國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

以執行監測爲基礎之別名分析

Profile-based Alias Analysis

鄭宇修

Yu-Hsiu Cheng

指導教授：劉邦鋒 博士

Advisor: Pangfeng Liu, Ph.D.

中華民國99年7月

July 2010

誌謝

在碩士班的這兩年間，首先要感謝的是我的指導教授劉邦鋒老師，因爲有了他的指導， 不論是在專業領域或是研究態度，都讓我在這兩年間有所成長，今日能順利得到碩士 學位，都要感謝老師的指導。

再來要謝謝研究室的學長以及一起努力的同學，謝謝他們兩年來的陪伴，不管在做研究 、修課、平日的相處，謝謝他們一路來所提供的幫助。

最後要感謝我的家人，因爲有他們一路來的陪伴及支持，我才能有動力突破各個難關， 獲得碩士學位，謝謝他們一路來對我的包容及關愛。

摘要

在許多程式分析中，別名分析扮演了一個重要的角色。跟傳統缺乏動態資訊的靜態別名分析相比， 使用執行監測的方法具有極大的潛力，因爲並不需要做過度的預測。 在這篇論文中，我們提出一個用執行監測的框架以用來做別名分析。 我們發現使用執行監測的方法可以比傳統的靜態別名分析獲得更多的非別名答案。 儘管我們可以獲得更準確的別名分析，但執行時間並沒有太多的成長。 我們也針對文本性質和域性質分別做了探討，並發現在文本性質中，非環式的方式較其他的方法 有效以及域性質對別名分析是有很大的影響等結論。

關鍵字　　別名分析、執行監測、Open64、記憶體分辨、文本性質、域性質

Abstract

Alias analysis plays an important role in various program analyses Comparing to traditional static alias analysis which struggles for precision under the lack of dynamic information, profile techniques show a great potential in this area since they don't need to make any over-approximation. In this paper, we propose an framework which uses profile techniques to perform alias analysis. We can get more 52% queries than the worst case for 429.mcf. We also find that although we have a more accurate alias analysis, the performance has insignificance gain. And we simulate several schemes in context sensitivity and field sensitivity. We find that acyclic scheme in context sensitivity is useful and field sensitivity is important for alias analysis.

Keywords Alias analysis, Profiling, Open64, Memory disambiguation, Context Sensitivity, Field Sensitivity

Contents

List of Figures

Chapter 1

Introduction

Alias analysis is important in program analyses as indirect memory references, such as pointers in C programs and references in Java programs, are ubiquitous. Recently, various applications such as program understanding, software verification, data race detection [1], and automatic lock generation [2], all heavily rely on alias information. Unfortunately, traditional static alias analysis has some limitations in precision because it can only over-approximate program behaviors. Thus, profile techniques have a great potential in this area since they could get exact dynamic information. In this section, we briefly introduce serveral common schemes, including the transfer functions for pointer assignments, flow sensitivity, context sensitivity, field sensitivity, and abstract names of memory in static analysis. We also identify their restrictions and describe why profile techniques doesn't have these problems.

Most static alias analyses can be classified by the transfer functions for pointer assignments into two categories: inclusion-based [3] and unification-based [4]. In inclusion-based analyses, a pointer assignment `p = q` implies that the locations pointed by `q` are a subset of the locations pointed by `p`. In unification-based analyses, the locations respectively pointed by `p` and `q` are unified. We could easily notice that the precision of inclusion-based analysis is better than unification-based analysis. Nevertheless, some unavoidable imprecision would be introduced and propagated around even in inclusion-based analyses. Profile could totally eliminate the transfer function and conquer these

```
if (a*b == 0)
        p = &c; // block 1
else
        p = &d; // block 2
... // block 3 irrelevant to a, b, p
if (a==0 || b==0){
        ... // block 4
}
```

Figure 1.1: points-to(p) should be the same in block 1 and 4

problems by capturing what is actually accessed at each independent location of the program.

Flow-sensitive alias analyses provide different alias information at different positions in the control flow graph. Thus, they are very important for applications caring about local information like some local optimizers in compilers. Even though the obvious benefits, this option is eliminated in many analyses because of its extra overhead. Furthermore, the artificial example in Figure 1.1 illustrates a situation that the fact points-to-set(p) $= \{c\}$ in block 4 could not be trivially captured without the help of other program transformation even by flow-sensitive analysis. The reason is that standard control flow graph doesn't capture the dependence of block 1 and 4, and the imprecision caused by conservatively merging the points-to sets from block 1, 2 into block 3 as points-to-set(p) $= \{c, d\}$. Similarly, this restriction could be resolved by profiling what actually accessed at each location, because it's caused by another transfer function.

Context-sensitive analyses distinguish different calling contexts of methods. Rather than generating false alias in callees like Figure 1.2 and in callers like Figure 1.3, context-sensitive approaches generate much more precise information, and is very useful in analyses requiring inter-procedural information. They are even more important in object-oriented languages like Java, where objects could be treated as contexts [5]. However, complete context-sensitive alias information is almost not reachable because there are usually cycles, resulting infinite possible calling paths, in call graph. Only dealing acyclic paths is also very challenging because the number usually grow exponential to the number

```
void func(int *p, int *q);
func(&a, &b);
func(&b, &a);
```

Figure 1.2: Imprecise points-to(p) = points-to(q) = $\{a, b\}$ is generated in callees when ignoring contexts

```
int *func(int *a) { return a;}
p = func(&a);
q = func(&b);
```

Figure 1.3: Imprecise points-to(p) = points-to(q) = $\{a, b\}$ is generated in callers when ignoring contexts

of procedures. Profile is capable to capture the boundaries of procedures during execution, so context-sensitive information is easy to obtain. Moreover, profile only captures real execution paths which form a much smaller space.

Field sensitivity is non-negligible when aggregate data, such as structures and unions in C programs, plays an important role in the analyzed program. Practical implementations like [6] make a compromise to distinguish fields but not between individual instances. The concept could be extended to deal with arrays as elements are actually fields of the whole array. Since the number of elements is normally tremendously larger than the declared variables, collapsing each elements as the whole array is common due to the scalability issues. Pearce et al. [7, 8] extended the inclusion-based alias analysis to reach field sensitivity. But according to their result, they think field sensitivity is expensive to compute. As profile could see the exact region of memory referenced, it is not difficult to design a scheme to reach complete field sensitivity. But according to [9], our result is not portable since the memory layout of structures is implementation dependent.

Naming schemes also need to be carefully selected for the precision purpose. User-declared variables and line numbers are the most intuitive names for memory objects on stack of activation frames and in the heap for dynamic data. Unfortunately, these are too coarse-grained. For example, there might be multiple instances of the same procedure on the stack of activation frames like Figure 1.4, and the variable name could not distinguish

```
void func(int c, int *p)
{
        int q;
        if (c!=0) func(c-1, &q);
}
```

Figure 1.4: p is not pointing to the local q

```
void *wrapper(){
        void *p = malloc(SIZE);
        ... // do initialization
        return p;
}
p = wrapper();  // instead of malloc()
q = wrapper();  // instead of malloc()
```

Figure 1.5: Wrapper Functions Like malloc()

the multiple instances of local variables. The wrapper function behaving like `malloc()` is another example. In Figure 1.5, the points-to sets of both p and q will be collapse to the same line number of the real call to `malloc()` instead of the calls to the wrapper functions. Dynamic behaviors complicate the naming problem too. As Figure 1.6, the fields of dynamic allocated objects are difficult to assign names. Memory objects dynamic allocated in a loop are also hard to assign names uniquely due to the nondeterministic number of iterations. Regardless of traditional names, profile can be aware of these issues by resolving accessed addresses to capture multiple instances on the stack of activation frames, monitoring `malloc()` to capture dynamic memory allocations, recording at each memory reference point to know the real propagation of dynamically allocated memory, and the real access patterns.

In this paper, we use the compiler profile techniques to do alias analysis. At compile time, we instrument profile library functions at every memory references, dynamic mem-

```
p = malloc(SizeA + SizeB + SizeC);
// access the block in the order of C, B, A
```

Figure 1.6: Access patterns could not be inferred by sources.

ory allocations, and procedures. During the execution, the instrumented program collects the point-to sets information as profile data. Since alias queries issued by optimizers in the compiler are used to evaluate our approach, we finally re-compile the program again and feed profile data for solving those queries.

The rest of this paper is organized as follows. Section 2 describes the related work. Section 3 describes the design of our system. Section 4 shows our experiment result and demonstrate this system could be used to construct prototypes of different alias analyses. Section 5 give the conclusions.

Chapter 2

Related Work

According to M. Hind [10], there are two dimensions in alias analysis, scalability and precision. How to get a reasonable trade off between these two conditions is still an unsolved problem. Most researchers have focused on improving scalability of static alias analysis [11, 12, 13, 14, 15, 16, 17, 18, 19] recently , but not too many discuss precision, since more precise information means more costs to analyze programs. If we can get more precise alias information, we'll have more opportunities to take more aggressive utilization.

To the best of our understanding, Mock et al. [20] have pioneered in applying profile techniques to alias analysis. Their goal is to provide program understanding tools a more useful dynamic points-to sets rather than potential candidate targets of a pointer. They use Calpa instrumentation tool to instrument their applications, and present a comparison of several static pointer analyses and dynamic behavior. Their results show that dynamic behavior can get that average points-to sets size close to 1 which is better than other static alias analyse. This means using dynamic tools could get more precise information than static alias analyses and could help improving static information.

Chen et al. [21] [22] have also used profile techniques on alias analysis. They studied the importance of the granularity of the naming scheme. Naming scheme is the most important part in their work. They assign names to heap-oriented memory objects according to different length of call paths which maintained by profile library, and assign names to global variables and local variables according to symbol table id. But this may still cause some imprecision since using symbolic names may get same name in two different memory objects. We extend their work and consider more dynamic information to fully utilize more benefits of profile techniques.

Ghiya et al. [6] has already studied the correspondence between their memory disambiguation framework and the performance of the compiled program. Their results of little extra performance improvement contributed alone by alias analysis are very similar to ours. Though their inter-procedural alias analysis is flow-insensitive and only uses the line number to deal with dynamically allocated memory, the deficiency is hided by other heuristics. We're different in ignoring the heuristics used in their framework and still achieve high disambiguation ability.

PIN [23] is used in monitoring the program behavior. However, PIN monitors the program at instruction level instead of at IR level. Although we can get the details of the program at runtime through PIN, we can't improve the program performance by this way since several reasons. First, we can't feedback the data into compiler. Second, the data may be too huge. Finally, we may have no source codes.

Zhang et al. [24] present an infrastructure using another dynamic instrument tool Valgrind [25, 26] to analyze data dependence. Their infrastructure could provide users the regions which are suitable to parallelize. Since it still need programmer to parallelize the codes, their infrastructure has some drawbacks.

Lin et al. [27] proposed a compiler framework with data speculation. They use speculative alias and dataflow analysis to improve optimization. Their result show that they can achieve some performace gain in their framework. The difference between their and our framework is we didn't use speculative optimization.

Chapter 3

System Design

Our alias profile framework with the clients of alias information in Open64 [28] is shown as Figure 3.1. There are three phases. In the first phase, the source code of the investigated program is compiled. During the compilation, the compiler will insert instrumentation code to the intermediate representation of the original program. Then, the instrumented code will be linked with our profile library, where our alias scheme is implemented, to generate an executable binary. We simply execute the instrumented program in the second phase. In addition to the original functionalities, it would generate the profile data containing the points-to set information. The final phase is recompiling the source code and feeding the profile data into compiler. In this phase, optimizers in Open64 are expected to get better alias information.

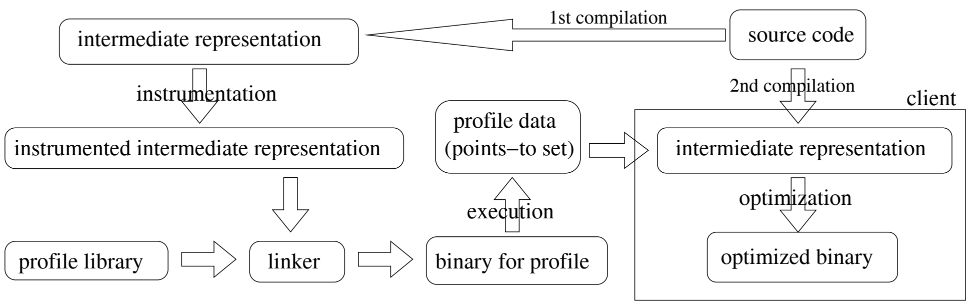

Figure 3.1: Framework

The original source code:

```
*p = ...;
```

The instrumented code:

```
*p = ...;
PFLIB_MEMORY_REFERENCE(ref_id ,address ,size)
```

Figure 3.2: Instrumentation to Memory Reference

3.1 Instrumentation

The primary goal in the first phase is to insert our library functions into the original program. The design philosophy is to collect sufficient information to the profile library for alias analysis. This section describes the instrumentation of memory references, procedures and dynamic memory allocations, and highlights the correctness and properties about instrumentation at intermediate representation.

Memory reference is the most important information we need, since we are interested in alias analysis and pointers are primary used as indirect memory references. Thus, our instrumenting compiler simply walks through the intermediate representation of the currently compiled program and inserts profile library function calls to collect the real address and size of the memory reference. The idea is depicted as Figure 3.2. Though we show it in the form of C code, the real work is done at IR level.

Procedures also need to be instrumented as mentioned. We need to instrument both before and after the procedures in callers since we need to maintain a stack to keep procedure information. The second reason is the callee may have multiple exits. It's pretty tedious and unreasonable to instrument every possible exits. And the last reason is we can establish the context relation by these functions. But we don't enable this feature now. Instrumenting profile library functions at the beginning of every procedure body is used to identify the exact procedure even when function pointers are involved. The idea of instrumentation is depicted in Figure 3.3. This approach could also deal with the situation that the callee is in pre-compiled libraries. In this case, our profile library could still

The original source code:

```
void f(){
}
void g(){
        f();
}
```

The instrumented code:

```
void f(){
        PFLIB_PROC_EXEC(proc_id_f);
}
void g(){
        PFLIB_PROC_EXEC(proc_id_g)
        PFLIB_BEFORE_PROC_CALL(call_id_x)
        f()
        PFLIB_AFTER_PROC_CALL(call_id_x)
}
```

Figure 3.3: Instrumentation to Procedure Call

do pre- and post-processing to the procedure call with the only loss of the information about the callee. And we need to instrumented a special version at the beginning of the `main()` to initialize our profile library and install a post-processing function to capture the termination of the original program via `atexit()` in the instrumented routine. The idea is shown in Figure 3.4.

Memory allocation is also needed to instrument for our alias analysis, we also generated a special version of instrumentation to the standard allocation-related routines as in Figure 3.5. Unfortunately, this approach could not capture memory allocation behaviors in pre-compiled libraries.

The correctness about whether instrumentation modifies the original behaviors of the program is worthy to discuss. For example, one might think instrumentation introduces new memory references as the result of inserted function calls and register spilling, so what we want to monitor is also changed. Undoubtedly, the behavior is changed in the instruction level. However, what we monitor is the behavior at the IR level. The memory reference behaviors in the original program are preserved in the instrumented code and

```
int main(){
        PFLIB_MAIN_EXEC()
}
```

A typical profile library:

```
void PFLIB_MAIN_EXEC(){
        atexit(PF_MAIN_EXIT, ...);
}
void PFLIB_MAIN_EXIT(){
}
```

Figure 3.4: Instrumentation to main()

The original source code:

```
p = malloc(size);
```

The instrumented code:

```
p = malloc(size);
PFLIB_MALLOC(alloc_id_x, p, size);
```

Figure 3.5: Instrumentation to Memory Allocation

information describing each reference is passed to the profile library via inserted function calls. The newly introduced behaviors like new memory references do not affect the original functionality of the application and aren't noticed by the profile library.

The properties of instrumentation at IR level rather than the instruction level should also be highlighted. The local variables whose addresses aren't taken as pointers are usually transformed by the compiler as pseudo-registers, and registers has no memory addresses to be taken. Thus, it is impossible to know every actual memory reference by our framework. On some architectures like IA-32, the number of architecture registers are so few that most memory references are accessing local variables. However, under the IR abstraction, most of these references are just data manipulations on registers and ignored by our mechanism. Thus, what could be monitored by our framework is somewhat biased relating to the real machine behaviors. However, as we only consider the memory references relating to alias analysis, there is no loss of interesting information. Moreover, it helped boosting the performance of the instrumented binary as the memory references to the irrelevant local variables and memory references caused by memory spilling are not noticed by the profile library.

3.2 Profile Library

The profile library is implemented with the most features of alias analysis in our framework, so it is the core part of the system. After the profile phase finishes, the profile library generates profile data containing the points-to sets information for each memory reference location. We describe how to reach flow, context, and field sensitivity and the naming scheme used in the proposed alias analysis.

Flow sensitivity is naturally achieved by our instrumentation framework because we have individually instrumented each memory reference. Thus, the points-to sets obtained at a particular point in the control flow graph is independent to other points. This is the essence of flow sensitivity. Although we can acheive this attribute easily, our framework only can get the information on real execution paths.

Context sensitivity analysis is also easy in our framework. We push the sensitivity to the each dynamic instance of procedure calls. As each procedure call information is passed into the profile library, the library generates an unique context identifier for subsequent record, and restore the previous context identifier after this procedure finishes. We simulate two schemes in context sensitivity, k-CFA scheme and acyclic scheme respectively. `k`-CFA scheme is used to distinguish calling context by last `k` procedure calls. Acyclic scheme doesn't consider cycle calling paths.

Our framework is field-sensitive since we can directly get the address and size of memory references. Through this mechanism, we can get the detail of a field if a memory reference accesses to the field. We are interested in the effect which field sensitivity cause, so we try to simulate field insensitivity. We need to map the adress of memory reference to entire memory object in field insensitivity scheme.

We take the advantage of dynamic profiling to directly use the memory address to describe the elements in the points-to sets. Since we consider each different context independently, there is no false alias between local variables on the activation frames and global variables when analyzing a particular frame. However, it might take the same addresses in the heap as the same objects, when the memory location is actually released and re-allocated. That means the same addresses on the heap could represent different memory objects over time. We try to use a naming scheme to solve this problem. That means, we assign a name to each memory object. Every memory object on the heap has a unique name. Thus, we can distinguish different memory objects even if they have the same address.

3.3 Feedback

The goal of the third phase is to feed the profile data containing points-to set information to the clients for improving its performance. In our current implementation, the clients are optimizers in Open64. This section describes the feedback mechanism, what is expected, and the correctness.

Feedback is a typical mechanism in profile-guided compilers. The compiler also walks through the IR during second compilation. At the point before instrumentation in the first phase, the program should be the same as what it is before instrumentation. Instead of inserting codes to the IR, the compiler reads the points-to set of each memory reference in the profile data, and annotates it to the corresponding location. Then, the profile data would be carried around with the IR during subsequent transformations. An alias query, which contains a pair of interested load or store, would be issued when an analysis need it. We intercept the issued alias queries, and give a better answer based on the profile data. Calculating the answer to any alias query could be done by the set intersection. Under the assumption that the points-to sets are completely profiled, the queried pair is solved as `NOT_ALIASED` if the intersection is empty. Otherwise, it is solved as `POSSIBLY_ALIASED`. The context information is encoded in the name of each accessed element, so information from different contexts won't confuse the set intersection. However, this mechanism doesn't fully utilize the context-sensitive alias information. To completely take the advantages of the context-sensitivity alias information, we should compile different versions of callees of different function calls if the points-to information is different in those function calls. We discarded this feature in our implementation since this approach would let the compiled program grow exponentially.

More definite `NOT_ALIASED` are expected to those queries. We have to highlight that the later queries issued by other analyses are changed as answers of the earlier queries changed. Comparing the percentage of queries solved as `NOT_ALIASED` is not mathematically meaningful since the queries themselves is changed, but it reflects how much information sent back to the client is definite rather than ambiguous. Undoubtedly, we could measure the improvement by the profiled information, and still keep the answers of static alias analysis unchanged. We discard the option since it doesn't reflect the usage of a typical client. For example, if accessed targets are already determined as `NOT_ALIASED`, a data race detector will not interested in the real locks protecting these areas.

A correct answer to an alias query should be conservative regarding to traditional optimizers. That means it is always safe to identify memory references as `POSSIBLY_ALIASED`. Once the analysis identifies them as `NOT_ALIASED`, they should never be aliased. Thus, if we identify a common element in the profile points-to sets, the answer `POSSIBLY_ALIASED` is definitely correct. In comparison, if the intersection of the two points-to sets is empty, that only means they are highly possible `NOT_ALIASED` because of the inadequate input. Our implementation doesn't measure the probability and aggressively reply `NOT_ALIASED`. The optimizers are slightly fault-tolerant that a incorrect optimization is usually caused by several erroneous answers. An incorrect optimization would lead to segmentation faults in the execution of the compiled program. In this paper, we only discuss precision on the basis of correctly compiled program for all selected inputs. To correctly utilize profiled points-to sets in real world, we should build the probability model and the clients should also consider it.

Chapter 4

Experiments

We implemented the framework by extending the instrumentation feature in the x86 Open64 compiler 4.2.3-1. We focus our study on the alias queries issued at optimization level -O2, which is the most stable configuration. We do instrumentation and feedback at code generation(CG) phase since the optimizers using alias information are only at this phase. Perlbench and libquantum are eliminated due to incorrect compilation caused by Open64 with instrumentation. Benchmarks are compiled into 64-bit binaries and tested on an x86-64 machine with Intel® Xeon® CPU X5550 at 2.67GHz and 24G memory. We use all the input sets provided in the SPEC 2006 suite as the possible input space, and carefully ensure all the compiled programs correctly executed for all the input sets. To simulate a typical usage of a profile-guided system, the smallest input set is used as the representative input for profile.

The queries issued from optimizers in Open64 are used to reflect the real usage of a client of alias information. The profile information is used to improve the default intra-procedural unification-based alias analysis in Open64, and this shows the ability to combine profiled information with high level information in the compiler. The details about these queries are showed in Figure 4.1 where redundant queries are only counted one. The type heuristic [29] is also listed for comparison because it is widely believed to be lightweight and efficient in memory disambiguation.

All the profiled-based schemes improve the results of alias queries than default unification-

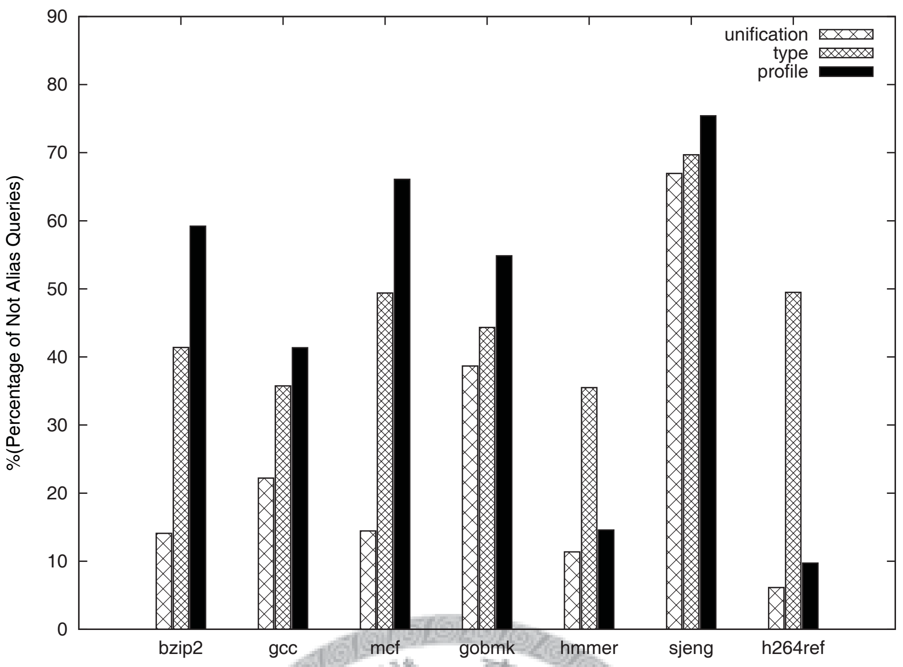

Figure 4.1: Queries Solved as `NOT_ALIASED`

based alias analysis. The result is also better than the type heuristic except hmmer and h264ref. The defects are caused by the low coverage of executed code. The profile scheme uses the most advantages of dynamic information and distinguishes the most points-to sets.

Suprisingly, littel improvement in the performance of optimized binaries is really achieved with better alias information. We measure the performance and the results are listed in Table 4.2. The performance of hmmer and h264 is also improved with our profiled alias information because the information falls in the hot region of the program. Nevertheless, we still have to mention that the performance measurement is not very appropriate that it is affected by many factors and may be unexpected. Performance of the benchmarks compiled with no alias information is listed as a proof of the insensitivity to alias information.

We simulate several schemes in context sensitivity. The aliasing pairs found in context senstivity is showed in Fig4.3. We can find that 0-CFA is satisfied for several benchmarks. But like gcc, it needs 2-CFA. Therefore we think, the effectiness of context sensitive alias

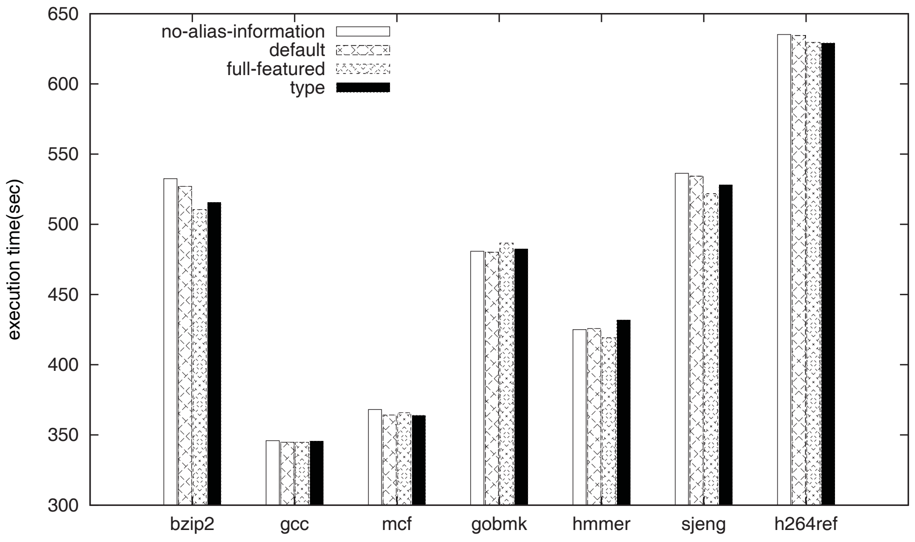

Figure 4.2: Execution Time

analysis depend on the structure of application. Wrapper functions, data structure, and constructor, all required an additional level of analysis. The result of acyclic scheme is similar to the result of unristicted path scheme.

The aliasing pairs found in field sensitivity and insensitivty are normalized as Fig4.4. We can see that field sensitivity is important since the number of aliasing pairs from field insensitivity scheme is about 3.9 times higher than from field sensitivity scheme.

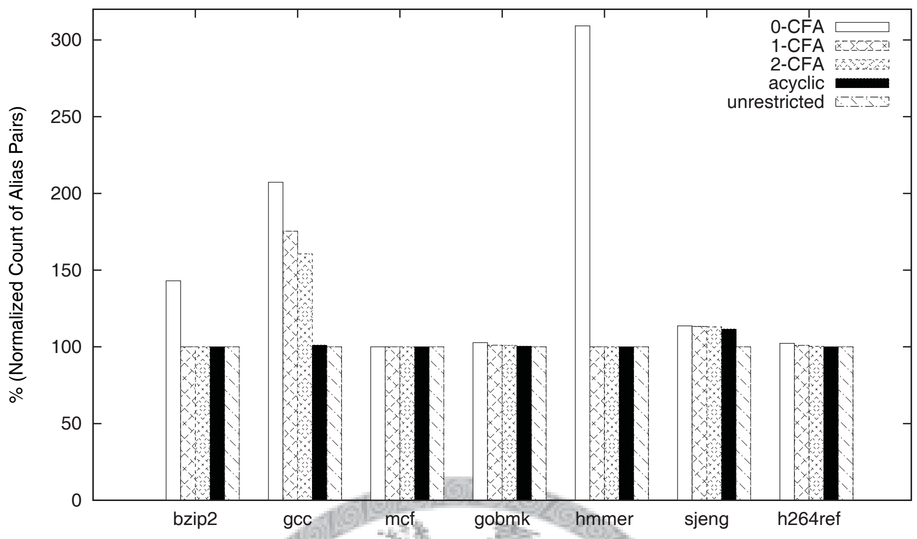

Figure 4.3: Context Sensitivity

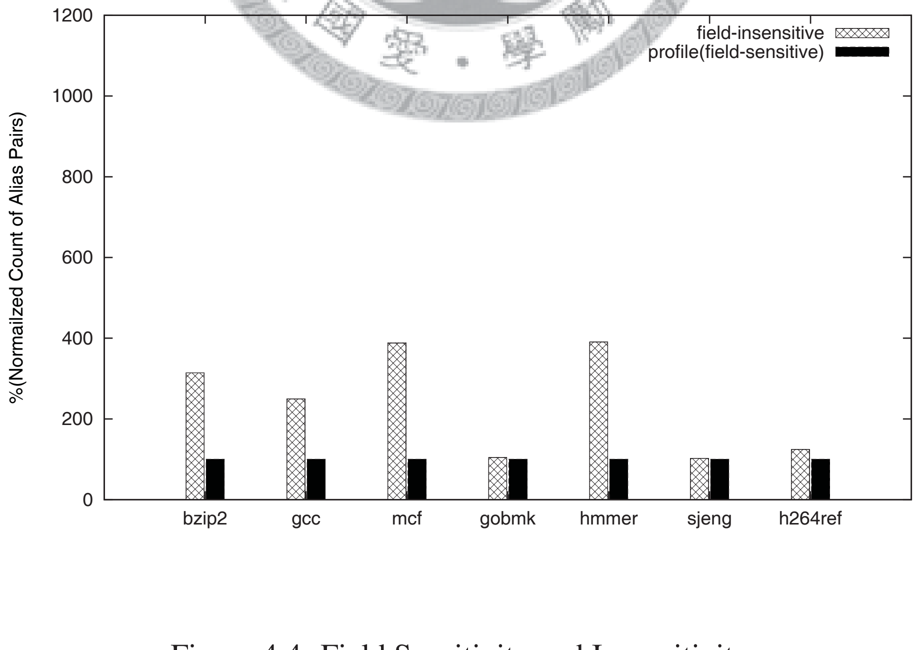

Figure 4.4: Field Sensitivity and Insensitivity

Chapter 5

Conclusion

We have identified several problems of transition functions , flow sensitivity, context sensitivity, field sensitivity and naming schemes in static alias analysis. Using profile techniques would not have the same problems and can easily achieve the above attributes.

We proposed a framework which uses profile techniques to collect dynamic information. When measuring intra-procedural aliasing pairs, the worst case for 456.hmmer is about 10 times than our profile scheme. When we measure alias queries, we can get more 52% queries than the worst case for 429.mcf. And we also find that although we have a more accurate alias analysis, the performance has small gain.

We believe that the alias profiling has a great potential as the probability model and clients can consider the probability nature of this alias information. As many current implementations of optimizations haven't utilized the inter-procedural alias information, there is only small improvement in performance of compiled programs with accurate alias information, and we urge the research and implementations of optimizations to take this advantage.

Bibliography

[1] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, New York, NY, USA, 2006. ACM.

[2] G. Upadhyaya, S.P. Midkiff, and V.S. Pai. Using data structure knowledge for efficient lock generation and strong atomicity. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 281–292, New York, NY, USA, 2010. ACM.

[3] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[4] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[5] A. Milanova, A. Rountev, and B.G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. ACM.

[6] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. *SIGPLAN Not.*, 36:47–58, May 2001.

[7] D.J. Pearce, P.H.J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for c. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 37–42, New York, NY, USA, 2004. ACM.

[8] D.J. Pearce, P.H.J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of c. *ACM Transactions on Programming Languages and Systems*, 30, 2007.

[9] S.H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 91–103, New York, NY, USA, 1999. ACM.

[10] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *In workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, New York, NY, USA, 2001. ACM.

[11] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, New York, NY, USA, 2007. ACM.

[12] F.M.Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 126–135, Washington, DC, USA, 2009. IEEE.

[13] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 226–238, New York, NY, USA, 2009. ACM.

[14] V. Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 2008 ACM SIGPLAN Conference on*

Programming Language Design and Implementation, pages 249–259, New York, NY, USA, 2008. ACM.

[15] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–289, New York, NY, USA, 2007. ACM.

[16] T.W. Sheng, W.G. Chen, and W.M. Zheng. A context-sensitive poiner analysis phase in open64 compiler. In *Open64 Workshop in conjunction with IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.

[17] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 254–263, New York, NY, USA, 2001. ACM.

[18] J. Whaley and M.S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM.

[19] D.J. Pearce, P.H.J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 3–12, 2003.

[20] M. Mock, M. Das, C. Chambers, and S.J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential application in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 66–72, New York, NY, USA, 2001. ACM.

[21] T. Chen, J. Lin, W.C. Hsu, and P.C. Yew. An empirical study on the granularity of pointer analysis in c programs. In *In 15th Workshop on Languages and Compilers for Parallel Computing*, pages 157–171, 2002.

[22] T. Chen, J. Lin, W.C. Hsu, and P.C. Yew. On the impact of naming methods for heap-oriented pointers in c programs. In *ISPAN '02: 2002 Proceedings of the International Symposium on Parallel Architectures, Algorithm and Networks*, Washington, DC, USA, 2002. IEEE.

[23] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40:190–200, June 2005.

[24] X.Y. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 47–58, Washington, DC, USA, 2009. IEEE.

[25] Valgrind. http://valgrind.org.

[26] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[27] J. Lin, T. Chen, W.C. Hsu, P.C. Yew, R.D.C. Ju, T.F. Ngai, and S. Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, pages 289–299, New York, NY, USA, 2003. ACM.

[28] Open64. http://www.open64.net.

[29] A. Diwan, K.S. McKinley, and J.E.B. Moss. Type-based alias analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 106–117, New York, NY, USA, 1998. ACM.