partial solution to this problem. These analyses can distinguish different fields within a record, such as a struct in C [30], or a class in Java [40]. However, they rely on syntax that is usually absent in the low level program representations adopted by compilers. We believe that actually implementing such analyses inside production compilers is still a challenge.

A proposition for alias analyses with offsets. Recently, we proposed an analysis that combines range analysis and pointer analysis [29]. The main idea is to disambiguate pointers by comparing their base pointers and possible addressable memory offsets. We use an abstract interpretation based algorithm where we compute offsets from *abstract* program locations. Our analysis strongly relies on a preprocessing that computes parametric ranges of numerical variables. The solution that we have proposed is sparse and relies on Extended Static Single Assignment E-SSA form [2] to achieve efficiency. Indeed, pointers are disambiguated within two steps: the first step is a global analysis that allows disambiguating pointers in the whole program by iterating and solving an abstract constraint system until achieving a fix point. A widening operator is used to ensure convergence. The second step is a local refinement that permits to disambiguate pointers inside loops.

Experimental results show that this solution implemented on the top of LLVM compiler is fast, scalable (quasi-linear in the number of lines of code) and succeeds in disambiguating more pairs

of pointers compared to basic alias analysis of LLVM. It was for instance able to disambiguate pointers $p + a$ and $q + b$ in procedure F (Figure 12) without using the restrict keyword and to remove the extra load and store as detailed in Example 3. However, our experiments show that the new alias analysis enables less optimizations than what was expected. We therefore suppose that a dysfunction between analysis passes and optimization ones is occurring and conclude that what matters most is not the efficiency of the analysis and its scalability but information shape provided to optimizing passes.

6 Conclusion

In this paper we have presented an overview of state-of-the art techniques for pointer analyses. Although this topic has received much attention in the last decades, we believe that new analyses remain to be defined to fit the special needs of optimization in state-of-the-art compilers. The designers of such analyses should pay a particular attention to the actual implementation of the compilers to design simple but efficient analyses. We also strongly believe that there remains work to be done on client analyses that intensively use the results of alias analyses.

Contents

1 Introduction **3**

2 Alias analysis: context and related work **4**
2.1 Problem statement 4
2.2 Static analyses of pointers 4
2.3 Complexity and scalability of pointer analyses 5
2.4 Static analyses with runtime checks 6
2.5 Dynamic analyses 6
2.6 Our setting: scalable correct static analyses 7

3 Applications of alias analyses inside compilers **8**
3.1 An application domain: program optimization. 8
3.2 Conclusion: a need for specialized information about pointers 10

4 Experimental evaluation of pointer analyses inside state-of-the art compilers **11**
4.1 C compilers: GCC, LLVM, ICC 11
4.2 GCC 11
4.3 LLVM 11
4.4 ICC 12
4.5 Restrict keyword 12

5 Discussion **14**

6 Conclusion **15**

References

[1] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.

[2] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 321–333. ACM, 2000.

[3] Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. SSI Properties Revisited. *ACM Transactions on Embedded Computing Systems*, 11S(1), 2012. Article 21, 23 pages.

[4] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Proceedings of Static Analysis Symposium*, pages 182–203. Springer, 2006.

[5] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of Computer and Communications Security*, pages 39–50. ACM, 2008.

[6] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of Symposium on Principles of Programming Languages*, pages 55–66. ACM Press, 1991.

[7] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. Checking cleanness in linked lists. In *Proceedings of Static Analysis Symposium*, pages 115–134. Springer, 2000.

[8] Michael D. Ernst. Invited talk static and dynamic analysis: synergy and duality. In *Proceedings of Programming and Analysis for Software Tools and Engineering*, page 35. ACM, 2004.

[9] Rakesh Ghiya, Daniel M. Lavery, and David C. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of Programming Language Design and Implementation*, pages 47–58. ACM, 2001.

[10] Bolei Guo, Youfeng Wu, Cheng Wang, Matthew J. Bridges, Guilherme Ottoni, Neil Vachharajani, Jonathan Chang, and David I. August. Selective runtime memory disambiguation in a dynamic binary translator. In *Proceedings of European Joint Conferences on Theory and Practice of Software*, pages 65–79. Springer, 2006.

[11] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of International Symposium on Code Generation and Optimization*, pages 289–298. IEEE Computer Society, 2011.

[12] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of Workshop on Program Analysis For Software Tools and Engineering*, pages 54–61. ACM, 2001.

[13] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.

[14] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of International Conference on Computer Aided Verification*, pages 226–239. Springer, 2007.

[15] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of Operating Systems Design and Implementation*, pages 147–163. USENIX, 2014.

[16] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of Symposium on Principles of Programming Languages*, pages 93–103, 1991.

[17] Per Larsen, Razya Ladelsky, Jacob Lidman, Sally A. McKee, Sven Karlsson, and Ayal Zaks. Parallelizing more loops with compiler guided refactoring. In *Proceedings of International Conference on Parallel Processing*, pages 410–419. IEEE Computer Society, 2012.

[18] Huisong Li, Bor-Yuh Evan Chang, and Xavier Rival. Shape analysis for unstructured sharing. In *Proceedings of Static Analysis Symposium*, 2015.

[19] Francesco Logozzo and Manuel Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Science of Computer Programming*, 75(9):796–807, 2010.

[20] Vincenzo Martena and Pierluigi San Pietro. Alias analysis by means of a model checker. In *Proceedings of Compiler Construction*, pages 3–19. Springer, 2001.

[21] Amira Mensi. *Analyse des pointeurs pour le langage C*. PhD thesis, MINES ParisTech, 2013.

[22] Reed Milewicz, Rajeshwar Vanka, James Tuck, Daniel Quinlan, and Peter Pirkelbauer. Runtime checking C programs. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 2107–2114. ACM, 2015.

[23] Markus Mock. Dynamic analysis from the bottom up. In *Proceedings of International Conference on Software Engineering*, 2003.

[24] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: a comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of Programming and Analysis for Software Tools and Engineering*, pages 66–72. ACM, 2001.

[25] Vaivaswatha Nagaraj and R. Govindarajan. Approximating flow-sensitive pointer analysis using frequent itemset mining. In *Proceedings of International Symposium on Code Generation and Optimization*, pages 225–234. IEEE Computer Society, 2015.

[26] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Validation of memory accesses through symbolic analyses. In *Proceedings of Object-Oriented Programming, Systems, Languages & Applications*, pages 791–809. ACM, 2014.

[27] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 475–484, New York, NY, USA, 2014. ACM.

[28] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 369–378. ACM, 2015.

[29] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Pereira. Symbolic range analysis of pointers. Manuscript submitted for publication, 2016.

[30] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *Proceedings of Program Analysis for Software Tools and Engineering*, pages 37–42, 2004.

[31] Fernando Magno Quintão Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of International Symposium on Code Generation and Optimization*, pages 126–135. IEEE Computer Society, 2009.

[32] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[33] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of Programming Language Design and Implementation*, pages 182–195. ACM, 2000.

[34] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. In *Proceedings of International Conference on Supercomputing*, pages 274–284. ACM, 2002.

[35] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.

[36] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of Symposium on Principles of Programming Languages*, pages 32–41. ACM Press, 1996.

[37] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp. Proving termination and memory safety for programs with pointer arithmetic. In *Proceedings of International Joint Conference on Automated Reasoning*, pages 208–223. Springer, 2014.

[38] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of Symposium on Security and Privacy*, pages 156–168. IEEE Computer Society, 2001.

[39] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. Effective dynamic detection of alias analysis errors. In *Proceedings of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 279–289. ACM, 2013.

[40] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 155–165. ACM, 2011.

[41] Sen Ye, Yulei Sui, and Jingling Xue. Region-based selective flow-sensitive pointer analysis. In *Proceedings of Static Analysis Symposium*, pages 319–336. Springer, 2014.

[42] Suan Hsi Yong and Susan Horwitz. Pointer-range analysis. In *Proceedings of Static Analysis Symposium*, pages 133–148. Springer, 2004.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399