Alias-free Parameters in C for Better Reasoning and Optimization

Medhat G. Assaad and Gary T. Leavens

TR #01-11
November 2001

Keywords: Aliasing, multibody procedures, dynamic dispatch, static dispatch, program verification, alias-free programs, compiler optimizations, ACL approach, C/ACL language, C language.
2000 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — formal methods; D.3.3 [*Programming Languages*] Language Constructs and Features — control structures, procedures, functions, and subroutines; D.3.4 [*Programming Languages*] Processors — compilers, measurement, optimization, performance; D.3.m [*Programming Languages*] Miscellaneous — dynamic dispatch, multiple dispatch; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — logics of programs; F.3.3 [*Logics and Meanings of Programs*] Studies of Program Constructs — control primitives, functional constructs.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Alias-free Parameters in C for Better Reasoning and Optimization

Medhat G. Assaad*
Department of Computer Science
Iowa State University
226 Atanasoff Hall, Ames, IA 50011 USA
+1 608 271 9000
medhat_assaad@yahoo.com

Gary T. Leavens
Department of Computer Science
Iowa State University
226 Atanasoff Hall, Ames, IA 50011 USA
+1 515 294 1580
leavens@cs.iastate.edu

ABSTRACT

Aliasing among formal parameters and among formals and globals causes problems for both reasoning and optimization. Whole-program static analysis could provide some knowledge about such aliasing, but this is not usually done, and in any case would have to be conservative. All aliasing patterns that are not ruled out by an analysis have to be considered possible both by a person reasoning about correctness and by a compiler trying to optimize code. For compilers, the conservative nature of the static analysis leads to missed optimization opportunities.

We have designed and implemented a small extension to C that partially solves the reasoning problem and leads to significantly better optimization. The extension guarantees that there will be no direct aliasing among arguments and globals inside procedure bodies, and yet allows aliasing among arguments and globals at the call site. This is done by having multiple bodies for each procedure, up to one for each aliasing pattern. Procedure calls are automatically dispatched to the body that matches the run-time aliasing pattern among the actual parameters and the globals.

We present experimental evidence that this approach is practical. It is easy to convert existing C code, because not every procedure in a program has to be converted, and because converted code can call code that has not been converted and vice versa. By following simple guidelines, one can convert a program in a way that usually makes it run faster than before. In our experiments with 6 of the SPEC 2000 integer benchmarks we found an average speedup of about 5%. In one case, we had a speedup of about 29%.

*Currently at Epic Systems Corp., 5301 Tokay Blvd., Madison, WI 53711 USA.

Keywords

Aliasing, verification, optimization, C/ACL language

1. INTRODUCTION

Aliasing adds to the complexity of a program, making it more difficult to reason about. For example, one must consider many cases to verify the correctness of a procedure when its formals may be aliased [12, 21, 32, 33].

Aliases also limit the amount of optimization a compiler can do. This is especially true of compilers that do not do high-quality, whole-program alias analysis [23, p. 293]. Such analysis is costly, and in any case, must be conservative.

In this paper we are only concerned with direct aliases between pointers and between pointers and global variables.[1] Two pointer variables p and q are *direct aliases* if the pointers contained in p and q both point to the same storage cell in memory. A pointer variable p and a global x are direct aliases if p points to the location named by x. If p and q are direct aliases, then modifications to the storage that p points to can be seen through q. Other kinds of aliases are possible: overlaps between array elements and arrays, and indirect aliases resulting from pointers inside distinct memory locations. However, indirect aliases are outside of the scope of the reasoning problem that motivates our work, and overlaps are outside the scope of the partial solution to the reasoning problem treated in this paper.

We describe an approach eliminating direct aliasing among formals and globals, and study its effectiveness. In the Alias Controlling Language (ACL) approach [20, 2] the programmer declares different bodies for the same procedure, each of which handles a different pattern of aliasing among the formal parameters and globals. Within each such body, only one name may be used for each alias; hence, within each such body there is no direct aliasing. Calls to such procedures are dispatched to the appropriate bodies according to the aliasing pattern at the call site.

In this paper we focus on the feasibility of this approach. We

[1] In a language like C, distinct variables are always stored in distinct locations, and so cannot be directly aliased. Hence distinct globals also cannot be directly aliased.

describe how to apply it to the C programming language, and then measure the running time of the resulting code. Limitations of C mean that we are not able to provide a complete solution to the reasoning problem, because we are not able to eliminate problems caused by overlaps. However, our adaptation of the approach to C is able to eliminate direct aliases among formals and globals.

Our main goal is to show that this approach does not slow down program execution at run-time. Although dispatching calls to the appropriate bodies slows down code, many calls can be statically dispatched, incurring no overhead at run-time. Furthermore, increased knowledge of aliasing leads to better optimizations. The experiments we conducted tried to measure these effects in a real compiler.

In the next section, we briefly review the problems caused by aliasing. Section 3 discusses related work on solving the aliasing problem for formal parameters and globals. Section 4 presents C/ACL — C with ACL extensions. Section 5 presents our experimental setup and method. Section 6 summarizes the results of our experiment. Section 7 discusses our pragmatic experience coding with C/ACL, and other related work. Finally, in Section 8 we present our conclusions, and some directions for future work.

2. PROBLEMS CAUSED BY ALIASING

Direct aliasing among parameters can happen in two ways. First, the same pointer can be passed into two parameters, in this case, this same location will have two different names, or direct aliases, inside the called procedure. These pointers can result from the use of call-by-reference (as in C++), indirect storage models (as in Lisp, Smalltalk, and Java), or from an "address of" operator (as in C and C++). The other way direct aliases can occur is when a pointer to a global is passed as one of the arguments. If this global is used inside the called procedure, then it will also have two names inside that procedure.

Such aliases can cause two kinds of problems. The first problem is that in the presence of aliases it is much harder to reason about the correctness of the code, because aliases violate referential transparency [32, 33]. For example, in Figure 1, the `sum` procedure is supposed to add up the elements of array `a` with indexes less than the value of `size`, and return this sum in the location pointed to by the parameter `total`. While this code may look like it will always return the required result, in the presence of aliasing, this may not always be true. It may return an incorrect result if `size` and `total` are direct aliases; for example if invoked with `sum(myArray, &size)`.[2] It is all too easy for specifiers and implementers to ignore such potential aliases. One reason for this is that informal summaries of the purpose of code often make implicit assumptions about lack of aliasing, as did our English description of `sum` above.

The second kind of problem is that the possibility of aliasing limits the ability of a compiler to achieve good optimization. In the code given in Figure 1, without knowing whether the global `size` and `total` are aliases, the compiler can not

```
extern int size;
void sum(int a[], int *total) {
  int i;
  *total = 0;
  for(i=0; i<size; i++) {
    *total += a[i];
  }
}
```

Figure 1: C code to sum the elements in an array.

assume that the value of `size` will not change in the loop body. Thus it must reload it from memory in each iteration. This happens even if we compile this code with GCC using all possible optimizations[3]

In addition to direct aliasing, overlaps may cause similar problems for reasoning and optimization. For example, in Figure 1, there is also a possible overlap between `*total` and the elements of `a`. Since C does not track the sizes of arrays at run-time, our changes to C do not handle overlaps. This makes the C/ACL language described in this paper an incomplete solution to the reasoning problem (unlike the ACL approach itself, which can handle overlaps [20]). However, our main concern in adapting the ACL approach to C was to experiment with the approach's efficiency.

3. RELATED WORK ON ALIASING OF FORMALS AND GLOBALS

Other research has tried two different approaches to solving the problems caused by direct aliasing of formals and globals. These approaches either restrict the aliasing possible, or give some hint to the compiler about the aliasing patterns that will occur at run time. Euclid [29] prohibits calls with aliasing among formals and globals. Some compilers [31, 17, 18] and the latest C standard [19] use hints to instruct the compiler about aliasing, so that this information can be used for optimization.

Euclid [29] is a variant of Pascal designed to aid program verification. Euclid has pointer variables and call by-reference, both of which can cause aliasing between parameters. Pointers in Euclid are indexes into collections of objects. Collections are explicitly declared variables that are similar to the implicit arrays accessed by pointers in other languages, like C. Therefore eliminating aliasing in Euclid is all about eliminating aliasing between arrays. The definition of Euclid prohibits any aliasing between the actual parameters passed to a function by generating checks in the code that will signal run-time errors if direct aliasing or any overlap is detected between the arguments.

In general, to satisfy Euclid's restriction that there is no aliasing between the actual parameters, the developer will have to add alias analysis code at the call site. Figure 2

[2]Recall that, in C, `&size` returns the address of `size`.

[3]Using GCC 2.95.2 with the options `-fstrict-aliasing -O3`; these are explained in Section 6.

shows such code for a function, p, that takes three pointer parameters; the idea is to pass pointers to a[i], a[j], and a[k] as the actual parameters. In that code p_abc, p_ab, p_ac, and p_bc are different versions of p that handle different aliasing patterns but achieve the same effect.[4] The problem with this approach is that, to ensure correctness, the programmer must add these checks at each call site. This is what the ACL approach automates.

```
if ( i == j && j == k) { p_abc(&a[i]); }
else if ( i == j ) { p_ab(&a[i], &a[k]); }
else if ( i == k ) { p_ac(&a[i], &a[j]); }
else if ( j == k ) { p_bc(&a[i], &a[j]); }
else /* unequal */ { p(&a[i], &a[j], &a[k]); }
```

Figure 2: Hand-coded alias analysis at the call-site [20]

Several authors have proposed variations of linear type systems or uniqueness annotations for pointers [3, 6, 16, 22, 5]; these ensure that the given pointer is the only one that points to the target location. These are often used in functional languages, where partial copies of data structures can be made to ensure uniqueness. In non-functional languages there are also some techniques, such as alias-burying [4] that can be used to avoid some of the copies. However, if the programmer is given the responsibility for ensuring uniqueness, or if one wishes to use unique pointers to transmit results from a procedure, then the programmer will have to ensure that the call site does not attempt to create two unique pointers to the same location. Thus the programmer in such a language would face the same problem that affects Euclid; for example, the programmer forming several unique pointers into an array as in Figure 2 has to either make copies (if the pointers are not being used for results) or must do run-time tests to avoid errors when the locations are not distinct.

Some compilers, like IBM's XL compiler [31, 18, 17], use a noalias pragma that allows the developer to indicate what parameters cannot be aliased. The noalias pragma can be used at the function definition site to instruct the compiler about the possible aliasing patterns at call sites. This pragma is used for optimization, but it does not pose any restrictions on the parameters used at call sites. Although, unlike Euclid, no run-time errors will be generated if these pragma-defined patterns do not hold at some call sites, the behavior of the code at such call sites is undefined. This can lead to errors, since the XL compiler does not give any warnings about possible violations of the pragma-defined pattern by procedure calls. One way around this difficulty is for the XL programmer to think like a Euclid programmer, and to write code as in Figure 2 where necessary. Such a programmer will encounter the same problems as the Euclid programmer would. Another way around this difficulty is to delete noalias pragmas as necessary to enable the procedure to be used at all call sites, but this sacrifices opportunities for optimization. The ACL approach does not suffer from this tradeoff, because it allows different bodies to be associated with the same procedure, each of which will handle a different aliasing pattern.

[4]A formal statement of such a postcondition often must have separate cases for each aliasing pattern; this is another motivation for the ACL approach.

In the C standard, a new type qualifier, 'restrict' [19, Section 6.7.3.1], can be used on a pointer declaration to instruct the compiler about the aliasing patterns that this pointer can be involved in. Figure 3 shows an example using restrict. In that example the compiler will assume that at each invocation of copy, x and y will not be aliases, and it may do optimization based on that assumption. But the developer must insure that none of the call sites breaks the assumed pattern, otherwise the behavior is undefined. So, as with IBM's XL compiler, either extra checks are needed at each call site, or one must trade optimization for generality.

```
void copy(int * restrict x, int * restrict y,
          int size) {
  int i;
  for (i=0; i<size; i++) {
    x[i] = y[i];
  }
}
```

Figure 3: The use of the restrict type qualifier in C

In summary, all of these approaches, either explicitly or implicitly, limit the options available for the developer at the call site. And, they either necessitate adding extra code at each call site to ensure correctness, or they have a tradeoff between optimization and generality. We will see that they can all benefit from the ACL approach.

4. THE C/ACL LANGUAGE

The ACL approach is a language-based technique for eliminating aliasing among formals and globals [20, p. 4]. In this approach, a separate body is provided for each aliasing pattern that can occur among the parameters and among the parameters and the globals, thus each procedure may have multiple bodies, each of which is guaranteed not to have any direct aliasing among the parameters and globals. At the call site, a call will be dispatched (statically if possible, or otherwise dynamically) to the body that matches the aliasing pattern among the actual arguments and globals; if that body does not exist an error will occur. Thus, the ACL approach automates and systematizes the Euclid style of dispatch coding shown in Figure 2.

Figure 4 shows a function written in C/ACL [2], i.e. C with ACL extensions. In this code we notice two things that give ACL its distinct flavor. First, a function must specify explicitly all the globals it uses with an imports clause. Second, a function may have several *multibodies*, each of which handles a different aliasing pattern among the arguments and among the arguments and the globals. In Figure 4 the second body of swap_with_a starts with "| alias (x, a)"; this body, which in this case does nothing, will be used when swap_with_a is called with an alias to the global variable a. In C/ACL only the first variable in the alias list can be used inside the corresponding body. So, in this example, only x can be used in the second body.

```
int a = 0;

void swap_with_a (int *x) imports (a) {
   int tmp = *x;
   *x = a;
   a = tmp;
} | alias (x, a) {
   /* do nothing */
}
```

Figure 4: C/ACL code with two multibodies

4.1 The 'imports' clause

The imports clause declares that a function is written in ACL style and also declares all globals that the function uses. It is a static error if an ACL style function uses a global it does not declare in the imports clause.

In C/ACL, one can have both normal C functions and ACL style functions. The imports clause tells the C/ACL compiler that the function is an ACL style function that can have multibodies. So, even if an ACL style function does not use any globals, it must have an imports clause —an empty one in this case— to be treated as an ACL function. A function that does not have an imports clause is treated as a normal C function, and is not allowed to have multibodies. This maintains C's semantics for functions that are not converted to ACL style.

Globals listed in the imports clause can be listed as identifiers, in which case they will have the same type inside the function as outside, or, they can be given a more restrictive type —in terms of constness— which can be used by the compiler when optimizing that body. Figure 5 shows an example in which size is a global integer, but inside the function array_sum it will be treated as a constant.

```
int size = 5;

int array_sum (int a[])
        imports (const int size) {
   int sum = 0;
   int i;
   for (i=0; i<size; i++) {
     sum += a[i];
   }
   return sum;
}
```

Figure 5: Globals with restricted types in the imports clause

4.2 Multibodies

Each ACL function can have more than one body implementing it. Inside the first such multibody all the parameters and imported globals can be used. Within this multibody, the compiler assumes that there is no direct aliasing among the formals and the globals. Subsequent multibodies start with an alias clause that has the following syntax:

`| alias (` [*alias_list* [`;` *alias_list*] ...] `)`

where each *alias_list* is a comma-separated list of identifiers. An *alias_list* can include any parameter that has a pointer type or any of the imported globals. The semantics is that inside that body, all the stated identifiers in each *alias_list* will be direct aliases to each other at run-time. For example, in Figure 4, in the second multibody at run-time x and a will be directly aliased, i.e., it will be the case that x == &a.

C/ACL allows formals and globals of different types to be declared as aliases in alias clauses. This is in keeping with the spirit of C. For example, C/ACL allows a pointer to an integer to be an alias to a constant pointer to an integer. Of course, such aliases are unsafe, as they can lead to unexpected results when used with aggressive optimization, because the compiler assumes that a pointer will only point to its type, and that a constant will actually be constant.

Inside each multibody only a subset of the formal parameters and imported globals can be used. The ones that can be used are the first identifier in each *alias_list* and those not included in the *alias_list*s. For example, Figure 4 shows a function with two multibodies. In the second multibody, only x can be used, since it is mentioned first in the *alias_list*. Formals and globals not listed in any *alias_list* are assumed to not be directly aliased with each other. Recall that all these assumptions are enforced at run-time by dispatching calls to the appropriate multibody.

5. THE EXPERIMENT

In order to determine the feasibility of the ACL approach we set up an experiment in which we could compare the execution speed of C code to that of C/ACL code [2].

5.1 Hypothesis

Our main hypothesis was that C/ACL code would run faster than unconverted C code. We hypothesized that this would occur for the following reasons:

Specially written multibodies. Since the developer can supply a separate multibody for each aliasing pattern, and since the developer knows the function's semantics, the developer can write each multibody in a way that makes efficient use of that pattern. An example is the second multibody in Figure 4 in which, knowing that x and a are aliases, we had only to give an empty multibody. Our hypothesis was that each multibody would be no less efficient than a correct implementation of the original function for that aliasing pattern.

Alias-free arguments. In an ACL style function, the arguments to each multibody will be guaranteed not to be directly aliased with other arguments or globals.

Our hypothesis was that this extra information would enable a compiler to do a better job when optimizing these multibodies. Certainly the generated code should be no less efficient than with a standard compiler.

Dynamic dispatch. The only run-time overhead from the ACL approach was hypothesized to be the dynamic dispatch code that matches an aliasing pattern to the corresponding multibody.

Static dispatch. Our final hypothesis was that static dispatch would be possible from most call sites.

In summary, we hypothesized that static dispatch, which avoids the overhead from dynamic dispatch, and better optimization would more than make up for the overhead of whatever dynamic dispatch proved necessary.

5.2 Setup

For testing our hypotheses we applied the ACL approach to an existing language and optimizing compiler and measured the performance difference between compiling code with and without ACL features. We selected C as the language because it has pointers and is widely-used. We chose GCC (the GNU Compiler Collection) because we needed a compiler that we had source access to, and that does aggressive optimization using aliasing information.

The implementation of C/ACL was accomplished by applying the design described in the previous section to the C front end of GCC. All experiments were done on an AMD-K2 380 MHz with 192MB of RAM, running RedHat Linux 7.1. The version of GCC modified was GCC 2.95.2 with Pentium-specific optimizations.

To test our compiler we used benchmarks from the SPEC CPU2000 benchmark suite. We chose to use this suite because it ensured that we thoroughly tested our compiler with real programs and big input data sets. The SPEC suite also compares the generated output with the expected output, helping to ensure that C/ACL produced correct code. Moreover, by changing real programs, we could unearth situations in which using C/ACL was difficult or did not give the desired improvement (or both).

When running the SPEC benchmarks, most benchmarks were measured three times. The median of the three measurements is reported as the result of the benchmark.

5.3 Testing method

For each of the benchmarks, we first profiled the unmodified source. Using the profile data, we selected functions to be converted to ACL style. These were functions that took a large amount of the execution time and passed pointers or used globals. Thus, in doing this conversion we followed the method detailed in Section 7.4 below. This method selected no more than 11 functions in each benchmark to convert. Although in the benchmark 'MCF' 42% (8 of 19) functions were converted, in the other benchmarks no more than 2% of the available functions were converted to ACL style.

After converting the chosen functions to ACL style, we ran the benchmark suite twice, the first time using the original source code, and then using the modified source code. In both cases we used the same optimization flags and the same C/ACL compiler. The optimization flags used were `-fomit-frame-pointer -fstrict-aliasing -O3`, which selects the most aggressive optimization levels.

To ensure consistency in the results, all tests were run from the console with no other users logged in, and most unnecessary daemons were stopped.

6. EXPERIMENTAL RESULTS

We ran the test on six SPEC 2000 benchmarks: Parser, MCF, GCC, VORTEx, GAP, and TWolf.[5] Parser is an English language parser, it takes as input a sequence of sentences and returns as output the analysis of the input sentences. MCF is a combinatorial optimization program, it generates the timetable for a single-depot vehicle schedule. GCC is an earlier version[6] of GCC, it includes only the C compiler which takes as input preprocessed C files and generates as output 88100 assembly code files. VORTEx is a single-user object-oriented database transaction benchmark. GAP implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming). And the TWolf placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips.

Recall that the compilation flags used for optimization were `-O3`, `-fomit-frame-pointer` and `-fstrict-aliasing`. The last flag instructs the compiler that a pointer will only point to its declared type. This flag had to be used as otherwise GCC assumes that any pointer can point to any memory location, and skips the alias analysis phase.

MCF, GCC, GAP, and VORTEx had good candidate functions. Converting these functions to ACL style achieved an improvement in the execution time of between 2.1% and 29.6%, as reported in Tables 1, 2, 3, and 4 respectively. Run times in these tables are reported in seconds.

Table 1: Results for MCF

	Original	With ACL	
Run	Run Time	Run Time	Speedup
1	4086	4177	
2	4278	3930	
3	4255	3870	
Reported	4255	3930	8.3%

After profiling 'Parser' and 'TWolf', we found that most of the candidate functions to be converted to ACL style either did not use pointer formals heavily in their bodies, and thus did not benefit from the extra aliasing information, or were mainly called outside the compilation unit (i.e., file) in which the ACL style function was defined; thus our C/ACL

[5]GAP was only measured once, instead of three times.
[6]The benchmark was based on GCC version 2.7.2.2

Table 2: Results for GCC

	Original	With ACL	
Run	Run Time	Run Time	Speedup
1	4089	3746	
2	4106	3835	
3	3582	4027	
Reported	4089	3835	6.7%

Table 3: Results for GAP

	Original	With ACL	
Run	Run Time	Run Time	Speedup
Reported	12985	12719	2.1%

Table 4: Results for VORTEx

	Original	With ACL	
Run	Run Time	Run Time	Speedup
1	4171	3196	
2	3956	3218	
3	4206	3327	
Reported	4171	3218	29.6%

compiler did not use static dispatch extensively for these benchmarks. The overhead of dynamic dispatch dominated in these benchmarks, resulting in slowdowns of 8.9% and 5.8%, as shown in Tables 5 and 6, respectively.

Table 5: Results for Parser

	Original	With ACL	
Run	Run Time	Run Time	Speedup
1	4016	4055	
2	3668	3832	
3	3648	4031	
Reported	3668	4031	-8.9%

Table 6: Results for TWolf

	Original	With ACL	
Run	Run Time	Run Time	Speedup
1	6240	6827	
2	6423	6620	
3	6171	6569	
Reported	6240	6620	-5.8%

To study the effect of varying the number of converted functions and the effect of dynamic dispatch, we made further investigations with 'Parser'. By reverting functions that are called outside of their compilation units to their original C code, we decreased the cases that incurred the penalty from dynamic dispatch; this gave a 3% runtime speedup versus the unmodified C code. To see the net effect of using static dispatch, we moved the definition of functions converted to ACL style into the compilation unit where they are called and measured the performance with the `-O3` flag, thus enabling static dispatch. In that case we got 6% runtime speedup versus the unmodified C code. Our conclusion is that a suitably sophisticated compiler could automatically eliminate dynamic dispatch in most cases, by making use of static information about what multibodies are available in the called function.

To see more of the effect of static dispatch, we measured the GCC benchmark with only the `-O2` flag. This in effect instructs the compiler to do less optimization than the previous tests, and in particular eliminates inlining and hence static dispatch. Without static dispatch the code converted to ACL style was slower than the original C code by 3%. This was slower than the static dispatch measurements for the ACL style code by about 10%. Thus static dispatch has a significant effect on performance.

In summary, our results indicate that static dispatch is easy to accomplish in most cases, even if the compiler does not propagate information about what multibodies are available across compilation units. Overall, the C/ACL compiler achieved a speedup of about 5% for the benchmarks we examined. Certainly, the details of our experiments validate our hypotheses.

7. DISCUSSION

7.1 Our implementation

The central task in implementing C/ACL was to accommodate for multibodies in a C program. The compiler converts each ACL style function having n multibodies into $n+1$ regular C functions. Of these $n+1$ functions, n correspond to each of the multibodies. These functions have compiler-generated names. The $n+1$-st generated function has the same name as the original C function, and does the dynamic dispatch to the n multibody functions. The dynamic dispatch function has decision code based on a decision tree algorithm. It thus makes the minimum number of tests to determine the direct aliasing pattern between the formals and the globals, to decide which of the n multibody functions to call. Since the dynamic dispatch function has the same name as the original ACL style function, any call to that function will run the dynamic dispatch code, which will then direct the call to the appropriate multibody function.

In the cases where static dispatch is possible, the compiler can bypass the dynamic dispatch code, and call the appropriate multibody function directly. In our compiler, this was achieved automatically by GCC's function inlining optimizations. That is, the dynamic dispatch code was inlined at call sites, and when the compiler knew enough about the aliasing pattern at the call site, other optimizations would automatically eliminate the decision tree code, leaving just the call to the appropriate multibody function — i.e., a static dispatch. The compiler would often know enough about the aliasing pattern at the call sites when these calls occurred within multibodies, since extra aliasing information was available there. Thus the conversion to ACL style has a beneficial effect on the aliasing information needed to make static dispatch possible.

Finally, we benefited from existing infrastructure in GCC that can inform the optimization passes about aliasing. That is, there was a pre-existing flag in GCC that we used to tell the compiler directly that there were no aliases among the formals and globals within each multibody.

7.2 Where the Speedup Comes From

By studying some code snippets and the resulting assembly code, we found that the speedup in ACL style functions mainly comes from the following reasons. By knowing that a memory location is not aliased, the compiler moves it to a register and can then access that value from the register instead of loading it from memory each time. In some other cases, a variable that is known to be not aliased can help in finding more loop-invariant code; such code is moved out of the loop, thus minimizing code used inside loops, which has a great impact on performance. In the cases where two variables are known to be aliases, like in multibodies with alias lists, these two variables can occupy the same register, thus freeing a register that can be used for another variable which can save memory accesses.

The other main reason for speedup is static dispatch. While not a reason for speedup by itself, static dispatch eliminates the only overhead that one incurs from using the ACL approach, which is dynamic dispatch. In our implementation of C/ACL, using GCC, we found that static dispatch is possible when using the optimization flag `-O3` which enables function inlining, and when the called C/ACL function is defined in the same compilation unit as the call site.

7.3 Pragmatics of C/ACL

Beside measuring the performance improvement gained from using C/ACL, we were interested in finding out whether it will be easy to adopt C/ACL and how practical it is. There are two main hurdles that we thought someone embracing the ACL approach could face.

First, having to write new multibodies for a function means that the semantics of the function must be understood. This can be a disadvantage compared to other optimization techniques that do not require human intervention. While it might be possible to use the ACL approach automatically in a more sophisticated compiler, our experience in applying ACL to the benchmarks was that having to write import lists and multibodies is not burdensome. In most cases, the additional multibodies are very simple. Even when the multibodies can not be deduced easily from the original function, one often can copy the same function to all the multibodies, and substitute the head of each alias list for that list's other members. This could certainly be automated. Leavens and Antropova explored this and other ways to automate writing of multibodies [20].

On the other hand, a human that understands the intended semantics of a function can write multibodies that are more efficient for certain patterns of aliasing. A simple example is shown in the second multibody of Figure 4. In our experiments we found several similar examples. For example, a function that compares two strings to see if one is less than or equal to another can return "true" if they are both aliased. Such optimized multibodies might be very difficult to construct automatically.

However, hand-optimized multibodies for cases where some formals and globals are aliased will probably have a small effect on efficiency, because these multibodies are rarely called. Our results indicate that most of the calls were dispatched to the multibody that handles the case where there is no aliasing among the formals and globals. Many times this was the only multibody that saw any calls in our experiments. However, for some benchmarks other multibodies did see significant numbers of calls. 'Parser' had one ACL style function where the other multibodies were run for 19% of the calls, in another they handled 18% of the calls, and in another 3% of the calls. So, while these other multibodies are probably not important for efficiency, they are important for correctness, since the program cannot abort when a call to an ACL style function is made with aliases among the arguments and globals.

Secondly, there is the problem of choosing which functions to convert. However, this also turned out to be straightforward. We developed a pragmatic method for converting functions which is described below.

7.4 Method for Conversion to C/ACL

Our method for choosing functions to convert to ACL style (i.e., which functions should have imports clauses and multibodies) is as follows.

First, one profiles the program with representative input data, and looks for bottlenecks. The functions that take the bulk of the execution time are candidates for conversion.

Of the candidate functions, one should convert to ACL style those that fall into one of the two following classes.

- Functions with at most three pointer parameters and imported globals. We found that it was burdensome to convert functions with four or more such parameters or globals, since the number of required bodies grows exponentially with the number of such parameters.
- Functions whose call sites only use a few aliasing patterns. Such functions can be converted to ACL style even if they have a large number of formal pointer parameters and imported globals, because one only has to write multibodies corresponding to the aliasing patterns used at the call sites.

Our experimental results show that this method usually results in execution time speedups. To be more effective, either the compiler needs to be able to do static dispatch across compilation units (which our compiler was unable to do), or the method needs to be revised to not convert functions to ACL style if they are both heavily used and called from other compilation units.

7.5 Other Related Work

It would be interesting to see if some tools could do the conversion to ACL style automatically; i.e., if the ACL approach could be incorporated into compilers. For example, one can consider the ACL approach to be a variant of Chambers and Ungar's idea of compiling different versions of a function body assuming different type information, guarded by a type test [8]. Indeed the idea of dispatch on the aliasing pattern of arguments was inspired somewhat by this idea and by an analogy to multiple dispatch. The multibodies that make up a function in C/ACL are similar to multimethods that make up a generic function in languages like CLOS [28], Dylan [30], Cecil [7], and MultiJava [11].

Predicate dispatch [14] is able to simulate the ACL approach, because one can dispatch to a method body based on a predicate that includes direct aliasing among formal parameters and globals. Indeed, such predicates would make it easy to combine various cases in ways that the ACL approach does not support. However, there have been no compilers that implement predicate dispatch making it unsuitable for our experiments. Furthermore, it is unclear how difficult it would be to link predicate dispatch to optimization based on aliasing information.

Our work can also be seen as complimentary to recent work on alias controlling type systems [1, 10, 9, 16, 26, 25]. These type systems use static type information to control problems caused by aliases into the internal state of an object from outside the class implementing it, such as representation exposure and argument exposure. Such type systems also seem important for reasoning in a modular fashion about the correctness of implementations of abstract data types [24]. These type systems work by partitioning memory into several *ownership contexts*, i.e., sets of locations, and by prohibiting references from certain ownership contexts to certain other ownership contexts [9]. The invariants maintained by such type systems can help in alias analysis, because the compiler can know that references between certain universes are prohibited. However, these type systems cannot be used to rule out all aliasing among formals and globals; in particular, they cannot rule out aliasing among formals and globals that are in the same ownership context.

RESOLVE deals with the problem of aliasing by eliminating it completely. It does this by using swapping [27, p. 26] [15] and by eliminating arrays and pointers as built-in types. Because there are no built-in arrays and pointer types, the programmer can always statically avoid aliasing of arguments and globals. However, not having these built-in types restricts the language's expressive power. The ACL approach, on the other hand, is able to eliminate aliasing among formals and globals without such restrictions.

8. CONCLUSION & FUTURE WORK

In this paper, we investigated the efficiency of code generated by GCC using the ACL approach. We added syntax to C that allows users to declare several multibodies for a function, each of which handles a specific pattern of direct aliasing between actual parameters and imported globals. We modified GCC so that it dispatched calls to such functions to the appropriate multibody. Thus, in each multibody, there is no direct aliasing among the formals and globals that the multibody is allowed to use.

It is quite easy to get a semantics similar to that of Euclid [29] by only writing a single multibody for the case where no arguments and globals are directly aliased. But unlike Euclid, because of limitations of C, C/ACL does not protect against overlaps between parameters and between parameters and globals. Nevertheless we demonstrated the feasibility of the ACL approach.

C/ACL resolves the tension between optimization and generality found in the IBM XL compiler [31] and the `restrict` mechanism of the new C standard [19]. Because one can specify multiple aliasing patterns and multibodies to handle each of them, our approach does not limit the aliasing pattern at the call site, thus giving extra freedom to the developer. On the other hand, with the `noalias` pragma of the IBM XL compiler and C's `restrict` mechanism, one can state what formals are not aliased, thus the single function body can handle several aliasing patterns, which the C/ACL user would have to write as several multibodies. Hence it is possible that a better syntax for the ACL approach may make it even less burdensome. Note however, that `restrict` is not able to declare that a formal pointer parameter and a global are not aliased, so the ACL approach is more powerful in this respect.

The ACL approach can be seen as complimentary to interprocedural analysis. Interprocedural analysis tries to find the effect of a call on the call-site, by propagating information from the called function back to the call site. On the other hand, the ACL approach propagates information about possible aliasing patterns at different call-sites to the called function. This information is recorded in the various multibodies of an ACL style function. Unlike the IBM XL compiler or the C restrict notion, the user is not limited to a single function body, hence different call-sites can have different multibodies tailored to their needs.

By choosing suitable functions to convert to ACL style, improvement in run-time performance can be achieved. We described a simple method for doing this in Section 7.4.

In our experiments we applied C/ACL to six of the SPEC 2000 benchmarks. When doing the conversion of functions to ACL style, we followed the method described in Section 7.4 above to choose the candidate functions. The average speedup achieved on these six benchmarks was approximately 5%. In four out of the six benchmarks cases, we got runtime speedup that averaged 11%, with a median of 7.5%. These results are indicative of the results that could be expected by a sophisticated compiler that is able to do static dispatch across compilation unit boundaries, since they were the cases where our compiler did static dispatch. In two benchmarks, our method for conversion to ACL style picked functions that were called across compilation unit boundaries; since our compiler cannot statically dispatch such functions, the overhead from dynamic dispatch resulted in slowdowns (versus normal C code) averaging 7.4%.

However, in C/ACL one can always compensate for the lack of static dispatch by not converting functions to ACL style if necessary. For example, in one benchmark where we observed a slowdown, we profiled the converted benchmark, and reverted the ACL style functions that were slower to not use the ACL style. By this process, we obtained a small runtime speedup, consistent with the other benchmarks. This validates our hypothesis that the only overhead of the ACL approach is due to dynamic dispatch. It also shows that a sophisticated compiler could eliminate this overhead.

The performance improvement from functions converted to ACL style is mainly due to the ability of the compiler to optimize the function knowing that it has alias-free parameters. The overhead from dynamic dispatch can be minimized by using static dispatch whenever possible. When the improvement from optimization dominates the dynamic dispatch overhead, a net gain in code efficiency is achieved.

In conclusion, we have demonstrated that the ACL approach is feasible. Furthermore, with static dispatch, significant speedups can be achieved. Since the programmer can compensate for slowdowns that result from our compiler's inability to do static dispatch, the C/ACL compiler itself may also be useful for optimization of C programs.

The sources for the C/ACL version of GCC are on-line at `ftp://ftp.cs.iastate.edu/pub/leavens/CACL-src.tar.gz`.

There are several directions for future work. One direction is to extend the syntax for function prototypes with information about implemented multibodies. Using that information, the compiler can emit the decision tree code to call the appropriate multibody of an ACL style function, instead of just calling the dynamic dispatch function. This will enable static dispatch across compilation units.

Another direction is to perform more experiments with more benchmarks. We plan to perform experiments to find what kind of functions and optimizations benefit most from the ACL approach. This could be used to refine the criteria upon which functions are chosen to be converted to ACL style. We are also planning to study the performance effect of converting functions in the C library to ACL style, although it appears that doing this will only be effective once the compiler is extended to do static dispatch across compilation unit boundaries.

In this paper we were mainly interested in code optimization. But having alias free parameters can also be useful in reasoning about programs. However, to be of true benefit for reasoning, one would have to apply the ACL approach to a language where one could handle indirect aliasing and overlaps (unlike C). Once this was done, one could study whether the ACL approach helps make reasoning about programs easier; for example, one could see if it makes programs less error prone, or if formal verification is easier. We also intend to investigate the interaction of the ACL style with formal specification.

Finally, we would like to extend the ACL approach to object-oriented languages. This would allow us to explore its effects on optimization and reasoning in that setting.

Acknowledgments

The work of Leavens was supported in part by the US National Science Foundation under grants CCR-9803843, CCR-0097907, and CCR-0113181. Thanks to Olga Antropova, Craig Chambers, Yoonsik Cheon, Curtis Clifton, Markus Lumpe, Ricky Kendall, Murali Sitaraman, and Bruce Weide for comments on a earlier drafts. Thanks to Jim Horning for earlier discussions, and to Olga Antropova for her pivotal role in developing the ACL approach.

9. REFERENCES

[1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer-Verlag, New York, NY, June 1997.

[2] M. G. Assaad. Alias-free parameters in C using multibodies. Technical Report 01-05, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 2001. Available from archives.cs.iastate.edu.

[3] H. G. Baker. Lively linear lisp — 'look Ma, no garbage!'. *ACM SIGPLAN Notices*, 27(8):89–98, Aug. 1991.

[4] J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

[5] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming: 15th European Conference, Budapest, Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 1–27, Berlin, June 2001. Springer-Verlag.

[6] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean: A language for functional graph rewriting. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 364–384. Springer-Verlag, Berlin, 1987.

[7] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, NY, 1992.

[8] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, pages 1–15, Nov. 1991. Published as Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, number 11.

[9] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming: 15th European Conference, Budapest, Hungary*, volume 2072 of *Lecture Notes in Computer*

Science, pages 53–76, Berlin, June 2001. Springer-Verlag.

[10] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, Oct. 1998.

[11] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, Oct. 2000.

[12] F. S. de Boer. A proof system for the language POOL. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 124–150. Springer-Verlag, New York, NY, 1991.

[13] S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors. *Formal Techniques for Java Programs.* Technical Report 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[14] M. D. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98: 12th European Conference on Object-Oriented Programming, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211, New York, NY, 1998. Springer-Verlag.

[15] D. E. Harms and B. W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[16] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 271–285, Nov. 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[17] IBM Corporation. *AIX Version 3 for RS/6000: Optimization and Tuning Guide for Fortran, C and C++.* SC09-1705.

[18] IBM Corporation. *IBM C Set ++ for AIX/6000 User's Guide Version 2.1.* SC09-1605.

[19] International Organization for Standardization. *Programming Language – C.* ISO/IEC 9899.

[20] G. T. Leavens and O. Antropova. ACL — Eliminating Parameter Aliasing with Dynamic Dispatch. Technical Report 98-08a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Feb. 1999. Available from archives.cs.iastate.edu.

[21] D. Luckham and N. Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Trans. Prog. Lang. Syst.*, 1(2):226–244, Oct. 1979.

[22] N. H. Minsky. Towards alias-free pointers. In P. Cointe, editor, *ECOOP '96 – Object-Oriented Programming: 10th European Conference, Linz Austria*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209, Berlin, July 1996. Springer-Verlag.

[23] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, Inc., 1997.

[24] P. Müller. *Modular Specification and Verification of Object-Oriented programs.* PhD thesis, FernUniversität Hagen, Germany, Mar. 2001.

[25] P. Müller and A. Poetzsch-Heffter. A type system for controlling representation exposure in Java. Published in [13]., 2000.

[26] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.

[27] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct 1994.

[28] A. Paepcke. *Object-Oriented Programming: The CLOS Perspective.* MIT Press, 1993.

[29] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the design of Euclid. *ACM SIGPLAN Notices*, 12(3):11–18, March 1977.

[30] A. Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley, Reading, Mass., 1997.

[31] K. E. Stewart. Using the XL compiler options to improve application performance. http://www.rs6000.ibm.com/resource/technology/options.html, 1998.

[32] M. Utting. Reasoning about aliasing. In *Proceedings of the Fourth Australasian Refinement Workshop (ARW-95)*, pages 195–211. School of Computer Science and Engineering, The University of New South Wales, Apr. 1995. Available from `http://www.cs.waikato.ac.nz/~marku`

[33] B. W. Weide and W. D. Heym. Specification and verification with references. In D. Giannakopoulou, G. T. Leavens, and M. Sitaraman, editors, *SAVCBS 2001 Proceedings: Specificaton and Verification of Component-Based Systems, Workshop at OOPSLA 2001*, pages 50–59, Nov. 2001. Department of Computer Science, Iowa State University, technical report TR #01-09a.