

The Limits of Alias Analysis for Scalar Optimizations

Rezaul A. Chowdhury¹, Peter Djeu¹, Brendon Cahoon²,
James H. Burrill³, and Kathryn S. McKinley¹

¹ Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712, USA,
{shaikat, djeu, mckinley}@cs.utexas.edu

² Conformative Systems, Austin, TX 78759, USA,
brendon.cahoon@conformative.com

³ Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA,
burrill@cs.umass.edu

Abstract. In theory, increasing the precision of alias analysis should improve the results of compiler optimizations on C programs. This paper compares the effectiveness of several popular alias analyses on nine scalar optimizations. We include an analysis that assumes no aliases to establish a very loose upper bound on optimization opportunities. We statically measure the number of optimization opportunities in the Scale compiler for each analysis performed on thirty-six C programs. We find that, in practice, the precision of the alias analysis rarely inhibits these optimization opportunities. Previous work finds similarly that the increased precision of specific alias algorithms provide little benefit for scalar optimizations, and that simple static alias algorithms uncover almost all dynamically determined aliases. This paper, however, is the first to provide a static methodology that indicates that additional precision is unlikely to yield improvements for a wide set of optimizations. For clients with higher alias accuracy demands, this methodology can help pinpoint the cases where additional accuracy is needed.

1 Introduction

An enormous amount of research is devoted to developing compiler alias analysis for C programs in order to determine if two distinct variables may reference the same memory location [1, 5, 9, 16, 23, 28, 29, 34, 35]. The literature shows an essential tradeoff: improving the precision of alias analysis increases the cost of performing it, and this increase can be substantial. In theory, a more precise alias analysis improves the clients' results. The clients of alias analysis are numerous, and include, for example, improving program performance [17], finding bugs [18], and pinpointing memory leaks [19]. In some cases the theory holds true. For instance, error detection [18] avoids many false positives through the judicious application of precise alias analysis. Certain parallelization optimizations, likewise, may significantly benefit from increased precision [34].

This paper studies using alias analysis on scalar compiler optimizations designed to improve performance. We implement three analyses: address-taken, Steensgaard [29], and Shapiro-Horwitz [27]. In addition, we include an upper bound methodology that assumes there are no alias relations. Most alias analysis evaluations count the number of aliases statically and assume that fewer aliases are better. We instead use a static upper bound of no aliases. This bound is not guaranteed to be tight, but the compiler is

never inhibited by an alias when applying optimizations. This bound is only useful for static evaluation.

We use nine scalar optimizations to determine the difference between using alias analyses and the upper bound on the number of optimizations that the compiler performs on thirty-six C programs from popular benchmark suites. The optimizations are sparse conditional constant propagation, global variable replacement, loop unrolling, loop invariant code motion, global value numbering, copy propagation, useless copy removal, array access strength reduction, and scalar replacement. We measure optimization applications individually and as a group across our set of alias analyses. Experiments show there is very only a small gap between Shapiro-Horwitz and our static upper bound. The increased precision of Shapiro-Horwitz attains minor improvements over Steensgaard with respect to optimization, and both are somewhat better than address-taken. The largest difference is for loop invariant code motion, for which the upper bound methodology detects on average six percent more loop invariant computations than the best analysis.

Our results are in agreement with the few other studies on the effect of alias analysis on scalar optimizations, which all suggest that a simple alias analysis will suffice [3, 12, 14, 15, 21]. For example, Hind and Pioli show that few additional scalar analysis opportunities come from increasing alias precision on twenty-three C programs [21]. Diwan et al. measure dynamically the additional opportunities for two optimizations on Modula-3 programs and find that improving alias analysis will not benefit these optimizations much, if at all [14]. Das et al. measure aliases dynamically, without respect to an optimization, and find that a simple analysis misses 5% of independent memory accesses in nine C programs [12].

In their work, Das et al. suggest the following [12]: “Ideally, we would like to repeat their study [Diwan et al.] for every conceivable optimization and every pointer analysis.” Although we of course do not study “every conceivable optimization,” the range is more numerous than that in previous work. The most important contribution of this paper, however, is obviating the need for evaluating all pointer analyses. For thirty-six C programs, no matter how much additional precision an alias analysis provides over Steensgaard or Shapiro-Horwitz, that extra precision is unlikely to benefit scalar optimizations.

The remaining sections of this paper are organized as follows. Section 2 further compares our methodology to previous evaluations of pointer analysis. Section 3 briefly describes the alias analyses: address-taken, Steensgaard [29], Shapiro-Horwitz [27], and the no aliases upper bound. Section 4 and 5 introduce our compiler framework, intermediate representation (IR), how the IR represents aliases, and the test set of scalar optimizations. Section 6 presents the experimental methodology. Section 7 presents measurements of optimization opportunities and compile times. It shows that a fast, simple alias analysis produces the same optimization applications as the loose upper bound, demonstrating that no additional amount of precision will yield many, if any improvements to scalar optimizations and thus performance. Other alias analysis clients can use this methodology to find opportunities for improvement, and of course, clients such as error detection, memory leaks, and data reorganization are likely to require additional accuracy.

2 Related Work

This section describes comparative studies of alias analyses. We focus on the closest related work that use the clients of the alias analyses for evaluation.

The evaluation of most new alias or points-to analysis algorithms reports the size of the static points-to information they compute. For example, Emami et al. [16], and Wilson and Lam [34] introduce new context-sensitive interprocedural points-to algorithms and evaluate them using the size of the points-to sets.

Other researchers evaluate alias analysis algorithms by reporting changes to the size of the static points-to information when the precision of the algorithm changes. Ruf evaluates the effect of context-sensitivity on the precision of alias analysis [25]. Ruf concludes that adding context-sensitivity does not improve the precision for the benchmarks he examines. Liang and Harrold introduce a context-sensitive flow-insensitive algorithm, and they compare their algorithm to three other algorithms [23]. Yong et al. present a tunable pointer analysis framework that distinguishes fields structures [35].

Other related work focuses on the clients. Hind and Pioli compare five alias analysis algorithms, and they study how the precision of alias analysis affects typical client analyses, along with two optimizations [21]. The client analyses are Mod/Ref analysis, live variable analysis, reaching definitions analysis, and interprocedural constant propagation. We corroborate their results, but we do so within the context of a new compiler and with a focus on a more comprehensive selection of client optimizations, rather than analyses. In earlier work, Hind and Pioli present an empirical comparison of four alias analysis algorithms with different levels of flow sensitivity [20]. They measure the precision of the analysis results, and the time and space to compute the results. They do not study the effect of analysis quality on optimizations in this earlier work.

Shapiro and Horwitz compare the precision of four flow and context-insensitive pointer analysis algorithms [27, 28]. They test the precision of the pointer analyses using GMOD analysis, live variable analysis, truly live variable analysis, and an interprocedural slicing algorithm. Shapiro and Horwitz conclude that more precise analysis does improve the results of some, but not all of the clients. Stocks et al. compare the flow-sensitive and context-sensitive analysis on Mod analysis [30]. They conclude that more precision helps improve the precision of Mod analysis. These two papers are focused on analysis clients rather than the optimization clients we use.

Das et al. measure the effect of pointer analysis on optimizations [12]. Their goal is to evaluate whether flow-insensitive pointer analysis is sufficient for compiler optimizations. Das et al. do not use any specific optimization or compiler, but instead develop a new metric for evaluating the precision of pointer analysis.

Diwan et al. evaluate three alias analysis algorithms using static, dynamic, and upper bound metrics [13–15]. They demonstrate the effect of the three analyses using redundant load elimination and method invocation resolution. They show that a fast and simple alias analysis is effective for type-safe languages. Bacon and Sweeney find similar results for C++ method resolution [3].

Ghiya and Hendren empirically show that their points-to analysis and connection analysis can improve loop-invariant removal and common subexpression elimination, array dependence testing, and program understanding [17]. They do not experiment

with the precision of the analysis, and they concede that a conservative analysis may provide the same benefits for the scalar optimizations.

Cooper and Lu use pointer analysis to perform register promotion, which is an optimization that converts references to scalar values in memory to a register [10]. Identifying aliases is important for this optimization, but Cooper and Lu do not show how the precision of the analysis affects optimization opportunities.

Our work is in the spirit of the last four studies, all of which focus on the client optimizations. We are, however, broader in scope in terms of the range of optimizations and the number of programs. In addition, we use a new methodology that computes the very loose static upper bound that shows, for our programs and optimizations, that no additional precision is needed.

3 Alias Analysis

We study the following alias analysis algorithms.

- Address-taken
- Steensgaard [29]
- Shapiro-Horwitz [28]
- Assume no aliases

Address-taken is very simple and is linear in the size of the input program. The compiler assumes all heap objects are potential aliases of each other, and includes in this set all variables for which the program explicitly takes their address. The address-taken algorithm produces the most conservative set of alias relations.

Steensgaard's algorithm is an interprocedural flow-insensitive analysis characterized by almost linear running time and linear space complexity, but does not necessarily produce precise results [29]. It is based upon type-inference methods using alias relations. It results in alias sets that are symmetric and transitive.

The Shapiro-Horwitz algorithm [28] extends and increases the precision of Steensgaard's algorithm without a significant effect on running time. A parameter specifies the precision between the least precise version (Steensgaard) to the most precise version which is equivalent to Andersen's algorithm [1]. The analysis time varies inversely with precision. We choose an intermediate point for our evaluation.

Assuming no aliases serves as a static metric for evaluating the effect of alias information on subsequent clients. It simply generates an empty set of alias relations to communicate to the optimizations that there are no aliases. Since it makes the typically false assumption that there are no aliases, the generated executable is usually incorrect. The purpose of this analysis is not to generate correct code, but rather, to establish a reference point for the maximum number of optimizations the compiler could perform. Hence, no aliases provides a loose upper bound.

4 The Scale Compilation System

This section briefly outlines our compilation framework, and then provides details about how it represents and uses alias information. The next section enumerates the client optimizations and how they use aliases.

Scale is a flexible, high performance research compiler for C and Fortran, and is written in Java [24,32]. Scale transforms programs into a control flow graph, performs alias analysis, and uses the results to build a static single assignment (SSA) [11], machine-independent intermediate representation (IR) that we call *Scribble*. Scale performs optimizations on Scribble, and then transforms Scribble to a low-level, more machine dependent RISC instruction-like IR on which it performs a variant of linear scan register allocation [31]. It outputs C or assembly for the Alpha and Sparc processors.

The Scale compiler transforms the control flow graph (CFG) to SSA form after it performs alias analysis. SSA form ensures that each use of a scalar variable, or a virtual variable created during pointer analysis, gets its value from a single definition [11]. Scale utilizes Chow et al.'s technique for representing pointers which makes a distinction between definitions that must occur and may occur [7]. Chow et al. define virtual variables to represent indirect variables (e.g., *p). They create a unique virtual variable for all indirect variables that have similar alias characteristics. They perform alias analysis on the virtual variables and the scalar variables. It is unclear which alias analysis algorithm they use.

Scale's analysis has a subtle difference; it performs alias analysis prior to creating the virtual variables. After performing the analysis, Scale defines a unique virtual variable for each alias group, where an alias group is the set of variables which share the same aliases.

In Scale, the SSA form thus includes may and must definitions which are linked to uses by corresponding edges. The optimizations traverse these edges to find definitions, recurrences, etc. All scalar optimizations in Scale (except useless copy removal) manipulate the SSA form of the control flow graph. The precision of disambiguation information derived from alias analysis directly affects the quality of this SSA form. Thus, alias analysis precision has a direct effect on the number of optimizations that a scalar optimization performs.

5 Scalar Optimizations

We study the effects that alias information has on the following scalar optimizations. Scale performs these optimizations on SSA Scribble format except for useless copy removal. We briefly describe each optimization and the *criteria* that we measure to expose optimization opportunities. These optimizations target scalar variables, loads, scalar expressions, array address arithmetic, and heap allocated arrays. We expect that alias analysis will have more effect on additional optimizations that specifically target heap pointers.

Loop Invariant Code Motion (LICM) LICM recognizes computations (including loads) in loops that produce the same value on every iteration of the loop and moves them to appropriate locations outside the loop. For nested loops, it moves computation out of as many inner-loops as possible without destroying program semantics. LICM speeds up program execution by reducing the number of instructions executed. In Scale, SSA use-def links indicate where the CFG node gets its definitions, and LICM moves computations to the outer-most basic block in which the definition is available. More precise alias information provides more accurate dependency information and thus allows safe

movement of CFG nodes out of more inner-loops. In order to preserve the program semantics, it only moves stores for temporary variables. Procedure calls and expressions involving global variables are never moved.

Criteria: number of expression moved.

Sparse Conditional Constant Propagation (SCCP) SCCP discovers variables and expressions that are constant and propagates them throughout the program. SCCP correctly propagates constants even in the presence of conditional control flow. This optimization speeds up program execution by evaluating expressions at compile time instead of runtime and improves the effectiveness of other optimizations, such as value numbering. Scale uses Wegman and Zadeck's SCCP algorithm on SSA-form [33]. If alias analysis is performed, there will be more instances where one can determine whether an expression is constant. Scale uses the alias analysis information to obtain the value through pointer operations with the may-use/may-def information.

Criteria: number of constants propagated.

Copy Propagation (CP) CP discovers assignments of the form $x \leftarrow y$ and replaces any later use of variable x by y as long as neither x nor y is changed by any intervening instructions. CP then removes the original assignment statement. Scale does not propagate a copy if (1) the right hand side variable of the assignment statement contains May-Use information indicating that it may be involved in an alias relationship, or (2) either of the two arguments in the assignment are global variables.

Criteria: number of copies propagated.

Global Value Numbering (GVN) Scale uses the dominator tree-based value numbering technique by Briggs et al. [4]. GVN determines whether two computations are equivalent and if so, removes one of them. It does so by assigning a *value number* to each computation in such a way that any two computations with the same value number always compute the same value. SSA form simplifies this process. GVN works on entire procedures instead of single basic blocks, as does traditional value numbering. It improves program running time by removing redundant computations.

Criteria: number of expressions removed.

Loop Unrolling (LU) LU replaces the body of a loop by several copies of the body and adjusts the loop control code accordingly. Aliases inhibit loop unrolling only if the loop control variables may be aliased with loop varying variables. LU reduces the number of instructions executed during runtime at the cost of increased code size. This optimization may also improve the effectiveness of other optimizations, such as common-subexpression elimination and strength reduction.

Criteria: number of loops unrolled.

Scalar Replacement (SR) Register allocators usually do not allocate subscripted variables to registers. Scalar Replacement tricks the allocator by replacing subscripted variables by scalars and thus making them available for register allocation. Dependence analysis is used to locate patterns of consistent re-use of array elements and then SR replaces those re-uses by references to scalar temporaries. The same trick is used for both array loads and array stores. This optimization reduces the number of loads and stores in programs and is very effective in reducing execution times.

Criteria: number of array loads replaced.

Global Variable Replacement (GVR) This transformation replaces references to global variables with references to local variables by copying the global into a local. The variable may not be aliased to another variable that the procedure modifies.

Criteria: number of loads to global variables replaced.

Array Access Strength Reduction (AASR) AASR uses the *method of finite differences* to replace expensive operators in array element address calculations with cheaper ones. Scale targets array index calculations in the inner-most loops, and replaces multiplications with additions when possible. It also moves any resulting loop invariants outside the loop and folds constant expressions as part of this process. AASR reduces the number of multiplication operations executed during runtime.

Criteria: number of array index calculations replaced.

Useless Copy Removal - (UCR) Useless copy statements are of the form $x \leftarrow x$. Scale creates these statements by transitioning to and from SSA form and via other optimizations. Because transitioning from SSA form introduces copies and new temporary variables, this optimization is very sensitive to the form of the SSA graph. Scale finds and removes useless copy statements in the CFG form.

Criteria: number of useless copies removed.

6 Methodology

Table 1 enumerates our test suite, which consists of programs from the following benchmarks: SPEC 95, SPEC 2000, Austin from Todd Austin [2], McCAT from McGill [16], and Landi-PROLANGS from Rutgers [22, 26, 25]. Our testing suite is closely modeled after Hind and Pioli’s [21] and all but two of their programs appear in our study. We omitted 052.alvinn from SPEC 92 because the benchmark suite is subsumed by SPEC 2000. We omitted 17.bintr from McCAT because of a Scale compilation bug with this program.

We used the most recent development version of Scale, as of the time of this report. Scale’s default parameters are used except for the following cases.

When invoking Scale, we specify a version of alias analysis from Table 2. Shapiro-Horwitz with one category behaves the same as Steensgaard (although the implementations are distinct). We select four categories as the input parameter to Shapiro-Horwitz so that it behaves as an intermediate point that is more precise than Steensgaard, but not as expensive as Andersen [1].

We either select a fixed sequence of optimizations or choose a single optimization, with all other optimizations turned off. Table 3 enumerates the optimizations, their Scale option letter, and our abbreviation.

We also collect data on Partial Redundancy Elimination (PRE), which Scale implements using Chow et al.’s algorithm for SSA [6]. This algorithm requires SSA form, but does not produce SSA form as output, which makes it difficult to measure and use in Scale. Furthermore, our PRE results show more optimization opportunities with Steensgaard and Shapiro-Horwitz than with no aliases. We believe this anomaly results from either a bug in Scale or an interaction with SSA. We decided to omit the PRE

Table 1. Benchmark suites

Abbr.	Benchmark suite
A	Austin's
MC	McCAT
LP	Landi-PROLANGS
S95	SPEC 95
S00	SPEC 2000

Table 2. Alias analysis algorithms

Abbr.	Algorithm
AT	Address-taken analysis
ST	Steensgaard's interprocedural algorithm
SH-4	Shapiro-Horwitz's interprocedural algorithm with 4 categories
NA	Assume no aliases

Table 3. Optimizations

Abbr.	Scale option	Optimization
AASR	a	Array Access Strength Reduction
SCCP	c	Sparse Conditional Constant Propagation
GVR	g	Global Variable Replacement
LU	j	Loop Unrolling
LICM	m	Loop Invariant Code Motion
GVN	n	Global Value Numbering
CP	p	Copy Propagation
UCR	u	Useless Copy Removal
SR	x	Scalar Replacement

results here since we believe the underlying problem is orthogonal to alias analysis. A companion technical report contains these results [8].

We measure compile times on a 502 MHz UltraSPARC-IIe Sun Blade 100 running SunOS 5.8 with 256 MB of RAM and 1475 MB of swap space. We specified an initial heap size of 100 MB and a maximum heap size of 1000 MB for Sun's Java virtual machine running Scale.

7 Results

This section first compares the compilation times of the benchmark programs and then optimization opportunities utilized by the scalar optimizations in Table 3. A companion technical report [8] contains complete per program results, and we summarize these results below.

7.1 Compile Time

Table 4 describes some characteristics of the 36 benchmark programs. The column marked "Source" identifies the benchmark suite to which the program belongs. The column marked "NCLC" reports the number of non-blank and non-commented lines of code in the program. The column marked "CFG Nodes" shows the number of nodes in the Control Flow Graph created by Scale for the program. This number gives us an idea on the size of the program as seen by the compiler. The programs in the table are arranged in ascending order of the number of CFG nodes. The next three columns list the compile times (in seconds) of the program: the first column is for the case when address-taken analysis is performed, the second one is for Steensgaard's interprocedural algorithm, and the third one is for Shapiro-Horwitz's interprocedural analysis with

4 categories. Each compile time is the smallest among 5 independent compiles of the same program with the same parameter values. The last two rows of the table report the normalized average compile times over all the programs. The compile time of the program for each alias analysis algorithm is first divided by the compile time of the program using address-taken analysis. Then, we take the arithmetic and geometric means (AM & GM) of those normalized compile times. Geometric mean reduces the effect of extreme values.

The means in Table 4 suggest that using Steensgaard instead of address-taken increases the compile time by 5-6% on the average while the average increase in compile time due to the use of Shapiro-Horwitz is 20-30%. But the results also show that for large programs (like 186.crafty, 300.twolf, and 099.go), these percentages may grow to 150-200% and 350-400%, respectively. We think this result is due to paging.

The compile times in Table 4 are represented as a bar graph in Fig. 1.

7.2 Optimization Opportunities

We measure the optimization opportunities utilized by each of the 9 scalar optimizations over each of the 36 benchmark programs over each of the 4 alias analyses based on the selected *criteria* (see Section 5). For each optimization, we report this counter and the higher the counter, the more effective the optimization is.

For a given optimization, we report the counter using each of the 4 alias analyses and normalize the counters by dividing by the value obtained for NA (no aliases). During the normalization process, if $numerator = denominator = 0$, we assume the normalized value is 1. If $numerator = 0 \neq denominator$, we set the *numerator* to 0.5 and proceed with the division. Although the case “ $numerator = denominator = 0$ ” occurs many times in our experiments, the case “ $numerator = 0 \neq denominator$ ” occurs only once: in 01.qbsort for LICM in Table 6. After calculating the normalized counters for all programs, we take the geometric mean of these values for each alias analysis.

We perform two sets of experiments. In the first set we enable all 9 optimizations in the order “jgcamnpxnmpu” (see Table 3) during each compilation and in the second set, we enable only one optimization per compilation. Table 5 summarizes the results for all optimizations enabled, and Table 6 for each one individually.

Table 5 contains one row for each of the 9 optimizations and one column for each of the 4 alias analyses. In each optimization row, in the column corresponding to each alias analysis, we report the geometric mean of the normalized optimization counters of all 36 programs after performing that analysis. Each mean is subscripted by the average deviation of the counter values from that mean. For a given optimization, the column marked “Total for NA” contains the summation of the NA counter values for that optimization over all programs. The bar graph in Fig. 2 presents this table.

To understand these results, we first define a sequence anomaly.

Sequence Anomaly: When a sequence of optimizations is applied on a program, effectiveness of the optimizations later in the sequence is influenced by the type and number of opportunities exposed by earlier optimizations. Since optimizations interact with each other in a non-linear fashion, it is possible for a more precise alias analysis

Table 4. Comparison of NCLC, number of CFG nodes and total compile times (in seconds) with all optimizations enabled in order “jgcampxnmpu”

Program	Source	NCLC	CFG nodes	Compile time (sec)		
				AT	ST	SH-4
15.trie	MC	311	197	5.0	5.1	5.5
fixoutput	LP	368	206	6.0	5.7	5.9
allroots	LP	155	272	10.0	9.9	10.2
01.qbsort	MC	200	294	6.9	7.5	8.0
04.bisect	MC	217	331	11.1	11.8	12.0
06.matx	MC	191	439	7.9	7.8	7.9
anagram	A	352	532	9.9	9.4	9.3
lex315	LP	598	658	8.2	8.1	9.0
ul	LP	472	773	11.2	11.0	11.2
129.compress	S95	1457	923	12.1	12.4	12.9
ks	A	585	987	11.6	10.9	10.9
09.vor	MC	984	1031	12.0	12.0	12.6
loader	LP	802	1082	15.8	16.0	17.0
ansitape	LP	1203	1113	17.0	16.1	17.1
08.main	MC	990	1115	12.0	11.7	12.6
ft	A	1113	1116	12.1	12.9	13.1
compress	LP	1071	1119	10.3	9.8	10.7
05.eks	MC	575	1498	22.0	21.9	23.0
xmodem	LP	1392	1718	19.0	19.6	20.0
181.mcf	S00	1482	1722	24.3	29.9	29.5
compiler	LP	2073	1789	19.0	18.1	19.0
assembler	LP	1891	2052	21.0	22.9	23.9
unzip	LP	2808	2637	26.1	29.1	29.7
patch	LP	2461	3248	24.1	24.4	27.2
simulator	LP	2881	3532	29.8	30.1	30.7
yacr2	A	2710	3753	27.2	27.8	31.5
256.bzip2	S00	3236	4888	38.0	35.4	39.8
flex	LP	4841	5405	42.5	44.1	48.2
bc	A	5449	5618	36.1	40.1	48.8
football	LP	1975	5765	143.1	140.4	149.7
agrep	LP	3434	8185	74.5	70.0	73.2
197.parser	S00	7921	15418	72.0	81.9	99.8
175.vpr	S00	11301	17935	111.0	144.2	238.8
186.crafty	S00	12985	22379	388.1	595.9	691.4
300.twolf	S00	17934	31414	209.3	365.9	788.1
099.go	S95	25895	35018	232.2	256.4	814.6
AM of norm. comp. times (wrt AT)				1.000	1.062	1.271
GM of norm. comp. times (wrt AT)				1.000	1.052	1.190

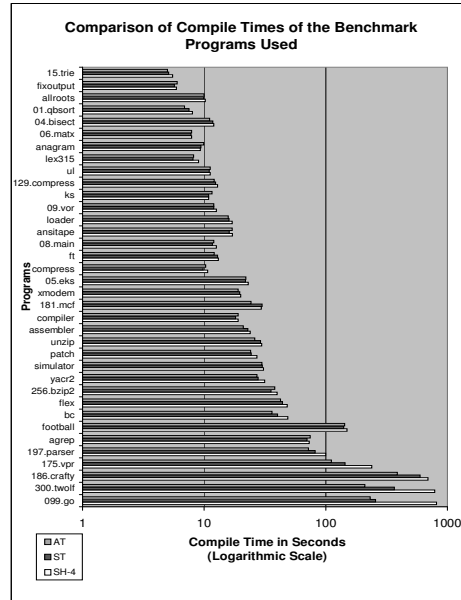


Fig. 1. Comparison of compile times of the benchmark programs used.

to have a negative impact on optimizations that come later in the sequence. We refer to these effects as *sequence anomalies*. Sequence anomalies cannot occur when only one optimization is applied to a program.

7.3 Optimizations in Sequence

We first summarize the average effect of AT, ST, and SH-4 using a fixed sequence of optimizations, and then summarize the results for each particular optimization.

Address-taken: For LU, GVR, and SR, the effectiveness of address-taken analysis is within 0.1% of that of any alias analysis, no matter how precise it is. For AASR, it is within 1%, and for SCCP, GVN and CP it is within 3% of the most precise alias analysis. It is least effective on LICM, but still within 9% of the effectiveness of the best possible analysis.

Table 5. Effectiveness of alias analysis on optimizations with all of them enabled in order “jg-camnpxnmpu” (Geometric mean of normalized (w.r.t. NA) criteria counts with avg. deviation from the mean as subscript)

Opt.	Criteria	AT	ST	SH-4	NA	Total for NA
LU	loops unrolled	0.999 _(0.002)	0.999 _(0.002)	0.999 _(0.002)	1.000 _(0.000)	1004
GVR	loads replaced	0.999 _(0.001)	0.999 _(0.001)	0.999 _(0.001)	1.000 _(0.000)	16825
SCCP	constants propagated	0.967 _(0.046)	0.970 _(0.043)	0.970 _(0.043)	1.000 _(0.000)	25258
AASR	calculations replaced	0.992 _(0.006)	0.999 _(0.003)	0.999 _(0.003)	1.000 _(0.000)	7996
LICM	expressions moved	0.912 _(0.123)	0.940 _(0.086)	0.940 _(0.086)	1.000 _(0.000)	2136
GVN	expressions removed	0.979 _(0.030)	0.992 _(0.010)	0.992 _(0.010)	1.000 _(0.000)	24054
CP	copies propagated	0.978 _(0.032)	0.987 _(0.020)	0.987 _(0.020)	1.000 _(0.000)	21247
SR	array loads replaced	0.999 _(0.002)	0.999 _(0.001)	0.998 _(0.003)	1.000 _(0.000)	8143
UCR	useless copies removed	1.017 _(0.025)	1.007 _(0.012)	1.006 _(0.012)	1.000 _(0.000)	101543

Table 6. Effectiveness of alias analysis on optimizations with only one optimization enabled at a time (Geometric mean of normalized (w.r.t. NA) criteria counts with avg. deviation from the mean as subscript)

Opt.	Criteria	AT	ST	SH-4	NA	Total for NA
LU	loops unrolled	0.999 _(0.002)	0.999 _(0.002)	0.999 _(0.002)	1.000 _(0.000)	1004
GVR	loads replaced	1.000 _(0.000)	1.000 _(0.000)	1.000 _(0.000)	1.000 _(0.000)	10701
SCCP	constants propagated	0.948 _(0.069)	0.948 _(0.070)	0.948 _(0.070)	1.000 _(0.000)	9357
AASR	calculations replaced	0.993 _(0.013)	0.993 _(0.013)	0.993 _(0.013)	1.000 _(0.000)	2675
LICM	expressions moved	0.924 _(0.101)	0.924 _(0.101)	0.924 _(0.101)	1.000 _(0.000)	1236
GVN	expressions removed	0.953 _(0.061)	0.985 _(0.020)	0.985 _(0.020)	1.000 _(0.000)	13273
CP	copies propagated	0.968 _(0.041)	0.968 _(0.041)	0.968 _(0.041)	1.000 _(0.000)	9203
SR	array loads replaced	0.998 _(0.003)	0.999 _(0.002)	1.000 _(0.001)	1.000 _(0.000)	5135

Steensgaard and Shapiro-Horwitz (SH-4): Steensgaard and Shapiro-Horwitz (SH-4) essentially have the same effect on all the optimizations we considered. For LU, GVR, AASR, and SR, they are within 0.1% of the most precise analysis. For GVN, they are within 1%, for CP, within 1.5%, and for SCCP, within 3% of the best possible analysis. Again, they are least effective on LICM, but still within 6% of the most precise alias analysis.

LU: Very little opportunity (only 0.1%) is left for improving LU beyond what is already achieved by applying AT, ST, or SH-4. We also found that for each program, exactly the same number of loops were unrolled when using AT, ST and SH-4. For only 3 programs (175.vpr, 300.twolf and 099.go), LU unrolled a few more loops when “no aliases” (NA) is assumed.

GVR: GVR behaved exactly like LU.

SCCP: ST and SH-4 already achieve about 97% of what NA achieves. In our experiments, ST and SH-4 behaved identically with respect to SCCP on all programs. For only 4 programs (loader, simulator, flex and 175.vpr), ST and SH-4 triggered more constant propagation than AT did. However, for about 40% (14 out of 36) of the programs, there is still a little room to improve SCCP.

AASR: The room for improvement is less than 1% and it is, in fact, about 0.1% with respect to ST and SH-4. For each program, AASR behaved identically with respect to ST and SH-4 and for only 3 programs (04.bisect, simulator, and 175.vpr) did applying ST or SH-4 instead of AT have any positive impact. For only 2 programs (04.bisect and simulator), application of NA instead of ST or SH-4 influenced ASSR positively. However, a sequence anomaly occurs for 099.go when NA is assumed.

LICM: The room for improvement is about 9% with respect to AT and 6% with respect to ST and SH-4. However, the large improvement opportunity with respect to AT is slightly misleading because for half of the programs (3 out of 6) on which LICM improved when NA was applied instead of AT, the number of expressions removed was quite low (AT vs. NA: 1 vs. 3 for 01.qbsort, 2 vs. 6 for loader, 1 vs. 2 for 181.mcf). For LICM, too, ST and SH-4 behaved identically for all programs and for only 2 programs (compress and simulator) had more positive impact than AT. However, a sequence anomaly occurs again for 099.go when NA is assumed.

GVN: The room for improvement is about 2% with respect to AT and about 1% with respect to ST and SH-4. Again, ST and SH-4 behaved identically on all programs, for 15 programs ST and SH-4 improved over AT, and for 14 programs NA improved over ST and SH-4. Two sequence anomalies occurred: one with 099.go, when NA was assumed, and another one with 04.bisect when either ST, SH-4 or NA was applied instead of AT.

CP: The room for improvement is slightly more than 2% with respect to AT and about 1.5% with respect to ST and SH-4. For 3 programs (129.compress, simulator and 175.vpr) ST improved over AT, for one program (197.parser) SH-4 improved over ST and for 11 programs NA improved over SH-4. For CP, five sequence anomalies occurred. For one program (099.go), AT was the most effective alias analysis for CP and NA was the least effective one. For another program (197.parser), SH-4 was the most effective analysis and both ST and NA were the least effective ones. For yet another program (300.twolf), AT was the most effective analysis and the rest were identical to each other. A similar pattern occurred with ST for 256.bzip2. For 186.crafty, AT was better than ST and SH-4, but the largest number of copies were propagated with NA.

SR: Alias analysis seemed to have very little impact on SR and the only room for improvement, if any, is around 0.1%. Only 4 programs (256.bzip2, 197.parser, 175.vpr and 099.go) were slightly affected. However, the effect did not seem to have any particular trend.

UCR: UCR depends on the form of the SSA graph and how other optimizations change that graph. Alias analysis influences UCR in a very complex fashion. However, in our experiments, a more precise alias analysis created fewer useless copy statements than a less precise one.

7.4 Optimizations Enabled Independently

Table 6 summarizes the results obtained by enabling each optimization independently. This table is structured almost identically to Table 5. It does not include UCR since Scale need not perform UCR if no other optimization is applied. Fig. 3 represents the data in a bar graph.

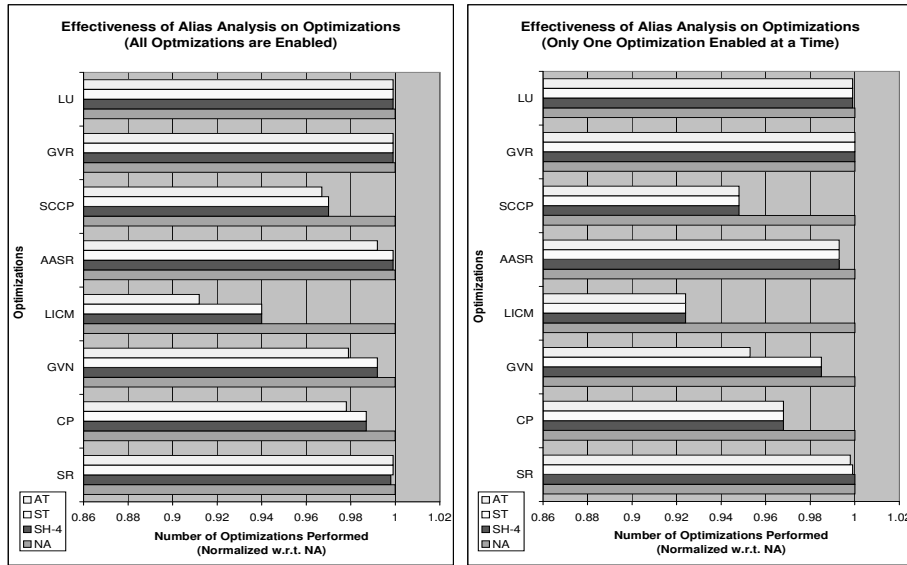


Fig. 2. Effectiveness of alias analysis on optimizations (all optimizations enabled).

Fig. 3. Effectiveness of alias analysis on optimizations (one optimization enabled at a time).

From the last column of Table 6, we observe that optimization opportunity is, in general, greatly reduced when each optimization is applied independently of others. For all optimizations except Scalar Replacement, ST and SH-4 behave identically on all programs. We first summarize the trends, and then examine each optimization.

Address-taken: For GVR, address-taken analysis is as good as any alias analysis. For LU and SR, its effectiveness is within 0.2%, for AASR, within 1%, and for SCCP, GVN, and CP, within 5% of that of the most precise analysis. For LICM, it is within 8% of the best possible analysis.

Steensgaard and Shapiro-Horwitz (SH-4): Steensgaard and Shapiro-Horwitz (SH-4) essentially have the same effect on all the optimizations we consider. For GVR, they are as good as the most precise analysis. For LU and SR, they are within 0.1%, for AASR, within 1%, for GVN, within 1.5%, and for SCCP and CP, within 5% of the effectiveness of the best possible alias analysis. Again, they are least effective on LICM, but still within 8% of the best possible analysis.

LU: This optimization behaved exactly in the same way in all aspects as it did with all optimizations enabled. This similarity comes from the fact that LU was the first optimization in the sequence of optimizations applied on programs in the previous set of experiments.

GVR: Alias analysis precision does not have any effect.

SCCP: For SCCP, the improvement opportunity is slightly more than 5%. For 13 programs, NA had a more positive impact on SCCP compared to ST and SH-4. For only one program (175.vpr), ST and SH-4 influenced SCCP more positively than did AT.

AASR: AT, ST, and SH-4 behaved identically for every program. For only 2 programs (04.bisect and simulator) further improvement (less than 1%) is possible.

LICM: About 8% improvement is possible for LICM. For each program, LICM was influenced identically by AT, ST, and SH-4. For 7 programs, NA exposed more optimization opportunities than did AT, ST or SH-4.

GVN: The margin for improvement is about 5% with respect to AT and about 1.5% with respect to ST and SH-4. For 10 programs, ST and SH-4 proved to be more effective than AT and for 13 programs, NA exposed more optimization opportunities than ST or SH-4.

CP: The room for improvement is slightly more than 3%. For each program, the impact of AT, ST and SH-4 on CP were identical. For 12 programs, there is a gap between ST/SH-4 and NA.

SR: Alias analysis precision has very little effect.

8 Conclusion

Our “assume no aliases” methodology provides upper bound analysis that is surprisingly tight for scalar optimizations and easy to implement. Our conclusions are expressly for the domain of scalar optimizations, and show that there is little room to improve scalar optimization by improving alias analysis. By studying this upper bound for other clients, researchers can explore the limits of alias analysis. When the bound and analysis match, there is no need to test more precise analyses, and in these cases, the methodology obviates an entire class of iterative testing.

Within other domains, such as error detection and memory leak detection, precise pointer disambiguation is either required for correctness or critical for good performance [18, 19]. However, this upper bound methodology can provide additional insight into optimization-program pairs that warrant further study. If an optimization applied to a program produces far fewer transformations than the number of transformations permitted in upper bound analysis, this case may benefit from more precise alias analysis and should be investigated further.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
2. T. Austin. Pointer-intensive benchmark suite, version 1.1. <http://www.cs.wisc.edu/~austin/ptr-dist.html>, 1995.
3. D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *ACM Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, San Jose, CA, Oct. 1996.
4. P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software—Practice and Experience*, 27(6):701–724, June 1997.
5. J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual ACM Symposium on the Principles of Programming Languages*, pages 232–245, Charleston, SC, Jan. 1993.
6. F. Chow, S. Chan, R. Kennedy, S. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 273–286, Las Vegas, NV, June 1997.

7. F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In T. Gyimothy, editor, *International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267, Linköping, Sweden, Apr. 1996. Springer-Verlag.
8. R. A. Chowdhury, P. Djeu, B. Cahoon, J. H. Burrill, and K. S. McKinley. The limits of alias analysis for scalar optimizations. Technical report, University of Texas at Austin, Oct. 2003. <http://www.cs.utexas.edu/users/mckinley/papers/alias-tr-2003.ps.gz>.
9. K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Proceedings of the 16th Annual ACM Symposium on the Principles of Programming Languages*, pages 49–59, 1989.
10. K. D. Cooper and J. Lu. Register promotion in C programs. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 308–319, Las Vegas, NV, June 1997.
11. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th Annual ACM Symposium on the Principles of Programming Languages*, pages 25–35, Austin, TX, Jan. 1989.
12. M. Das, B. Liblit, M. Fahndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *The 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 260–278, Paris, France, July 2001. Springer-Verlag.
13. A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117, Montreal, June 1998.
14. A. Diwan, K. S. McKinley, and J. E. B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23(1):30–72, Jan. 2001.
15. A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and effective analysis of statically-typed object-oriented languages. In *ACM Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 344–355, San Jose, CA, Oct. 1996.
16. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.
17. R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, Jan. 1998.
18. S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *International Static Analysis Symposium*, pages 214–236, San Diego, CA, June 2003.
19. D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 168–181, San Diego, CA, June 2003.
20. M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analysis. In *The 5th International Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 57–81, Pisa, Italy, Sept. 1998. Springer-Verlag.
21. M. Hind and A. Pioli. Which pointer analysis should I use? In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'00)*, pages 112–123, Portland, OR, Aug 2000.
22. W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. *ACM SIGPLAN Notices*, 28(6):56–67, 1993.
23. D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 199–215, Toulouse, France, Sept. 1999.

24. K. S. McKinley, J. Burrill, B. Cahoon, J. E. B. Moss, Z. Wang, and C. Weems. The Scale compiler. Technical report, University of Massachusetts, 2001. <http://ali-www.cs.umass.edu/~scale/>.
25. E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, CA, June 1995.
26. Rutgers. PROLANGS benchmark suite, data programs. <http://www.prolangs.rutgers.edu/public.html>, 1999.
27. M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In P. V. Hentenryck, editor, *Lecture Notes in Computer Science, 1302*, pages 16–34. Springer-Verlag, 1997. Proceedings from the *4th International Static Analysis Symposium*.
28. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, pages 1–14, Paris, France, Jan. 1997.
29. B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, pages 21–24, St. Petersburg, FL, Jan. 1996.
30. P. A. Stocks, B. G. Ryder, W. Landi, and S. Zhang. Comparing flow and context sensitivity on the modifications-side-effects problem. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 21–31, Clearwater, FL, Mar. 1998.
31. O. Traub, G. Haolloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 142–151, Montreal, June 1998.
32. Z. Wang, D. Burger, K. S. McKinley, S. Reinhardt, and C. C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 388–398, San Diego, CA, June 2003.
33. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.
34. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, June 1995.
35. S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 91–103, Atlanta, GA, June 1999.