Restrictifier: a tool to disambiguate pointers at function call sites

Victor Campos[1], **Péricles Alves**[1], **Fernando Pereira**[1]

[1]Universidade Federal de Minas Gerais (UFMG)
Avenida Antônio Carlos, 6627, 31270-010, Belo Horizonte, MG

{victorsc,periclesrafael,fernando}@dcc.ufmg.br

Abstract. *Despite being largely used in imperative languages, pointers are one of the greatest enemies to code optimizations. To circumvent this problem, languages such as C, C++, Rust and Julia offer reserved words, like "restrict" or "unique", which are able to inform the compiler that different function parameters never point to the same memory region. Even with proven applicability, it's still left to the programmer the task of inserting such keywords. This task is, in general, boring and prone to errors. This paper presents Restrictifier, a tool that (i) finds function call sites in which actual parameters do not alias; (ii) clones the functions called at such sites; (iii) adds the "restrict" keyword to the arguments of these functions; and (iv) replaces the original function call by a call to the optimized clone whenever possible.*

Video tutorial: https://youtu.be/1blYSsDV5Qc

1. Introduction

One of the most important characteristics of languages such as C and C++ is the existence of pointers. These have been included in the project of those languages as a way to allow a better control over the memory usage along the life cycle of a program. In spite of providing a broader control over the memory management to the developer, pointers make the operation of optimizing compilers more complex. This is due to, in general, the fact that static languages' compilers aren't able to determine whether two or more pointers may reference the same memory region at some point in the control flow of an application. This limitation blocks several code optimizations [Landi and Ryder 1991]. Aiming the mitigation of this problem, in the last decades there has been a considerable amount of research about efficient techniques of alias analyses [Wilson and Lam 1995, Hind 2001, Whaley and Lam 2004].

In a similar effort, the C standard from 1999 (C99) included the keyword "restrict". This modifier allows the programmer to inform the compiler about pointers that never reference overlapping regions. More specifically, when applied on a pointer argument, the "restrict" word determines the memory region pointed by this parameter cannot be accessed by any other variable inside the function. Despite being available for more than a decade, the usage of "restrict" has been very little among the C community. This phenomenon has been led by two factors: (i) the use of this modifier requires a deep knowledge about memory resources across the program's lifecycle; (ii) by demanding manual intervention from the programmer, the insertion of "restrict" is prone to errors. Our tool aims to move such task to the compiler.

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

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 }
```

Figure 1. Example of a target program for Restrictifier.

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

Our technique starts with the cloning of functions that take two or more pointers as input, applying the "restrict" modifier to the arguments of the cloned version. The clone of *prefix* is the function *prefix_noalias*, shown in Figure 2. Those "restrict" modifiers allow the compiler to assume *src* and *dst* do not overlap. This information permits, for instance, to substitute the assignment to *dst[i]* in the inner loop (Figure 1, line 6) by an assignment to a temporary variable (Figure 2, line 6), executing the store operation only once for each iteration of the outer loop (Figure 2, line 8), thus reducing the number of memory accesses. In this report, we present a tool to promote substitution of function calls by their cloned version, optimized with "restrict", using compiler static analyses.

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 }
```

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

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

3. Pointer Disambiguation

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

As initially mentioned, the purely static method employed by the Restrictifier tool uses alias analyses to distinguish, in compilation time, memory regions. There are several algorithms described in the literature that tackle the pointer aliasing problem. In LLVM, there are six different implementations that complement each other, given that they handle different cases. Thus, our tool uses them in conjunction to maximize effectiveness:

- **Basic AA**: the simplest implementation (yet, one of the most effective) present in LLVM. This analysis uses several heuristics, all based in tests that can be done in constant time to disambiguate pointers.
- **Type-based AA**: based on the C99 property stating that pointers of different types cannot overlap.
- **Globals AA**: based on the fact that global variables whose address is not taken cannot overlap other pointers in the program.

- **SCEV AA**: uses *scalar evolution* of integer variables to disambiguate pointers. A variable's scalar evolution is an expression that describes the possible values it may assume during execution. Such information is extracted from loops. For instance, in the loop `for(i = B; i < N; i += S)`, variable `i` has the following evolution: $e(\texttt{i}) = I \times S + B$, I being the loop iteration. Regions indexed by integer variables overlap if their scalar evolutions have a non-empty intersection.
- **Scoped NoAlias**: preserves aliasing information, found by the compiler frontend, between different scopes in the program. This is rather important to achieve a limited form of inter-procedurality of the alias analyses, which are primarily all intra-procedural in LLVM.
- **CFL AA**: this is the most refined [Zhang et al. 2013] alias analysis among the LLVM implementations. However, it is also the most expensive in terms of computational cost, reaching $O(n^3)$ in the worst case.

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

4. Tutorial

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

4.1. Web tool

The web tool is a simpler demonstration of the Restrictifier's features. As such, it accepts only one C or C++ file as input, which can be either uploaded or have its contents inserted in a form. Given that it works on a single file as an independent unit, this file should contain (i) the functions to be optimized, i.e. functions that receive pointer arguments, and also (ii) their calls. Failing to comply to both conditions will naturally undermine our optimizer.

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

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

Having the optimized LLVM assembly file, you need to have it integrated to your project. Before linking it with the other modules of your C/C++/Objective-C program, you need to compile it for your specific machine. For this, you ought to have installed

the LLVM binaries. These can be acquired by either building LLVM yourself (our page has directions for this), or by installing them using a package manager of your operating system:

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

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

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

And finally compile the native assembly to an object file.

```
g++/clang++ module.s -o module.o
```

4.2. Local use

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

In order to use Restrictifier on your own computer, you need to compile it. Our tool's source code is available at our web page, and it is compatible with LLVM 3.6. To use our tool locally, you are required to build LLVM yourself. A very good guide for this can be found in LLVM's tutorial[1]. We recommend that you also build Clang, the C/C++/Obj-C frontend, alongside with LLVM[2]. After building LLVM and Clang, it's the turn of building Restrictifier itself. This process is the same as building any custom LLVM pass. As such, the LLVM's custom pass tutorial[3] can help you.

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

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

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

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

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

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

```
opt -mem2reg -cfl-aa -libcall-aa -globalsmodref-aa -scev-aa
   -scoped-noalias -basicaa -tbaa
```

[1] `http://www.llvm.org/docs/GettingStarted.html`
[2] `http://clang.llvm.org/get_started.html`
[3] `http://llvm.org/docs/WritingAnLLVMPass.html`

```
-load PATH_TO_Restrictifier/AliasFunctionCloning.(so|dylib)
   -afc -O3 -disable-inlining
-o program.optimized.bc
```

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

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

What is left to do is just translating it from LLVM assembly to native assembly:

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

And assemble it:

```
g++/clang++ program.s -o program.exe
```

Now you have an optimized version of your program.

5. Use case: prefix sum

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

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

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

We applied the Restrictifier onto this example and assessed the results. For this small experiment, we used a computer with an Intel Core i5 1.6 GHz processor 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.

From the Table 1, we clearly see how much the program earned from using Restrictifier. The addition of "restrict" to the arguments of function *prefix* enabled the reduction of the quantity of memory accesses and also the loop's vectorization. These

```
1  void fill(int *array, int N) {
2    int i;
3    for (i = 0; i < N; i++) {
4      array[i] = rand();
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 }
```

Figure 3. Use case of Restrictifier: prefix sum.

optimizations, only possible because of our tool's work, permitted our use case program to achieve 3.2x of speedup for this input.

We also state that Restrictifier takes a negligible time to run. To illustrate this, we applied our optimization on the largest benchmark from SPEC CPU2006 [Henning 2006], called `403.gcc`, which contains 235,884 lines of code. Restrictifier took 0.67 seconds to run, which is less than the time taken by Global Value Numbering, 3.7 seconds. The latter is an important optimization commonly performed by compilers to eliminate redundant code.

6. Conclusion

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

Restrictifier can be found at:

`http://cuda.dcc.ufmg.br/restrictification/`.

References

Henning, J. L. (2006). SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17.

Hind, M. (2001). Pointer Analysis: Haven't We Solved This Problem Yet? In *PASTE*, pages 54–61. ACM.

Landi, W. and Ryder, B. G. (1991). Pointer-induced Aliasing: A Problem Classification. In *POPL*, pages 93–103. ACM.

Whaley, J. and Lam, M. S. (2004). Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI*, pages 131–144. ACM.

Wilson, R. P. and Lam, M. S. (1995). Efficient Context-sensitive Pointer Analysis for C Programs. In *PLDI*, pages 1–12. ACM.

Zhang, Q., Lyu, M., Yuan, H., and Su, Z. (2013). Fast Algorithms for Dyck-CFL-reachability with Applications to Alias Analysis. In *PLDI*, pages 435–446. ACM.