EECS 583: Profiling Pointer Aliasing

Morgan Borjigin-Wang,
Louis Gouirand, Matt Qemo

Abstract

Static dataflow analysis enables many powerful optimizations. However, statically tracing data through a program can become difficult or even impossible when that program uses pointers. Static alias analysis lessens the uncertainty introduced by indirect dataflow, and can help prove that certain optimizations are safe or unsafe. However, static alias analysis is generally restricted to identifying whether two given pointers "may", "must", or "never" alias with one another. These possible analysis results are very coarse-grained, and naturally support conservative optimizations. In this project, we aim to support more aggressive speculative optimizations by using runtime profiling to provide fine-grained alias analysis information. We implement our alias analysis using two LLVM passes, and demonstrate optimizations through LLVM passes implementing pure function CSE and speculative LICM.

1. Introduction & Background

In C and C++, it is typical for programs to interact with their data through the indirection of pointers. These pointers offer a low-level, minimalistic interface on the memory of a program. By referring to the *address* of a certain variable, they offer advantages like runtime allocation and reduced copying. On the other hand, pointers can be difficult to work with and sometimes impose some overhead from the indirection, especially when it is multi-leveled.

One way to improve the runtime performance is to elucidate the pointer relationships throughout a program. For this project, we decided to look specifically at pointer aliasing. Two pointers are said to alias if they point to the same address in memory. Information on pointer aliasing opens up avenues for various optimization involving scheduling or speculation, some of which we explored.

Inspiration for this project stemmed from a 2004 paper on data dependence profiling [1]. In this paper, they described the advantages of looking directly at memory accesses to avoid potential false dependencies that pointer alias analysis might create. Particularly, these arise when aliased pointers refer to a `struct` or `class`, but reference distinctly separate data members of that class (Figure 1). Although their analysis shows data dependence profiling matching or outperforming pointer analysis profiling in their tested applications [1], it piqued our interest as a concept.

```
struct Line {

    int start_x, start_y, end_x, end_y;
};

Line* line1 = (Line*) malloc(sizeof(Line));

Line* line2 = line1; //line1 and line2 alias

line1->start_x = 3;

line2->end_x = 4; //but they are not data dependent
```

Figure 1. Alias analysis may sometimes highlight false data dependencies.

For this paper, a pair of pointers can fall under three distinct categories: must-alias, may-alias, and never-alias. While some must-alias and never-alias pairings are easily identified statically, the remaining pairings can only be categorized as may-alias. This limits the compiler's flexibility with those pairings. Due to runtime uncertainty, compilers are hesitant to optimize any pointers whose aliasing data can't be statically deduced.

One way the programmer can help the compiler here is through the `__restrict` keyword. Available in C and C++, `__restrict` denotes that a pointer does not alias in the current scope. But this only addresses the never-alias situation, and offers little in robustness. Oftentimes, a holistic understanding of pointer relations depends on the input of the user and requires a runtime analysis. In this project, we have developed a line of LLVM passes that illuminate the runtime behavior of a program's pointers, and use this profiling information to guide optimizations.

Specifically, we wrote four individual passes. The first pass instruments a program to record pointer values when it runs. The second pass reads from the log generated by the instrumented program and computes

alias data. The third and fourth passes use the alias data to perform specific optimizations, to help showcase the benefit of our alias analysis.

2. Motivation

The main motivation behind our examination of pointer aliasing is to improve runtime by leveraging runtime pointer profiling. A number of optimizations can be performed that cater to each type of alias relationship. For example, if profiling indicated that pointers `p` and q have a high probability of aliasing, and a pure function were invoked with argument `p` and then afterwards with `q`, we may be able to avoid the second (or subsequent) calls all together. On the other hand, we can take advantage of extremely unlikely aliasing for optimizations like speculative LICM. Regardless of which way the entries in a pointer alias profile lean, any kind of detected pattern from the profiled input has potential to be exploited in the name of performance.

3. Implementation

Our product is built upon four LLVM passes which were implemented in `profilepass.cpp`, `analysispass.cpp`, and `optimpass.cpp`. In addition to these passes, we created a number of short C programs to illustrate our project. These are named `simple.c`, `sqrt.c`, `classex.c`, and `licm.c`.

3.1 Profiler

Our first pass aims to generate a memory profile for a program. This will help us speculate on the relationship between pairs of pointers that can only be classified as may-alias by static analysis.

To find the relevant pointers to add to our profile, we look for load and store instructions throughout the program, and focus on the pointer operand they use. Because LLVM uses Static Single Assignment, we need only log that pointer operand's value (i.e. where it "points") when it's initialized. To perform this logging, we define a C function that is included in the input program. The function only takes 2 arguments: an ID and the memory address the pointer holds, and logs it to the relevant file (Figure 2). The ID is a way for us to map back the log line to the instruction it relates to. Additionally, implementing this functionality at the source level instead of at the IR level enables us to introduce complexity more easily and make future improvements easier.

```
1   12
2   0x7ffe1db4386c
3   5
4   0x601038
5   2
6   0x601048
7   13
8   0x7ffe1db43868
9   14
10  0x7ffe1db43864
```

Figure 2. The log file generated by the first pass.

3.2 Alias Analyzer

This second pass is run after our profile is collected and we are doing compilation for the final optimized program. Its role is to extract the relevant information from the profiler-generated log, and provide an interface for optimization passes.

Each line of our log essentially represents a point in time during the execution of the program. For this reason, when looking for potential aliases between pointers, we need only worry about the most recent value held by each pointer - previously held values are irrelevant. Our process is then to do pairwise comparisons between every pair of pointers using their most recent values as obtained from the log. We can then compute the probability of a pair of pointers aliasing in the profiled program by looking at how often in the log that pair of pointers contained the same address.

The interface we provide for future passes is therefore a function returning that very probability. This gives additional freedom to the optimization pass compared to LLVM's static alias analysis. Indeed, the static alias analysis only returns discrete values (must-alias, may-alias, never-alias), whereas our pass returns a double between 0 and 1.0, leaving the freedom of choosing the cutoff values to apply code transformations up to the optimization pass. The LLVM-level interface for our analysis pass mimics the standard interface for LLVM alias analysis passes, accepting llvm::MemoryLocation objects.

3.3 Optimizer

The final passes use the pointer profiling analysis to perform optimizations speculatively. It's important to highlight that the alias probabilities from the alias analysis are only as accurate as the profile input is. For this reason, the optimization pass should most typically perform its optimization speculatively and introduce fixup code in case the speculated relationship is not true at runtime.

This optimization can take many forms, and for this reason, it was not the focus of our project. However, in order to demonstrate the benefits our pointer profiling can provide, we provide example passes. An interesting aspect we will see is that we can enable optimizations by speculating that two pointers never alias, but also by speculating that they will always alias.

3.4 Optimizable Use Cases

In the following two sections, we will describe the optimizations which we implemented to demonstrate the use of our alias analysis pass.

3.4.1 Common Subexpression Elimination with Pure Function Calls

An example optimization we can provide is to look at pure function calls, specifically when the arguments to two different calls of the same function end up being equal because of pointer aliasing. Because the function is pure and therefore the output only depends on its input, the second function call can be seen as redundant since the return value will be identical to that of the first call (Figure 3).Using our pointer alias profiling, we can detect such cases and speculatively remove the second call.

The process for this is to look at the arguments to the function, and compare the values provided between the first and second call. If a value comes from a load, then the analysis pass may be able to detect if those loads frequently alias, i.e. if those arguments will have the same values. In the case that the alias probability reaches our optimizer's threshold for speculation, we alter the code to do a runtime check for whether the functions are truly redundant and execute the second call conditionally in a new basic block. We use LLVM's phi nodes to resolve the final result of the function call.

Assuming the threshold probability for such speculation to be performed on a function call is reasonably high, the added cost of our optimization is that of a conditional branch with a low mis-speculation rate, and that of a phi node. On the other hand we save a function call to a potentially very expensive pure function such as computing the n^{th} fibonacci number.

```
while (loopCount --> 0) {
  Num_t a;
  Num_t b;
  int i1rand = GetRandIdxSometimes(percent_likelihood_same, i1);
  int i2rand = i2;
  ConstructNum(&a, arr + i1rand);
  a.isInf = 0;
  ConstructNum(&b, arr + i2rand);
  double res_a = fn_PURE_(*a.valPtr);
  double res_b = fn_PURE_(*b.valPtr);

  printf("results are: %f %f \n", res_a, res_b);
}
```

Figure 3. Example program (`classex.c`) that could benefit from pure function common subexpression elimination. The alias frequency dependency between `a.valPtr` and `b.valPtr` varies based on the `percent_likelihood_same`, which is an input to the program.

3.4.2 Speculative loop invariant code motion with pure function calls

The second example we examined was speculative LICM for almost-invariant loads in loops. LICM allows loop-invariant code to be hoisted into the preheader of loops, which has the potential to drastically reduce the cost of some operations in the loop body. When an instruction is almost loop invariant (i.e. when it is loop invariant for most iterations of the loop), the same optimization can be leveraged speculatively, assuming correct insertion of fixup code for cases when the instruction is not loop invariant.

Our speculative LICM pass first finds all loads which are almost loop-invariant. It does this by collecting all loads in a loop which either have either a never-alias or almost-never-alias relationship with every other store in the loop body. We define almost-never-alias to be when our alias analysis pass statistics demonstrate a very low (< 2%) probability of aliasing. It hoists these loads to the loop preheader, and additionally hoists any pure function calls which then

become loop-invariant (perhaps because they depend solely on the hoisted load). Fixup code is then injected after every store which may-alias.

Our motivation for specifically targeting dependent pure-function calls was simplicity; we are able to leverage some of the code from our CSE optimization described in the previous section. Moreover, while it's very possible to implement an optimization which hoists additional instructions that become loop-invariant beyond just pure function calls, these optimizations are used merely as examples of the use of our alias analysis pass, and are not the focus of this project.

The performance gains for speculative LICM are very similar to those of CSE, as they are dependent on the probability of mis-speculation (whenever a store in the loop aliases to a hoisted load) and dependent on the cost of the hoisted pure functions.

4. Evaluation

We evaluated the benefits of our pointer alias profiler by creating toy program benchmarks for our two example optimization passes to run on. While those programs are very simple and delimit a smaller scope for our project than using more robust and diverse benchmarks, they are a proof-of-concept that runtime improvements can be made from our project.

4.1 Common Subexpression Elimination with Pure Function Calls

When running the `classex.c` program through our LLVM passes, we saw a direct correlation between alias frequency of the pointer used to call `_fn_PURE_` (Figure 4) and the optimized runtime. The code inserted by our optimization pass checks for pointer aliasing due to promising profile information. In the case that they alias with 100% frequency, we saw the runtime cut in half (Figure 5). Even when the test input has a never-alias relationship, the fixup code does not impose any substantial overhead since it simply makes the second function call in the way the unoptimized program would, with the addition of a branch instruction.

```
/* represents an arbitrary somewhat computationally expensive pure functi
double fn_PURE_(double x) {
        return sqrt((x * x * 3.1415926 + 3948.23891) / 27.5);
}
```

Figure 4. A pure function we used to demonstrate CSE.

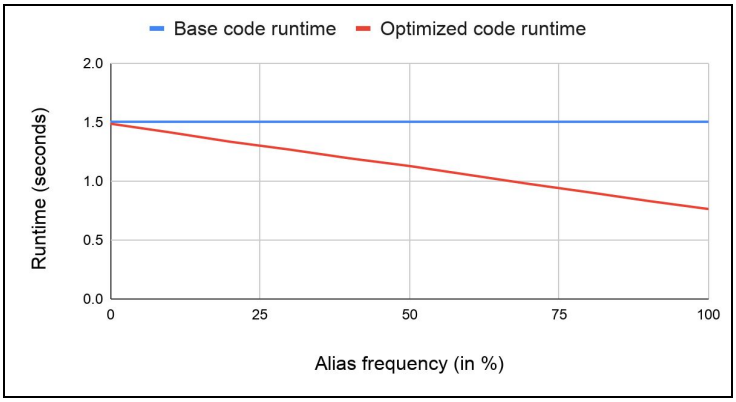

Figure 5. Runtime performance comparisons between unoptimized and optimized compilations of `classex.c`.

4.2 Speculative Loop Invariant Code Motion with Pure Function Calls

To provide an example of speculative LICM using pointer profiling, we created a simple program with a loop running many iterations (`licm.c`). The loop assigns a value to a varying index in an array. The value assigned each iteration is the output of a somewhat computationally expensive math function which performs 1 addition, 1 division, 1 multiplication and one call to sqrt(). The input to that pure function is almost loop invariant and thus, the result of the pure function is therefore almost loop invariant too. The rest of the program is very simple so a large proportion of the runtime is spent within the pure function.

The user controls the frequency at which the store aliases with the input to the pure function. We vary the aliasing frequency using this parameter and report our gains compared to the unoptimized version of the benchmark program (Figure 6).

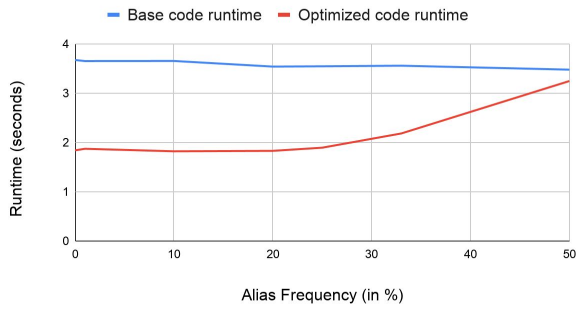

Figure 6. Runtime performance comparisons between unoptimized and optimized version of `licm.c`

We can see there are significant gains from our optimized code on alias frequencies less than 30%, almost a 2x improvement compared to the baseline. The reason we are able to get such significant improvements is due to the fact the benchmark program's runtime is heavily dominated by the pure function calls, which are specifically targeted by our speculative optimization.

Conclusion

By supplementing alias analysis with fine-grained profile information, we can allow extensive speculative optimizations relying on the runtime behavior of programs. In some programs, the simple addition of distinguishing between may-alias and likely-alias is sufficient to power very effective optimizations. The addition of profile information also helps power tuning, as application developers can make very fine-grained decisions about when and how aggressively to speculatively optimize.

Additionally, because of the interface similarity with existing alias analysis tools, profile-guided alias analysis can be easily integrated by compiler developers looking to leverage runtime information about pointer aliasing.

We anticipate that future extensions such as optimizing the instrumented code and building out more extensive use-cases will come naturally.

Availability

All source code can be accessed at https://github.com/mattqemo/eecs583fp

Works Cited

[1]
Chen, Tong & Lin, Jin & Dai, Xiaoru & Hsu, Wei-Chung & Yew, Pen-Chung. (2004). Data Dependence Profiling for Speculative Optimizations. 2985. 2733-2733. 10.1007/978-3-540-24723-4_5.

[2]
`__restrict` | Microsoft Docs [Online]. Available: https://docs.microsoft.com/en-us/cpp/cpp/extension-restrict?view=msvc-160

[3]
LICM.cpp [Online]. Available: https://llvm.org/doxygen/LICM_8cpp_source.html