Dynamic Points-To Sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization

Markus Mock*, Manuvir Das+, Craig Chambers*, and Susan J. Eggers*

Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, W 98195-2350
{mock,eggers,chambers}@cs.washington.edu

Microsoft Research
Redmond, WA 98052
manuvir@microsoft.com

Abstract

In this paper, we compare the behavior of pointers in C programs, as approximated by static pointer analysis algorithms, with the actual behavior of pointers when these programs are run. In order to perform this comparison, we have implemented several well known pointer analysis algorithms, and we have built an instrumentation infrastructure for tracking pointer values during program execution.

Our experiments show that for a number of programs from the Spec95 and Spec2000 benchmark suites, the pointer information produced by existing scalable static pointer analyses is far worse than the actual behavior observed at run-time. These results have two implications. First, a tool like ours can be used to supplement static program understanding tools in situations where the static pointer information is too coarse to be usable. Second, a feedback-directed compiler can use profile data on pointer values to improve program performance by ignoring aliases that do not arise at run time (and inserting appropriate run-time checks to ensure safety). As an example, we were able to obtain a factor of 6 speedup on a frequently executed routine from *m88ksim*.

Keywords

Points-To analysis, alias analysis, dynamic analysis, program understanding, program optimization, Calpa, program instrumentation.

1. INTRODUCTION

Many programming languages in use today, such as C, allow the use of pointers. Pointers are used extensively in C programs to simulate call-by-reference semantics in procedure calls, to emulate object-oriented dispatch via function pointers, to avoid the expensive copying of large objects, to implement list, tree or other complex data structures, and as references to objects allocated dynamically on the heap. While pointers are a useful and powerful feature, they also make programs hard to understand, and often prevent an optimizing compiler from making code-improving transformations.

In an attempt to compensate for these negative effects, many pointer analysis algorithms have been devised over the past decade [1,4,5,8,9,12,13,18,20,21,22]. These algorithms produce a conservative approximation of the possible sets of variables, data structures, or functions a particular pointer could point to at a specific program point; these are referred to as *points-to sets*. These sets can be used, for instance, by an optimizing compiler to determine that two expressions might be *aliased*, i.e., refer to the same object.

Several kinds of pointer analysis algorithms have been designed. 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 [6,11] suggests that for typical C programs (e.g. SPEC benchmarks) flow- and context-sensitivity produce only insignificant improvements over Das's fast One-Level Flow algorithm [5], which has been shown to scale up to over a million lines of C code. However, even though it is generally the most precise among scalable algorithms, its points-to sets are still on the order of tens or even hundreds of objects. Clearly, such points-to sets are too large to be very useful in a program understanding tool, where the user might like to know what objects a pointer store might modify.

Instead of designing yet another pointer analysis algorithm, we wanted to find out how well the statically computed points-to sets agree with actual program behavior, i.e., how many different objects are referenced at a particular pointer dereference compared to the number of objects in the points-to set computed by a scalable pointer analysis algorithm. Dynamic points-to sets may tell us how close actual algorithms are to the theoretical optimum[1], may be used to improve program understanding tools, and enable dynamic optimizations that take alias relationships into account.

For example, instead of presenting the user with hundreds of potential candidate targets of a pointer dereference `*p`, the program understanding tool could use the dynamically observed targets of `*p` and present those to the user; in addition, when the

PASTE'01, June 18-19, 2001, Snowbird, Utah, USA.

static and dynamic sets agree, it could also inform the user that the static information is in fact optimal and not just a conservative approximation. This could be used to find out where a more precise, but more expensive, algorithm might be beneficial. Moreover, in some situations the potentially unsound dynamic sets are more useful for program understanding than optimal points-to sets: if the user is interested in what a pointer pointed to in a particular program run, for instance when debugging a program, it is actually more desirable to present only the dynamically observed targets rather than all potential targets for all possible executions.

In order to obtain dynamic points-to information in this study, we used a slightly modified version of the Calpa instrumentation tool [15] to observe the dynamic points-to sets of a set of programs taken from the SPEC95 and SPEC2000 benchmark suites. The static average points-to set sizes ranged from 1.0 to 177 for the best of the scalable static pointer analysis algorithms we used, while the dynamic points-to set sizes were on average (geometric mean) a factor of 5 smaller. Additionally, for the large majority (over 97%) of dereferences the dynamic points-to sets were actually singletons. This means that 97% of the time a tool similar to ours that is integrated into a program understanding tool, would be able to tell the user the exact target of a dereference (for the particular input data set). This is a much higher fraction than what is possible with static analysis alone (14% of executed dereferences), demonstrating the considerable potential benefit to program understanding systems.

Furthermore, dynamic optimizers can take advantage of the fact that a dereference accesses only one object at run time. Having smaller dynamic points-to sets may cause fewer expressions to be aliased in a program, which may allow the optimizer do a better job; Section 2 shows an example of exploiting dynamic alias information and the ensuing performance benefits.

This paper makes the following contributions:

- we present a tool to observe points-to information at run time;
- we show that there is a large gap between the sizes of dynamic points-to sets and static points-to sets produced by scalable static analyses, which produce sets that are in general an order of magnitude larger;
- we show that dynamic points-to sets are almost always (97% of the time) of size 1; with the average size being close to 1 across all dereferences executed at run time;
- we outline how these results might be used to improve both program understanding tools and dynamic program optimizers.

The rest of the paper is organized as follows: in Section 2 we present an example illustrating the optimization potential of dynamic pointer information. We describe our instrumentation methodology in Section 3. Section 4 discusses our experimental results. Section 5 discusses related work, and in Section 6 we present our conclusions.

[1] Since the dynamic points-to set sizes may be distinct for different inputs, they are potentially unsound and could be smaller than the optimal sound solution, which in general is not computable, since pointer-analysis has been shown to be undecidable [17]. However, for programs exercising a large fraction of their execution paths, such as the SPEC benchmarks, we expect the dynamic sets to be not much smaller than an optimal solution.

2. OPTIMIZATION EXAMPLE

The following example illustrates the potential benefits of exploiting dynamic pointer information:

```
void align(uint* low, uint* high, uint*
result, uint diff) {
    for (*result = 0; diff>0; diff--) {
        *result |= *low & 1;
        *low >>= 1;
        *low |= *high << 31;
        *high >>= 1;
    }
}
```

The example shows a simplified version of a routine found in the *m88ksim* SPEC95 benchmark. The routine is called from a number of places in the code and none of the static alias analyses we looked at was able to determine that at run time the arguments `low`, `high`, and `result` were not aliased. Therefore, a code optimizer would have to assume that the store `*result` might overwrite the value of `*low`, preventing a register allocation of `*low`. Similarly, `*high` or `*result` cannot be allocated to registers but have to be reloaded from memory each time.

If dynamic points-to sets are available, however, and indicate that `low`, `high`, and `result` are not aliased, a feedback-directed optimizer could allocate their memory targets to registers, inserting a run-time check to ensure that the arguments are in fact not aliased. If the aliasing check fails, the slower code version, where the memory targets are reloaded before each use, would be executed instead. The resulting code would look as follows:

```
void align_opt(uint* low, uint* high, uint*
result, uint diff) {
    if ALIASED(low, high, result) {
      /* slow version with reloading */
       for (*result = 0; diff>0; diff--) {
         *result |= *low & 1;
         *low >>= 1;
         *low |= *high << 31;
         *high >>= 1;
        }
    } else {
      /* fast register-allocated version*/
       register uint r, l=*low, h=*high;
       for (r=0; diff>0; diff--) {
         r |= l & 1;
         l >>= 1;
         l |= h << 31;
         h >>= 1;
       }
       *result = r; *low = l; *high = h;
    }
}
```

Dynamic points-to information is required to ensure that this transformation will be beneficial, because the execution time penalty that is incurred by the run-time check that assures soundness will only be recouped if the faster code version is selected sufficiently often at run time (this is generally unknown in the absence of profile information).

We hand-simulated register-allocation for the example by loading the memory targets of the arguments into local variables and storing them back before procedure return. To make the

transformation sound, we also inserted a check at the beginning of the routine to test whether the arguments were aliased. The speedup over the unoptimized version was a factor of 6.2 for the routine alone, and 1% for *m88ksim* as a whole, when executed on a Compaq True64 Unix workstation with an Alpha 21264 processor running at 667 Mhz. Both versions were compiled with the vendor compiler and optimization flags -O2. These speedups show the potential benefits of using dynamic pointer information.

3. INSTRUMENTATION

We used the Calpa instrumentation tool [15] to instrument our applications. In the Calpa [14,15] system, the instrumenter is used to obtain a value profile of the variables and data structures of a program. When a variable or data structure is accessed via a pointer `p`, the instrumenter inserts a call to a runtime library function that compares the pointer value to the addresses of potential target data structures or variables for that pointer dereference; the potential target objects are identified by a static alias analysis that is run before the instrumenter. Once the actual target object has been identified, the object's value profile is updated.

For this study we changed the instrumenter to simply count how often each potential target object of a pointer dereference was accessed during a program run. For each load or store instruction the instrumenter inserts an array of counter variables; a distinct array variable is created for each dereference point, and its size is made equal to the size of the static points-to set at the particular dereference. Before the load or store the instrumenter inserts a call to a library routine that matches the pointer address with the addresses of the potential target objects at the dereference. At run-time this matching routine returns an integer which identifies which object matched, and the corresponding counter is incremented. For example, a pointer store `*p = val` is changed to:

```
temp = match_object(p, object_addrs[]);
counter[temp]++;
*p = val
```

where `object_addrs[]` is a data structure created and updated by code inserted by the instrumenter to always contain the addresses of the variables or data structures that `p` might refer to at that point. For example, if the static points to set size for `p` is `{x, y, z}`, `object_addrs[]` would contain `{&x, &y, &z}`; both `object_addrs[] and counter` are specific to the particular program point.

To be able to match data structures on the heap, calls to memory allocation routines such as `malloc`, are instrumented as well. At run time a list of `(address, malloc-site)` pairs is maintained, so that an address corresponding to a heap-allocated data structure can be matched to the corresponding malloc site. A *malloc site* is a program point where a memory-allocating function (e.g. *malloc*) is called. Since a particular *malloc* call may be executed multiple times at run time, multiple heap objects may be represented by the same malloc site. Consequently, when a dynamic points-to set of size 1 represents a malloc site, it may correspond to multiple heap data structures at run time. This is similar to static points-to analyses which typically represent a malloc site by one symbol; some even represent all heap data structures by just one representative heap symbol.

Once the instrumented program has finished, the contents of all counter variables are written out to disk. Using a map file that maps a particular counter variable to the corresponding static points-to set, the dynamic points-to set is computed as the set of objects that had a non-zero counter value. For instance, if the values of the counter array in the example are `{0, 100, 200}`, we know that variable `x` was never accessed, `y` was accessed 100 times, and variable `z` 200 times at that dereference. Therefore, the dynamic points-to set for `*p` would be `{y,z}` with a dynamic points-to set size of 2.

Since the counters also tell us how often a particular dereference was executed (in the example 300 times), we can use these execution frequencies to compute a weighted average of all dereferences executed in the program. For instance, if there is only one other dereference *q in the program which is executed 200 times and has a dynamic points-to set size of 1, the (unweighted) points-to average would be 1.5, whereas the weighted average would be 1.6. Since this measure gives more importance to heavily executed dereferences, the weighted average may be more significant in a context where not only the points-to set size but also the execution frequency is relevant, for instance, in dynamic optimizations.

4. EXPERIMENTS

To compare static and dynamic points-to sets, we first ran an alias analysis on each application. We used the fast and scalable algorithms developed by Steensgaard [20] and Das [5], as well as an extension of Steensgaard's algorithm proposed by Shapiro and Horwitz [18], where feasible within time and memory-constraints[1]. Both the points-to algorithms and the instrumentation tool are implemented using the Machine SUIF infrastructure [10,19].

For each algorithm we measure the points to set sizes at each executed dereference point in the program (a load or a store instruction in the SUIF intermediate representation), and compute the average over all dereference points; in addition we compute an average weighted by the execution frequency of each dereference.

4.1 Workload

Our workload consists of the SPEC95 (*m88ksim*, *perl*) and SPEC2000 (the others) benchmarks shown in Table 1. With each benchmark we list a short description of the benchmark, the number of lines of C code (in thousands), and the static average dereference size for each pointer analysis algorithm.

We also looked at the programs in Todd Austin's pointer-intensive benchmark suite [2] but did not use them in this study. In addition to being relatively small they showed the same dynamic points-to sets results as the benchmarks we used in this study.

4.2 Static Points-To Sets Results

Table 1 shows the average points-to set sizes for the alias analysis algorithms that we implemented. The average static points-to sets sizes produced by the One-Level Flow (OLF) algorithm ranged

[1] The Shapiro-Horwitz algorithm is parameterized by the number of symbol categories, and the number of runs. We randomly assigned symbols to 5 categories, and ran the algorithm 2 times, taking the intersection of the resulting points-to sets as the final result.

Table 1. Description of the workload. The average dereference sizes shown are arithmetic means over all dereference points in the program; shown are the results for the One-Level Flow (OLF), Steensgaard, and Shapiro-Horwitz algorithms. Empty entries indicate that our implementation ran out of memory before finishing.

Program	Description	KLOC	Average dereference size		
			OLF	Steensgaard	Shapiro-Horwitz
equake	seismic wave propagations simulation	1.2	1.00	1.04	1.00
art	image recognition, neural networks	1.2	1.14	1.27	1.00
mcf	combinatorial optimization	1.9	2.9	2.9	2.9
bzip2	compression	3.9	1.0	1.9	1.0
gzip	compression	7.6	7.9	35.9	25.0
parser	Word processing	10.3	6.7	66.2	
vpr	FPGA circuit placement and routing	13.6	2.5	12.6	
m88ksim	Motorola 88000 instruction set simulator	19.4	5.7	96.8	
perl	perl interpreter	26.8	21.2	56.1	
gap	group theory interpreter	62.5	7.2	86.4	
mesa	3D graphics library	81.8	247.3	423.8	

from 1 for equake to 247 for mesa. Steensgaard's algorithm fares worse, with points-to sets of up to a factor of 16.8 larger (*m88ksim*) than those produced by the OLF algorithm. The largest application for which we could run our implementation of the Shapiro-Horwitz algorithm was *gzip*; for the larger applications it ran out of memory.

The points-to set sizes that we report for Steensgaard's and Das' algorithm, are slightly different from the ones reported in [5]. The differences come from a number of sources: (1) We use a different intermediate representation for the C programs. In particular, a structure field access `s.f` typically creates a pointer dereference in our representation but not in Das'. (2) We treat constant strings differently; since they are read-only (which is, however, typically not enforced by compilers), they can be ignored as targets (of stores). Das' numbers ignore them completely; our results include one representative for all string constants.

Like [5] we do not include pointers to functions in the points-to sets at dereference points. The only benchmarks for which the points-to set sizes would be significantly different if procedure targets were included are *vortex* and *gap*, which both simulate object-oriented dispatching via function pointers.

4.3 Dynamic Points-To Sets Results

To obtain the dynamic points-to sets we instrumented the applications as described in Section 3, and executed the instrumented applications on the SPEC-provided test inputs. We chose the test inputs, because the reference inputs take much longer to run. Since the instrumentation slows down the applications by about 2 orders of magnitude, running the reference inputs would generally not finish within a day. However, we expect the results to be largely unchanged for the larger reference inputs, which tend to run the same program parts, only more often.To confirm this intuition, we ran *gzip*, *parser*, *perl* and *mesa* also on the reference inputs, and found the results to be the same. Tuning our profiling infrastructure for speed, would allow us to also run with larger inputs in reasonable time (less than a day or two which it takes now for the reference inputs).

In Table 2 we compare the average sizes of the dynamically observed points-to sets for each application with the average sizes of the corresponding static sets. While in Table 1 the static points-to set sizes include all dereference points, in Table 2 only those dereference points are included, that were actually executed.[1] This allows a comparison with dynamic points-to sets, which are determinable only at executed dereferences. We show simple average points-to sets sizes, and a weighted average points-to-set size, weighted by the execution frequency of each dereference.

While the average dynamic points-to set sizes are very close to 1, ranging from 1 to a maximum of 1.2, the averages produced by static analyses are much higher. For the generally most precise analysis used in this study, the One-Level Flow algorithm, the sizes were 1.1 to 171 times larger. For *equake*, *art*, and *bzip2* all algorithms were able to produce the same or almost the same result as the dynamic points-to sets; common to these programs is that they pass around pointers to just one data structure, so that the context-insensitivity does not appear to introduce any imprecision for them.

Table 2. Dynamic versus static points-to set size average; both simple and weighted averages are shown.

Program	Average dynamic dereference size		Average static size of executed dereferences					
			One-Level Flow		Steensgaard		Shapiro-Horwitz	
	simple	weighted	simple	weighted	simple	weighted	simple	weighted
equake	1	1	1	1	1.04	1.00	1.00	1.00
art	1	1	1.11	1.10	1.17	1.20	1	1
mcf	1	1	2.83	2.63	2.83	2.63	2.83	2.63
bzip2	1	1	1.00	1.00	1.82	1.23	1.00	1.00
gzip	1.20	1.06	9.01	8.46	39.6	50.7	28.5	38.5
parser	1.12	1.68	6.84	8.38	66.7	68.7		
vpr	1.04	1.01	2.63	2.95	13.2	15.9		
m88ksim	1.14	1.01	6.65	10.6	99.7	136		
perl	1.04	1.06	21.2	23.1	57.3	59.0		
gap	1.00	1.01	6.86	6.95	86.3	86.8		
mesa	1.03	1.05	177	215	423	450		

Other than these three applications, Steensgaard's algorithm fared much worse, with ratios of static to dynamic sizes ranging up to 411. In general, our version of the Shapiro-Horwitz algorithm produced points-to sets larger than the OLF algorithm, but smaller than the sets produced by Steensgaard. Table 2 includes the numbers for those benchmarks for which we were able to run the algorithm to completion before running out of memory.

One of the most striking results of our experiments is shown in Table 3, which shows the number of points-to sets that were singletons, i.e., that had a size of 1. For the dynamic points-to sets 89.4% to 100% were singletons, with an average of 97%. This result is surprising because even in a single program run some routines are called multiple times from different places and with different arguments (e.g. addresses of different locals). The result implies that in almost all cases a program understanding tool that uses the dynamic information could present the user with a single target of a pointer dereference.

The number of singleton sets produced by static analysis is generally much smaller. For the OLF algorithm, with the exception of *equake* and *bzip2*, where the number of dynamic and static singleton sets were identical, the number of singleton sets is a factor of 1.1 to 37 smaller. Consequently, an average (geometric mean) of 14% of all executed dereferences with dynamic singleton sets also had static singleton sets. For the remaining 86% of dereferences, a program understanding tool, or an optimizer with purely static information, would deal with uncertainty (sets of size 2 or larger), whereas the dynamic information would pinpoint exactly one pointer target (at least for the particular input data set).

In summary, the static points-to sets were a factor of 5 larger than their static counterparts (geometric mean). Comparing the weighted average points-to sets sizes improves the ratios for some programs (*art*, *mcf*, *parser*), whereas they get worse or stay the same for the others.

Table 3. Number of dereferences with singleton dynamic points-to sets and percentage of total dereferences

Program	Number of static singleton sets	Number of dynamic singleton sets	Percentage of dynamic singleton sets (weighted)
equake	395	395	100 (100)
art	124	124	100 (100)
mcf	165	424	100 (100)
bzip2	400	400	100 (100)
gzip	137	361	89.4 (97.0)
parser	442	2365	96.3 (87.7)
vpr	782	1749	97.9 (99.6)
m88ksim	397	970	91.6 (99.2)
perl	165	1711	97.9 (95.3)
gap	178	6596	99.9 (99.5)
mesa	733	2053	97.9 (95.4)

[1] The executed dereference points represented between 9% (*mesa*) to 78% (*equake*) of all dereferences present in the program.

We also computed a lower bound on the number of cases when the static pointer information was optimal, by counting the number of static points-to sets that were identical to the dynamic sets.[1] With the exception of *bzip2* and *equake*, for which all of the dereferences were optimal, the percentages ranged from 88.7% (*art*) down to 2.7% for *perl*, with an averge of 48.0% of optimal static points-to sets.

Consequently, for about one out of two dereferences a tool like ours would be able to establish the optimality of the statically computed information, which could be used, for instance, to direct more expensive, but more precise static analyses, to those references for which optimality could not be established. Similarly, in a program understanding system, this information might be used to assign a confidence rating to pointer targets, focusing the user's attention on definitely feasible pointer targets. Furthermore, run-time optimization techniques, such as the redundant load elimination (as shown in the example in Section 2) can be used to optimize code in those cases where purely static optimizations are impeded by conservative static information.

5. RELATED WORK

As far as we know, our work represents the first application of program instrumentation to observe points-to sets at run time and compare them to their static equivalents.

Previous work in dynamic memory disambiguation by Bernstein et al. [3] attempted to improve execution time for numeric programs with array accesses. They used compile-time heuristics to select inner loops that might benefit from optimization assuming no aliasing. Such loops are duplicated, and at run time an aliasing check selects the appropriate code version. To avoid slowdowns, their heuristic had to be conservative, consequently, their approach often achieved no speedups. Dynamic aliasing data is likely to expose many more optimization opportunities than a purely static heuristic alone.

Diwan et al. [7] used binary instrumentation to measure the effectiveness of redundant load elimination based on memory disambiguation through alias analysis. They compared a number of simple algorithms for the strongly-typed object-oriented languages Java and Modula-3 by measuring the remaining percentage of redundant loads after eliminating those loads identified as redundant by the particular alias analysis algorithm. For the languages and benchmarks in their study they were able to show that their best static algorithm was close to optimal. This suggests that static alias analysis is more successful for strongly-typed languages than for a weakly-typed language such as C that is examined in this paper.

[1] As mentioned in the introduction, the dynamic sets may be smaller than the optimal sets if they are unsound. However, static points-to sets that are equal to their dynamic counterpart, must definitely represent optimal information.

Recently, Postiff et al. [16] have proposed a hardware extension for processors to support register allocation of variables that may possibly be aliased. Using a hardware table and compiler support, loads and stores to register-allocated aliased variables are forwarded to the register in which they are allocated. In their simulation they found a reduction of up to 35% of the loads and 15% of the stores. While their scheme requires a change in the processor hardware, the scheme we sketched in Section 2 requires no hardware modifications.

Das et al. [6] looked at lower and upper bounds for the number of possibly aliased data references in procedures. For each procedure they used the memory references occurring in the procedure to form pairs of memory references. For these pairs they used static points-to sets to compute whether the pairs could be aliased and compared these alias relationships to a lower bound where a pair was considered aliased only if it consisted of two identical references (e.g. `(x,x)` or `(*p, *p)`). They were able to show that with existing static, scalable analyses the number of references reported as aliased is very close to this lower bound. For the remaining pairs, however, and also in cases where the actual points-to set size matters (e.g. in debugging), dynamic points-to sets would be useful.

6. CONCLUSIONS

In this paper we have presented a comparison of pointer analysis information produced by static analyses and actual dynamically occurring behavior. Using a slightly modified instrumentation tool developed in the context of the Calpa system, we observed dynamically occurring points-to sets. We found that while static points-to sets are on the order of tens or hundreds of objects per dereference, even for the best scalable algorithm, the actual dynamically occurring sets are much smaller, with 97% of the sets being singletons, and average sizes close to 1. This suggests that a tool like ours can be used to supplement program understanding tools and significantly enhance their usefulness by improving on purely static information. Furthermore, profile data on pointer values can be exploited in feedback-directed optimization with potentially high performance benefits.

To assess the potential improvement that program understanding tools might obtain by using dynamic points-to information, we are currently integrating dynamic points-to sets information into a program slicer for C. We plan to compare slices that use static points-to information with slices based on dynamic points-to information and to quantify the potential advantages of the smaller dynamic sets.

7. ACKNOWLEDGMENTS

We would like to thank Mike Smith and Glenn Holloway for Machine SUIF and technical help using it.

8. REFERENCES

[1] L. Anderson. *Program Analysis and specialization for the C programming language*. Ph.D. thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[2] T. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290-301, June 1994.

[3] D. Bernstein, D. Cohen, and D. E. Maydan. Dynamic memory disambiguation for array references. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 105-111, November 1994.

[4] J. D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232-245, January 1993.

[5] M. Das. Unification-Based Pointer Analysis with Directional Assignments. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 35-46, June 2000.

[6] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. *Microsoft Research Technical Report 2001-20.* January 2001. Also to appear in *Proceedings of 8th International Static Analysis Symposium,* July 2001.

[7] A. Diwan, K. S. McKinley, and J. E. Moss. Type-Based Alias Analysis. in *SIGPLAN '98 Conference on Programming Language Design and Implementation,* pages 106-117, June 1998.

[8] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN '94 Conference on Programming Language Design and Implementation,* pages 242-256, June 1994.

[9] R. Ghiya and L.J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, December 1996.

[10] G. Holloway and C. Young. The flow and analysis libraries of machine SUIF. In *Proceedings of the 2nd SUIF Compiler Workshop*, August 1997.

[11] M. Hind and A. Pioli. Which pointer analysis should I use? In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, pages 113-123, August 2000.

[12] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 56-67, June 1993.

[13] D.Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference and ACM SIGSOFT Foundations of Software Engineering*, pages 199-215, September 1999.

[14] M. Mock, M. Berryman, C. Chambers, and S.J. Eggers. Calpa: A tool for automating dynamic compilation. In *2nd Workshop on Feedback-Directed Optimization*, November 1999.

[15] M. Mock, C. Chambers, and S. J. Eggers: Calpa: A Tool for Automating Selective Dynamic Compilation. In *Proceedings of the 33rd Annual Symposium on Microarchitecture*, pages 291-302, December 2000.

[16] M. Postiff, D. Greene, and T. Mudge. The store-load address table and speculative register promotion. In *Proceedings of the 33rd Annual Symposium on Microarchitecture*, pages 235-244, December 2000.

[17] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467-1471, September 1994.

[18] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Conference Record of POPL'97: Symposium on Principles of Programming Languages*, January 1997.

[19] M. D. Smith. Extending SUIF for machine-dependent optimizations. In *Proceedings of the first SUIF compiler workshop*, pages 14-15, January 1996.

[20] B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[21] R.P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C program. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1-12, June 1995.

[22] S. H. Yong, S.Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 91-103, May 1999.