Proceedings of the GCC Developers' Summit

June 22nd–24th, 2005
Ottawa, Ontario
Canada

Compilation Wide Alias Analysis

Daniel Berlin
IBM T.J. Watson Research Center
dberlin@dberlin.org

Kenneth Zadeck
NaturalBridge Inc.
zadeck@naturalbridge.com

Abstract

The intermediate code in a compiler can be divided into three broad categories: register operations, branching instructions, and storage operations. Register operations and branch instructions[1] have transparent semantics, i.e. they are easy to understand. Storage operations are more difficult to understand because of their underlying addressing computations. Because many different address expressions can map to the same address, care must be taken when either reordering or removing memory operations to assure that the source program still computes the same result.

While there is a one-to-one mapping between the register operations and values computed in the program, no simple relationship exists for the storage operations. Loads and stores typically reference sets of memory locations, and two seemingly independent address expressions may evaluate to the same storage location. Thus, while it is fairly straight-forward to reorder register operations, the ability to reorder storage operations is limited by the ability to reason about the addresses that may be created in the reordered computations. The analysis of the computations that produce the addresses is called *alias analysis*.

In the absence of alias analysis, each write operation becomes a *barrier* that no other memory reference can cross. Barriers inhibit many optimizations, especially those that attempt to either remove redundant computations, or move operations to locations where the execution is less costly.

We present both an alias analysis phase as well as several transformations that make use of this information. The paper concludes with some simple measurements as well as directions for further work.

[1]with the exception of computed goto statements

1 Alias Analysis

The literature contains a large number of papers that describe techniques for alias analysis. There are many parameters that can be used to characterize the algorithms:

scope How much of a program can be analyzed at one time? Common values range from small regions, like single loops to entire programs.

flow sensitivity Does the algorithm attempt to understand the control flow of the program?

In a *flow-sensitive* algorithm, information is propagated along the control flow graph edges. In a *flow-insensitive* algorithm, the ordering of the statements is ignored.

Many algorithms that have their scope limited to a single function or smaller can afford to be flow-sensitive. It is unusual to find flow-sensitive interprocedural algorithms that can be performed fast enough to be used in a compiler.

unit of analysis What is the smallest unit of analysis? For many algorithms, each variable is analyzed independently. For others, all the variables of the same type are aggregated together.

The choice of intermediate representation can make a difference here. If the unit of analysis are variables, translation to static single assignment (SSA) form can be of great benefit. SSA form provides many of the benefits of a flow-sensitive algorithm when using a flow-insensitive framework without incurring the cost of flow-sensitivity. This is because SSA form encapsulates many aspects of flow into its representation.

records and structures Does the algorithm assume that an assignment to a structure may modify all of the fields of the structure or can each field be modeled separately?

arrays and lists Does the algorithm assume that an assignment to an array may modify all of the elements of the array or can each element, or groups of elements, be modeled separately?

type system Does the algorithm take into account any restrictions that the type system or inheritance system places on acceptable programs?

time and space complexity How much time and space are used to compute the result? The limiting factor of many techniques is the time complexity.

Understanding how the parameters interact is important to understanding what aliasing can do. Most techniques have a sweet spot, where they provide useful information at a reasonable cost. But if you try to push the technique beyond that point, the algorithm may become prohibitively expensive.

For this reason, most compilers implement several different alias algorithms, each providing a different kind of information useful for a different set of transformations. For instance, in GCC, the scalar evolution pass is flow-sensitive and only deals with arrays and index variables; however, its scope is limited to loops that have very restricted structure. When outside this particular domain, the algorithm provides no useful information about the variables.

1.1 New Alias Analysis for GCC

There are two approaches we could have taken to decide what to add to GCC: a top-down approach: search the literature for the latest hot algorithm or a bottom-up approach: address some obvious shortcomings in GCC. We decided on the latter approach.

The shortcomings we decided to address are:

- All of the aliasing in GCC is only done on small units of analysis, such as basic blocks, single loops, or at most, single functions. This narrow focus is historical, GCC was originally a line-at-a-time compiler that has since evolved into function-at-time-compiler. However, C, C++, Java, and ObjC are file oriented languages where a single file may contain many closely related functions. There are three file level targets to look at:
    - Most files contain a significant number of local functions. Aggressive inlining eliminates the need for some

of the information that interprocedural alias analysis could provide. However, it is not desirable to inline everything, so opportunities for more precise alias information may be lost.

  - There are a significant number of variables that are defined at the file-local level. A file-at-a-time analysis allows these variables to be analyzed in a limited closed world framework.

    Function-at-a-time compilation forces these variables to be handled as if they were global variables with a localized name. However, file-at-a-time analysis, for those static variables that do not escape the file, analyzes their entire lifetime before the functions that use them are compiled.

  - There are a significant number of types whose scope can be determined to be local to the file. A file level approach to any variable of one of these types provides limited closed world analysis for these variables.

- Structures are handled poorly. At least for the purposes of aliasing, an assignment to any element of a structure is modeled as a kill of the entire structure.[2]

The underlying analysis algorithm in our approach is rather straight forward. However, the implementation required significant engineering because the correctness of this analysis depends on seeing the *entire compilation unit* at one time. While this is routine in most production compilers, it had never been attempted in GCC and required significant re-engineering of several components to accomplish this goal.

[2]This problem has been addressed in 4.1.

1.2 The Algorithms

There are five kinds of analysis that are performed by this pass. All of them share a single scanning pass of the intermediate code and declarations. Data structures are built during the scan, and in some cases the further analysis is performed on the data structures.

The scope of our framework is an entire compilation unit. The method of attack we have chosen is to use flow-insensitive interprocedural techniques, and apply them to variables as well as types. Several of the problems deal explicitly with structures and inheritance but in all cases we aggregate arrays into single blobs. The worst case time and space are associated with number of functions times the number of static variables, which is easily a manageable quantity.

1.2.1 Addressable and Readonly Analysis for Static Variables

GCC associates an `addressable` bit with each variable. This bit is set whenever the address of the variable is actually needed. For variables where this bit is clear, more aggressive transformations can be performed. In the past this bit has only be considered valid for local variables since it was not possible to see the entire usage of a variable with larger scope.

By analyzing an entire compilation unit before the individual functions are compiled, all of addressing operations associated with static variables can be seen. Any static variables that have no addressing operations have their addressable bit cleared. This enables the compiler to determine that the address is not needed, allowing further optimization of these variables.

Likewise, we also track assignments to static variables, and any static variable that is not ad-

```
static int foovar;

void func1()
{
  return foovar + 6;
}

int bar()
{
  func1();
  return foovar;
}
```

Figure 1: Example of static variable that is neither addressed nor clobbered

dressable and has no stores to it aside from the static initialization are marked `readonly`.

All of this information can be gathered with a single scan of the entire compilation unit. None of this requires any type of propagation or deep analysis.

1.2.2 Type Escape Analysis

Types, especially complex ones, can be analyzed in a manner similar to what is done with static variables. If no instance of a type `t` escapes from the compilation unit, it is possible to improve the compilation of all instances of `t`.

There are many ways that a type `t` can escape a compilation unit:

- An instance of a pointer to `t` can be a parameter or return type of a global function.
- An instance of a variable of type `t` can be a global variable.
- A pointer to `t` can either be in a cast where the other type is an opaque or general type (such as `char *` in C).
- An operation other than plus, minus or times is applied to the pointer to `t`.
- A field is declared of type `t` in a record or union with a type that escapes.

As with the pointer escape analysis, most of the processing is a single pass that examines the entire compilation unit. This is followed by a recursive walk of all of the type trees. Note that this analysis differs in a significant way from the addressable analysis in Section 1.2.1. In the analysis here, a type does not escape if an instance of the type has its address taken, it only escapes if that address escapes the compilation unit or is cast to or from escaping type.

Two very important special cases are made with respect to the flow-insensitivity of this algorithm. These are flow-sensitive processing, but limited to flow within a single basic block:

- The result of any function with the `ECF_MALLOC` flag set is exempted from the cast rule. This flag corresponds to functions that are like `malloc`, and return new memory.
- The parameters passed to functions with the `ECF_POINTER_NO_ESCAPE` flag set (such as `free`) are exempted from the cast rule. This flag corresponds to functions that guarantee the pointers will not escape.

These very common cases allow objects to be allocated and freed without being marked as escaping.

1.2.3 Structure Analysis

GCC currently makes very pessimistic assumptions about accesses to structures, records and unions. GCC does not make effective use of type and structure layout information in deciding if a store to one structure or field can effect a pointer to another record.

There is a very powerful assumption that is consistent with the language semantics of C, C++ and ObjC[3] that make it possible to be aggressive about disambiguating structure references.

> Assume that you have a pointer `p` to object `o`. There are a limited set of operations that can be performed on `p`. Namely `p` can be used to reference anything within `o` but it cannot be used to reference objects before `o` or after `o` in memory.
>
> Thus, if the address of a field `f` is taken, no other fields at the same level or at outer levels within that record can modified through the pointer `&f`.
>
> However, if `f1` is itself a record or union within `f`, the pointer `&f` can be used to access any fields declared within `f1`.

This forms the basis of a simple analysis: mark a field `f` if it has its address taken within a record. If `f` or any of its containing records do not have their address taken, then stores through pointers of type `tf` (where `tf` is the type of `f`) cannot modify any record that contain `f`.

There are two boundary cases:

- If type `t` escapes, all fields within `t` escape.
- If type `t` escapes, all supertypes of `t` escape.

[3]Since Java and Fortran do not allow address to be explicitly taken, they are free of any of these problems.

As with the previous analysis problems, this algorithm consists of scanning of all of the operations in the compilation unit followed by modest amount of post-processing. The data structure used is similar to the current sets in `alias.c` with two exceptions:

- For this analysis, structure and union types are only represented if they do not escape. If they do escape, the analysis falls back on the existing code.
- For this analysis, for the types that are represented, the only fields listed as conflicting within the type record are those who have their address taken (and their subtypes). In `alias.c`, all fields that are accessed in the current function are listed and are assumed to cause conflicts.

These differences are significant: `alias.c` pessimistically assumes that every field that is referenced can cause a conflict, unless proven otherwise. Here, we optimistically assume that a field can cause a conflict only if there is evidence of the address being taken or the record type escapes the compilation unit. This provides for a much smaller potential set of conflicts.

1.2.4 Call Side Effect Analysis

For functions within the compilation unit, it is possible to summarize some of the effects that a call to that function *may* have, namely which static variables may be read and which may be modified. Only static variables that are not addressable are considered for the analysis.

The algorithm consists of four phases:

1. A scanning phase which processes all if the intermediate code and declarations.

The output of the phase is a pair of bit vectors for each function. The first bit vector describes the set of variables locally read and the second describes the set of variables locally stored. In addition, there is a bit vector that contains the universe of static variables that this analysis is applied to. Any static variable that has its address taken or has the used attribute is excluded from this set.

2. An analysis of the call graph of the compilation unit. The structure of the information being propagated through the call graph is such that any information that reaches any node that is in a cycle in the call graph reaches all nodes in that cycle. Thus, it is profitable to use Tarjan's depth first cycle reduction first to build a derived acyclic graph.
3. Propagation of the information produced in the first step along the derived graph of the second step. This produces two bit vectors for each node in the reduced graph: one for the variables that may be read and another by the set of variables that may be written by executing a call to one of the functions represented by that node.
4. Expansion of the information into a form that is acceptable to the rest of the compilation unit. All of the bit vectors are stored with bits indexed by the `DECL_UID` field. However, the APIs used at the SSA level are based on the `var_ann UID` field.

Steps 2 and 3 are required because it is not enough to just understand the potential side effects of an immediate callee, `a`, a correct picture can only be had processing all of `a`'s callees and, recursively, the callees of `a`'s callees. For this analysis, the more functions that are contained in the compilation unit, the more precise the information will be.

Of course most compilation units do not contain the entire program so conservative approximations must be employed for the parts of the program that are not available. Calls to functions outside the compilation unit fall into two categories, ones whose side effects are understood, and everything else. Calls to the first category generally do not effect the side effects since those that can possibly call back into the compilation unit are excluded. For the everything else category, it is assumed that a call to any of these can read or modify all static variables. The need for such a pessimistic assumption is that it must be assumed that there is a call path from the external function being called to every globally visible function in the compilation unit. Functions are globally visible if they are *extern* or have their address taken.

It is interesting to note there are a set of standard library functions that fail the well known and understood test. The obvious ones are `bsearch` and `qsort`. However, the GNU C library contains obscure extensions that disqualify a large number of functions from this special treatment. The standard IO functions like `printf` allow callbacks to be registered for formatting certain data types. Since these callbacks can, in general, call any function and reference any variable, this analysis cannot take advantage of these common functions. In practice, this extension severely limits the usefulness of the analysis. That said, this analysis improves regular examples such as Figure 1 because the compiler no longer believes that `func1` clobbers `foovar`.

1.3 Pure and Const Function Detection

With all of the scanning machinery in place it is not difficult to detect `pure` and `const` functions during scanning.

Pure functions are functions which depend only on the arguments passed, and reading from memory (they cannot write to memory).

Const functions are a subset of pure functions which are only allowed to read from readonly memory.

Currently this detection is done at the RTL level. The processing is much the same as was previously done at the RTL level with one exception; Since all the processing is done at once, it is possible to remove all dependencies on function ordering on the processing: recursive functions are handled properly and their are no cases where a function is referenced by another function before it is marked as being `pure` or `const`.

2 The Engineering

The difficult part of the implementation was convincing the existing front ends of the compiler that there had to be one specific time when absolutely all functions and variables were available. Prior to implementation, the C++ frontend would often create variables or functions in the middle of compilation, and later shoehorn them into the callgraph. The correctness of this analysis depends on seeing *everything* because, for instance, if the one missing fragment is the one place that takes the address of a variable, that address has still been taken and can be spread throughout the entire program.

The task of making the front ends give up all of the secrets up front fell on Jan Hubicka of SuSE Labs. He had been implementing an interprocedural pass analysis (IPA) framework to facilitate CFG-aware inlining and was interested in more clients. As with many things in GCC, we did not understand the magnitude of the hiding problem until we were both well into implementation. As it turns out, inlining is not a particularly good test to determine if you are really getting the front ends to behave because the inlining will be correct even if you miss a few functions or variable declarations. For more sophisticated analysis, promises made must be kept.

However, Jan, with the help of many other people was able to get all of the front ends to behave.

2.1 Whole Program Analysis

In theory these analysis algorithms would work with no further modification when applied in *whole program mode*. In this mode, all of the modules are compiled at one time. Variables and functions that are declared global in a normal computation are converted to static variables.

In practice whole program mode still needed a lot of engineering. The problem is the way types are represented at the top level, especially in C and C++, which do not have a formal module system. It is perfectly legal to have to two types declared with the same name but in some modules the types have one representation and in other modules have a second representation. What is missing from whole program mode is a pass that performs type unification across all the instances of the types. Without such a mechanism, the whole program mode likely contains many latent bugs.

For this analysis, we implemented the most rudimentary type unification system. This was sufficient for our purposes, but should not be considered a blueprint for what is really needed, in that if two types do not unify, the system just assumes that there is no information available about the variables of those types.

2.2 Enhancements to the Analysis

The analysis algorithms discussed above are quite simple. All involve only a scanning of the intermediate code followed by some simple data reduction. There are ways of enhancing them:

SSA Form All of the problems discussed above could benefit by performing constant propagation and dead code elimination before applying the analysis. In the context of the existing GCC infrastructure this involves converting the intermediate code to SSA form before applying the analysis.

However, it is difficult to have all of the functions in SSA form at the same time because of an unfortunate data structure decision. There is a constraint that the `var_ann UID` field be densely packed for each function. The current SSA builder does this, but in the process renumbers this field for the static and global variables also. Thus, SSA form, with its current datastructures cannot exist simultaneously for all functions in the compilation unit. This will be fixed in either 4.1 or 4.2 which will allow a richer set of analysis and transformations to be done at the IPA level.

Types v.s. Values Type escape analysis (Section 1.2.2) and structural analysis (Section 1.2.3) could both benefit by performing the analysis over each individual variable. It is not always true that all of the instances of a type are connected and partitioning the variables into disconnected components could allow better information to be obtained for some partitions that assuming the worst for all partitions.

This is clearly true for some of the SPEC2000 benchmarks. Some of them contain places where a variable of a pointer type `pt` is converted from `pt` to a `char *` and then back a different variable of type `pt`. There are no bad operations performed on the value while it is a `char *` and this could be easily detected with a flow-insensitive algorithm, especially one run over the intermediate code in SSA form.

While somewhat odd at the source level, this kind of casting code may be common after inlining. If a variable of type `pt` is passed to a function expecting a `char *` and that function is inlined, then the same code would exist (depending on what the function body did to the variable) as in the SPEC benchmarks.

3 Transformations

We have implemented a number of transformations based on this analysis. There are many other possibilities.

3.1 Call Clobber Analysis

Using the side effect information that we obtain in Section 1.2.4, we are able to remove many static variables from the call clobber sets of function calls. This has required a certain amount of reengineering of the `tree-ssa-operands: add_call_clobber_ops` since the recently added cache was not call site specific.

3.2 Static Variable Promotion

In the past it has not been possible to promote a static variable to a register. Each load or store to the value has required a memory reference.

Under certain conditions this is neither necessary nor desirable. The side effect information in Section 1.2.4 provides enough information to decide if it is safe to promote a static variable to a register. It also provides enough information to determine at which function calls the variable must be stored back before the call or reloaded after the call.

The only other remaining issue is if it profitable to promote a static to a register:

- Statics with the `readonly` attribute are not promoted since it was considered better to simply teach constant propagation to replace the loads with references to the constant value.
- Arrays are not promoted because the cost of loading or storing the large aggregates makes them not profitable in almost all cases. We have considered adding code to the scan all of the references and if the only constants are used for the indexing to consider promotion.
- Non-readonly scalars are always promoted.
- Aggregates whose type passes `tree-sra: sra_type_can_be_decomposed` are promoted.

Promotion code for each variable `v` may be inserted into seven contexts in function `f`:

- At the entry of the function, `v` is loaded into virtual register `r`.
- The code that loads `v` is replaced with code that copies from `r`.
- The code that stores `v` is replaced with code that copies to `r`.
- If there were any stores to `v` in `f`, `r` is stored to `v` before any function call.
- After any function call that may modify `v`, `r` is reloaded from `v`.
- If there were any stores to `v` in `f`, `r` is stored to `v` before any return.

After this promotion code is added, SSA form is then built for the registers. The cases cover the entire function `f`, not just where the `v` was live. An enhancement to dead code elimination removes the live ranges that were not necessary.

3.3 Enhanced Alias Analysis

A function has been added which takes a record type and a field type, returns true if the store to a pointer to the field type can not clobber the record type.

This has been integrated at the tree-ssa level in `tree-ssa-alias.c: may_alias_p` and at the RTL level in `alias.c: nonoverlapping_memrefs_p`.

No changes to the underlying representation of aliases has been done, nor have any of the clients of alias analysis been modified in order to support this enhancement.

4 Results

The performance for the call clobber analysis and static promotion have been modest. They essentially provide slightly better code but the situations where either is a critical problem are rare. Both of the optimizations are crippled because of the *enhancements* to `libc`. Most large software projects contain debugging code in a large percentage the functions. Having to make the most pessimistic assumptions about such code because of these rarely used enhancements seems like something that should be considered.

4.1 Results for Enhanced Alias Analysis

The enhanced alias analysis currently has a limited effect at the SSA level because it has been difficult to fit this kind of information into the alias sets of the current compiler. This will change as the underlying representation is enhanced.

At the RTL level, the story is completely different. Turning on the alias analysis has an effect of ±20%. Small compilation units tend to improve and large compilation units tend to degrade. On standard benchmarks, such as Spec2000, some benchmarks improve significantly: `171.swim` improves 10%, `179.art` improves 5%. The rest are a mixed bag. Some improve by 1%, some degrade by roughly 1%.

While at first glance this may seem problematic, it is actually really good news. Poor analysis tends to make a compiler behave as if it is a machine with a very viscous lubricant. As the aliasing improves, the machine parts are able to spin faster and accomplish more. GCC was developed with either no or very poor aliasing information available. It is not surprising that the transformation that depend on aliasing make globally poor decisions when suddenly given more freedom to work than they were designed or tuned for.

Particularly hard hit are scheduling, register allocation and spilling. At no point does the scheduler have any understanding of how many registers the machine has or how many it is currently using. It simply tries to separate the loads from their uses without any understanding of the consequences. The register allocation is also a problem because there is no rematerialization available to reduce register pressure and the spilling choices are generally not good.

For small programs this is not a problem because, even with very good information, there is simply not enough code to pile on a lot of live ranges. However, larger compilation units provide a lot of opportunities for procedure inlining to bulk up the inner loops with a lot of instructions that can be moved around. With no controls on how much to do, it is inevitable that better information produces poorer results.

5 Conclusions

The future of the alias analysis described here is not clear. Danny Berlin is working on a more precise algorithm which tracks values rather than types. This should provide still better alias information. Until the tree-ssa alias representation is enhanced, it is unlikely that Danny's algorithm will show much at the tree level; at the RTL level the better aliasing may exacerbate the problems discovered with this more modest analysis.

However the cost of that analysis has yet to be measured. If the time and space are close to this algorithm, then a few of the edges that are not covered in that algorithm will be mined from this algorithm and the code will be abandoned. However, it is more likely that this will prove to be much cheaper (since this just consists of a scan of the program followed by a modest number of bit vector operations) and that this will become the default technique at `-O1`.

What is clear is that the RTL level squanders a lot of performance because it does a bad job of managing the resources available on the machine. When this is fixed, we should see substantial improvement.