On the Impact of Naming Methods for Heap-Oriented Pointers in C Programs

Tong Chen, Jin Lin, Wei-Chung Hsu and Pen-Chung Yew
Department of Computer Science
University of Minnesota
{tchen, jin, hsu, yew}@cs.umn.edu

Abstract

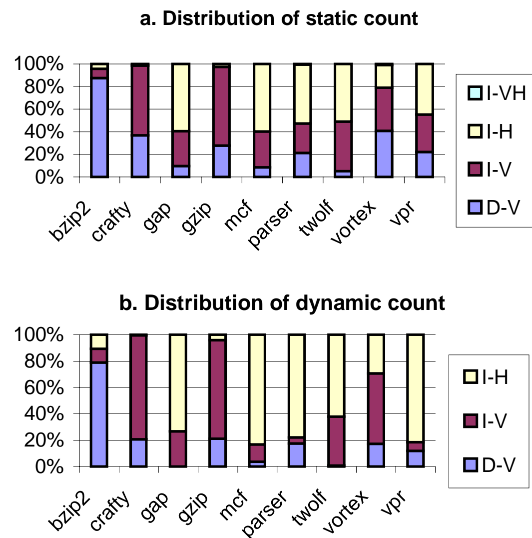
Many applications written in C allocate memory blocks for their major data structures from the heap space at runtime. The analysis of heap-oriented pointers in such programs is critical for compilers to generate high performance code. However, most previous research on pointer analysis mostly focuses on pointers pointing to global or local variables. In this paper, we study points-to analysis of heap-oriented pointers using profiling information. An instrumentation tool and a set of library routines are developed to measure points-to sets of memory references at runtime. Different naming methods for heap-oriented pointers are studied. We found that it is very important to adopt appropriate naming methods to recognize wrapper functions for memory allocation and memory management functions defined by users. Based on these naming methods, the approaches in pointer analysis, such as flow sensitivity and context sensitivity, are examined with the runtime tool. The program characteristics are observed at runtime to evaluate what kind of compiler analysis is needed. Experiments are conducted on SPEC CPU2000 integer benchmarks. We found that flow sensitivity and context sensitivity have little impact on the analysis of heap-oriented pointers.

1. Introduction

Pointers are used in many applications written in C. These pointers could pose a problem to compilers because it is often unclear what locations may actually be accessed by pointer-based indirect memory references at runtime. Due to the lack of knowledge of the targets for these memory references, compilers may have to make conservative assumptions, and consequently, prohibit many useful optimizations, resulting in less efficient code

The pointer analysis aims to inform the compilers whether memory references access the same memory blocks or not. One approach, the points-to analysis [1, 2, 3], tries to identify all of the targets of each memory reference so that it can be determined whether two references are aliased or not by comparing their target sets. The result of points-to analysis can also be used in other analyses, such as data dependence test [12] and shape analysis [4, 15].

A pointer may point to global or local variables, called a stack-oriented pointer; or memory blocks allocated from the heap space at runtime, called a heap-oriented pointer; or to both [5]. The use of heap memory gives programs flexibility to adapt their memory usage according to the size of the problems. Therefore, many programs allocate their major data structures from the heap space. Figure 1 reports the distribution of memory references, read or write, in some of the SPEC CPU2000 integer benchmarks. The references through heap-oriented pointers take up a large portion, in both static count at compile time (Figure 1.a) and dynamic count at runtime (Figure 1.b). The analysis of heap-oriented pointers is thus critical to these applications.



D-V: direct references accessing variables.
I-V: indirect references accessing variables.
I-H: indirect references accessing heap blocks.
I-VH: indirect references accessing both variables and heap blocks.
Memory references caused by register spill are not included because they are irrelevant to a pointer analysis.

Figure 1. Distribution of memory references

Researchers have been trying to find efficient and yet effective heuristic algorithms for pointer analysis [7, 8, 9, 10, 11] because a precise pointer analysis is

· The work was supported in part by the U. S. National Science Foundation under Grants EIA-9971666 and MIP-9610379 and a grant from the Intel Corporation

IEEE COMPUTER SOCIETY

an NP- hard problem [6]. The effectiveness of a pointer analysis depends on the characteristics of pointers in real applications.

Most of the previous studies focused mainly on stack-oriented pointers. The pointer analysis of heap-oriented pointers should start with a naming process that gives names to anonymous heap memory blocks so that they can be expressed as points-to targets in pointer analysis. Such a naming method implicitly determines the *granularity* of the named objects. For instance, the compiler can give the same name to all of the memory blocks allocated in all of the instances of the same *malloc* statement, or it can give a different name to each memory block allocated in each instance of the same *malloc* statement. The granularity of the named objects, and the number of names associated with the allocated heap space can be quite different. Consequently, the results of the analysis on heap-oriented pointers, such as the size of points-to sets and the percentage of points-to sets with only one target, may vary significantly with different naming methods. The characteristics of heap-oriented pointers under different naming methods require a more careful study.

Instead of implementing and comparing different pointer analysis methods directly in a compiler, we developed an instrumentation and profiling tool based on the Intel's ORC compiler [14]. Different naming methods and the precise points-to sets based on each naming method are collected at runtime. Alias pairs can then be identified by their points-to sets, and we can use them to evaluate the upper bound potential of each naming method. We also examine the characteristics of heap-oriented pointers, such as the size of their points-to sets, and the effect of flow sensitivity and context sensitivity. Our experiments are conducted on SPEC CPU2000 integer benchmarks.

Our approach has several advantages. First, it is much easier to develop such a profiling tool than to implement different naming methods and pointer analyses in a real compiler. We can quickly study the characteristics of applications and evaluate the effectiveness of different naming schemes and pointer analyses before we actually implement them. Secondly, the precise points-to sets for each naming method can be collected at runtime and used as a measure to evaluate the effectiveness of the pointer analysis. Since these results are collected at runtime, they can be regarded as approximated upper bounds that these naming methods and pointer analyses can be expected to achieve. The third advantage of our approach is that the results thus obtained are independent of a particular compiler and the idiosyncrasy of how its naming methods and pointer analysis are implemented. This could be a distinct advantage, because the effectiveness of inter-procedural pointer analysis depends heavily on how they are implemented.

However, such a profiling method also has its limitations. Since our results are collected during runtime, they could be input dependent. They do not cover the entire program, but rather only the parts that are actually executed. Hence, these results may not reflect the exact characteristics of a particular program when it is actually analyzed by a real compiler. Nevertheless, the main focus of our study is to compare the effectiveness of different naming methods and pointer analyses on the entire benchmark suite, not on each individual program. We believe that these limitations will not significantly affect the results of our study. However, a more detailed study using only compile time analysis will definitely be needed.

The main contributions of our study include:

- An instrumentation and profiling methodology to study heap-oriented pointers. This tool is capable of calculating precise points-to sets for each memory reference, including both stack-oriented and heap-oriented memory references. It does not require a compiler to conduct pointer analysis in advance [13].
- A comprehensive study on different naming methods for heap-oriented pointers. We found that the widely used simple naming method is inadequate. Wrapper functions and self-management functions that contain system memory allocation functions need to be carefully analyzed.
- Based on a proper naming method, we found that flow-insensitive points-to analysis is quite effective for heap-oriented pointers. Flow sensitive analysis only adds marginal improvement.
- Based on a proper naming method, we found that some procedures have different side effects at different call sites. However, their alias patterns in each procedure remain mostly the same in different calling contexts. This shows that schemes, such as partial transfer function [7], may be quite effective.

The rest of the paper is organized as follows: we describe our approach and our methodology in the next section. Section 3 reports the experiment results. Finally the conclusions can be found in section 4.

2. Approach

In this section, we start with an introduction of the related background knowledge. Later, the tool to collect points-to sets at runtime and the methods to evaluate different naming methods are discussed in detail.

2.1 Background

The requirement of a naming method is the main difference between the analysis of the stack-oriented pointers and that of the heap-oriented pointers. Since all variables have explicit names in a program, a target of a stack-oriented pointer can be represented by its given variable name. However, no explicit names are given to dynamically allocated memory blocks from the heap space in the program. The compiler needs to

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)
1087-4089/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

assign names to those dynamically allocated memory blocks to facilitate its later analyses and optimizations.

Typical system functions that dynamically allocate memory blocks from heap space at runtime include: *malloc*, *calloc*, and *allocate*. For our convenience, the *mallac* will be used to refer to all the memory allocation functions in the later discussions. We did not include the function *realloc* because even though *realloc* may allocate a new memory block to change the size of an existing heap block, the new block can always inherit the name of the old one. There is no need to give it a new name.

There are several possible naming methods for dynamically allocated memory blocks in the heap space. In this paper, we classify them by what is used in the naming methods as follows:

- One name. Assign only one name to the entire heap space. Namely, all of the dynamically allocated memory blocks will belong to the same named object. The consequence of such a naming scheme is that all of the heap-oriented pointers are all aliased together. Since the result of this method is obvious, we will not measure it in our paper.
- Line numbers. A memory block dynamically allocated in a statement, for instance, by a *malloc* function, is named by the line number of the statement. This is the most common naming scheme used in existing compilers. This naming scheme will allow different memory blocks dynamically allocated in different statements to have different names. This is a significant improvement over using only one name. However, it still cannot differentiate memory blocks dynamically allocated in the same statement, but from different call sites of that procedure. This will make all of the dynamically allocated memory blocks from different call sites to become aliased.
- Calling paths. To improve the precision in naming heap objects, we can name dynamically allocated heap memory blocks by their calling paths in addition to its statement line number. The calling path is a sequence of call sites from the *main* function to this *malloc* call. We can also limit the length of each calling path to a fixed number L. When a calling path is partially used, we usually select the call sites backwardly, instead of the call sites starting from the *main* function. For example, if procedure *A* is called by procedure *B* which is in turn called by procedure *C*, and a *malloc* function in procedure *A* is in statement at line number 100. If we set L=2, the dynamically allocated heap memory blocks will be named "*B*-*A*-100", and with L=3, it will be named "*C*-*B*-*A*-100". Using this convention, the calling path length of the previous method is L=1. To an extreme, a naming method can use the entire calling path no matter how long the path is. We call this naming method L=a, where “a” means “all” parts of a calling path. We will show the effect of different calling path lengths in the next section.
- Memory blocks. We found some of the *malloc* functions are actually called within loops. Using calling paths, all of the dynamically allocated heap memory blocks within a loop will be assigned the same name and cannot be differentiated. To further differentiate them, we give each memory block allocated in loops a unique name. This is obviously beyond what a compiler can do statically at compile time, and we will use this method only as the upper bound for all of the other naming methods. We use L=u, "u" for "unique", to represent this method.

2.2 The profiling tool

Our profiling tool includes two major components: an instrumentation tool developed based on the Intel's ORC compiler [14], and a set of library routines written in C. An application program is instrumented by the modified ORC compiler. Corresponding library routine calls are inserted at every entrance and exit of a procedure, every invocation of system memory allocation function (such as *malloc*), and every indirect memory reference. These instrumented library routine calls will maintain the current calling path, assign names to dynamically allocated memory blocks according to the specified naming method and then calculate the points-to set of each memory expression at runtime. We could then analyze all alias pairs based on their points-to sets. The application programs we studied include a subset of SPEC CPU2000 integer benchmark programs[1].

The advantage of our proposed scheme is that the address value of an indirect memory reference is known at runtime. We only need to associate its address value with its corresponding name. To facilitate efficient lookup of the address value with its assigned name, we associate a shadow location with each memory location. The shadow location keeps the name of the memory object associated with the memory location. For global and local variables (i.e. stack-oriented variables), the shadow location stores its symbol table ID. For an indirect memory reference, it will store the name assigned according to the specified naming method. A separate hash table is set up to keep track of the points-to targets associated with each indirect memory reference. The table is then post-processed to determine the pair-wise aliases. Two indirect memory references are aliased if there exist at least one common target in their points-to sets.

2.3 Evaluate naming methods

After the target sets are computed, each naming method is evaluated by the percentage of aliased reference pairs in all pairs of memory references. A

[1] Since there are two input data sets for *vpr* to cover different parts of this program, *vpr* is split into *vpr1* and *vpr2* according to the input data.

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN’02)
1087-4089/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

memory reference is not included if it cannot be reached at runtime.

This measure is used because the size of a points-to set [11] is no longer appropriate for heap-oriented pointers. The size of a points-to set depends heavily on the naming method and not necessarily reflects the effectiveness of a pointer analysis. For example, when only one name is given to all heap blocks, the size of points-to set is one. Although the size of points-to sets is low, this is obviously not an effective pointer analysis.

To count alias pairs, we compare the references within a procedure because most optimizations that are the consumers of the results from a pointer alias analysis are applied only in the scope of a procedure. This measurement reflects the impact of pointer analysis on the generated code.

Reference pairs, instead of variable pairs, are chosen for measurements because reference pairs can be flow-sensitive.

3. Experiment results

Several measurements have been conducted to study the characteristics of heap-oriented pointers. First, we study the impact of naming methods on heap-oriented pointers. Then, the impact of flow sensitivity and context sensitivity in pointer analysis is evaluated based on the proper naming methods.

3.1 Impact of naming methods

For the different naming methods discussed in the previous section, the references in each procedure are checked pair-wise and the percentages of alias pairs in each program are reported in Figure 2. Only memory references that access heap memory blocks at runtime are considered. On average, 57.1% of static indirect memory references in the benchmarks are reached at runtime.

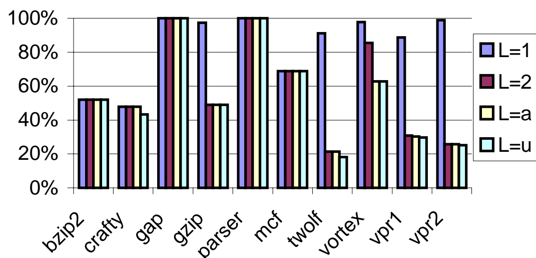


Figure 2. Percentage of alias pairs with different naming methods

From figure 2, we observed:

- In half of the benchmarks, naming heap blocks with only the line number, namely L=1, is not enough. The number of alias pairs can be greatly reduced when more precise naming methods are used.
- For programs, such as *twolf*, *vpr* and *vortex*, heap memory allocation functions are wrapped in utility functions of these programs. Therefore L=2 is quite sufficient to recognize these wrapper functions.
- L>2 does not show much further improvement in these benchmarks.
- Some benchmarks, such as *parser* and *gap* get no help from traditional pointer analysis. These programs allocate a large chunk of memory block once, and then manage this memory block by themselves.

The number of distinct names used is shown in Table 1. These data could explain, to some extent, why the percentage of alias pairs changes with respect to the naming methods, and also indicate possible overhead associated with the naming methods. For L=1, it is the number of system memory allocation functions actually reached at runtime. For L=2 and L=a, they are the numbers of names generated according to the calling contexts. The size of name space is an indication of how precise the naming could be. The number of alias pairs may be reduced when the size of name space is increased. This is why benchmarks of *bzip2*, *crafty* and *parser* show no improvement over different naming methods: the size of their name space remains the same. The size of name space is also an indication of the cost for points-to analysis. More names used, more time and space will be needed in points-to analysis.

Table 1. Average size of the name space

	L=1	L=2	L=a
bzip2	8	8	8
crafty	6	6	6
gzip	3	3	3
parser	1	1	1
mcf	3	3	3
twolf	2	142	170
vortex	4	6	2891
vpr1	2	84	166
vpr2	2	40	52

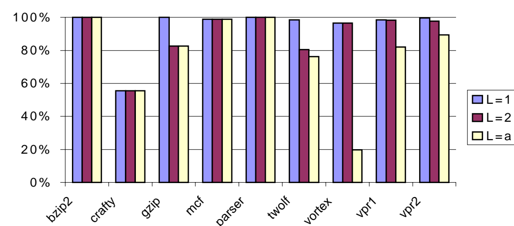


Figure 3. Percentage of single target references

Table 2 shows the average size of points-to sets, and Figure 3 shows the percentage of singleton (points-to set with only on target). The average size of points-to sets of heap-oriented pointers is very close to 1 in all of the naming methods, execept for vortex with L=a. The percentage of singletons is high for L=1 and L=2. These data indicate that naming heap

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)
1087-4089/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

memory blocks with a limited length of calling paths do not introduce much overhead. However, some pragram may have very large points-to sets, if we use the complete calling path. Since the naming method of L=2 is the best considering both the effectiveness and efficiency for these benchmarks, it is used in the following flow sensitivity and context senstivity study.

Table 2. Average size of points-to sets

	L=1	L=2	L=a
bzip2	1.00	1.00	1.00
crafty	1.40	1.40	1.40
gzip	1.00	1.30	1.30
mcf	1.01	1.01	1.01
parser	1.00	1.00	1.00
twolf	1.01	1.24	1.34
vortex	1.04	1.05	15.97
vpr1	1.02	1.02	1.55
vpr2	1.00	1.03	1.17

3.2 Flow sensitivity

One of the key cost factors in pointer analysis is its flow sensitivity. A flow sensitive pointer analysis propagates the value of pointers on the control flow graph, while a flow insensitive pointer analysis does not consider the control flow. It takes more time for flow sensitive analysis, but the result may be more accurate.

The source of inaccuracy in flow insensitive analysis is that pointers may be assigned different names. There is no kill and all the definitions will be propagated to each read point, though some of the definitions may be unreachable when control flow is taken into account.

To observe the impact of flow sensitivity on heap-oriented pointers, we instrument each assignment statements to record which location is written to and what value is written to. All of the values assigned to a location are then propagated to all indirect reference expressions to simulate flow insensitive analysis.

Table 3. Comparison of flow sensitive and flow insensitive

	Flow senstivie analysis			flow insensitive analysis		
	size of points-to set		alias (%)	size of points-to set		alias (%)
	average	max		average	max	
twolf	1.24	5	76.2	1.53	9	76.2
vpr1	1.02	2	82.0	1.04	2	83.6
vpr2	1.03	2	89.4	1.04	2	89.8

Table 3 compares the average size, maximum size of points-to sets, and percentage of alias pairs in a flow sensitive and a flow insensitive analysis. The programs that have small name space are excluded because the results for those programs are not very interesting. Though the points-to set is increased in some benchmarks, the percentage of alias pairs does not change. This result shows the flow insensitive analysis can be effective for these benchmarks.

3.3 Context Sensitivity

A context-sensitive pointer analysis analyzes a procedure for each of its calling contexts because different calling contexts may lead to different alias patterns. The context-sensitive pointer analysis is usually very expensive.

In this measurement, we study how programs behave under different contexts. First, we study whether the side effect of a procedure will change in different calling contexts or not. Second, we study whether the alias patterns will change in different calling contexts.

The method used in context sensitivity study is an extension of the method used in calculating points-to set under different naming methods. The points-to sets calculated previously are context insensitive. The points-to targets coming from different calling contexts are recorded together. Now we will distinguish the points-to targets from different calling contexts. As discussed in the previous section, our tool maintains a calling path from the *main* function to current function at runtime. To find out the impact of context sensitivity, each points-to target is attached with its calling path ID. The read/write set and the alias pairs of a procedure for each calling path are then computed. In a context sensitive analysis, two references are aliased only when they have the same target with the same calling path ID.

Table 4. Changes caused by context sensitive pointer analysis

	side effect of procedures	alias pairs	alias pairs that are reached
bzip2	0.0%	0.7%	0.0%
crafty	13.7%	0.0%	0.0%
twolf	2.6%	0.4%	0.0%
vortex	7.4%	17.3%	0.0%
vpr1	10.2%	5.1%	0.0%
vpr2	4.1%	0.2%	0.0%

Table 4 reports the changes of analysis results when points-to are calculated in a calling context sensitive way. The first data column reports the percentage of procedures that have different side effects in different calling contexts. The side effect of a procedure is measured by the set of memory objects that are read and the set of memory objects that are written in the procedure. The second column shows the percentage of alias pairs reduced when alias pairs are checked with their calling path IDs. An alias pair may becomes non-alias due to the change of the control flow in different calling contexts. Some branch paths are not executed and the references in those branch paths have no targets. This kind of change is related to the control flow, and has less

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)
1087-4089/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

impact on a pointer analysis. What we care about is the spurious alias pairs in which either reference is reached at runtime. Only these pairs may be eliminated by a context sensitive analysis. An example of such spurious alias pairs is illustrated in Figure 4. The percentage of procedures that have spurious alias pairs is reported in the last column.

```
foo (int *p, int *q) {
        *p = *q+k;
}

call sites:
        foo(&a, &b);
        foo(&b, &a);
```

The pair of **p* and **q* is aliased in a context insensitive analysis because they both point to *a* and *b*. However, this pair is not aliased in a context sensitive analysis.

Figure 4. An example of context sensitivity

The data in Table 4 shows that some procedures have different side effects for heap memory references and different alias pairs at different call sites. However, the change of alias pairs is only caused by the change of the control flow and the alias patterns of procedures remain almost the same. This result indicates that flow insensitive pointer analysis may be powerful enough to most programs. However, context sensitive analysis may be important for the side effect analysis.

4. Conclusions

This paper presented a new approach to study the pointer analysis for heap-oriented pointers. A tool is developed to study different naming methods for heap-oriented pointers, and precise points-to sets are collected at runtime. With the help of this tool, we found that simple naming methods used in many existing compilers may be insufficient, and more sophisticated naming methods are needed.

Based on an effective naming method, the flow sensitivity and context sensitivity of pointer analysis for heap-oriented pointers are studied. A flow insensitive analysis for such pointers is found to be powerful enough. We also found that the side effects of procedures do change with calling contexts while alias patterns of procedures are context insensitive.

References

[1] J. Choi, M. Burke, and P. arini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and sife-effects. In Proceedings of the ACM 20th Symposium on Principles of Programming Languages, pages 232-245, January 1993.

[2] W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, page 235-248, July 1992.

[3] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, pages 242-256, June 1994.

[4] X. Tang, R. Ghiya, L. J. Hendren, and G.R. Gao. Heap analysis and optimizations for threaded programs. In Proc. Of the 1997 Conf. On Parallel Architectures and Compilation Techniques, Nov. 1997

[5] Nevin Heintze and Olivier Tardieu. Demand-Driven Pointer Analysis. ACM SIGPLAN Conference on Programming Language Design and Implementation 2001.

[6] W. Landi. Undecidability of static analysis. ACM Letters on Programing Languages and Systems, 1(4):323-337, Dec, 1992.

[7] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, pages 1-12, June 1995.

[8] Bjarne Steensgaard. Points-to analysis in almost linear time. In Conference Record of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, Pages 32-41, January, 1996.

[9] Michael Hind and Anthony Pioli. Evaluating the effectiveness of Pointer Alias Analysis. Science of Computer Programming, 39(1):31-35, January 2001

[10] Bixia Zheng. Integrating scalar analyses and optimizations in a parallelizing and optimizing compiler. PhD thesis, February 2000.

[11] Ben-Chung Cheng. Compile-time memory disambiguation for C programs. PhD. Thesis, 2000.

[12] Rakesh Ghiya, Daniel Lavery and David Sehr. On the Importance of Points-To Analysis and Other Memory Disambiguation methods For C programs. In Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation, page 47-58, June 2001.

[13] Markus Mock, Manuvir Das, Acraig Chambers, and Susan J. Eggers. Dynamic Points-to Sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimzation. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis 14for Software tools and Engineering, June 2001.

[14] Roy Ju, Sun Chan, and Chengyong Wu. Open Resource Compiler for the Itanium Family. Tutorial at the 34th Annual International Symposium on Microarchitecture.

[15] M. Sagiv, T. Reps and R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. ACM Transactions on Programming Languages and Systems, 20(1):1-50, January 1998.

Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)
1087-4089/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY