Enhancements to Aliasing

Danny Berlin, *Esq*.
IBM T.J.Watson Research Center
dberlin@dberlin.org

Kenneth Zadeck
NaturalBridge, Inc.
zadeck@naturalbridge.com

Roadmap

Analysis Enhancements

- Call side effect analysis.
- Read-only and non-addressable variable detection.
- Pure and const function detection.
- Type based alias analysis.

Transformation Enhancements

- Promotion of static variables.
- Refinements to call clobbering.

Interprocedural Analysis

- What is new for GCC is that these analysis passes are applied to an entire compilation unit.
- In the past, the only real way to do compilation unit wide analysis was in the front end.
  - The front ends lied about what was the whole program.
- Changing this required a lot of work.

•Jan Hubicka

•Steven Bosscher

•Dale Johannesen

•Stuart Hastings

The world is now safe for doing interprocedural analysis.

Call Side Effect Analysis

- Determine which static variables *may* be read or written as a side effect of a call.
- Requires:
  - Local summary of static variables read and written by each function.
  - Complete and correct call graph.

Call Side Effect Analysis (Technique)

1. Compute local read and write information for the variables whose scope is contained in the compilation unit.
2. Collapse cycles in call graph.
    - All nodes within a cycle share same information.
3. Assume worst case for functions outside of compilation unit.
    - All statics are read and written.
    - Calls may call back into compilation unit.
4. Propagate reads and writes along call graph edges.

Call Side Effect Analysis (Problems)

- Must assume that any function that is not seen can call back into the module being compiled and thus, get access to the static variables.
- For a *Standard* C library you can do better (except qsort and bsearch).
- GLIBC is not standards conformant.
- Extensions added to printf mean that any function with some debugging code in it causes worst case assumptions to be used.

Read-only and Non-addressable Variable Detection.

- Small amount of additional code in the Call Side Effect Analysis pass.
- This is also done in the front ends.
- We still find more than the front ends do.
  - Are these front end bugs?
  - Do we want to fix each of the front ends?
  - Does it make sense to have the front ends gather this information?
- This pass is run at -O2 and above for C and at -O1 and above for other languages.

Pure and Const Function Detection

- Replaces the phase done at the RTL level.
- Many positive differences in results:
  - Does not get confused by profiling code.
  - Handles recursive functions correctly.
  - Does not get confused by low level RTL constructs.
- One (current) regression:
  - Misses some cases because constant propagation and dead code are not run before the detection step.

Type Based Alias Analysis

Simple Idea – If the *address is never taken* for any instance of type T, then no instance of type T can alias anything else.

Complex Problem – What does *address is never taken* really mean?

The answer depends on the language, and the way that one interprets the languages specification.

This analysis is one of the sanity tests for the structure reorganization.

What Does *address is never taken* Mean?

structs.h:

```
struct A {int aa;};
struct B {float bb};
```

Module A:

```
#include “structs.h”
struct A a1 = get_a();
struct B b1 = get_b();
```

Can a1 interfere with b1?

What Does *address is never taken* Mean?

structs.h:

```
struct A {int aa;};
struct B {float bb};
```

Module A:

```
#include “structs.h”
struct A a1 = get_a();
struct B b1 = get_b();
```

Can `a1` interfere with `b1`?

Module B:

```
#include “structs.h”
union X {
    struct A a;
    struct B b;
};
static union X x = xalloc …
struct A get_a() {
    return x->a;
}
struct B get_b() {
    return x->b;
}
```

What Does *address is never taken* Mean?

```
struct C {
    int d;
} c1;
struct B {
    struct C c;
} b1;
struct A {
    struct B b;
} a1;
```

What can be done with `x` in `x = &a1.b.c`?

- Any fields in `c` may be accessed.
- Offsets can be added to `x` to access any field in `a` or `b`.
- However, if no bad operations are done to `x`, none of the types `A`, `B` or `C` need escape.

What Does *address is never taken* Mean?

- Taking the address of something does not cause the type to escape.
  - Using the pointer to access sub-fields is fine.
- Using the pointer in a bad way causes the type, and many connected types, to escape.
  - Doing math with the pointer.
  - Upcasting with the pointer.
  - Passing it to an external function.

Type Based Analysis Algorithm, Part I – Scan the Code

- Record the type of all structure *address of* operations.
- Build a table of all of the types seen in the compilation unit.
- Mark the types as escaping if:
  - a pointer to that type appears as a parameter or return type of a public function.
  - the type is a public variable.
  - the type appears in a *bad cast* or pointer arithmetic operation.

Type Based Analysis Algorithm, Part II – Transitive Closure

- Flow insensitive analysis.
- The closure is performed over the type system.
- If a type X escapes:
  - x’s subtypes escape.
  - x’s supertypes escape.
  - the types of X’s contained fields escape.
  - the types w containing x if there was a pointer operation of the form &...W.X...

Type Based Analysis Algorithm

- In practice, there are three limiting factors:
  - The algorithm is flow insensitive.
  - The types that escape across module boundaries.
  - Poor representation of aliases at the tree level.
- Malloc an free are special cased to keep these from killing everything.
  - Abstracted version of these functions still cause problems.
- In whole program mode this is algorithm is very effective since taking the address of fields of a structure is rare.

Type Based Analysis Algorithm

- Originally motivated for structure reorganization.
- Not meant as a replacement for points to analysis.
  - Type analysis has severe limitations since it does not track instances.

Transformation Enhancements:

Promotion or Static Variables, Part I

- Static scalar variables and structures are promoted if their types can be "scalarized" by the SRA pass.
- Arrays and constant variables are not promoted.
- The side effect analysis code provides information about which variables must go back into memory when crossing call sites.
- Promotion occurs before SSA form is built.

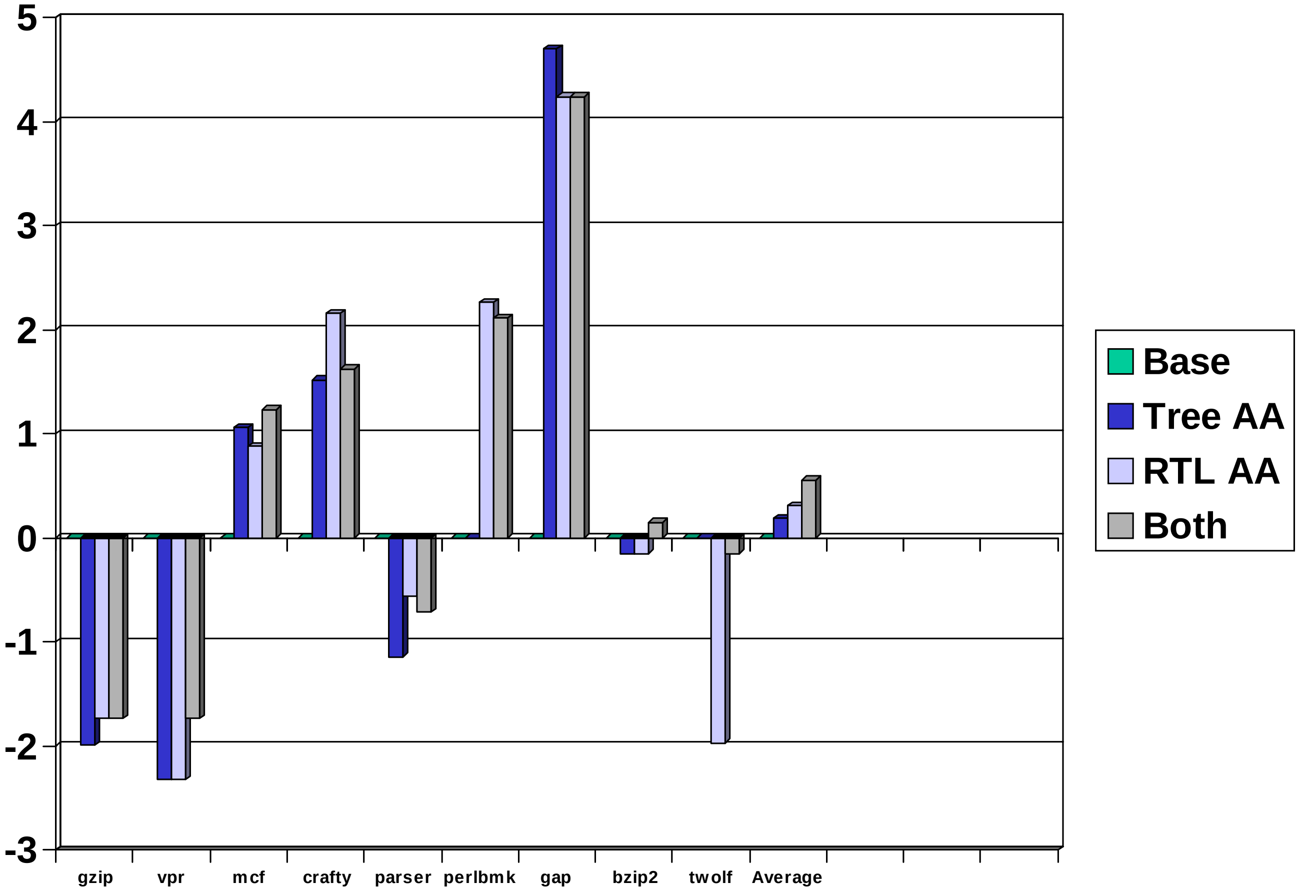
Transformation Enhancements:
Promotion or Static Variables, Part II

- Loads are inserted:
  - at the top of the function.
  - after calls that may modify the variable.
- Stores are inserted (if the value is modified)
  - at returns.
  - before calls that may read or modify the variable.
- A special enhancement to dead code elimination removes these variables where they are not live.

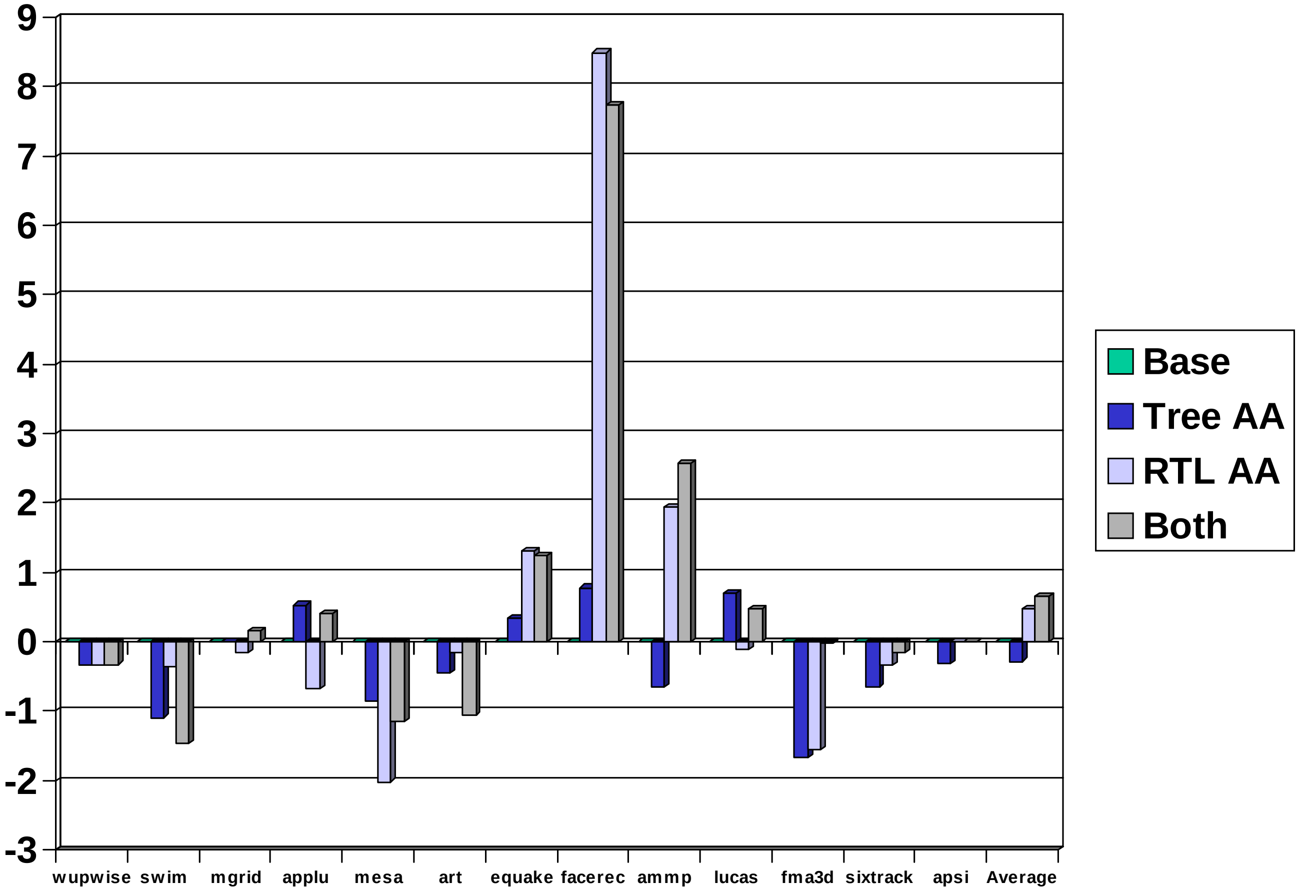
Transformation Enhancements: Refinements to Call Clobbering.

- Reduce the number of variables listed as being call clobbered.
- Call site specific (must redo the caching).
- The side effect analysis code provides information about which static variables may be modified by a specific call.

Spec 2000 Integer Percentage Improvements



Spec 2000 Floating Point Percentage Improvements



Conclusions

- This is the first round of interprocedural analysis phases to be added to GCC.
- Most of these only provide modest improvement when compiling a single module.
- Occasionally some trigger big changes.
- Many times, the improvement is lost because the analysis overwhelms downstream transformations.