Why Programmer-specified Aliasing is a Bad Idea

Markus Mock
University of Pittsburgh, Department of Computer Science
Pittsburgh, PA 15260, USA
mock@cs.pitt.edu

Abstract

The ISO C standard C99 has added a special keyword, named `restrict` to allow the programmer to specify non-aliasing as an aid to the compiler's optimizer and to thereby possibly improve performance. However, it is the programmer's responsibility to ensure that the annotations are correct. Therefore, in practice, `restrict` will be useful only when the programmer's effort is rewarded with noticeable performance improvement. To assess the performance potential of the `restrict` annotation, we automatically generated best-case `restrict` annotations for SPEC CPU2000 benchmarks by using pointer profiling. However, even though we used the best possible `restrict` annotations, we found an average program speedup of less than 1% on average when using two state-of-the art optimizing compilers that implement the `restrict` pragma. Since the typical performance benefits do not warrant significant user effort and potential errors, we conclude that having the programmer specify non-aliasing is a bad idea.

Keywords: C, C99 standard, restrict, compilers, program optimization, aliasing, dynamic points-to analysis

1 Introduction

In most programming languages it is possible to access a memory location in multiple ways using different names. For instance, in programming languages that support call-by-reference parameters, the name of the formal and actual argument refer to the same location and are therefore aliased. Aliases do also frequently arise due to pointers to named variables: if `p` points to variable `x`, both `x` and the pointer dereference `*p` access the same memory location.

To ensure that correct code is generated in the presence of aliases, compilers have to perform an *alias analysis* to determine the aliases in the program and disambiguate memory accesses. Consider the example in Figure 1, which shows the fragment of a C program. Procedure `vmul` takes two arrays of floating point numbers, `a` and `b` and their size `n` as inputs and computes the square of each element of `a` and stores it element by element in array `b`. Without further knowledge and without special hardware support, the compiler must assume that `a` and `b` may refer to the same array or overlapping parts of an array so that the loop cannot be parallelized or software-pipelined because it has to be ensured that an update of `b[i]` is performed before the next value `a[i+1]` is loaded. This constraint also prevents the compiler from generating loads for multiple array elements of `a` at the same time since the subsequent store to `b[i]` may modify elements of array `a`.[1]

To compute the aliases in a program, many algorithms have been developed. Flow- and context-sensitive algorithms potentially produce the most precise results, but they generally do not scale well, which limits their applicability to relatively small programs (50,000 lines of code at most). In addition, recent work [5] suggests that for many C programs flow- and context-sensitivity produce only insignificant improvements over Das's fast One-Level Flow algorithm [4], which has been shown to scale up to over a million lines of C code.

Because flow- and context-sensitive pointer analyses are usually too expensive to be used in production compilers, often only very simple pointer analysis is performed in practice. For instance, the analysis may simply assume that a pointer dereference may define any variable that has its address taken in the program. Unfortunately, the conservative results ensuing from such simple algorithms may prevent the compiler from performing aggressive code optimization.

Even though more expensive algorithms potentially produce better results, compared to actual run-time pointer behavior their results are still very conservative. Mock et al. [19] showed that while the points-to sets of SPEC CPU2000 benchmarks computed by several well-known scalable points-to analyses typically contained hundreds or even thousand of potential pointer targets, at run-time pointers typically point to only a few, in fact in most cases only one, logical location.[2] This means that even compilers that use more sophisticated pointer analysis algorithms may in

[1] When the processor provides mechanisms for data speculation, the compiler may decide to speculatively load the value into a register and rely on the hardware to detect whether the value has to be reloaded. For example, on the Intel Itanium processor the compiler can use an advanced load instruction in conjunction with a check instruction to perform this kind of data speculation (cf. Section 4.2 for more details).

[2] A logical location is either a program variable or a heap allocation site. There may be multiple instantiations of a single logical variable in the case of local variables and multiple distinct objects allocated at the same memory allocation (heap) site.

```
void vmul(int n, double* a, double* b) {
   int i;
   for (i=0; i<n; i++)
      b[i] = a[i] * a[i];
}
```

Figure 1: A simple vector multiplication routine. If it is known that arrays `a` and `b` are not aliased, the compiler can software-pipeline or parallelize the loop.

```
void vmul(int n, double* restrict a, double* restrict b) {
   int i;
   for (i=0; i<n; i++)
      b[i] = a[i] * a[i];
}
```

Figure 2: The vector multiplication routine whose arguments have been annotated with the `restrict` pragma to indicate to the compiler that the arguments are not aliased, thereby allowing the compiler to perform more aggressive optimization, such as software pipelining. If the annotation is incorrect, the resulting code may be incorrect.

practice be prevented from performing some optimizations even when those would be sound in reality.

Pointer analysis imprecision is a particularly severe problem for the C programming language, where widespread pointer use, a weak type system, and pointer arithmetic often lead to very imprecise analysis results. This has been long recognized by C users and compiler writers alike and led to a community effort to add a language pragma to the C language standard that could be used by C programmers to communicate to the compiler that certain expressions in a program are not aliased. The latest C99 standard (formally defined as ISO/IEC standard 9899:1999 [13]) introduces a `restrict` keyword, which can be used to qualify formal pointer-type arguments of procedures.[3]

The definition of the `restrict` keyword specifies that "an object that is accessed through a restrict qualified pointer requires that all accesses to that object use, directly or indirectly, the value of that particular restrict qualified pointer. Any access to the object through any other means may result in undefined behavior. The intended use of the restrict qualifier is to allow the compiler to make assumptions that promote optimizations." (quoted from the README file distributed with the Sun C compiler, which implements the C99 standard, one of the compilers used in this study).

Figure 2 shows an example of how `restrict` is supposed to be used. The two pointer-typed arguments `a` and `b` are qualified with the `restrict` keyword to tell the compiler that the array pointed to by `a` is only accessed via pointer `a` and any pointers derived from it (e.g., accesses such as `a[i]` which use an address obtained by adding an offset to `a`). Similarly, the array pointed to by `b` is only accessed via pointer `b` and pointers derived from it. In particular, these assertions state that `a` and `b` may not point to overlapping or identical arrays. Consequently, based on these assertions, the compiler can perform (software) parallelization and other transformations whose correctness depend on `a` and `b` not being aliased. Compared to the routine shown in Figure 1, the restrict-qualified routine in Figure 2 executes 24% faster on a Sparc workstation and even 2.9 times faster on an Itanium-based workstation with two processors.[4]

Since the `restrict` keyword is an assertion specifying that the qualified function arguments are not aliased at run time, it has to be used with utmost care because the compiler may perform some optimizations that are not sound when the arguments are in fact aliased. While a compiler user who is familiar with the program he/she is compiling may be able to use the `restrict` keyword correctly, it will still be a tedious task and, without any support by an automatic tool, likely be error prone. In practice, therefore, users will only accept this tedium if they can expect a noticeable performance gain. While previous work [3] found that in some cases memory disambiguation can result in significant speedups for kernel array codes, it is not clear what performance gains are achievable with the `restrict` pragma on general C programs of realistic size.

In this paper we show that even optimistic annotations with `restrict` pragmas (i.e., annotations that may only be sound for some but not all inputs) lead only to minor performance improvements for C programs of significant size and variety. We were able to verify that this result appears to hold across compilers and architectures by using two distinct commercial optimizing compilers on two different processors (Sparc V9 and Itanium 2) and obtaining very similar results. Therefore, we conclude that is generally a bad idea to put the specification of non-aliasing into the programmer's hands.

However, while the typical performance improvements are too small to warrant extra programmer effort, we would like to realize those improvements if this could be done with no or insignificant programmer effort. Fortunately, this is

[3]The C standard also defines the use of `restrict` on structure fields to specify that `restrict`-qualified fields are not aliased to other program expressions.

[4]The Sparc speedup was measured on a Sun Blade workstation with version 7 of the Sun Studio One C compiler and highest (-xO5) optimization level; the speedup for the Itanium processor on a HP ZX6000 machine with 2 Itanium 2 processors, and Intel's ecc compiler with highest (-O3) optimization. In both cases, a vector size of 100 was used and the loop was executed 1,000,000 times.

feasible: with simple pointer profiling, `restrict` annotations can be derived automatically. We present an approach that ensures that such annotations are sound by generating a run-time check that redirects execution to conservatively optimized code (assuming aliasing) when the run-time check fails.

In more detail, this paper makes the following contributions:

1. By observing alias relationships at run-time and using them to place `restrict` pragmas, we obtain an upper bound on the potential performance improvement arising from `restrict` pragmas in two real highly optimizing C compilers;

2. we show that for a large class of diverse, compute-intensive applications (SPEC CPU2000 benchmarks), only minor improvements can be achieved – on average less than 1% and no more than 8% in our experiments;

3. we demonstrate how `restrict` pragmas generated by run-time observation of alias relationships can be made sound across all inputs by outlining how to generate guard code that selects a conservatively optimized version of a function when aliases occur at run-time;

4. and finally, we show that placing `restrict` pragmas based on static alias analysis alone is in general much less effective than generating guarded `restrict` pragmas using run-time information.

The rest of this paper is organized as follows: Section 2 gives some background on pointer analysis and describes how we obtained the dynamic alias information used for our optimization experiments. Section 3 describes our experimental setup and the workload used in this study. Section 4 presents the results of our experiments and analyzes the reasons for the observed speedups. In Section 5 we propose an automatic approach to generate sound uses of `restrict` pragmas, and in Section 6 we discuss related work. Finally, Section 7 presents conclusions and directions for future research.

2 Alias Analysis

In the C programming language aliases can arise in two ways. First, different fields of a union data structure are aliased to each other. Second, variables that have their address taken can be accessed both via their name and a pointer dereference. Aliases arising via union data structures are easily identified, potential aliases via pointers, on the other hand, require a pointer analysis.[5] This analysis is often performed as a *points-to analysis*, which computes for each pointer `p` and pointer dereference expression `exp` the set of logical locations that may be accessed via `p` or `exp`. Some pointer analyses compute the set of possible aliases due to pointers directly (e.g.,Landi et al.'s algorithm [16]).

2.1 Points-to Analysis

Aliases can be computed easily from points-to sets: given variable x and pointer-valued expressions $exp1$ and $exp2$, x is aliased to $*(exp1) \iff x \in pts(exp1)$, where $pts(exp)$ denotes the points-to set of expression exp, and $*(exp1)$ and $*(exp2)$ are aliased $\iff pts(exp1) \cap pts(exp2) \neq \emptyset$.[6]

Traditional *static* points-to analyses compute an approximation of the set of objects to which a pointer may point. They are conservative in the sense that their results must be correct for any input and execution path of the program. In addition, for the C programming language they have to make various conservative assumptions when analyzing a program, for instance, because of C's weak type system.

An alternative way of gathering points-to data is to perform a *dynamic* points-to analysis. A dynamic points-to analysis records the targets of program pointers during actual program execution, by instrumenting the program source with calls to an appropriate data-capturing routine. Since dynamic points-to sets only capture the targets of pointers during a particular program execution, they are in general unsound (i.e., optimistic). Mock et al. [19] showed that the typically observed dynamic points-to sets are 10–100 times smaller than the points-to sets computed by Das's highly-scalable One-Level Flow algorithm [4], which generally produces results as precise as Andersen's well-known algorithm [2]. Andersen's algorithm, in turn, has been shown [17] to be of comparable precision as some other well-known pointer analysis algorithms [17, 20]. Mock et al. [19] additionally showed that the majority of program variables in the SPEC CPU2000 benchmarks point to only a single logical location during execution with the SPEC-provided test inputs. Although more expensive flow-sensitive algorithms [22] can obtain better average points-to sets than the scalable analyses used in [19], they still do not in general yield points-to sets as small as the dynamic sets.

[5] Sometimes the terms *pointer analysis*, *points-to analysis*, and *alias analysis* are used interchangeably in the literature. In this paper, we use the term alias analysis for an analysis that determines for a memory-access expression the set of other expressions (variable names or pointer dereferences) that may refer to the same memory location.

[6] This rule does not take aliases via union data structures into account. For the rest of the paper we do not distinguish fields of structures or unions, i.e., a reference of a structure or union field is treated as if the whole structure or union were referenced. Since distinguishing structure and union fields in a pointer analysis can potentially make the algorithm much less efficient, many pointer analyses do not distinguish them, e.g., [4, 23].

	Source Lines	Reachable Functions	Description
art	1,270	22	image recognition, neural networks
equake	1,513	27	seismic wave propagation simulator
mcf	1,909	24	combinatorial optimization
bzip2	4,639	63	compression
gzip	7,757	62	compression
parser	10,924	297	word processing
ammp	13,263	161	molecular dynamics
vpr	16,973	255	circuit placement and routing
twolf	19,748	167	placement and global routing
vortex	52,633	643	object-oriented database
mesa	49,701	770	graphics package
gap	59,482	826	group theory interpreter

Table 1: Sizes and descriptions of the programs used in the experiments.

Since all logical locations contained in a dynamic points-to set must also be contained in its static counterpart, using dynamic points-to sets to compute aliases in a program enables us to obtain a lower bound on the aliases present in a program. Using these optimistic aliases in the compiler's optimization phase therefore provides an upper bound on the improvement we can hope to achieve by using `restrict` pragmas in a program. Obviously, the `restrict` pragmas obtained by this method are optimistic, i.e., only guaranteed to be sound for the program input that was used to compute the alias relationships. For a bound on the potential impact on optimization, however, this is immaterial. When used for optimization in practice, however, we must guarantee that the `restrict` pragmas are sound for all inputs; Section 5 outlines how to generate run-time guards to make `restrict` pragmas safe and enable aggressive optimizations at the same time.

2.2 Dynamic Alias Analysis

To generate the dynamic alias information used to place `restrict` pragmas, we used a slightly modified version of the instrumentation tool *Tumi* [18, 19], which works in three steps. First, a static points-to analysis is run on the application source code. For each pointer and dereference point, it computes a conservative approximation of the set of logical locations a pointer may point to.

Second, the application is instrumented, inserting a call to a run-time routine (at the entry of each procedure) that, for each procedure argument `arg`, matches the address contained in `arg` with the run-time addresses of potential pointer targets for `arg` (identified by the static points-to analysis of the first step). [7] At run-time, when the run-time library routine identifies the same logical location for two distinct arguments, the arguments are marked as aliased.[8]

As the final step, the instrumented application is compiled, and executed on some input. Upon termination, the instrumentation code saves a record of the aliasing relationships that occurred during execution. In this process the address matching step is essential: since distinct run-time addresses may refer to the same logical location, simply recording the pointer addresses is not sufficient to construct the aliasing relationships at run time. More details about the instrumentation algorithm can be found in [18, 19].

To obtain the dynamic aliasing relationships for the applications in this study, we used the SPEC-provided reference inputs to ensure that the resulting `restrict` pragmas were sound for the timing runs, which use the reference inputs. Das's One-Level Flow algorithm [4] was used as the static points-to analysis of the first step.

3 Experimental Setup and Workload

To determine the performance impact of `restrict` pragmas on realistic programs, we chose to use applications from the SPEC CPU2000 benchmark suite since SPEC benchmarks are of considerable size, cover a wide range of tasks (e.g., simulations, group theoretic computations, graphics, databases, word processing), are actually used in practice, and are generally used to evaluate CPU performance, i.e, they are generally considered to be good indicators of CPU and compiler performance. Table 1 shows the programs used with their sizes (in lines of C code) and the number of statically reachable functions, for which we generated `restrict` annotations.

We performed our experiments on two different platforms. The first platform was a Sun Blade-100 workstation with a 500MHz UltraSPARC-IIe processor, 256 MByte of RAM, and running Sun OS 5.8 (Solaris). For this platform

[7] Since arguments that have empty (static) points-to sets are guaranteed not contain pointers at run-time, only arguments with non-empty static points-to sets are instrumented.

[8] In addition, for the `restrict` annotation to be correct, other memory references inside the procedure must also not be aliased to the procedure argument. In our experiments, we verified that this was the case by manual inspection of all dynamic points-to sets within a procedure.

	Un-aliased at run time	Aliased statically
equake	26	16
mcf	3	2
parser	22	18
ammp	12	2
vpr	50	35
twolf	11	7
vortex	397	397
mesa	18	18
gap	49	48

Table 2: Column two shows the number of functions for which none of the arguments were found to be aliased at run-time; column three the number of those that were reported to be aliased by the static pointer analysis. For instance, for `vpr`, there were 50 functions with no aliases in the arguments at run time; out of those 50, 35 were reported to have aliased arguments by the static points-to analysis. `art, bzip2` and `gzip` are not shown in the table since the static alias analysis was able to determine that no aliases were present in these applications for the procedure arguments.

---

we used the Sun Studio One C compiler, version 7, and compiled the benchmarks with the highest optimization option -xO5, using cross-file optimization (-xcrossfile) and the -fast option, which turns on a couple of optimizations designed to improve execution speed (e.g., data alignment along double word boundaries to improve memory access times).

The second platform was a Hewlett Packard ZX6000 workstation with two 900 MHz Itanium 2 processors, 1 GB of RAM, and running Redhat Linux 7.2. For this platform we used the ecc compiler from Intel, version 7.0 at highest optimization level (-O3), with interprocedural optimization across files (-ipo), and the -restrict option to enable the use of the `restrict` qualifier.

For both platforms, all timings were obtained using the `runspec` script provided with the SPEC CPU2000 benchmark suite. We ran the benchmark several times on otherwise unloaded machines and took the best (shortest) execution time of all runs. We used the reference input data sets, which represent the largest input data sets available in the SPEC benchmark suite. We did not use feedback directed optimizations in our experiments to avoid any interaction of profiling with the `restrict` optimization.

4 Results

4.1 Aliasing Results

Table 2 shows for each benchmark the number of reachable functions with at least one pointer-type argument (on which the `restrict` pragma might be used) and for which none of the arguments was found to be aliased at run-time. Column three, for comparison, lists the number of those functions for which the static pointer analysis (Das' algorithm [4]) reported that some arguments might be aliased. For instance, for `gap` 49 functions with at least one pointer-type argument did not have any aliases at run-time in their arguments. For 48 of those 49 functions, however, the static points-to analysis reported that the arguments might be aliased. With the exception of `mcf` and `ammp`, generally the alias information produced by the static pointer analysis was not a very accurate representation of the run-time alias relationships. This shows that there is an opportunity for exploiting run-time alias information in practice, but how effectively can it be realized with `restrict` pragmas?

4.2 Best Case Speedups with restrict

To answer this question, we generated the `restrict` pragmas, compiled and executed the programs as described in Section 3. Figure 3 shows the speedup for each application when compiled with `restrict` pragmas versus the execution time of the same program that was compiled without any `restrict` pragmas. In both cases, all other optimizations options were identical (`-xO5 -fast -xcrossfile` for the Sparc platform and -O3 -ipo -restrict for the Itanium platform).[9] Since no aliases where found by the static pointer analysis for `art, bzip2` and `gzip`, they have been omitted from the graphs.

On average (geometric mean) across all benchmarks, we see only a modest improvement of just a little under 1% on the Sparc platform; on Itanium, the average was close to one. The best speedup improvement was achieved for application `ammp` on the Sparc platform, which improved by 7.5%.

While `ammp` was the benchmark with the largest speedup on the Sparc platform (in fact, the largest speedup in general), its performance was identical to the unannotated version on the Itanium platform. We even found that the

[9] In the Sun Studio One C compiler recognition of the `restrict` pragma is on by default, for the Intel ecc compiler it has to be turned on explicitly with the -restrict switch.

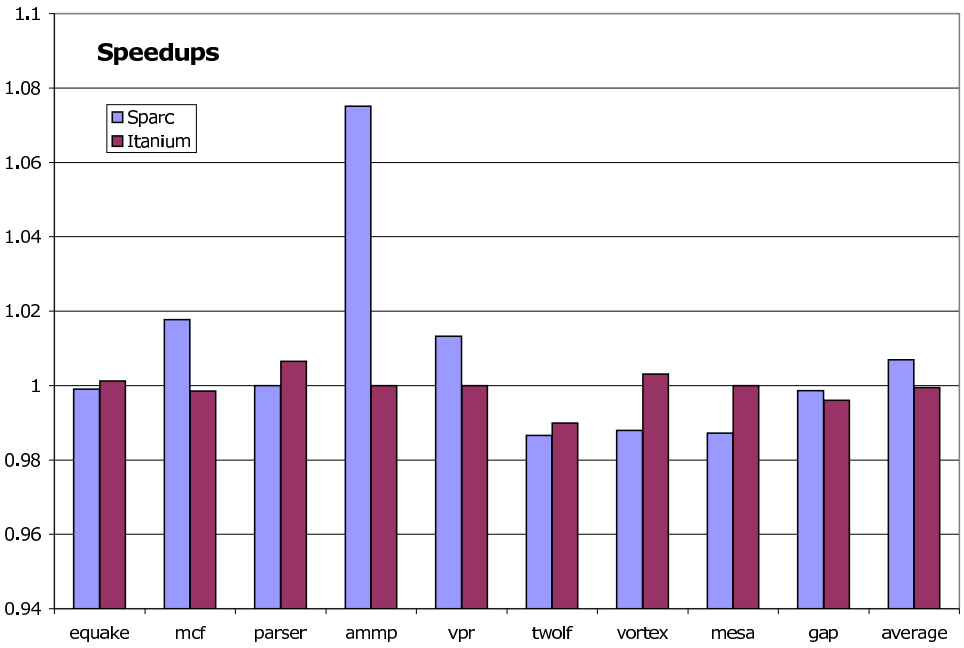

Figure 3: Upper bound speedups of the applications obtained by placing `restrict` on all arguments that were found to be unaliased when run on the reference input.

---

ecc compiler produced identical binaries for the benchmark with and without `restrict` pragmas. Since in other benchmarks (and as mentioned in Section 1) ecc actually is able to take advantage of the pragma we looked more closely at the binary that was generated.

The compiler performs aggressive data speculation exploiting the advanced load instruction on the Itanium (for instance `ldfd.a`) which loads a value into a register and stores the memory address from which the data was loaded in the *Advanced Load Address Table* (ALAT) hardware structure. Before the use of a value loaded with an advanced load instruction, the compiler inserts a `chk.a` instruction. This instruction will branch to fix-up code to reload the value if its load address is not in the ALAT and recompute any speculatively computed values based on the value loaded with the advanced load. Since the processor automatically removes an entry from the ALAT when a store to an address in the ALAT occurs, this ensures correctness in case of an intervening store. Using this feature, the compiler apparently found no additional optimization opportunities with the `restrict` annotations beyond the normal data speculation that is already performed for possibly aliased loads.

We ascertained this explication using the Itanium processor's performance counters. They showed that during the execution approximately 35 million check instructions were execution of which less than 0.04% failed the check, i.e., required a reload from memory. That is, the compiler is able to overcome poor static alias information by using the Itanium's hardware support for data speculation, breaking the pointer-induced data dependences, and so aggressively optimizing the application.

For the Sparc platform, `mcf` and `vpr` were the only other benchmarks for which the `restrict` pragma resulted in speedups. For several of the benchmarks, the code with `restrict` actually ran somewhat slower. We believe this to be due to degraded instruction cache performance since code compiled with `restrict` tends to be considerably larger than the original code because the compiler performs loop parallelization, aggressive unrolling etc. which can improve run-time but result in larger instruction cache footprints.

For the Itanium platform, `equake` and `parser` also showed minimal speedups, however, below 1%. For `vortex`, the `restrict` annotation led to a performance degradation of about 1% on the Sparc platform, and minimal improvement on the Itanium platform. `gap`'s performance was virtually unchanged on both platforms with an insignificant degradation in both cases.

5 Sound, Aggressive restrict Annotations

The results in Section 4 indicate that there is generally too little performance gain from `restrict` annotations to justify the additional programmer effort and potential for errors by incorrectly placed annotations. However, some applications may benefit noticeably (e.g., `ammp` on the Sparc platform in our experiments), and for others even minor improvements may be welcome if they can be obtained without additional programmer effort.

However, to realize those improvements where possible without user intervention, `restrict` annotations placed by an automatic tool have to be sound. The data shown in Table 2 demonstrates that a tool relying on static (pointer) analysis alone will generally not be able to place `restrict` annotations aggressively and would usually not be able to effectively exploit non-aliasing. Yet, annotations generated exclusively based on dynamic alias information, which is guaranteed to detect non-aliasing when present, is unsound since it cannot guarantee the absence of aliasing across all inputs. Therefore, we propose the following approach that combines dynamic and static analysis with run-time checks to generate aggressive yet sound alias annotations.[10]

To ensure the soundness of a routine some of whose arguments have been qualified with `restrict`, guard code is added, which checks at run-time whether the arguments are in fact alias-free. If an alias is detected at run-time, the guard condition dispatches to a conservatively optimized version of the routine, otherwise the aggressively optimized routine (optimized assuming no aliasing) is executed. Figure 4 shows an example.[11] The original `vmul` routine has been replaced by a routine that checks for an alias for `a` or `b`. If the check returns `true` (no alias is present), then the routine that was optimized with the `restrict` annotation is called (renamed to `vmul-r`); otherwise `vmul-s` is called, the original `vmul` routine without any `restrict` pragma (i.e., optimized assuming potential aliasing). Effectively, we have specialized routine `vmul` for the presence or absence of aliases. To avoid additional call overhead, the `vmul` routine, which only serves as a trampoline to select the correct specialized code version, will be inlined into its caller (indicated by the `inline` keyword in the example). If the transformation is performed at the intermediate code level, this inlining could be performed at compile time if all code is available then, or at link time when the code for all the callers of `vmul` becomes available.

Since the run-time check `check_vmul_alias` adds an additional run-time cost to the original routine, it is important that its cost be recouped by sufficiently often executing the more aggressively optimized version. To determine whether the transformation is worthwhile, the compiler should assess the likely ensuing benefit, for instance by using an approach similar to the inline trials used by Dean et al. [7]. At the same time, it is important to make the run-time alias check as cheap as possible.

Our instrumentation tool Tumi provides library routines that can be linked with an application to automatically detect aliases at run-time. The same steps used to identify the logical location pointed to by a pointer can be used to check whether aliasing occurs: if address matching for the possibly aliased locations returns the same logical location, aliasing is present and the conservatively optimized code version can be dispatched, otherwise the aggressively optimized version. Unfortunately, currently this requires that the application be run with some instrumentation code in place, which degrades performance considerably.[12] Looking at possible hardware mechanisms to make the instrumentation more efficient, is an interesting area for future research.

While using Tumi's general alias detection mechanism is typically too expensive, in many cases much cheaper tests can be generated as guard code. If, for instance, the pointer analysis can determine statically that aliases with `a` and `b` will either not occur or that that the two arguments will point to the same base address, `check_vmul_alias(a,b)` can be implemented as a simple pointer comparison of `a` and `b`. How to generate those cheaper tests automatically and with minimal additional run-time (pointer profiling) overhead, is part of future research.

6 Related Work

Bernstein et al. [3] use a static memory disambiguator for array references to generate run-time conditions that check whether an alias actually occurs at run-time in those cases where the static analysis cannot rule out aliasing. In their approach, tests for these alias conditions are inserted into inner loops and (statically) two code versions are generated for the loop: one optimized assuming no aliasing and one with aliasing. To minimize the run-time cost of selecting the appropriate code version, they perform their transformation only when the dynamic condition is loop-invariant so that it can be hoisted out of the loop. On the downside, limiting their transformation to this subset of all inner loops also diminishes the applicability of their approach. On integer codes they found no improvements, and with the exception of two applications, floating point benchmarks also showed no speedups. The two applications that showed significant speedups (`alvin` 117% and `ear` 37%), are very small compared to the SPEC CPU2000 benchmarks; `alvin` is 200 lines and `ear` 3,000 lines of C code, i.e., really significant speedups were only achieved for kernel-size applications.

Postiff et al. [21] investigated hardware support to enable the promotion of a variable from memory to a registers. In the presence of an alias, this can generally not be done (at least without repeatedly reloading the register which defeats the optimization purpose). They propose a combined compiler-architecture approach, where the compiler

[10] Note that this idea per se is not new, and has been applied in similar fashion by several researchers [3, 6, 11]. While previous approaches typically relied on heuristics to decide when code speculation based on aliasing should be performed, the approach proposed here is based on dynamically gathered alias profiles that represent likely alias relationships at run time.

[11] The example shows the transformation in C code; in practice, the transformation would most likely be performed at the compiler's intermediate representation level.

[12] For all benchmarks in this study, the slowdown incurred by Tumi's instrumentation was over 10% [18]. Therefore, none of the best-case speedups achieved with `restrict` lead to actual speedups when Tumi's current alias detection mechanism to dispatch to the appropriate code version at run time is used.

```
inline void vmul(int n, double*  a,  double* b) {
   if (check_vmul_alias(a,b))
       vmul-s(n, a, b); /* aliased */
   else
       vmul-r(n, a, b); /* not aliased */
}
void vmul-r(int n, double* restrict a, double* restrict b) {
   int i
   for (i=0; i<n; i++)
      b[i] = a[i] * a[i];
}
void vmul-s(int n, double* a, double* b) {
   int i
   for (i=0; i<n; i++)
      b[i] = a[i] * a[i];
}
```

Figure 4: Guard code example. The original routine `vmul` from Figure 1 has been converted into a trampoline routine that is inlined into callers of `vmul`. Its only purpose is to select the correct code version of `vmul` at run time: `vmul-r`, i.e., `vmul` optimized with `restrict` annotations is selected if the run-time non-aliasing check for the `restrict`-annotated arguments succeeds; otherwise, `vmul-s` the safely optimized version of `vmul` is selected. By inlining the `vmul`-trampoline routine no additional function call overhead is added. If the alias check `check_vmul_alias` is simple enough (for checking that arrays `a` and `b` are not aliased a simple pointer comparison may suffice), it can also be inlined, further reducing the run-time checking overhead.

---

loads the address of a promoted variable into a special processor data structure (the store-load address table). When aliased accesses to the variable occur, they are automatically intercepted by the processor and the value is forwarded directly into the register into which the variable was promoted. In their simulation they found a moderate reduction in the number of loads and stores on average, however, it is unclear what the bottom-line performance effect of their combined approach might be in practice. Note however, that the proposed hardware structure is very similar to the ALAT structure on the Intel Itanium processor [12].

Das et al. [5] compare several static pointer analyses to estimate the potential impact on optimization arising from more precise static pointer analysis. They demonstrate the number of aliases reported by fast control-flow insensitive analyses is not significantly different from the number reported by more expensive flow-sensitive algorithms. Their works differs from ours inasmuch as they do not look at the potential for performance improvement arising from compiler pragmas (such as `restrict`), nor do they compare concrete execution times resulting from different alias analysis precision.

Ghiya et al. [10] do exactly this by comparing the run times of SPEC benchmarks when compiled with different memory disambiguation algorithms in their compiler for the IA-64 (Itanium) architecture. Similar to Das et al., they found that more expensive algorithms in most cases did not provide additional benefits. In practice, a more important property for the optimization of their C benchmarks was whether or not the algorithms distinguished structure fields.

Foster and Aiken [1, 9] define a precise semantics for the `restrict` qualifier and then present a type analysis that automatically infers whether `restrict` annotations written by a programmer obey the semantics. Their goal is to detect violations of certain program correctness properties, for instance, that a lock is not acquired again before releasing it. While their approach might benefit from dynamic alias information, in many cases the static alias information seems to be sufficient for detecting programming errors.

Koes et al. [14, 15] present work similar to ours. They propose another programmer annotation similar in spirit to `restrict`, that allows the programmer to specify the non-aliasing of two arbitrary pointers and study its usefulness for performance optimization. They also use profiling to determine when the annotations would be useful. However, they do not present an automatic scheme to ensure soundness and their performance numbers are only simulated and therefore have to be taken with a grain of salt. Their simulation results, however, show the same trend: for most applications there was only minimal improvement, and best-case (simulated) speedups were under 30% in all cases for realistic processor assumptions.

Fernandez and Espasa [8] present an approach for performing alias analysis on executable code at link time. In addition, they mention using run-time profiles to perform speculative optimization, without however, describing an automatic approach to achieve soundness.

Huang et al. [11] perform speculative disambiguation for the LIFE VLIW architecture. They produce specialized code that disambiguates memory at run-time, similar in nature to the approach outlined in Section 5. The difference lies in the granularity of the approach: they look at individual memory dependences (e.g., a RAW (read-after-write) dependence) for individual load and store instructions and build a dependence tree to determine the affected instructions. At run-time, either the speculative code or conservatively optimized code (obeying the potential dependence) is executing using the VLIW's predication mechanism.

Davidson and Jinturkar [6] use a software-based dynamic memory disambiguation approach to enable aggressive

loop unrolling (to increase instruction level parallelism) that is otherwise impossible because of possible memory aliases and show that significant performance improvements are possible with this technique, another indication for the potential of run-time alias information.

7 Conclusions and Future Work

In this paper we have shown that for large C programs, the impact of using `restrict` annotations is usually limited. While those annotations can dramatically improve performance of small kernels, larger applications do not significantly benefit from even optimistic `restrict` annotations. Since the potential benefits are meager but the potential for introducing errors high, we conclude that programmer-specified non-aliasing is a bad idea in general for improving program performance.

An alternative way of enabling compilers to perform aggressive optimization even when static point analyses indicate it might be unsafe to do so, is dynamic alias analysis. In those cases in which noticeable performance improvements are theoretically possible, the run-time cost for guard code that ensures soundness of the annotations becomes important. Exploring ways to make these guards cheap, possibly by special (simple) hardware support and making the automatic generation of `restrict` more efficient, are interesting areas for future research. Moreover, we are planning on integrating the dynamic alias information directly into the optimizer thereby obviating the need for `restrict` annotations completely.

References

[1] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 129–140. ACM Press, 2003.

[2] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Ph.D. dissertation, University of Copenhagen, DIKU, May 1994.

[3] David Bernstein, Doron Cohen, and Dror E. Maydan. Dynamic memory disambiguation for array references. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 105–111, San Jose, CA, USA, November 1994.

[4] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, Vancouver, BC, Canada, June 2000.

[5] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS '01: The 8th International Static Analysis Symposium*, Lecture Notes in Computer Science, Paris, France, July 16–18 2001. Springer-Verlag.

[6] Jack W. Davidson and Sanjay Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 125–132. IEEE Computer Society Press, 1995.

[7] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 273–282, 1994.

[8] Manel Fernandez and Roger Espasa. Speculative alias analysis for executable code. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques (PACT '02)*, pages 222–231, Charlottesville, Virginia, September 2002.

[9] Jeffrey S. Foster and Alex Aiken. Checking Programmer-Specified Non-Aliasing. Technical Report UCB//CSD-01-1160, University of California, Berkeley, October 2001.

[10] Rakesh Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–58, Snowbird, UT, USA, June 2001.

[11] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative disambiguation: a compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 200–210. IEEE Computer Society Press, 1994.

[12] Intel Corporation. *Intel Itanium Architecture Software Developers Manual*, 2002.

[13] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 9899:1999 Programming languages – C*, 12 1999.

[14] David Koes, Mihai Budiu, Girish Venkataramani, and Seth Copen Goldstein. Programmer specified pointer independence. Technical Report CMU-CS-03-128, Carnegie Mellon University, April 2003.

[15] David Koes, Mihai Budiu, Girish Venkataramani, and Seth Copen Goldstein. Programmer specified pointer independence. In *Proceedings of 2nd ACM SIGPLAN Workhop on Memory System Performance, MSP 04*, June 2004.

[16] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM '92 SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, CA, USA, June 1992.

[17] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, pages 199–215, Toulouse, France, September 1999.

[18] Markus Mock. *Automating Selective Dynamic Compilation*. Ph.D. dissertation, University of Washington, Department of Computer Science & Engineering, August 2002.

[19] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, Snowbird, UT, USA, June 2001.

[20] Hemant D. Pande, William Landi, and Barbara G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[21] Matthew Postiff, David Greene, and Trevor Mudge. The store-load address table and speculative register promotion. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-33)*, pages 235–244, Los Alamitos, CA, December 10–13 2000. IEEE Computer Society.

[22] Barbara G. Ryder, William Landi, Philip A. Stocks, Sean Zhang, , and Rita Altucher. A schema for interprocedural side effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, March 2001.

[23] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, FL, USA, January 1996.