An Empirical Study on the Granularity of Pointer Analysis 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

Pointer analysis plays a critical role in modern C compilers because of the frequent appearances of pointer expressions. It is even more important for data dependence analysis, which is essential in exploiting parallelism, because complex data structures such as arrays are often accessed through pointers in C. One of the important aspects of pointer analysis methods is their granularity, the way in which the memory objects are named for analysis. The naming schemes used in a pointer analysis affect its effectiveness, especially for pointers pointing to heap memory blocks. In this paper, we present a new approach that applies the compiler analysis and profiling techniques together to study the impact of the granularity in pointer analyses. An instrumentation tool, based on the Intel's Open Resource Compiler (ORC), is devised to simulate different naming schemes and collect precise target sets for indirect references at runtime. The collected target sets are then fed back to the ORC compiler to evaluate the effectiveness of different granularity in pointer analyses. The change of the alias queries in the compiler analyses and the change of performance of the output code at different granularity levels are observed. With the experiments on the SPEC CPU2000 integer benchmarks, we found that 1) finer granularity of pointer analysis show great potential in optimizations, and may bring about up to 15% performance improvement, 2) the common naming scheme, which gives heap memory blocks names according to the line number of system memory allocation calls, is not powerful enough for some benchmarks. The wrapper functions for allocation or the user-defined memory management functions have to be recognized to produce better pointer analysis result, 3) pointer analysis of fine granularity requires inter-procedural analysis, and 4) it is also quite important that a naming scheme distinguish the fields of a structure in the targets.

1. Introduction

The pervasive use of pointer expressions in C programs has created a serious problem for the C compilers. Without proper pointer analyses, compilers would not have accurate knowledge of what memory objects may have been accessed by indirect references. Consequently, many other important analyses, such as data dependence analysis on arrays and complex data structures, may suffer from the conservative assumptions about the targets of pointers. Hence, pointer analysis plays a critical role in C compilers in exploiting parallelism [12]. It provides the analysis base for other analysis and parallelizing techniques.

Many pointer analysis methods have been proposed ([1, 2, 3]). Among all the pointer analyses, the points-to analysis [2, 4, 5, 6, 7, 8, 9, 10, 15] is the most widely used. A points-to analysis aims to produce a set of potential targets for each indirect reference so that the alias relationship among pointers can be determined by comparing their target sets. Efforts have been put in searching for a good points-to analysis [11, 12, 13, 21].

The effectiveness of a pointer analysis is generally determined by two factors: the *algorithm* used, and the *granularity* of the points-to targets specified in the compiler. For example, the algorithms used by compilers may have different flow-sensitivity or context sensitivity. The algorithm may also be applied inter-procedurally or only intra-procedurally.

To calculate the target sets, the address space of memory objects in a program should first be assigned *names*. The *granularity* of the names represents the precision of the *naming schemes* used in pointer analysis. Different naming schemes may lead to different granularity in pointer analyses. In general, there are two types of memory objects: the local or global variables defined in the program, and heap memory blocks allocated at runtime. The pointers that point to global or local variables are called stack-oriented pointers; and the pointers that point to memory blocks are called heap-oriented pointers [5]. For heap-oriented pointers, their target objects are anonymous. Compilers have to assign them names internally before the target sets could be calculated. For example, if the compiler assigns the entire heap space with only one name, the entire heap space will

be viewed as only one large memory object. All of the pointers point to different memory locations in the heap space will have the same target in their target sets, and they will all be aliases. On the other hand, for stack-oriented pointers, global and local variables usually have explicitly given variable names in the program, and with well-defined types. However, if the compiler treats an entire data structure with many fields as a single memory object, all of the pointers point to the different fields of the data structure will be aliases.

The granularity of the target objects and its related naming schemes not only affect the results of a pointer analysis, but also the efficiency of its algorithm. Finer granularity will allow better distinction among different memory objects, and hence, fewer aliases. However, it may lead to a larger name space and possibly larger target set sizes, and hence, longer time and more storage requirement for a points-to analysis.

Various naming schemes have been proposed in the past [10, 16, 24, 27, 28]. For anonymous heap memory objects, the place where they are allocated is used to name them. For memory objects of structure type, the field names may be used in their names. Some experiments have been done [25, 29] and showed the importance of proper naming methods. However a comprehensive study on the impact of the granularity on pointer analysis has not been done. Most of the previous studies focus primarily on the algorithms. One reason is that it is not trivial to implement different naming schemes in conjunction with various pointer analysis algorithms. Another reason is that the heap memory objects have not received enough attention in the past. In most compilers, only very simple naming schemes are used for heap memory blocks. However, a recent study shows that the number of heap-oriented pointers is quite significant in most SPEC CPU2000 programs [17]. Hence, it is important to look at the impact of naming schemes and the granularity on the pointer analysis and the optimizations that use the results of the pointer analysis.

In this paper, we study this problem using a new approach that combines the profiling techniques and the compiler analysis. We developed an instrumentation and profiling tool set based on the Intel's Open Research Compiler (ORC) [14]. Different naming schemes are simulated and the precise target sets of indirect references (e.g. pointers) are collected at runtime for the points-to analysis. We then feed the results of the points-to analysis back to the ORC compiler. The improvement on the results of alias queries in other compiler analysis and optimizations and the performance of the code thus generated are also measured. Our experiments are conducted on SPEC CPU2000 integer benchmarks and on Intel Itanium computers.

The suggested approach does not have to implement pointer analyses with different granularity in a compiler. It is much easier to simulate these analyses with a runtime tool. The points-to set collected by this tool is an upper bound result and reveals the potential of different granularity. Using the optimizations in the ORC compiler as consumers makes the measurement of effectiveness meaningful. However, we have to admit that some import issues, such as the impact of the algorithm, are not covered in this paper.

The main contributions of this paper include:

- A comprehensive study on the naming schemes and the granularity of the pointer analysis. We found that the widely used simple naming schemes are inadequate. Wrapper functions and self-management functions that contain system memory allocation functions (such as *malloc()*) need to be carefully analyzed. It is also important for a pointer analysis to consider the fields of a data structure.
- A set of instrumentation and profiling tools to study issues related to pointer analysis. We develop a tool that is capable of calculating precise target sets for each pointer reference. This tool set is independent of the pointer analysis used in a compiler.
- The impact of the pointer analysis on compiler optimizations. We feed the target sets collected at runtime back into the ORC compiler to help later analyses and optimizations, and measure the performance improvement on Itanium. It provides a very direct way to study the impact of naming schemes and granularity on performance.

The rest of the paper is organized as follows: The background knowledge of points-to analysis is introduced in the next section. Section 3 and section 4 describe, in detail, how the instrumentation and profiling tool works, and how the runtime results are fed back to the ORC compiler to evaluate different naming schemes and granularity levels. The experiment results are presented in section 5. The conclusions are presented in section 6.

2. Background

In a points-to analysis, memory objects, such as variables and heap memory blocks, need their *names* so the compiler can identify them as the targets of pointers. A naming scheme sets up a mapping from the memory address space to the symbolic name space. These naming schemes differ in the way memory objects are grouped together, and the names assigned to them. As a result, the naming schemes

implicitly determine the granularity of memory objects used within the compiler.

Global variables have explicit and fixed variable names in a program. Therefore, using the variable names sets up a precise one-to-one mapping between their corresponding memory locations and their names. The local variables within a procedure also have explicit variable names. But there may be many instances of a local variable at runtime if the procedure is called recursively. A name for a local variable may represent many instances of the variable in different procedure instances. However, such many-to-one mapping is usually thought as a quite precise.

Heap memory objects have no explicit names assigned to them in the program. The number of memory blocks allocated at runtime by the *malloc()* function is unknown at compile time. The compiler has to group those heap memory blocks and assigns them a name to facilitate points-to analysis.

These anonymous memory objects created by all of the *malloc()* functions in the program could be assigned the *same* name[26]. If that is the case, all references accessing to any memory block allocated by the *malloc()* are aliases. This obviously is not very desirable. Hence, the compiler often assigns names to memory blocks according to the line number of the statement which contains *malloc()* function in the program. This allows memory blocks allocated at different call sites of the *malloc()* function to have different names, and hence, be treated as different points-to targets. This is significantly better than the previous naming scheme. However, if the *malloc()* function is called within the procedure X, and the procedure X is called several times at different call sites. All of the memory blocks allocated at different call sites of procedure X will have the same name.

To avoid such a problem, the compiler can also assign a name according to the calling path at the invocation site of the *malloc()* function in addition to the line number [10]. For example, if procedure X calls procedure Y which in turn calls procedure Z, and a *malloc()* is called within procedure Z. The memory blocks allocated by the *malloc()* can be assigned a name according to its calling path X-Y-Z in addition to its line number. To control the complexity of such a naming scheme, the compiler can use only the last *n* procedures of a calling path in its naming scheme. In the last example, if n=2, Y-Z will be used. Different n will thus give different levels of granularity to the named memory objects.

When a memory object is a structure type with many fields, the granularity of the memory object can be made even finer by considering each of its field as a different memory object. As a result, two pointers that point to different fields of a memory object of the structure type can be distinguished. However, since C is not a strong-typed language, type casting has to be monitored carefully when fields are considered. Notice that the naming of the dynamically allocated memory blocks and separating the fields of the structure-type memory objects are orthogonal, i.e. they can be used independently in determining the granularity of memory objects.

In the following discussion, the granularity level, G, of a naming scheme will be represented by these two considerations. For example, G=n means that the last n procedures in the calling path are used, but fields are not considered. When n=0, it is the degenerate case of assigning the entire heap space with only one name; when n=1, only the line number is used. G=nf means the fields are also considered in addition to the calling path.

3. Target sets in different naming schemes

3.1. Overview

We developed an instrumentation and profiling tool to simulate different naming schemes and collect their target sets of indirect references. Our approach takes advantage of the fact that the addresses of memory objects and references are all available at runtime.

The selected naming scheme is simulated by setting up a mapping at runtime from the addresses of memory objects to their names according to the naming scheme. Targets of a pointer are identified by looking up the mapping with the addresses of the references to their names. The target sets thus obtained represent approximately the best results that these naming schemes and pointer analyses can be expected to achieve.

To facilitate the lookup process, shadows are used to record the address-name mapping. There are three contiguous data segments in a program: the global variable segment, the heap memory segment and the local variables segment. A library routine for system memory allocation is provided to assure that the heap space is allocated in a compact space so as to keep the shadow space for heap compact. A corresponding shadow entry in the shadow segment is assigned to each of the memory blocks allocated. The sizes of the shadow segments can be dynamically adjusted to be large enough to hold the address-name mapping for all of the memory blocks allocated at runtime. The name of a memory object is stored in its shadow entry in the shadow segment with the same offset as that in the data segment (see Figure 1). As a result, the offset can be used in the lookup process to quickly locate the shadow entry that stores the name. Such a shadow data structure makes its modification very easy - just overwrite the

old value and no delete operation is needed. However, this method doubles the size of memory required by a program.

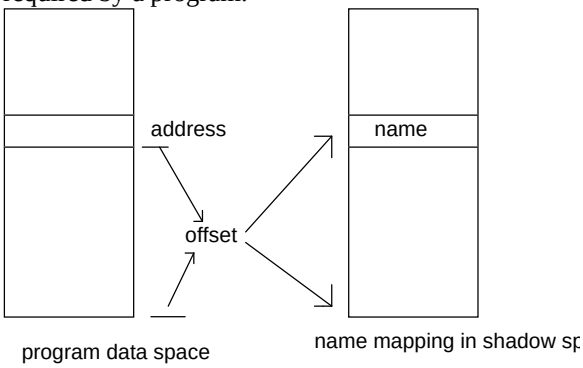


Figure 1: The shadow for naming schemes

There are several advantages using this approach. First, this tool provides a uniform platform to study the granularity of the points-to analysis. The effectiveness of different granularity levels can be compared using this framework. It is much easier to develop such a profiling tool than to implement different naming schemes and pointer analyses in a real compiler. Secondly, the precise target sets for each naming scheme can be collected at runtime. These results are roughly the best any compiler implementation can be expected to achieve. Hence, the obtained results do not depend on the quality of the implementation of these naming schemes and points-to analyses in a real compiler. This is a very significant advantage especially because the results of an inter-procedural points-to analysis are heavily dependent on how it is implemented. The third advantage is that the results of our measurements can be fed back to the ORC compiler, and we can study their actual impact on the other analyses and the optimization phases that are the clients of the points-to analysis. The fourth advantage is that we can study the potential performance improvement on a real machine, i.e. Itanium, not on a simulator.

However, such a profiling method also has its limitations. Since our results are collected during runtime, they could be input dependent and the coverage of the program limits our studies only to the parts that are actually executed at runtime. With the measurements from a suite of benchmarks and the focus of the study is not on a particular program, we believe that the results of our study can reflect the general characteristics of real applications.

Our profiling tool has two major components: an instrumentation tool developed on the Intel's ORC compiler [14], and a set of library routines written in C. Application programs are first instrumented by the modified ORC compiler to insert calls to the library routines. Then at runtime, these library routines simulate different naming schemes and collect the target sets of indirect references.

3.2. Instrumentation

The instrumentation tool in the ORC compiler inserts function calls to invoke our library routines to generate and process traces. They simulate different naming schemes for every memory object, and calculate target sets for every indirect reference. We describe some of the details in the followings:

- Procedure calls. At every entrance and exit of a procedure call in the program, a library call is inserted with the call site ID of the procedure passed as one of the parameters. The call site ID is pushed into or popped out of the calling path stack to maintain the current calling path.
- Memory objects. When a memory object becomes alive, a library call is inserted with the starting address, the length, and the name (for variables only) of the memory object passed as its parameters. The name of the variables helps us to identify which variable is actually referenced when the runtime results are fed back to the compiler. The way a name is assigned to a heap memory block is determined by the selected naming scheme. This library call sets up the mapping from the addresses of this memory object to its name by writing the name in the corresponding shadow entry. The number of entries to be written is determined by the size of this memory object. Global variables, local variables and heap memory blocks are instrumented differently:
- Global variables become alive at the beginning of a program. The mapping of global variables is initialized only once when the program starts. Scope may be an issue for global variables. Global variables are visible only in the files in which they are declared. The initialization procedure for global variables is instrumented in each file as a new procedure at the end of the file, and these procedures are invoked at the beginning of the main function. The starting address of a global variable can be accessed by the address-of operation. The length is determined by the type.
  - Local variables become alive each time the procedures in which they reside are called. The address for a local variable may not remain the same for each invocation of the procedure. Therefore, we have to insert library function calls at the beginning of each procedure to set up the address-name mapping for local variables. Variables can be ignored if their addresses are not taken. The starting address of a local variable can be accessed by the address-of operation.

- Heap memory blocks become alive when they are allocated through calls to system memory allocation functions, such as *malloc*() and *calloc*(). Library function calls are insert after these functions. The starting address is the return value of the memory allocation function, and the size of the memory blocks can be obtained from the parameters of these memory allocation functions.

- Indirect references. Each indirect memory reference is instrumented with its address and the reference ID passed as parameters to the library function call in order to collect its target set at runtime.
- Typecast. The instrumentation of type cast is needed only when we want to identify the type of a memory object. The instrumentation tool also generates a file to describe the layout of each structure type. Therefore, the heap memory blocks for data structures with fields can be sliced into smaller objects according to their fields.

3.3. Assign Names

Global and local variables already have their given names. Hence, there is no need to assign names to them. For heap memory blocks, we simulate naming schemes by using different lengths of the calling path. The calling path stack is maintained by instrumented library functions. When a heap memory block is allocated, the top *n* elements in the calling path stack are checked, if G=n.

When the fields are considered, the field ID associated with the name assigned to the memory object is written into the shadow. The instrumentation tool generates a file to describe the layout of each structure type to help break down memory objects to their fields.

For example, there is a memory object, and its name determined by the calling path is g_*name*. The memory object's starting address is addr_*start* and its size is *object_size*. If this memory object is of structure type or array of structure type, the *k*th field of this memory object will be assigned the name (g_*name, k*). Assume the offset and the size of this field are *offset* and *field_size*, and the size of the structure is *struct_size*. To set up the mapping, all address, *addr*, in this memory object will be given name (g_*name, k*), when the following two conditions hold.

1. *addr_start* ≤ *addr* < addr_start+object_size
2. *offset* ≤ *(addr-starting) mod struct_size* < *offset+ field_size*.

When references accessing different fields of this memory, the targets can be distinguished because they have different field IDs.

3.4. Collect target sets

The target of each instance of reference is collected by looking up the shadow with the address value of the reference. The target set of a reference is accumulated according to the reference ID and stored in a hash table.

The target sets computed by the tool are flow sensitive and path sensitive. Only the targets that can reach a reference at runtime are put into its target set. The previous value of a pointer is overwritten after the pointer is re-assigned. The possible targets in not-taken branches are also ignored.

If we want to make the target sets context insensitive, targets coming from different calling contexts are not distinguished and are stored together. We can also make the target sets context sensitive by attaching each target a tag to indicate its call site. However, our evaluation method requires calling context insensitive results, because it is not directly supported in the ORC compiler to generate multiple versions for different calling contexts.

4. Evaluate naming schemes

The effectiveness of naming schemes is evaluated by feeding the target sets collected at runtime back to the ORC compiler, and observing the changes in the alias queries and in the performance of the generated code. The optimizations in the ORC compiler are used as typical clients of the points-to analysis.

4.1. The ORC compiler

The Open Research Compiler, or the ORC compiler [14], originated from the Pro64 compiler [19] developed by the Silicon Graphic Inc. The ORC compiler is for C, C++ and Fortran90. It has most of the analyses and optimizations available in modern compilers. It performs pointer analyses, scalar optimizations, loop transformations, inter-procedural analyses, and code generation. Profiling and feedback-directed optimizations are also supported by this compiler.

There are three stages of analysis for each procedure: loop-nest optimizations (LNO), scalar global optimizations (WOPT), and code generation optimizations (CG). The LNO stage does loop related optimizations [23], such as parallelization, and unimodular transformations. The WOPT stage contains some general optimizations, such as partial redundancy elimination [22], copy propagation and strength reduction. The CG stage focuses on generating optimized binary code. The inter-procedural analysis is supported by the IPA component.

The pointer analysis in the ORC compiler starts from a flow-free pointer analysis, which is similar to Steengaard's algorithm [8]. This pointer analysis is done inter-procedurally when the inter-procedural analysis is turned on. A flow-sensitive pointer analysis is then applied intra-procedurally to get more precise results. Some simple rules, such as the address-taken rule, are used to help alias analysis. The alias information stored in the internal representations is maintained across different stages.

When using the ORC compiler as a base for comparison, we try to tune the compiler so that the best results could be brought about by the change of the naming scheme. The optimization level is always set at O3. The inter-procedure analysis is turned off, because the current version of the ORC compiler has unstable inter-procedural analysis which may fail in some benchmarks. Therefore the result of ORC compiler just represents the capability of a practical compiler, not a state-of-art compiler. However, the moderate pointer analysis in the ORC compiler actually makes the changes in granularity clear If the ORC had very powerful pointer analysis, it is unclear where the pointer analysis is overdone.

4.2. Feedback

The target sets of indirect references are fed back to the ORC compiler. The target sets may be different when different naming schemes are used, and thus the results of the optimizations in the compiler may be different. Two things are measured: the performance of the generated code on Itanium, and the results of alias queries within the optimization phases.

The changes in the performance on Itanium directly reflect the impact of different naming schemes in the ORC compiler. However, the performance changes are determined by many factors. In this study, we also measure the changes in the result of alias queries in the optimization phases, which somewhat reflect the subtle changes in the pointer analysis.

The major optimizations are done in the WOPT and CG stages. In order to feed back to different stages, the instrumentation is done at different stages so that the feedback information can match. The instrumentation is also done incrementally because the impact of the feedback to WOPT should be considered when the instrumentation at CG is done. The target sets collected at runtime by the profiling tool are fed back to the two stages, replacing the alias analysis result produced by the ORC compiler. In the WOPT stage, the static single assignment (SSA) form [20] is generated based on the target sets fed back from the runtime. Many optimizations in WOPT, such as partial redundant elimination and dead code elimination, are built upon the SSA form. In the CG stage, the results of alias queries are also replaced by the target sets fed back from the runtime. We instrument the ORC compiler to record the changes in alias queries.

The profiling information is limited to the portions in a program that is reached during the execution. There is no alias information for the references that are not reached at runtime. These references are conservatively assumed to be aliased with all other references.

5. Experiment Results

Experiments are conducted on the SPEC CPU2000 integer benchmarks. First, the distribution of the results of alias queries in the ORC compiler is reported. Then each benchmark is instrumented, and target information for each indirect references at different granularity levels are collected at runtime. The benchmarks are compiled again with the collected alias information. Due to the improved alias information, some alias queries which used to return may alias now return no alias. The changes of alias queries are reported again to show the impact of pointer analysis with different granularities. Finally, the compiled benchmarks are executed again to measure the impact on execution time.

5.1. Alias queries

As in typical compilers, an alias query in the ORC compiler returns one of the following three results: not alias, same location, and may alias. The first two cases are accurate results, while the third one , may alias, is conservative and could be improved by more precise pointer analyses. Since a pointer expression references either a variable or a heap memory block, the alias pairs that return may alias can be further classified into: three categories: between two variables (v-v), between a variable and a heap memory object (v-h), and between two heap memory blocks (h-h). Figure 2 shows the distribution of the returned values from the original ORC compiler. On average, the queries which return may alias accounts for 54.4% of all queries. This high percentage indicates that there could be great potential for improvements. As shown in Figure 2, the majority of the may alias queries are related to heap memory blocks. Although there are frequent v-h (variable to heap objects) type queries returning may alias, many of them should be turned into no alias by a stronger inter-procedural pointer analysis. For the rest of aliases among heap blocks, the following experiments are conducted to study the impact of granularity levels on pointer analyses.

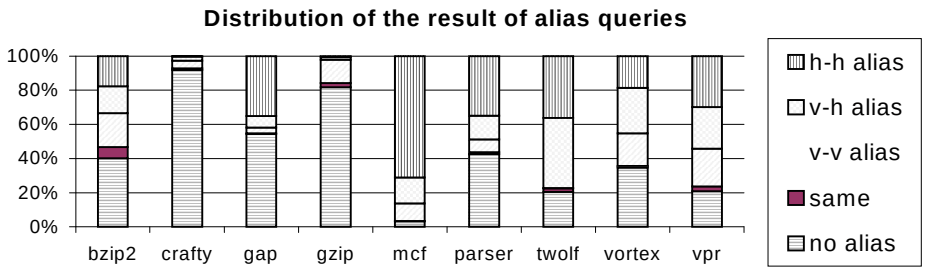
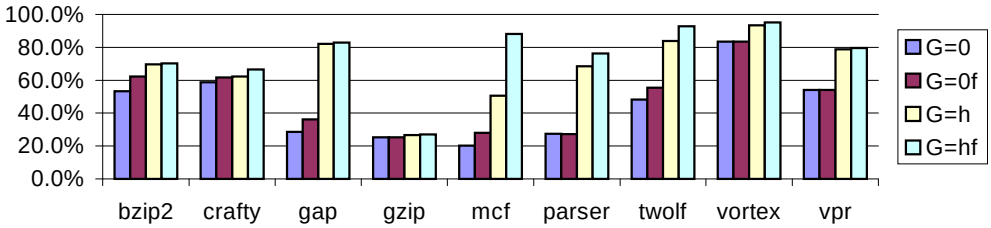


Figure 2: Distribution of the result of alias queries



G = 0: all heap memory blocks are given one name.
G=1, 2, 3: the calling path of length 1, 3 or 3 is used to name the heap memory blocks.
G=a: the whole calling path is used to name the heap memory blocks.
G=m: the user memory management function is recognized to name the heap memory blocks.

Figure 3: Percentage of no-alias queries changed with granularity

5.2. Query enhanced by feedback

After program instrumentation and runtime collection of target sets information, the benchmarks are compiled with the ORC compiler again. This time, the ORC compiler is provided with target information for pointer expressions collected from instrumented runs. Now the ORC compiler is able to give more accurate answers to alias queries. Some queries that used to return *may alias* now may return no alias. The percentage of the changes is reported in Figure 3. The queries involving un-reached references are excluded.

There are several observations based on Figure 3.

- There are more than 30% improvements even when G is 0. The reason is that the ORC compiler uses a default symbol to represent all memory objects outside of a procedure to simplify inter-procedural analysis. Such granularity is too coarse. A normal inter-procedural points-to analysis can do much better. .
- For most of the benchmarks, except for bzip2 and mcf, heap memory analysis with line number (G=1) does not improve much. However, for twolf and vpr, G=2 greatly reduces the number of may alias. Further increase of the calling path for heap pointer analysis (G=3) makes little difference.
- G=a does not bring further improvements. Therefore, there are little incentives to consider very long calling path. Some simple analyses, for example, suggested in Intel's compiler group [12], are sufficient.

5.3. User managed memory

In the benchmark gap and parser, the pointer analysis is insensitive to the naming scheme for heap memory objects. The reason is that the heap memory space is managed by programmers. Therefore, the calling path of system memory allocation does not help. If the functions in which the user manages the heap memory can be recognized, our tool can treat them like malloc(). For example, after we explicitly recognize user managed memory allocation functions, the query improvement improved drastically from 30.8 %

to 82.2% in gap, and from 29.9% to 68.4% in parser. See G=m in Figure 3.

Although the user managed memory allocation functions are very difficult, if not impossible, for compiler to recognize them. The major difficulty is to trace the size of memory space accessed through each pointer so that the no overlap can be proved. For programs with user managed memory allocation functions, speculation or dynamic optimization may be needed.

5.4. Fields of heap memory blocks

The fields can affect the pointer analysis in two ways: 1) the pointer analysis can distinguish the points-to sets of different fields that are defined as pointer type; and 2) the pointer analysis can distinguish the targets pointing to different fields of a structure. In our approach, the target sets collected at runtime have the same effects as considering fields in points-to set. Whether to consider fields in target sets is another potential variation.

It is easy to divide a structured variable into finer granularity using their type definition. However, there is no data type defined for heap memory blocks. They can be divided into finer granularity using their fields of structure type only when the memory blocks with the same name are cast to and used as the same type. The type casting of heap memory blocks are traced to identify conditions in which this analysis is applicable. The naming scheme could be based on G=1 or G=2, or G=m such that the heap memory blocks in the same group have the same type. We represented such granularity as G=hf.

The change of queries when fields are considered is reported in Figure 4. By comparing the result of G=0 and G=0f, and comparing the result of G=h and G=hf, it can be observed that it is important for pointer analysis to consider the fields of both variables and heap memory blocks.

5.5. Performance enhanced by profiling

Pointer analyses at finer granularity might significantly improve the results of alias queries. It is also interesting to know what would be the impact on the actual optimizations. In this section, the target sets collected at runtime are fed back to the WOPT and the CG phases in the ORC compiler. Optimizations in the two phases are performed with the feedback information, and thus improved results of alias queries. The performance improvement of the benchmark is shown in Figure 5. After the user memory management functions are recognized in gap and parser, the performance improvement is 20.1% and 12.3%, respectively.

The performance improvement is in proportion to the improvement to alias queries to a lower less a degree. The performance gain of an optimization may depend on many other analyses and the characteristic of the code. Therefore, the improvements of alias queries may not always contribute to overall performance. Half of the benchmarks achieved more than 10% of improvement in performance with finer granularity.

6. Conclusions

We conduct a comprehensive study on the naming schemes and the granularity of the pointer analysis. We implement a set of instrumentation and profiling tools to study issues related to pointer analysis. Each benchmark is instrumented with our tool to collect target sets information at runtime. Such target sets information is fed back into the ORC compiler automatically to help later analyses and optimizations. This approach provides a direct way to study the impact of naming schemes and granularity on performance.

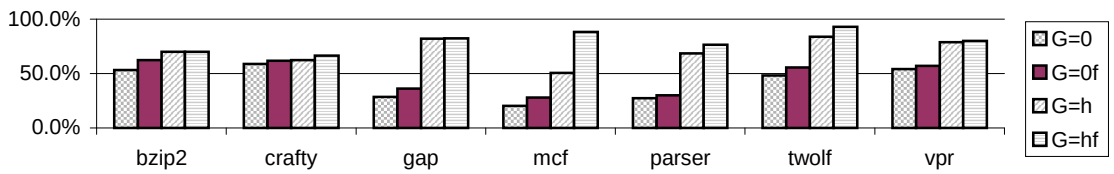


Figure 4: Percentage of no-alias queries changed with field granularity

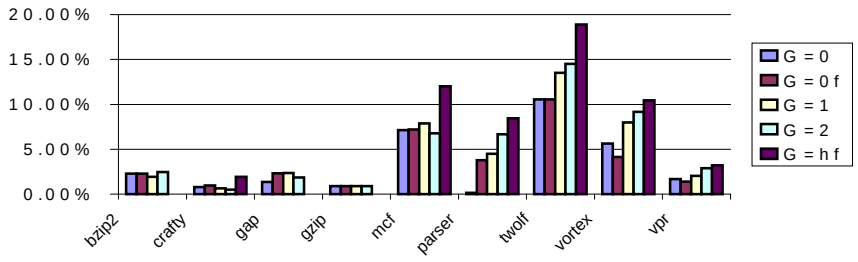


Figure 5: Performance improvement for different granularity levels

Our experiment results suggest that pointer analysis for heap memory blocks may yield a good return. The commonly used naming scheme that names memory objects with the statement line number of the malloc() function call improves only slightly over the approach that treats heap memory blocks as one entity. However, naming such dynamic allocated memory objects with respective calling path contributes more. Some programs have their own dynamic memory allocation and management routines. It is important for the compiler to recognize such routines to enable more effective naming schemes.

By simulating naming schemes with calling path and field information, the point-to information provided to the ORC compiler greatly improves the results of alias queries. The improved results from alias queries in turn significantly increase the effectiveness of compiler optimizations. Since the point-to information fed back to the compiler is collected at runtime, this approach may not be used directly to generate real code. However, it provides a useful guideline to the potential of pointer analyses at finer granularity.

References

[1] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In Proceedings of SIGPLAN'90 Conference on Programming Language Design and Implementation, page 296-310, June 1990.

[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] 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

[4] Choi, M. Burke, and P. Carini. 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.

[5] 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.

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

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

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

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

[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, Craig 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 Research Compiler for the Itanium Family. Tutorial at the 34th Annual International Symposium on Microarchitecture.

[15] N. D. Jones and S. S. Muchnick,. A Flexible Approach to Interprocedural Flow Analysis and Programs with Recursive Data Structures. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1982.

[16] S. Zhang, B. G. Ryder, and W. Landi. *Program decomposition for pointer aliasing: A step towards practical analyses*. In Proceedings of the 4th Symposium on the Foundations of Software Engineering, October 1996.

[17] Tong Chen, Jin Lin, Wei-Chung Hsu and Pen-Chung Yew, On the Impact of Naming Methods for Heap-Oriented Pointers in C Programs, International Symposium on Parallel Architectures, Algorithms, and Networks, 2002.

[18] Spec CPU2000, http://www.specbench.org/osg/cpu2000/.

[19] G. R. Gao, J. N. Amaral, J. Dehnert, and R. Towle. The SGI Pro64 compiler infrastructure: A tutorial. Tutorial presented at the International Conference on Parallel Architecture and Compilation Techniques, October 2000.

[20] Fred Chow, Raymond Lo, Shin-Ming Liu, Sun Chan, and Mark Streich, Effective Representation of Aliases and Indirect Memory Operations in SSA Form, Proc. of 6th Int' l Conf. on Compiler Construction, pp. 253-257, April 1996.

[21] Shapiro, M., and Horwitz, S., The effects of the precision of pointer analysis. Static Analysis 4th International Symposium, SAS ' 97, Lecture Notes in Computer Science Vol 1302, September 1997.

[22] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu. Anew algorithm for partial redundancy elimination based on SSA form. In Proc. of SIGPLAN 97 Conference on Programming Language Design and Implementation, page 273-286, May 1997.

[23] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen, Combining Loop Transformations Considering Caches and Scheduling, Int' l J. of Parallel Programming 26(4), page 479-503, August 1998.

[24] Amer Diwan, Kathryn S. McKinley, J. Eliot and B. Moss, Type-Based Alias Analysis**,** SIGPLAN Conference on Programming Language Design and Implementation, pages 106--117, June 1998

[25] Yong SH, Horwitz S, Reps T. Pointer Analysis for Programs with Structures and Casting. SIGPLAN Conference on Programming Language Design and Implementation, vol 34, pages 91-103, 1999-

[26] Erik Ruf. Context-Insensitive Alias Analysis Reconsidered. In ACM SIGPLAN ' 95 Conference on Programming Language Design and Implementation (PLDI' 95), La Jolla, California, vol 30, pages 13-22, June 1995.

[27] D. Choi, M. G. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer induced aliases and side effects. In Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages, pages 232--245, January 1993.

[28] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher, A Scheme for Interprocedural Modification Side-Effect Analysis with Pointer Aliasing, ACM Transactions on Programming Languages and Systems (TOPLAS), 23(2), March 2001, pages 105--186.

[29] Michael Hind and Anthony Pioli, An Empirical Comparison of Interprocedural Pointer Alias Analyses. IBM Report #21058, *December 1997.*