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 and Engineering
University of Washington
Box 352350, Seattle WA 98195-2350
{mock, chambers, eggers}@cs.washington.edu

[+]Microsoft Research
Redmond, WA 98052
manuvir@microsoft.com

UW CSE Technical Report 01-03-01
Microsoft Research Technical Report MSR-TR-2001-38
March 2001

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 and Engineering
University of Washington
Box 352350, Seattle WA 98195-2350
{mock, chambers, eggers}@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 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). For example, we were able to obtain a factor of 6 speedup on a frequently executed routine from *m88ksim*.

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 [1,3,4,6,7,10,11,16,18,19,20]. These algorithms try to give 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 classes of pointer analysis algorithms have been designed. While flow- and context-sensitive algorithms potentially produce more precise results, they generally do not scale well. In addition, recent work [5,9] suggests that for typical C programs (e.g. SPEC benchmarks) Das's fast and highly scalable algorithm produces results as good as those of a context-sensitive algorithm. However, even its points-to sets are often 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 set 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 state-of-the-art pointer analysis algorithm. Dynamic points-to sets may tell us how close actual algorithms are to the theoretical optimum[1], and they can also 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 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. 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 useful to

[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 [15]. For programs exercising a large fraction of their execution paths, such as the SPEC benchmarks, however, we expect the dynamic sets to be not much smaller than an optimal solution.

present only the dynamically observed targets rather than all potential targets for all possible executions as an optimal (undecidable) static analysis would.

To obtain dynamic points-to information in this study, we used a slightly modified version of the Calpa instrumentation tool [13] 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 21.2 for the best of the scalable static pointer analysis algorithm we used, while the dynamic points-to set sizes were on average (geometric mean) a factor of 3.3 smaller. Additionally, for the large majority (over 98%) of dereferences the dynamic points-to sets were actually singletons. This means that in 98% of the time a tool similar to ours integrated into a program understanding tool, would be able to exactly tell the user the target of a dereference, a much higher fraction than what is possible with static analysis alone (41% of dereferences) demonstrating the substantial 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 dynamic points-to sets are almost always (98% of the time) of size 1, with the average size being close to 1 across all dereferences executed at run time, even though the static points-to sets are an order of magnitude larger in general;
- we show that these results can 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 we present our conclusions.

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, for instance, 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 in routine `align`, 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 the arguments 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 arguments 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 */
    } else {
      /* fast register-allocated version*/
    }
}
```

Note that dynamic points-to information is required to ensure that this transformation will be beneficial. The execution time penalty 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, which is unknown in the absence of profile information.

We hand-simulated register-allocation for the example by loading 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 are aliased. On a Compaq True64 Unix workstation with an Alpha 21264 processor running at 667 Mhz compiled with the vendor compiler and optimization flags -O2, this resulted in a speedup of

6.2. This speedup clearly demonstrates the potential benefits of using dynamic pointer information.

3. Instrumentation

We used the Calpa instrumentation tool [13] to instrument our applications. In the Calpa [12,13] 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 this purpose the instrumenter inserts an array of counter variables at each load or store instruction; 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. The instrumenter also 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[1].

Once the 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.

[1] To be able to match data structures on the heap, the instrumenter instruments calls to memory allocation routines such as `malloc`, associating the returned address with the allocation site (for which the alias analysis created a distinct symbol).

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, 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 proposed by Steensgaard [18] and Das [4], as well as an extension of Steensgaard's algorithm proposed by Shapiro and Horwit [16], where feasible within time and memory-constraints[2]. Both the points-to algorithms and the instrumentation tool are implemented using the Machine SUIF infrastructure [8,17].

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[3] for each pointer analysis algorithm.

[2] 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.

Program	Description	KLOC	Average dereference size OLF	Steensgaard	Shapiro-Horwitz
equake	seismic wave propagations simulation	1.2	1.02	1.05	1.02
art	image recognition, neural networks	1.2	1.22	1.35	1.09
mcf	combinatorial optimization	1.9	2.88	2.88	2.88
bzip2	compression	3.9	1.01	1.88	1.01
gzip	compression	7.6	7.42	35.2	24.4
parser	Word processing	10.3	6.74	66.2	
vpr	FPGA circuit placement and routing	13.6	2.51	12.6	
m88ksim	Motorola 88000 instruction set simulator	19.4	5.74	98.2	
perl	perl interpreter	26.8	21.2	56.1	
gap	group theory interpreter	62.5		88.5	

Table 1. Description of the Workload.

4.2 Dynamic Points-To Sets Results

To obtain the dynamic points-to sets we instrumented the applications as described in Section 3. Then we ran the instrumented applications on the SPEC-provided test inputs. We chose the test inputs because the reference inputs take much longer to run, and since the instrumentation slows down the applications by about 2 orders of magnitude, running the reference inputs was generally impractical. 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 *bzip2* and *mcf* also on the reference inputs, and found the results to be same.

One of the most striking results of our experiments is shown in Table 2, 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 90% to 100% were singletons, with an average of 98% of all sets being singletons. This means 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 *bzip2* and *equake*, where the number of dynamic and static singleton sets differs only by 2, the number of singleton sets is a factor of 1.3 to 10.2 smaller, representing only from 9.6% to 99.5% of all executed dereferences, with an average percentage of singleton sets of 41%. For the remaining 59% of dereferences, a program understanding tool, or an optimizer with purely static information, would deal with uncertainty (sets of size 2 or larger).

In Table 3 we compare the average sizes of the dynamically observed points-to sets for each application with the average sizes of the static sets. While in Table 1 the static points-to set sizes include all dereference points, in Table 3 only those dereference points are included, that were actually executed. This allows a comparison with dynamic points-to sets, which are determinable only at executed dereferences. We show average points-to sets sizes, and (in parentheses) 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 1.18, the averages produced by static analyses are much higher. For the generally most precise analysis, Das' One-Level Flow algorithm, the sizes were from 2.8 to 20 times larger, with the notable exception of *bzip2* and *equake*, where the OLF algorithm was able to produce the same (*bzip2*) or almost the same (*equake*) result as the dynamic points-to sets. On average the static points-to sets were a factor of 3.3 larger (geometric mean). Comparing the weighted average points-to sets sizes

[3] The points-to set sizes that we report for Steensgaard's and Das' algorithm, are slightly different to the ones reported in [4]. 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 include one representative object for all references to string constants which are not included in Das's numbers, and (3) we include points-to set of calls through function pointers in our counts.

improves the ratios for some programs (*equake*, *art*, *mcf*, *parser*), whereas they get worse or stay the same for the others. Figure 1 shows the unweighted average points-to set sizes for the OLF algorithm and the dynamic set sizes pictorially.

In the comparison of static to dynamic points-to set sizes, Steensgaard's algorithm fared much worse, with ratios ranging from 1.8 to 88. Weighting the sizes by the execution frequency had the same effect on the ratios as for the OLF algorithm.

In general, the Shapiro-Horwitz algorithm produced points-to sets larger than the OLF algorithm, but smaller than the sets produced by Steensgaard. Table 3 includes the numbers for those benchmarks for which we were able to run the algorithm to completion before running out of memory.

Program	Number of static singleton sets	Number of dynamic singleton sets	Percentage of dynamic singleton sets (weighted)
equake	393	395	100 (100)
art	92	124	100 (100)
mcf	171	431	100 (100)
bzip2	401	403	100 (100)
gzip	151	407	90.4 (97.1)
parser	448	2375	96.3 (87.7)
vpr	772	1773	98.0 (87.7)
m88ksim	576	1233	93.2 (99.3)
perl	169	1719	97.9 (95.3)
gap[a]		6690	99.7 (99.5)

Table 2. Number of dereferences with singleton dynamic points-to sets and percentage of total dereferences

a.Due to a memory leak bug in our OLF alias analysis, we are unable to report OLF numbers for gap at this tim. However, we'll have those numbers for the final version of the paper, where we also plan to include at least one other large application (mesa).

To find out how often the dynamic points-to information may allow to establish that static points-to sets are optimal, we counted the number of statically computed sets that were identical to the dynamic sets.[1] With the exception of *bzip2* and *equake*, for which at least 99.5% of the dereferences were optimal, the percentages ranged from 74.2% (*art*) down to 9.8% for *perl*, with an averge of 51.9% 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

[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.

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 [2] 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.

Recently, Postiff et al. [14] 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. [5] looked at lower and upper bounds for the number of possibly aliased data references in procedures. Using points-to sets to compute alias relationships, they were able to show that with existing static, scalable analyses the number of references reported as aliased is very close to the lower bound. However, they did not look dynamic points-to sets, and their potential uses.

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

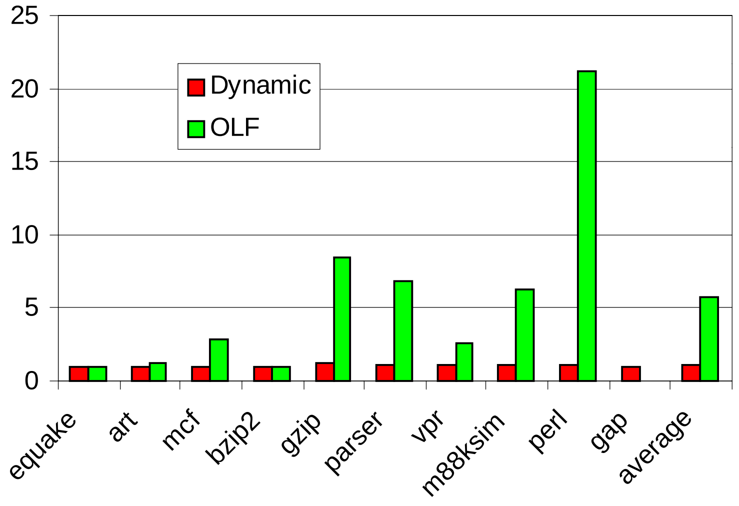

Figure 1. Comparison of average static and dynamic points-to set size for each benchmark, and the average over all benchmarks.

observed dynamically occurring points-to sets. We found that while static points-to sets even for the best scalable algorithm are on the order of tens or hundreds of objects per dereference, the actual dynamically occurring sets are much smaller, with 98% 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.

Program	Average Dynamic Dereference Size	Static Size of Executed Dereferences One-Level Flow	Steensgaard	Shapiro-Horwitz
equake	1 (1)	1.01 (1.00)	1.02 (1.00)	1.01 (1.00)
art	1 (1)	1.26 (1.11)	1.37 (1.20)	1.15 (1.00)
mcf	1(1)	2.81 (2.63)	2.81 (2.63)	2.81 (2.63)
bzip2	1 (1)	1.00 (1.00)	1.82 (1.23)	1.00 (1.00)
gzip	1.18 (1.06)	8.44 (7.85)	38.6 (49.8)	27.8 (37.6)
parser	1.12 (1.68)	6.81 (8.38)	66.6 (68.7)	
vpr	1.03 (1.01)	2.63 (2.95)	13.1 (15.9)	
m88ksim	1.11 (1.01)	6.3 (10.6)	93.3 (137.8)	
perl	1.04 (1.06)	21.2 (22.9)	57.3 (59.0)	
gap	1.01 (1.03)		88.5 (88.8)	

Table 3. Dynamic versus static points-to set size average; averages weighted by execution frequency are shown in parentheses.

Acknowledgements

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

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] 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, 1994.

[3]. D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *2oth Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232-245, Jan. 1993.

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

[5] M. Das, B. Liblit, M. Fahndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. *Microsoft Research Technical Report 2001-20.* 2001.Submitted to SAS 2001.

[6] 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. *SIGPLAN Notices, 29(6)*

[7] 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.

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

[9] 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, Aug. 2000.

[10] 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. *SIGPLAN NOtices 28(6)*

[11] D.Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In O. Nierstrasz and M. Lemoine, editors, *Lecture Notes in Computer Science, 1687*, pages 199-215, Springer-Verlag, Sept. 1999. Proceedings of the *7th European Software Engineering Conference and ACM SIGSOFT Foundations of Software Engineering*.

[12] 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.

[13] 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, Dec. 2000

[14] 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, Dec. 2000

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

[16] 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.

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

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

[19] 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. *SIGPLAN Notices, 30(6)*

[20] 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, 1999.