

1 Restrictifier: a tool to disambiguate pointers at 2 function call sites

3 Victor Campos, Péricles Alves, and Fernando Pereira¹

4 UFMG – Avenida Antônio Carlos, 6627, 31.270-010, Belo Horizonte
5 {victorsc,periclesrafael,fernando}@dcc.ufmg.br

6 **Abstract.** Despite being largely used in imperative languages, pointers
7 are one of the greatest enemies to code optimizations. To circumvent this
8 problem, languages such as C, C++, Rust and Julia offer reserved words,
9 like “restrict” or “unique”, which are able to inform the compiler that
10 different function parameters never point to the same memory region.
11 Even with proven applicability, it’s still left to the programmer the task
12 of inserting such keywords. This task is, in general, boring and prone to
13 errors. This paper presents Restrictifier, a tool that (i) finds function call
14 sites in which actual parameters do not alias; (ii) clones the functions
15 called at such sites; (iii) adds the “restrict” keyword to the arguments of
16 these functions; and (iv) replaces the original function call by a call to
17 the optimized clone whenever possible.
18 Video: <https://youtu.be/1b1YSsDV5Qc>

19 1 Introduction

20 One of the most important characteristics of languages such as C and C++ is the
21 existence of pointers. These have been included in the project of those languages
22 as a way to allow a better control over the memory usage along the life cycle of a
23 program. In spite of providing a broader control over the memory management
24 to the developer, pointers make the operation of optimizing compilers more
25 complex. This is due to, in general, the fact that static languages’ compilers
26 aren’t able to determine whether two or more pointers may reference the same
27 memory region at some point in the control flow of an application. This limitation
28 blocks several code optimizations [2]. Aiming the mitigation of this problem, in
29 the last decades there has been a considerable amount of research about efficient
30 techniques of alias analyses [4,1,3].

31 In a similar effort, the C standard from 1999 (C99) included the keyword
32 “restrict”. This modifier allows the programmer to inform the compiler about
33 pointers that never reference overlapping regions. More specifically, when applied
34 on a pointer argument, the “restrict” word determines the memory region pointed
35 by this parameter cannot be accessed by any other variable inside the function.
36 Despite being available for more than a decade, the usage of “restrict” has been
37 very little among the C community. This phenomenon has been led by two
38 factors: (i) the use of this modifier requires a deep knowledge about memory
39 resources across the program’s lifecycle; (ii) by demanding manual intervention

40 from the programmer, the insertion of “restrict” is prone to errors. Our tool aims
41 to move such task to the compiler.

42 Restrictifier combines code versioning, in the form of function cloning, with
43 static alias analyses to boost the number of disambiguated pointers, which in
44 turn enables more aggressive optimizations on functions that take pointers as
45 arguments. Our method consists in constructing, for each function that receives
46 two or more pointers as arguments, a new version in which these parameters
47 are annotated with “restrict”. We then utilize a completely static approach to
48 substitute calls to such functions by calls to their optimized versions. This static
49 approach relies on the different alias analyses implemented by our compiler of
50 choice, LLVM.

51 2 Overview

```
1 void prefix(int *src, int *dst, int N) {  
2     int i, j;  
3     for (i = 0; i < N; i++) {  
4         dst[i] = 0;  
5         for (j = 0; j <= i; j++) {  
6             dst[i] += src[j];  
7         }  
8     }  
9 }  
10  
11 int main() {  
12     int N = 100;  
13     int A1[N], A2[N];  
14     prefix(A1, A2, N);  
15 }
```

Fig. 1: Example of a target program for Restrictifier.

52 In this Section, we will use the program in Figure 1 to introduce the tech-
53 nique which Restrictifier incorporates. The function *prefix* receives two pointers
54 as arguments and stores to *dst* the sums of prefixes for each position of the ar-
55 ray *src*. If the compiler was able to infer that *src* and *dst* point to completely
56 distinct regions, it could apply more aggressive optimizations on *prefix*, such
57 as Loop Unrolling, Loop Invariant Code Motion, automatic parallelization, and
58 vectorization, making it significantly faster. It seems rather simple, but modern
59 compilers, like LLVM, are not able to optimize fragments of code such as this
60 one.

61 Our technique starts with the cloning of functions that take two or more
62 pointers as input, applying the “restrict” modifier to the arguments of the cloned

63 version. The clone of *prefix* is the function *prefix_noalias*, shown in Figure 2.
64 Those “restrict” modifiers allow the compiler to assume *src* and *dst* do not over-
65 lap. This information permits, for instance, to substitute the assignment to *dst[i]*
66 in the inner loop (Figure 1, line 6) by an assignment to a temporary variable
67 (Figure 2, line 6), executing the store operation only once for each iteration of
68 the outer loop (Figure 2, line 8), thus reducing the number of memory accesses.
69 In this report, we present a tool to promote substitution of function calls by
70 their cloned version, optimized with “restrict”, using compiler static analyses.
71 Next, we present how Restrictifier works on the example in Figure 1.

```
1 void prefix_noalias(int * restrict src, int * restrict dst, int N) {  
2     int i, j;  
3     for (i = 0; i < N; i++) {  
4         int tmp = 0;  
5         for (j = 0; j <= i; j++) {  
6             tmp += src[j];  
7         }  
8         dst[i] = tmp;  
9     }  
10 }  
11  
12 int main() {  
13     int N = 100;  
14     int A1[N], A2[N];  
15     prefix_noalias(A1, A2, N);  
16 }
```

Fig. 2: Optimized version of *prefix* and the calling context after being processed by Restrictifier tool.

72 In the context of *prefix*'s function call (Figure 1, line 14), the input arrays
73 clearly do not overlap, since they were allocated separately in the program's
74 stack. For this context, it's possible to substitute the called function by its cloned
75 version using only compilation time analyses, i.e. static alias analyses can detect
76 those memory references are independent. This way, we say this context could be
77 solved by a completely static approach, which defines the presented Restrictifier
78 tool.

79 3 Pointer Disambiguation

80 In this tool, we utilize a static approach to disambiguate pointers. This method
81 uses traditional pointer analyses to determine whether memory regions may over-
82 lap or not. Efficiency is its greatest advantage: the pointer disambiguation, being
83 statically performed, does not incur cost in the running time of the program.

84 As initially mentioned, the purely static method employed by the Restrictifier
85 tool uses alias analyses to distinguish, in compilation time, memory regions.
86 There are several algorithms described in the literature that tackle the pointer
87 aliasing problem. Our tool uses all the six different implementations available in
88 LLVM in tandem to maximize its effectiveness:

- 89 – **Basic AA**: the simplest implementation (yet, one of the most effective)
90 present in LLVM. This analysis uses several heuristics, all based in tests
91 that can be done in constant time to disambiguate pointers.
- 92 – **Type-based AA**: based on the C99 property stating that pointers of dif-
93 ferent types cannot overlap.
- 94 – **Globals AA**: based on the fact that global variables whose address is not
95 taken cannot overlap other pointers in the program.
- 96 – **SCEV AA**: uses *scalar evolution* of integer variables to disambiguate point-
97 ers. A variable’s scalar evolution is an expression – affine – that describes
98 the possible values it may assume during execution. Such information is ex-
99 tracted from loops. For instance, in the loop `for(i = B; i < N; i += S)`,
100 variable `i` has the following evolution: $e(i) = I \times S + B$, I being the loop iter-
101 ation. Regions indexed by integer variables overlap if their scalar evolutions
102 have a non-empty intersection.
- 103 – **Scoped NoAlias**: preserves aliasing information, found by the compiler
104 frontend, between different scopes in the program. This is rather important
105 to achieve a limited form of inter-procedurality of the alias analyses, which
106 are primarily all intra-procedural in LLVM.
- 107 – **CFL AA**: this is the most refined [5] alias analysis among the LLVM imple-
108 mentations. However, it is also the most expensive in terms of computational
109 cost, reaching $O(n^3)$ in the worst case..

110 The static approach, which we describe in this report, does not generate any
111 test to be executed during runtime. Therefore, its runtime overhead is *zero*.

112 4 Tutorial

113 In this Section we show how to use Restrictifier to optimize programs. Our tool
114 can be found in <http://cuda.dcc.ufmg.br/restrictification/>. Restrictifier
115 can be used in two ways: via the web tool or installing it on your computer. The
116 former is more contrived, and it should be used for demonstration purposes. To
117 wholly profit from the optimization opportunities, we recommend installing and
118 using our tool locally.

119 4.1 Web tool

120 The web tool is a simpler demonstration of the Restrictifier’s features. As such, it
121 accepts only one C or C++ file as input, which can be either uploaded or have its
122 contents inserted in a form. Given that it works on a single file as an independent

123 unit, this file should contain (i) the functions to be optimized, i.e. functions that
124 receive pointer arguments, and also (ii) their calls. Failing to comply to both
125 conditions will naturally undermine our optimizer.

126 The web interface of Restrictifier produces to its user two assembly files writ-
127 ten in the LLVM intermediate program representation. The first is the original
128 version of the program, without any optimization from our tool nor LLVM's. The
129 second one, on the other hand, is the final version after going through the entire
130 optimization chain of Restrictifier and LLVM, which takes advantage of “restrict”
131 keywords automatically inserted to aggressively optimize your program.

132 Along with these two resulting files, the user can also ask to see statistics of
133 the Restrictifier run, which show information such as how many function calls
134 have been substituted by their optimized clones, how many functions have had a
135 optimized clone created for, and how long it has taken to run our tool. Not only
136 that, but the resulting optimized assembly can also be output into the page, if
137 so desired.

138 Having the optimized LLVM assembly file, you need to have it integrated to
139 your project. Before linking it with the other modules of your C/C++/Objective-
140 C program, you need to compile it for your specific machine. For this, you ought
141 to have installed the LLVM binaries. These can be acquired by either building
142 LLVM yourself (our page has directions for this), or by installing them using a
143 package manager of your operating system:

```
144 - Ubuntu Linux: apt-get install clang llvm  
145 - Mac OS X (you need to have Homebrew installed): brew install llvm
```

146 After having LLVM binaries installed, you can translate from the LLVM
147 assembly to native assembly:

```
148 llc optimized.ll -o module.s
```

149 And finally compile the native assembly to an object file. On Linux:

```
150 g++ module.s -o module.o
```

151 On Mac OS X:

```
152 clang++ module.s -o module.o
```

153 4.2 Local use

154 Restrictifier is available as a web interface, so that it can be easily used. However,
155 we also provide a downloadable version of it. Using Restrictifier locally provides a
156 better degree of optimization. For instance, one can optimize the entire program,
157 linked, with our tool. As such, functions that are defined inside one unit, but
158 called from others, can also be candidates for optimization. This is not the case
159 when the web tool is used.

160 In order to use Restrictifier on your own computer, you need to compile it.
161 Our tool's source code is available at our web page, and it is compatible with
162 LLVM 3.6. Since LLVM usually changes a lot between versions, it's very likely
163 that Restrictifier won't work with earlier versions. To use our tool locally, you
164 are required to build LLVM yourself. A very good guide for this can be found in
165 LLVM's tutorial¹. We recommend that you also build Clang, the C/C++/Obj-C
166 frontend, alongside with LLVM². After building LLVM and Clang, it's the turn
167 of building Restrictifier itself. This process is the same as building any custom
168 LLVM pass. As such, the LLVM's custom pass tutorial³ can help you.

169 Assuming you've correctly built LLVM, Clang and Restrictifier, it's pretty
170 simple to use our tool. You should have the LLVM binaries in your path now.

171 First, we can translate a C/C++ file into an LLVM bitcode using Clang:

```
172 clang -c -emit-llvm file.c -o file.bc ${CFLAGS}  
173 clang++ -c -emit-llvm file.cpp -o file.bc ${CXXFLAGS}
```

174 After all units have been compiled, you need to link them together:

```
175 llvm-link file1.bc file2.bc file3.bc -o program.bc ${LDLFLAGS}
```

176 Now it's time to optimize your program. As such, use the LLVM optimizer:

```
177 opt -mem2reg -cfl-aa -libcall-aa -globalsmodref-aa -scev-aa  
178     -scoped-noalias -basicaa -tbaa  
179     -load PATH_TO_Restrictifier/AliasFunctionCloning.(so|dylib) -afc -O3  
180     -disable-inlining  
181     -o program.optimized.bc
```

182 This is quite a long command. Let's break it down:

- 183 1. The first line invokes the optimizer and calls every alias analysis that LLVM
184 contains.
- 185 2. The second line loads the shared library, calls the Restrictifier, and finally
186 performs the whole suite of optimizations (-O3) taking advantage of Restrictifier's work.
187

188 What is left to do is just translating it from LLVM assembly to native as-
189 sembly:

```
190 llc program.optimized.bc -o program.s
```

191 And assemble it:

```
192 g++ program.s -o program.exe
```

193 Now you have an optimized version of your program.

¹ <http://www.llvm.org/docs/GettingStarted.html>

² http://clang.llvm.org/get_started.html

³ <http://llvm.org/docs/WritingAnLLVMPass.html>

194 5 Use case: prefix sum

195 In this section, we present a use case for Restrictifier. In Figure 3, we have
196 a program that computes the prefix sums of a source array and stores them
197 to a destination array. This example was already presented in Section 2. Here,
198 though, we observe the effects of using Restrictifier on the running time of the
199 program.

```
1 void fill(int *array, int N) {
2     int i;
3     for (i = 0; i < N; i++) {
4         array[i] = i*i;
5     }
6 }
7
8 void prefix(int *src, int *dst, int N) {
9     int i, j;
10    for (i = 0; i < N; i++) {
11        dst[i] = 0;
12        for (j = 0; j <= i; j++) {
13            dst[i] += src[j];
14        }
15    }
16 }
17
18 int main() {
19     int N = 400000;
20     int A1[N], A2[N];
21     fill(A1, N);
22     prefix(A1, A2, N);
23 }
```

Fig. 3: Use case of Restrictifier: prefix sum.

200 We chose an array size of 400,000 so that runtimes could be long enough to
201 be properly compared. The source array, *A1*, is filled with integers before being
202 passed on to the core function of the program, *prefix*.

203 It is easy to see that, if the compiler has knowledge that *src* and *dst* point
204 to non-overlapping regions, *prefix*'s code can be optimized to reduce the number
205 of memory accesses. Particularly, the statement in Figure 3, line 13, which stores
206 to *dst[i]* at every single iteration of the inner loop, can be transformed into a
207 sum that stores to a register, which would then be stored to memory at position
208 *dst[i]* outside of the inner loop. Such optimization can be seen in Figure 2,
209 page 3.

210 We applied the Restrictifier onto this example and assessed the results. For
211 this small experiment, we used a computer with an Intel Core i5 1.6 GHz pro-
212 cessor running Mac OS X 10.10.3. We have taken the mean of 15 runs.

Version	Mean	Std. deviation
Regular	41.743s	1.642s
Restrictified	13.075s	0.826s

Table 1: Comparison of runtimes between regular and optimized versions.

213 From the Table 1, we clearly see how much the program earned from using
214 Restrictifier. The addition of “restrict” to the arguments of function *prefix*
215 enabled the reduction of the quantity of memory accesses and also the loop’s
216 vectorization. These optimizations, only possible because of our tool’s work,
217 permitted our use case program to achieve 3.2x of speedup for this input.

218 6 Conclusion

219 This report described our Restrictifier tool. The technique behind our tool is
220 to automatically insert the “restrict” keyword to function pointer arguments,
221 which indicates that these pointers do not reference overlapping memory regions.
222 Such property enables more aggressive compiler optimizations, hence improving
223 efficiency of programs. We developed a web version of our tool to demonstrate
224 its features and we also provide its source code along with a tutorial teaching
225 users how to deploy Restrictifier.

226 Restrictifier can be found at:
227 <http://cuda.dcc.ufmg.br/restrictification/>.

228 References

- 229 1. Michael Hind. Pointer Analysis: Haven’t We Solved This Problem Yet? In *Workshop*
230 *on Program Analysis for Software Tools and Engineering*, PASTE ’01, pages 54–61,
231 New York, NY, USA, 2001. ACM.
- 232 2. William Landi and Barbara G. Ryder. Pointer-induced Aliasing: A Problem Classi-
233 fication. In *Symposium on Principles of Programming Languages*, POPL ’91, pages
234 93–103, New York, NY, USA, 1991. ACM.
- 235 3. John Whaley and Monica S. Lam. Cloning-based Context-sensitive Pointer Alias
236 Analysis Using Binary Decision Diagrams. In *Conference on Programming Language*
237 *Design and Implementation*, PLDI ’04, pages 131–144. ACM, 2004.
- 238 4. Robert P. Wilson and Monica S. Lam. Efficient Context-sensitive Pointer Analysis
239 for C Programs. In *Conference on Programming Language Design and Implemen-*
240 *tation*, PLDI ’95, pages 1–12, New York, NY, USA, 1995. ACM.
- 241 5. Qirun Zhang, Michael Lyu, Hao Yuan, and Zhendong Su. Fast Algorithms for
242 Dyck-CFL-reachability with Applications to Alias Analysis. In *Conference on Pro-*
243 *gramming Language Design and Implementation*, PLDI ’13, pages 435–446. ACM,
244 2013.