McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Fast Points-to and Side-effect Analysis for the McCAT C Compiler

Guirlyn Olivar

ACAPS Technical Memo **
April, 1997

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7
FTP: ftp-acaps.cs.mcgill.ca • WWW: http://www-acaps.cs.mcgill.ca/

Abstract

Pointer and side-effect analyses are important components of optimizing and parallelizing C compilers. Good pointer analysis is useful for many subsequent analyses and transformations including reaching definition analysis, constant propagation, and dependence testing.

In general, there is a tradeoff between the accuracy and computation time for these analyses. The McCAT (McGill Compiler/Architecture Testbed) C compiler currently supports an accurate context-sensitive points-to analysis and an associated context-sensitive side-effect analysis (also called read/write sets analysis)[3]. This analysis provides good results, but can be computationally expensive for large programs.

The purpose of this project is to implement a fast points-to analysis based on the type inference algorithm presented by Steensgaard [10]. Based on this algorithm, a new, fast, side-effect (read/write) analysis is developed and implemented.

Experimental results are provided that compare the accuracy and computation time for the context-sensitive and fast analyses. As expected, we find that the fast version exhibits linear-time behavior, and computes points-to results significantly faster than the context-sensitive analysis. The results show that the fast version is quite adequate in many cases, but there are also cases where the fast version does incur significant inaccuracy.

Contents

List of Figures

List of Tables

1 Introduction

Fast Points-to Analysis is an interprocedural flow-insensitive analysis that follows the techniques previously defined by Bjarne Steensgaard's type inference algorithm [10].

The most attractive feature of this algorithm is its complexity time. This algorithm has shown to behave with almost-linear time complexity. This almost-linear complexity is highly appreciated when considering the processing cost of large programs with context-sensitive points-to analysis.

The flow insensitiveness of fast points-to analysis will generally lead to conservative assumptions. This principle forces the fast points-to analysis to generate less accurate results compared to the results obtained with a context-sensitive/flow-sensitive points-to analysis.

1.1 Background and Motivation

Alias analysis [1] is a well known process of estimating when two variables references refer to the same memory location. In C, the aliasing problem gains more complexity than just resolving aliased variable names. The existence of "pointers" supported by the operators `&` and `*` introduces new ways of accessing memory locations. We are interested in this more difficult problem which is often solved by estimating points-to relationships [4, 8, 11, 7]. The *points-to information* serves as the source of knowledge that can be used by subsequent analysis in order to resolve the ambiguities created by pointer accessed memory locations in a program. Some of these analysis are: live variable analysis, data dependence analysis, common subexpression elimination, among others.

A clear example of the benefit of using points-to information is on the analysis of obtaining read/write sets. In figure 1 we show an example program where the availability of the points-to information helps to build the program's read and write sets. In this case we can see how the presence of points-to information for statement `S3` makes it possible for read/writes set analysis to determine that variable `a` should be in the read set.

```
int *p, a, b;      Points-to inf.           Read - Write Sets

S1: a = 1;           {}                Read = {}      Write = {a}
S2: p = &a;        {p->a}              Read = {a}     Write = {p}
S3: b = *p;        {p->a}              Read = {p, a}  Write = {b}
```

Figure 1: Importance of Points-to Information on Read/Write Sets

Another use, not less important, is in the construction for the call graph of a program, where function pointers are very likely to introduce uncertain or unknown function calls. These function calls can be properly resolved with the knowledge provided by the points-to information.

1.2 McCAT-The Compiler Testbed

The McGill Compiler Architecture Testbed, **McCAT**[5] offers a convenient framework on which the flow-insensitive points-to analysis can be developed and adapted. The McCAT compiler translates a C program into a high-level representation referred to as FIRST, and with a series of tree transformations creates SIMPLE-AST, a simplified Abstract Syntax Tree.

The SIMPLE-AST[6] internal representation represents complex statements with a series of basic statements maintaining the high-level variable references and type casting information. Name resolution is also accomplished by this representation through the association of unique declare nodes for each variable in a program.

The name resolution is an important feature since, regarding the variable name, we always want to obtain the correct variable reference and the SIMPLE representation provides the way of correctly refer to a particular variable by distinguishing among the variable declarations.

The McCAT compiler is already equipped with an implementation for context-sensitive points-to analysis. This work is presented in [4] and fully described in [3]. Our addition to McCAT offers an alternative, less expensive, mechanism for generating points-to information. As an alternative technique for performing points-to analysis, we extend the features of the McCAT compiler by providing a flow-insensitive interprocedural points-to analysis.

Fast Points-To Analysis, our implementation of the flow-insensitive interprocedural points-to analysis, is an adoption of Steensgaard's type inference algorithm [10]. This points-to analysis is based on the construction of a storage model of a program that is obtained with the application of a set of rules on basic statements. As we will see, a storage model is built aside the SIMPLE-AST by associating an abstract location to each variable declare node, also there is a λ-definition associated to each function declare node of a program which offers support for the information regarding function pointers. This storage model can be accessed independently from the SIMPLE-AST or through an abstract location of a particular declare node.

1.3 The General Process

The general process involved in achieving fast points-to analysis can be described by four steps:

1. **Create the Initial Storage Model**. One pass throughout the program definition is enough to create the initial storage model. This pass initializes all declare nodes to have distinct abstract memory locations.

2. **Calculate the Final Storage Model**. Each statement is visited only once, applying one or more inference rules that modify the storage model's original shape.

3. **Create the Points-To Sets**. Retrieve from the storage model the points-to sets. By accessing the storage model we are able to associate short-cuts that creates appropriate points-to sets for each pointer variable. This step mainly creates the points-to groups that are generated by the equivalence classes.

4. **Retrieve information from the points-to sets**. This step is used to retrieve the points-to information of pointer variables. This becomes the relevant step for further analyses.

Although McCAT has implemented a robust read/write sets analysis with embedded points-to information obtained by the context-sensitive points-to analysis; a separate fast read/write sets analysis has been implemented to use the points-to information generated by the fast points-to analysis.

The availability of these two read/write sets analyses provide a convenient background for obtaining comparative results between using the context-sensitive versus the flow-insensitive points-to versions.

For the purpose of clarity, we will use the name fast read/write sets analysis to distinguish the read/write sets analysis that uses the points-to information generated by the fast points-to analysis.

The overall process of performing the fast read/write sets analysis can be globally viewed as a process of performing two tasks. The first task corresponds to the intraprocedural read/write sets analysis and the second task is referred as the interprocedural read/write sets analysis. The former task will deal with retrieving the program information available at the level of a procedure body. The latter task is in charge of updating the read/write sets with contextual information.

The terminology maintained throughout the remaining of this report will be the following:

FP for fast points-to;

CSP for context sensitive points-to;

FRWsets for fast read/write sets;

RWsets for read/write sets.

This report is organized into three more sections. Section 2 describes the design and implementation issues involved in achieving McCAT's fast points-to analysis. Section 3 concentrates on the implementation of fast read/write sets analysis. Section 4 presents a discussion on experimental results obtained with both versions of points-to and read/write sets analyses, using a suite of benchmarks. Section 5 gives a brief description of related work, and section 6 presents conclusions and future work.

2 Fast Points-To Analysis

Our analysis works as a mechanism that tracks the data flow of pointer values using a conservative storage model. This storage model describes all possible dynamic storage configurations of a program, which is, how storage is used by a program at runtime. This analysis is based on a set of typing rules that derives from the correctness of basic statements on well-typed programs. An abstract location is associated with each memory location of the program and is used to reveal if a possible content for that memory location can be a pointer value.

2.1 Equivalence Class

Each variable is known to have a name (through which we access it) and a content (better known as the value of the variable). The content of a variable is stored and made accessible through a specific memory location. We identify each accessible memory location as a distinct abstract location.

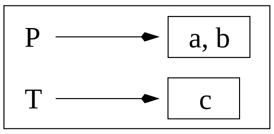
The content of a memory location may be a constant value or a pointer to another memory location. We are interested in resolving the content of the memory location as a pointer by using equivalence classes.

When the value held as the content of a variable is a memory address, we say that the variable is a pointer, and we associate the corresponding link between the variable and the referenced abstract location.

A pointer may hold different references depending on some runtime conditions. The conservative approach is to group these references to belong to the same class, called an *equivalence class*.

In figure 2, variables `a` and `b` can be pointed-to by the variable `P`. In this case we say that the abstract memory locations for the two variables are in the same equivalence class. Variable `c` belongs to a different equivalence class which is referenced by the variable `T`.

```
if (condition)
   P = & b;
else
   P = & a;
T = &c;
```



(a) Program (b) Storage Model

Figure 2: Equivalence Classes in the Storage Model

2.2 Data Structure

A *declare node* is an internal representation of a program's variable. We introduce the term ECR to refer to an abstract location, this term stands for an Equivalence Class Representative. For each declare node we assign a first-level ECR representing the memory location of the variable given by that declare node. But we also assign a couple of second-level ECR'S, these are:

- REF ECR

  that represents the possible memory location to which the variable given by the declare node points to.

- LAM ECR

  which represents a possible function prototype in the case of the variable being a function pointer.

The storage model indicates a reference assignment by associating the appropriate abstract locations. Consider, figure 3 where the first declare node represents a program variable `x`. The first-level ECR is intended to represent the memory location for `x` and the second-level ECR would represent in our storage model the memory location of `*x`.

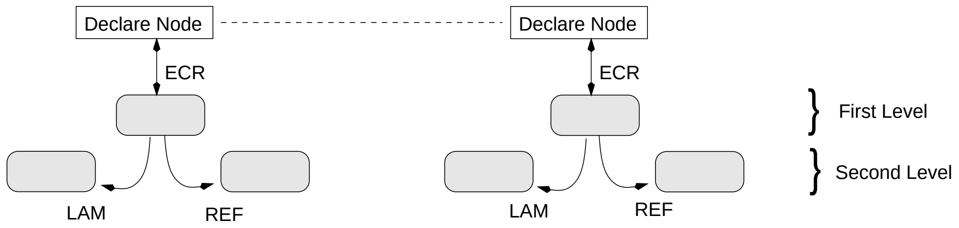


Figure 3: Assignment of an Abstract Location (ECR)

The analysis basically consists of joining the ECR'S related to variables that may point to the same memory location during the runtime of the program. The ECR'S are joined following the typing rules of selected statements, this ensures the well-typedness of the program.

Each program statement is visited once and if it is a selected statement the visit will contribute with the storage model by producing the appropriate joining.

An internal look at an abstract location is revealed by figure 4.

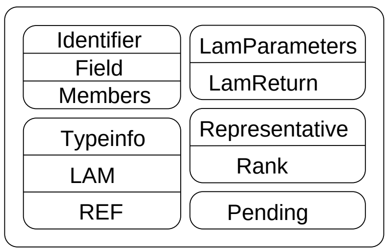


Figure 4: The internal structure of an Abstract Location (ECR)

The points-to information of a variable is mainly given by the location pointer component REF and the function signature component LAM fields. Other fields such as members and field are also important for retrieving the points to information. The first-level ECR'S have a non-empty value for field and are used to resolve second-level ECR'S when joining them.

2.2.1 Join Operation

Joining two ECR's is the basic operation of the analysis, we ensure that the ECR data structure is equipped with fast union/find operations. The data structure on which an ECR is based is known as *Disjoint Sets* [2]. The join operation is illustrated in figure 5.

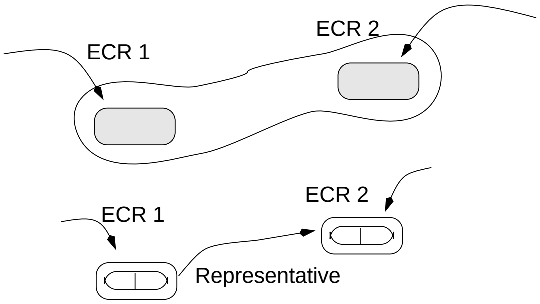


Figure 5: Fast Union/Find Operations

The union operation applied on the ECR's has asymptotically constant time. This is easily achieved through the update of a pointer. In our case each ECR is initially a singleton set, its **representative** is itself. When two sets are joined through the union operation, the **representative** of both sets will become the same. The selection of the new **representative** is determined by the **rank** of the sets involved at the time of the union.

2.2.2 Conditional Joins

A special type of join operation is required to treat unknown references, this type of join operation is referred to as *conditional joins*. A conditional join is used whenever it is required to update the storage model with an indirect reference related information. It may occur, that while updating the storage model a join operation is required on an abstract location that refers to an unresolved reference. As the reference is currently undefined, the join operation is recorded to keep track of the intended update. For this purpose, the conditional join introduces the use of *pending lists*. A pending list holds information that can not yet be resolved due to unknown information. A pending list will be resolved once the proper resolution to an unknown reference is made by subsequent updates to the storage model.

2.3 Algorithm

The fast points-to analysis is based on the storage model of the program. From the storage model we are able to extract information such as: a variable is a pointer, or two different variables may share the same memory content sometime during runtime, or a function body can be referenced by dereferencing a particular variable.

The storage model acts like a story book that keeps track of what the program does with its variables. By concentrating only on a small set of particular statements we are able to update the storage model to reflect the consequences created by these particular basic statements.

We begin our analysis by creating the initial storage model of the program. We start by assuming that all the variables of the program have a different abstract location set. So, we associate each variable with an independent equivalence class. Then we proceed to merge abstract location sets as necessary following the well-typedness rules of the program[10]. The task of performing a points-to analysis has now been reduced to the task of inferring the minimal solution for the storage model under which a program is well-typed.

Merging two abstract location sets is made by using a fast union/find data structure. Joining two abstract location sets will never make a statement that was well-typed be no longer well-typed. As merging abstract location sets is done only when necessary to ensure the well-typedness of the program, the final solution will be the minimal solution for our storage model.

The type inference rules derive from a set of statements that directly affect the shape of the storage model. We begin by identifying a set of basic rules, each of which is associated with one of the following simple statements:

- Address assignment: `x = &y`
- Simple assignment: `x = y`
- Assignment with indirect reference: `x = *y`
- Indirect reference assignment: `*x = y`
- Dynamic memory allocation: `x = alloc(size)`
- Assignment with type casting: `x = (cast)y`
- Function call on an assignment: `x = f(args)`

Other SIMPLE statements also affect the storage model in a direct way, but their effect can be easily modeled by a composition made of two of the previous SIMPLE statements and a temporary variable. We describe the modeling of each of these special statements in table 1.

SIMPLE Basic Statement	Modeled Statement
`*x = &y`	`temp = &y` and `*x = temp`
`*x = *y`	`temp = *y` and `*x = temp`
`*x = (cast)y`	`temp = (cast)y` and `*x = temp`
`*x = f(args)`	`temp = f(args)` and `*x = temp`

Table 1: Extended Rules

2.3.1 Basic Rules

There are certain statements in the program that contribute to shape the storage model. In general, these statements will collapse memory locations into the same equivalence class or will add more memory locations(i.e. via dynamic allocation).

There are four basic statements that contribute to the storage model by introducing well typing constraints that impose merging some abstract location sets. After processing each statement only once, we obtain the storage model that indicates the distribution of the memory locations.

Rule 1. Statement X = &Y: When a statement of the form `X = &Y` is identified, the rule to follow indicates that the memory location of `*X` and the memory location for `Y` must be the same. For this reason we produce a join operation with `REF(ECR(X))` and `ECR(Y)`. Where `ECR(Y)` is the first-level ECR representing the memory location of the variable `Y` and `REF(ECR(X))` is the second-level ECR obtained by the REF pointer of the ECR of the variable `X`; it represents the memory location for `*X`. Rule 1 is illustrated in figure 6.

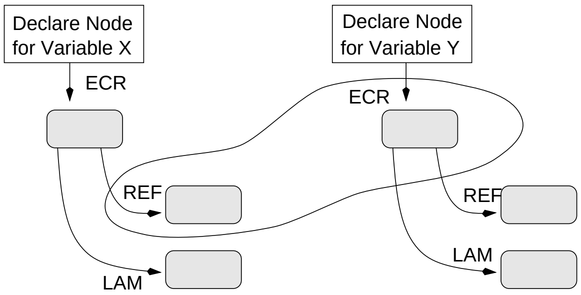


Figure 6: Basic Rule 1

Once the join operation is produced, the ECR'S involved become part of the same abstract location set. Internally, the join operation updated the **representative** pointer of `REF(ECR(X))` to be `ECR(Y)`. Also, we can say that we resolved the second-level ECR given by `REF(ECR(X))` by joining it with the first-level ECR, `ECR(Y)`.

Rule 2. Statement X = Y: If a location pointer value, say P_v, may be assigned to `X` when executing the statement then the location component of both `ECR(X)` and `ECR(Y)` must be the location pointer value, this is, P_v. If a function pointer value may be assigned, then both `ECR(X)` and `ECR(Y)` must have the same function prototype component. Rule 2 is illustrated in figure 7.

The first-level of ECR'S are not affected, it is their value represented by the second-level of ECR'S that change. But, this change is subject to the previous resolution of the ECR'S of the right-hand side of the statement. The right-hand side pointer value that is given by its second-level ECR'S may or may not be resolved. Therefore, a conditional join is

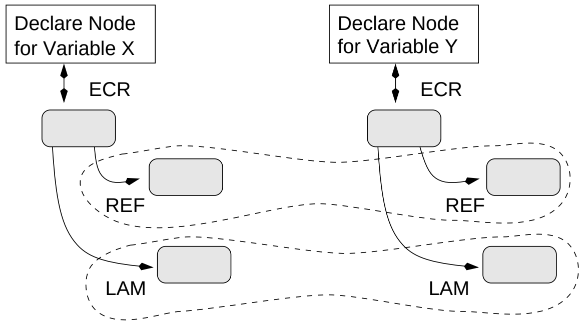


Figure 7: Basic Rule 2

performed instead of a direct join, to ensure that the contribution of this statement to the storage model will be achieved.

Rule 3. Statement X = *Y**:** This analysis is flow insensitive which in other words suggests that the result obtained must always be the same, regardless of the processing order of statements.

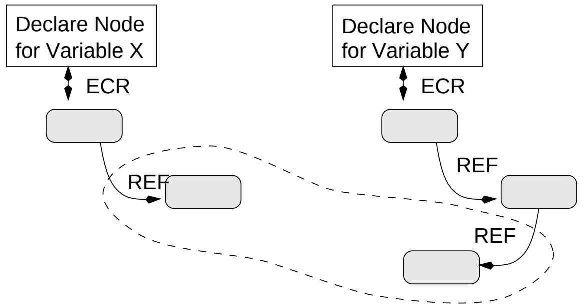


Figure 8: Basic Rule 3 with non-empty indirect reference

In this statement, we are accessing an indirect reference through the variable `Y`. This leads us to distinguish two different scenarios that may happen at the moment of processing this statement. One, where the value of the indirect reference is known to be a location pointer at the time of processing this statement; this case of rule 3 is illustrated by the figure 8. And two, the indirect reference is yet to be defined as a location pointer, or simply its value is unknown; this case of rule 3 is illustrated by figure 9.

If at the moment of execution of the statement `X = *Y` the value for `*Y` is known to be a location pointer, then we must update our storage model to indicate that the memory location of `*X` and the memory location of `**Y` must be the same (must belong to the same abstract location set).

In other words, the value hold in the memory location of `X` and of `*Y` is the same, what means that they point to the same memory location.

In this case, the second-level ECR `REF(ECR(Y))` must have already been resolved, such that its **representative** has a non-empty REF pointer, say REF(REF(ECR(Y))), which must be merged with REF(ECR(X)), by a conditional join.

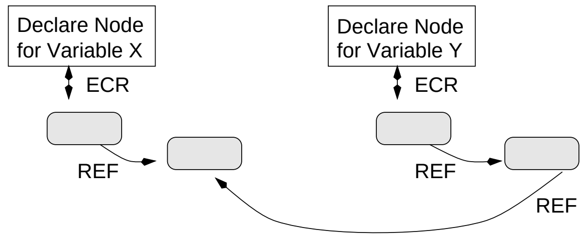


Figure 9: Basic Rule 3 with manual setting

It is possible to come up to a statement of the form `X = *Y` where the indirect reference given through the variable Y may not be defined at the moment of processing the statement. In this case, we have no other alternative than to manually set the REF pointer that represents the memory location for **Y to be the ECR representing *X. By this, we force the abstract location set for *X and **Y to be the same.

This manual setting implies a third-level of ECR by connecting through a REF pointer one second-level ECR to another. But, this third-level ECR notion is really a mechanism for the second-level ECR to keep track of the contribution that this statement does to the storage model.

Rule 4. Statement *X = Y: Again, in this statement, we must deal with an indirect reference. This time, the indirect reference is given through the left-hand side of the statement, which is the updated value. This rule is also treated under two possible scenarios.

Although the principle still remains, which is, that the content of *X and Y must be the same. In the case that this value refers to a location pointer, then the memory location for **X and *Y must be the same, this is, belong to the same abstract location set.

A first case applies when the value for *X is known to be a location pointer at the time of processing this statement, we proceed similarly as before, producing a conditional join with the second-level REF ECR of Y and the indirect reference made through *X. This case is illustrated by figure 10

The merge done by the conditional join will make REF(REF(ECR(X))) and REF(ECR(Y)) belong to the same abstract location set; in our storage model, this means that these are both names to the same memory location.

The second case for this rule requires a different handling which involves a manual setting. Figure 11 illustrates this case involving unknown references. If we are not able to access REF(REF(ECR(X))) we must manually set the REF pointer of REF(ECR(X)) to be REF(ECR(Y)).

This manual setting implies that the memory location obtained through **X is the same as the one obtained through *Y; this is, the pointer value (P_v) is at the same time hold

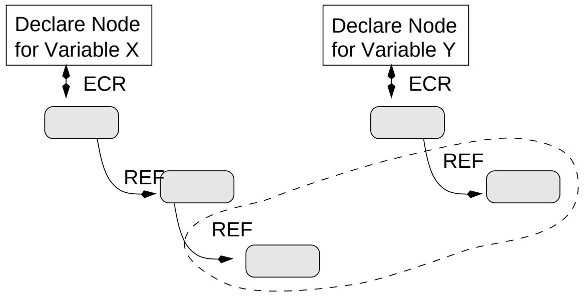


Figure 10: Basic Rule 4 with non-empty indirect reference

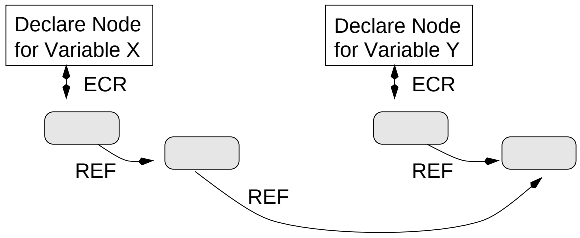


Figure 11: Basic Rule 4 with manual setting

by the indirect reference `*X` and the variable `Y`.

2.3.2 Compound Variables

Our basic rules will always assume variables with whole memory locations, that is, no identifiable partition within the variable memory location. For this reason, no field or offset information is held by our data structures. This assumption forces us to always maintain the most conservative principles when handling variables that are structures or arrays. We represent the effect that these variables produce on our storage model in table 2.

Statement with compound variable	Modeled Statement	Considered Rule
`a[i] = b`	`*(a+i) = b`	`*a = b`
`a->f= b`	`*(a+offset(f)) = b`	`*a = b`
`b = a[i]`	`b = *(a+i)`	`b = *a`
`b = a->f`	`b = *(a+offset(f))`	`b = *a`

Table 2: Compound Variables

In terms of pointers we can handle each of the statements presented as their corresponding modeled statement, this helps us to simplify our analysis for compound statements and apply one of our basic rules.

For practical reasons, the ECR of a variable is able to store certain information relative to statements with compound structures which is useful for further tasks. Although we are not able to identify a variable's partition from another, we can remember whenever a variable has been set to point to a field of another variable.

We maintain a descriptor for the location type of a pointer variable. Given a particular variable `X`, the descriptor will provide use with the following information:

- if the location type of `X` is ECR_IS_FIELD_LOCATION then `X` is a pointer to part of a composed variable(i.e. `X = &a.f` or `X = a[i]`).
- if the location type of `X` is ECR_IS_FULL_LOCATION then `X` is a pointer to a whole variable and not just part of it.

2.3.3 Dynamic Allocation

When memory is dynamically allocated in the program we must associate the storage model with a representation for the heap. We introduce a special abstract location to represent the program's heap memory allocation. We can manage the heap in two different fashions, providing our fast points-to analysis with two different flavors.

Many Heap Representation: One way to handle dynamic allocation is by creating a new and different abstract location set at each static allocation statement. In this way we will be representing the program's heap by a set of abstract locations in our storage model. To achieve this representation, we create an internal name to tag the memory location that was newly allocated on the heap.

One Heap Representation: If we are not interested in keeping track of the heap very thoroughly. We can always let the abstract location that represents the heap be the same at each static allocation statement. In this case we only need to maintain a unique representation for the program's heap, and every time we need to associate a variable to the heap we will refer to the same heap location.

Independently from the two ways of managing the heap, the action taken when processing a statement with dynamic allocation is similar to the one taken by the rule number 1. But with the addition of associating the heap to the abstract location determined by one of the resolved representation.

For example, consider that we use the many-heap representation, and we are processing the statement `X = malloc(SIZE);` then the contribution to the storage model is the one shown in figure 12. In this case, `Heap` is the internal name created to identify the newly-allocated memory location in the heap.

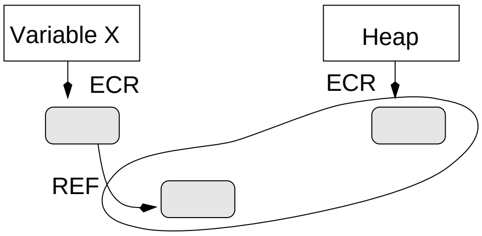


Figure 12: Dynamic Allocation on the Storage Model

2.3.4 Pointer Operations

A variable can be assigned to hold the value of a location pointer through a statement involving simple arithmetic. The analysis of points to information must be able to detect all possible flow of pointer values among variables. Therefore, the analysis takes action when detecting statements of the form:

- `X = Y + Constant`
- `X = Y - Constant`

where `X` and `Y` are variables, possibly holding pointer information and `Constant` is a constant value. The contribution to the storage model by statements of this form is the same as the one obtained when processing statements of the form `X = Y` as seen before in the description for Rule 2.

2.3.5 Function Pointers

Function Pointers are specifically handled through the second-level ECR'S referenced by LAM pointers. This *lambda* ECR describes a function prototype or signature by representing the abstract location for each formal parameter of the function as well as the abstract location of it's return value.

The function pointers contribute to the storage model through two types of statements. Statements where the definition of the prototype of a function is issued and statements where there is a function call.

Function Prototype: A first constraint dealing with lambda ECR'S is applied when a function declaration is discovered in the program. The appropriate lambda fields are filled up with the function declaration characteristics. Figure 13 reveals the proper lambda fields in the storage model related to a sample function.

We can identify the function name by a declare node and associate a first-level ECR indicating the memory location for the name of the function.

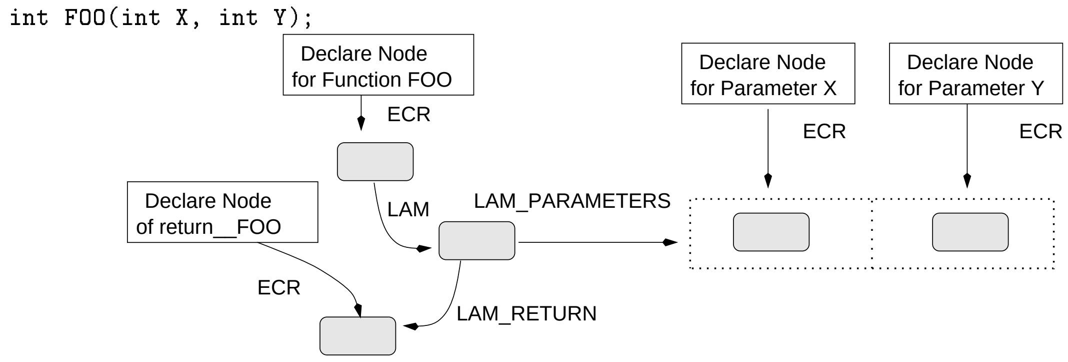


Figure 13: A Function Prototype in the Storage Model

The second-level ECR accessed through the LAM pointer will be filled up to include an abstract location set for each of the formal parameters and also will it maintain an abstract location set for the return value of the function. This last one, will depend on a symbolic variable created for the purpose of identifying the memory location of the function return value.

Function Call: A second constraint will apply on statements where a function call is issued. In general terms, this constraint will impose a merge between formal and actual parameters.

In the case where the function return value is assigned to a variable, it will produce a merge between the return value maintained in the function's lambda and the variable of the left-hand side of the statement. We update the storage model by merging formal and actual parameters. The effect of this merge is to indicate that the memory location that is referenced by a formal parameter is the same as the memory location that is referenced by the corresponding actual parameter. The merge will affect the second-level ECR'S of both formal and actual parameters. By performing this merge the formal parameters will adopt the memory references of the corresponding actual parameters. Figure 14 illustrates the join operations taking place at a function call statement.

In the case of an expression statement with a function call on the right-hand side, where a non-empty return value is stored; we produce a merge between the second-level ECR of the left-hand side variable and the second-level ECR of the symbolic return variable that is maintained by the function's lambda ECR, in order to keep track of any possible flow of pointer values.

Return Statements: A function in C can always specify a return value. We need to maintain record of what may happen during this process, this is, be aware of any pointer information that might require an update to our storage model. This is handled through an extra rule that verifies the return statements of each function and link the proper abstract locations.

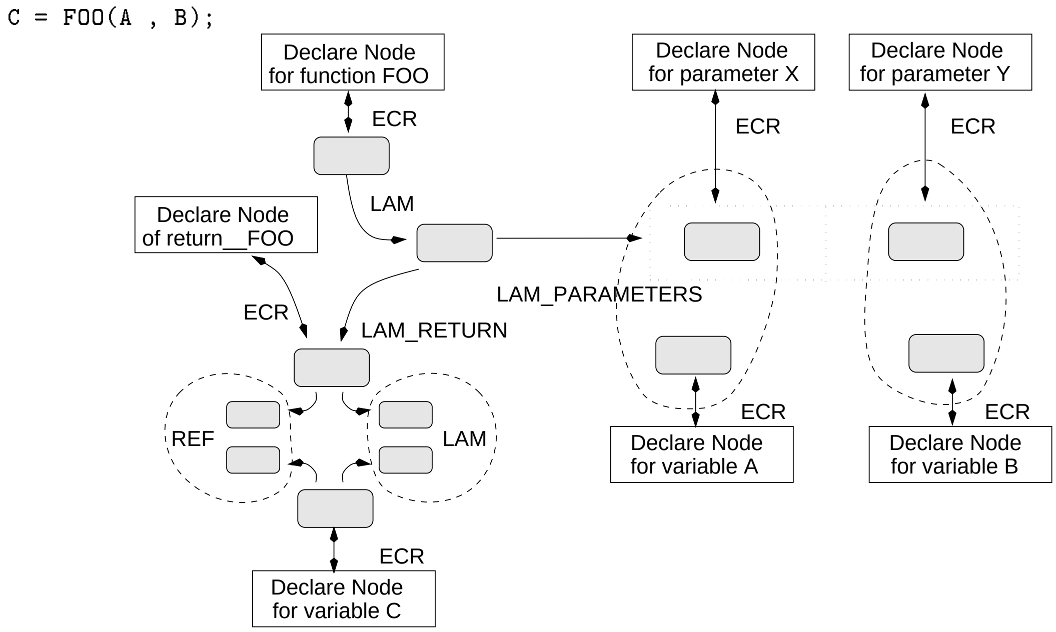


Figure 14: A Function Call in the Storage Model

All return statements are considered for the application of this extra rule. This extra rule will apply the join operations on each of the proper components involved in a return statement. Consider for example the program in figure 15. In order to resolve the indirect reference at statement `S5` we need to have points-to information for the variable `q`. At statement `S3` we apply our return statement rule. This rule produces a handle to the underlying indirect reference in our storage model. The abstract location that is associated to the return value of the function `FOO` is the ECR for `return_FOO` which is obtained through `FOO`'s prototype or `LAM` pointer.

In this example there are two steps involved:

1. `return_FOO = p`.
   Apply rule 2 to the ECR's of the variable `p` and the `return_FOO`, where `return_FOO` is the symbolic return variable associated with `FOO`.
2. `a = q`.
   Apply rule 2 on the variables `a` and `q` to link the proper `REF` pointer of `a` so that it points to `global`

A function pointer: Now that we have seen how the storage model deals with a function prototype and a function call, it is also important to understand how a variable that acts like a function pointer is supported in our storage model.

Consider, for example, the program example in figure 16. Given a function `FOO` in the program, whenever a variable `V` gets updated with the address of the function declare

```
                                              Points-To Information
   int global;                                ---------------------
   int *FOO()
    {
            int *p;
S1:         p = &global;                     { p-> global }
S2:         *p = 5;                          { p-> global }
S3:          return p;                       { return_FOO-> global }
    }
   main()
    {
            int *q, *a;
S4:         q = FOO();                       { q-> global }
S5:         a = q;                           { q-> global, a-> global }
    }
```

Figure 15: Program example for the Return Statement Rule

node of FOO, we have created a new access to FOO. But also we have updated the storage model of the program, since V can be used to access the function body of F. The storage model needs to reflect this ability by connecting the correct links among the abstract locations. In figure 16 we also show how a function pointer will look like in the storage model.

In general, we are able to detect if a variable is a function pointer by checking the equivalence class in which its REF pointer belongs and searching for non-empty lambda values (LAM_RETURN and LAM_PARAMETERS). In our example of figure 16 we can see how REF(V) and ECR(FOO) belongs to the same equivalence class, therefore, as ECR(FOO) has non-empty lambda values, so will REF(V) making the variable V a function pointer.

3 Fast Read/Write Set Analysis

In order to use the benefits of our fast points-to analysis, we extended McCAT with an implementation of the read/write sets analysis that directly makes use of the points-to information generated by fast points-to. This implementation will generate information in the same format as does the pre-existing read/write sets analysis that uses the points-to information produced by the context-sensitive points-to analysis. Therefore, further optimization tasks can use the read/write sets information regardless of the points-to analysis used to provide points-to information. The read/write sets analysis is a data flow analysis that gathers the following information for each of the program statements:

```
main()                      int FOO(int X, int Y)
{                                 {
   int a, b;                        ....
   int (*V)(int, int);            }

   V = &FOO;
   V(a,b);
}
```

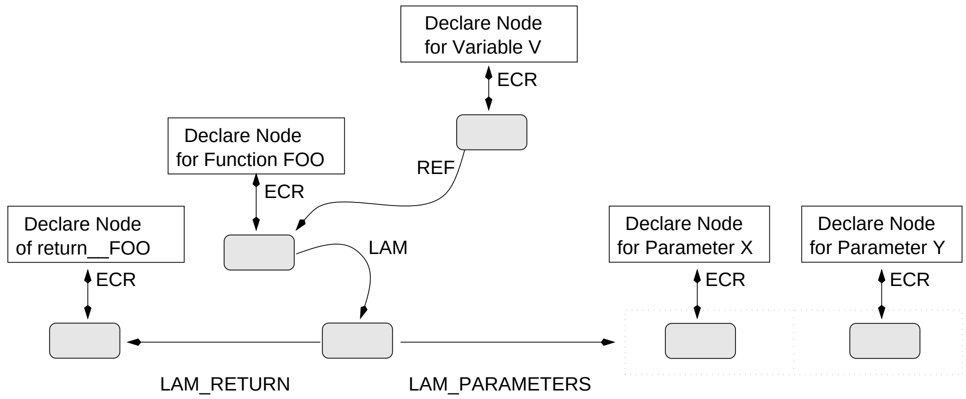


Figure 16: Example of a Function Pointer

Read Set: This set collects the variables that are *definitely read* in a statement. Generally this set includes the variables present on the left-hand side of an expression statement.

Write Set: This set collects the variables that are *definitely written* in the statement. Generally this set includes the right-hand side variables belonging to an expression statement.

May Read Set: This set collects the variables that are *possibly read* in a statement.

May Write Set: This set collects the variables that are *possibly written* in a statement.

The read/write sets analysis can be seen as a process that is applied to a program, at two different levels. A first level will concern individual statements inside a function. A second level of application deals with the contextual information that is generated through function calls. These two levels of application are described as follows:

- **Intraprocedural Read/Write Sets**

  For each function of the program, the read/write set analysis collects the data flow information without accounting for any contextual information that may exist among different functions. This is, it will ignore any effect that a function call may produce in the current collection of read/write sets. This way, we only concentrate on the read/write sets of statements inside a function body and ignore the effect of function calls.

- **Interprocedural Read Write Sets**

  Once we have collected the information for each of the function bodies we proceed to include the contextual information. This is, the side effect that a function call may have into the read/write sets of the invocation statements. This contextual information is obtained by following the order in which function calls are produced in the program. This order will lead to create a hierarchy of the functions in the program, that gives the analysis enough support to include contextual information into the read/write sets.

3.1 Intraprocedural Read/Write Sets

The intraprocedural read/write sets analysis will collect the read/write sets for every statement within a function body. The procedure followed for this purpose is a walk through the function body. Each statement is considered separately in order to record the appropriate information encountered. Compound statements will incrementally include their read/write sets information with the collected information from their constituent statements.

This walk through procedure collects all information in a recursive fashion, such that, at the end of the intraprocedural step, each function will keep records for the general read and write sets produced by the function body.

3.1.1 Fast Points-to Information

Whenever a statement with an indirect reference is analyzed, the appropriate points-to information is required. The points-to information is available from the fast points-to storage model and can be directly accessed through the declare node of any program variable or function name. In order to retrieve the points-to information for a particular indirect reference, we need to follow the proper links from a given variable. The steps to follow are enumerated below:

- ECR_VARIABLE: Obtain the abstract location of the pointer variable.
- ECR_POINTS_TO: Obtain the reference pointer of this abstract location. This reference pointer gives us the first referenced variable.
- ECR_VAR_DECL: Obtain the variable declare of the first pointed-to variable.
- ECR_ALIAS: Obtain the alias variables of the reference variable.
- ECR_ALIAS_DECL: Obtain the variable declare of a second pointed-to variable.
- ECR_ALIAS_NEXT: Follow the members list for other pointed-to variables.

Figure 17 illustrates the points-to information that is available from the storage model for a sample case. Variable `X` is a pointer to the variable `Y`, but also to the variables `Y'`, `Y''`, and `Y'''`. In other words, `X` points to the abstract location that has `Y` as its representative, the other members of this abstract location are called aliases since their abstract locations are joined to indicate that they may be accessed through the same pointer.

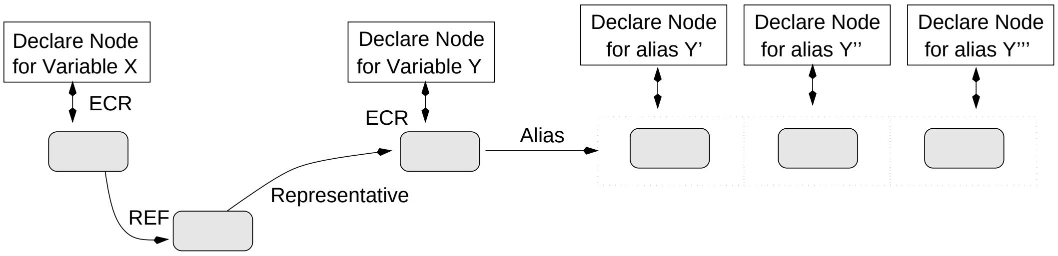


Figure 17: Retrieve Points-to Information

The general procedure for retrieving points-to information for a given pointer variable `X` is the following:

1. ecr = ECR_VARIABLE(X)

   `/* Obtain the abstract location the variable X */`

2. Y = ECR_VAR_DECL(ECR_POINTS_TO(ecr))

   `/* Get the first variable from the set of variables pointed by X*/`

   `/* Y is pointed by X */`

3. alias = ECR_ALIAS(ecr)

   `/* Obtain an anchor for the other variables that are pointed by X */`

4. While `alias` is different from BOTTOM do

   (a) Y' = ECR_ALIAS_DECL(alias) `/* Y' is pointed by X */`

   (b) alias = ECR_ALIAS_NEXT(alias)

3.2 Interprocedural Read/Write Sets

Contextual information is the one obtained by including into the data flow analysis the effects of function calls. Given a program sequence, we are now interested in capturing the information that is produced at a function call. Consider for example the following sequence:

```
int global;
   F() {                              G() {
        ....                               ....
        global = ...                       F();
        ....                               ....
        }                                  }
```

It is important to acknowledge that the function `F` can introduce valuable information into the read/write sets that `G` generates, which is, the modification of the variable `global`. This is, with contextual information, we are now able to include into the write set of `G` the variable `global`.

The notion of context is necessary for achieving more accurate data flow information. We can obtain this notion by creating a hierarchy based on the function calls produced in a program. The process of obtaining the hierarchy of functions will provide the analysis with the relative location of a function with respect to another in the calling context. In order to achieve this process we divide the interprocedural analysis into two sequential steps, which are:

1. Build the program's call graph.
2. Process the flow of information using a work list that is obtained by the call graph.

3.2.1 Call Graph

A *call graph* is a directed graph data structure that assigns to each function of the program a node. The edges of the graph connect nodes, whose associate functions, have a caller-callee relationship in the program. The caller-callee relationship of two functions in the program, say `p` and `q`, is determined as follows: if the function `p` calls the function `q` then there will be an edge going from the call graph node that represents `p` to the call graph node that represents `q`.

In C, it is possible to invoke a function through a variable by use function pointers. This feature introduces extra, potentially spurious, edges in the call graph.

When building a call graph the points-to information becomes a valid source of information since it provides a list of all possible functions pointed-to by the function pointer. An example shown in figure 18 reveals the benefit of having points-to information regarding function pointers on our construction of the programs call graph.

In the presence of recursion the call graph introduces some cycles. In figure 19 we show the presence of recursion on a sample call graph.

3.2.2 Worklist

Once we have support for the program structure, we proceed to produce the appropriate flow of information and obtain the interprocedural results. In order to correctly propagate the

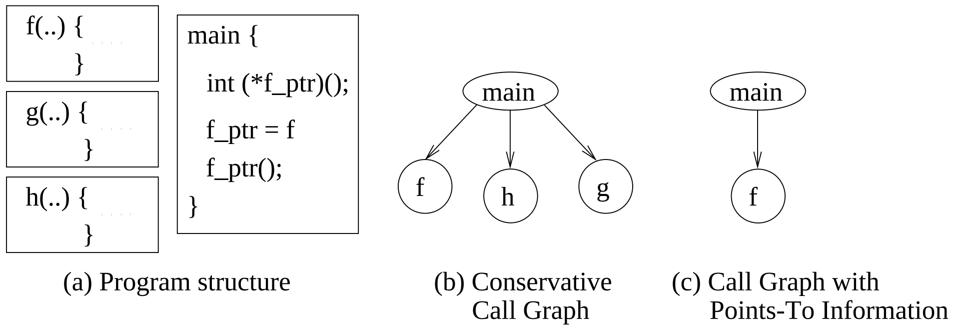


Figure 18: Example of a Call Graph with Function Pointers

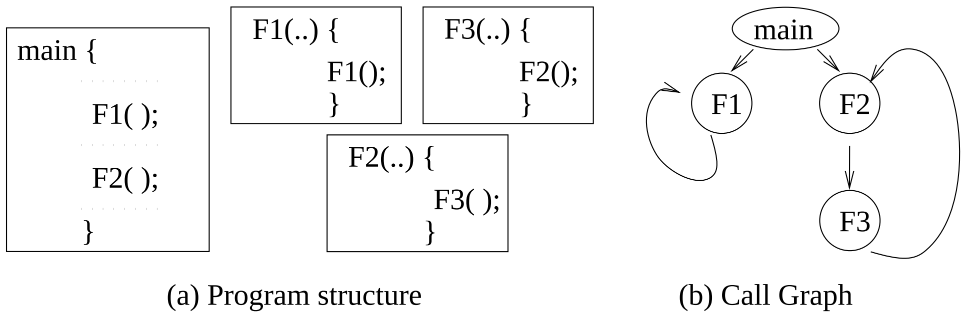


Figure 19: Example of a Call Graph with Recursion

information along the call graph we introduce a *worklist* data structure.

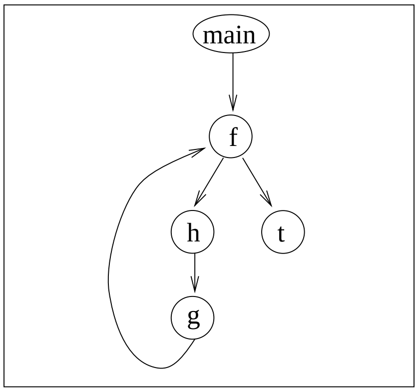
The worklist can be seen as an ordered list of the functions of the call graph. We create the worklist depending on the nodes of the call graph. This is done following a Depth First Search algorithm (DFS), such that the main function will appear as the last function in any generated worklist.

We use the worklist to process each node of the call graph by exporting the filtered read/write sets of a function to all of its predecessors in the call graph. The algorithm used to process a node in the worklist is the following:

```
ProcessNode(n)
          1. Merge info(n) with info(p) where p is in parents(n)
          2. Take_Out_From_WorkList(n)
          3. if info(p) changes for some p in parents(n)
                       AND
             some c in children(n) is still in the worklist
          3.1 Put_back_into_WorkList(n)
```

The worklist gives us the order in which we must process each node of the call graph. We will always process the first node of the worklist. `Put_back_into_WorkList(n)` will append the node `n` at the end of the worklist.

A node of the worklist must be reconsidered only if it belongs to a cycle (recursive call). We know that a node is in a cycle if it's child is eventually its ancestor; that is, if its child is still in the worklist. Figure 20 shows the step by step application of the `ProcessNode` procedure on the worklist of a sample call graph.



Worklist: {g, t, h, f, main}

1. Process g

Merge RW(g) with
RW(h) and RW(f) and RW(main)
As RW(f) has changed and
f is a child of g,
g must enter the worklist again

Worklist: {t, h, f, main, g}

2. Process t

Merge RW(t) with
RW(f) and RW(main)
As no child of t has changed
t leaves the worklist

Worklist: {h, f, main, g}

3. Process h

Merge RW(h) with
RW(f) and RW(main)
As no child of h has changed
h leaves the worklist

Worklist: {f, main, g}

4. Process f

Merge RW(f) with
RW(main) and RW(g)
As RW(g) has changed and
g is a child of f
f must enter the worklist

Worklist: {main, g, f}

5. Process main

main has no parents,
main leaves the worklist

Worklist: {g, f}

6. Process g

Merge RW(g) with
RW(f) and RW(main)
As RW(f) has not change
g leaves the worklist

Worklist: { f }

7. Process f

Merge RW(f) with
RW(g) and RW(main)
As RW(g) has not changed
f leaves the worklist

Figure 20: Worklist example with Recursion

Each function will obtain the proper interprocedural read/write sets information through the use of the worklist. In order to propagate the read/write sets information that results from the interprocedural analysis along the affected compound statements of a function's body, an update procedure is followed. This propagation follows a similar mechanism to the one used for the intraprocedural task. The call statements will contain the valid interprocedural information, and the walk through procedure along the statements of the function's body will make sure that each compound statement will get properly updated. This mechanism will ensure the interprocedural analysis that the interprocedural information will flow through all nested compound statements.

4 Experiments and Results

We have discussed so far that the benefits of performing flow-insensitive points-to analysis over context-sensitive points-to analysis mainly relies on the almost-linear complexity cost for the processing time and space requirements. We demonstrate the benefits of using our fast points-to analysis by providing some experimental results on standard benchmarks. Also, we describe some fast points-to statistics to reflect the characteristics of the storage model of a set of benchmarks. We present a discussion on how the accuracy of the points-to information may be affected when using flow insensitive points-to analysis results. We will see how this effect will turn into a major drawback for the application of fast points-to analysis on some type of programs. We base our comparison on the read/write sets results obtained from using the points-to information generated by the fast points-to analysis (FP) versus the one generated by the context-sensitive points-to analysis (CSP).

4.1 Experimental Results on Points-To Analysis

In this section we demonstrate the the benefits of using flow insensitive points-to analysis. The experiments were made on a Sun SPARCstation 20 (model 51) running SunOS v4.1.3_u1, with 128 megabytes of memory. A brief description for the benchmarks used for some of our experiments is given in table 3.

Experimental results on a set of benchmarks to determine the processing time consumed during the two versions of points-to analysis are listed in table 4. For each benchmark listed in table 4 we have (from left to right):

Size : The benchmark size counted as lines of code of the program, excluding lines of comments and empty lines.

FP-Time : the processing time reported by the flow-insensitive points-to analysis. This time is reported in seconds.

CSP-Time : the processing time reported by context-sensitive points-to analysis. For optimization reasons we generated these times using the optional flag -falias-func-memoize, in order to obtain the best times for CSP analysis. This time is reported in seconds.

Ratio : Calculated as the relation $\frac{CSP-Time}{FP-Time}$

In table 4 we can appreciate the speedup of points-to information processing time by using flow-insensitive points-to analysis against context-sensitive points-to analysis. In general terms we may say that the difference between both processing time is, for any case, considerable big. As example we can observe that for the benchmark livc has ratio of 286.8 and for benchmark travel the ratio reaches to 192.83.

The results shown in table 4 clearly demonstrates the gain on time, indicating that time costs is one of the major benefits obtained by using the flow-insensitive points-to analysis; this is, obtaining the points-to information from elaborating on the storage model of the program.

The impact on speed is highly noticed on the processing time of larger programs. The benchmarks results of large programs like cq or livc claim an extremely high cost on processing time of CSP analysis. This high cost may act as a drawback for the application of the CSP analysis in some cases. Therefore an alternative, less expensive, points-to analysis could fill in for obtaining points-to information.

Another appreciable issue from the FP time results is the almost linear behavior manifested by the numbers of table 4 for fast points-to analysis. This behavior corresponds to the almost-linear complexity cost for time and space mentioned in earlier sections.

Benchmark Name	Description
backprop	this program uses backpropagation to train a neural network
unroll	matrix multiplication
nas-appsp	C version for NAS CFD code: APPSP
exptree	find the expression tree
anagram	text string encoding, anagram generator
travel	the traveling salesman problem
ks	graph partition via Kernighan-Lin or Kernighan-Schweikert
li	lisp interpreter
sc	a spreadsheet calculator
stanford	stanford baby benchmark
clinpack	the C version of Linpack
compress	reduces the size of files using Lempel-Ziv coding
ft	algorithm for finding the minimum spanning tree
sim	find local similarities with affine weights
config	checks all the features of the C-language
circuit	parallel version of sparse matrix for Sequent
yacr2	channel Routiner
ear	simulates the propagation of sound
cq	performs a series of tests on a C compiler
bc	the grammar for a POSIX compatible bc processor
espresso	minimizes boolean functions with a logically equivalent function
eqntott	translates a logical representation equations into a truth table

Table 3: Benchmarks Description

Benchmark	Size	FP-Time	CSP-Time	Ratio
backprop	243	0.04	0.23	5.75
exptree	348	0.04	0.56	11.2
anagram	477	0.05	1.52	30.4
travel	851	0.06	11.57	192.83
ks	668	0.07	2.63	37.57
chomp	428	0.08	4.68	58.5
rubik	445	0.08	7.20	90
stanford	865	0.11	1.28	11.63
clinpack	1037	0.09	2.94	32.6
compress	1206	0.14	5.50	39.28
ft	1296	0.09	2.49	27.6
sim	1340	0.12	9.58	79.83
livc	1759	0.25	71.70	286.8
config	2036	0.36	14.90	41.38
circuit	2392	0.14	3.78	27
yacr2	2734	0.26	17.85	68.65
ear	3089	0.35	8.32	23.32
cq	4753	0.47	26.11	55.55

Table 4: Processing Times for Points-to Analysis.

In figure 21 we plot the almost linear behavior of the processing time with respect to the program size for flow-insensitive points-to analysis. In this figure we can appreciate how the growth on the program size claims great penalty on the time of processing CSP analysis.

4.2 Fast Points-to Statistics

Flow-insensitive points-to analysis uncovers other countable characteristics. Table 5 presents a summary of these characteristics.

Total EQC : Is the total number of equivalence classes found in the storage model, this number is obtained by counting the different representatives of each class.

Avg. Size EQC : Is the sum average of the size of all equivalence classes, this number corresponds to

$$\frac{\sum_{i=1}^{N} size(\text{EQC}_i)}{N},$$

where N is the total number of EQC and $size(\text{EQC}_i)$ returns the number of elements that belong the the equivalence class.

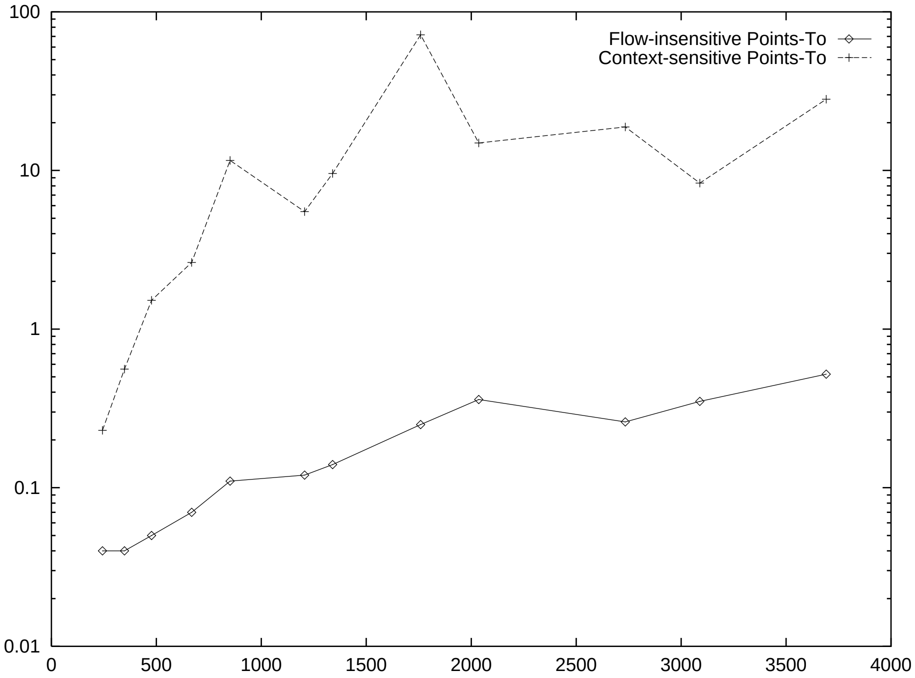


Figure 21: Processing time of flow-insensitive points-to analysis against context-sensitive points-to analysis on a y logarithmic scale

Max. Size EQC : the maximum size found for an EQC of the generated storage model. This number is a result of many JOIN operations.

To illustrate how we obtain the fast points-to statistics we show in figure 22 the computation for **Total EQC** and **Avg. Size EQC** on a sample storage model.

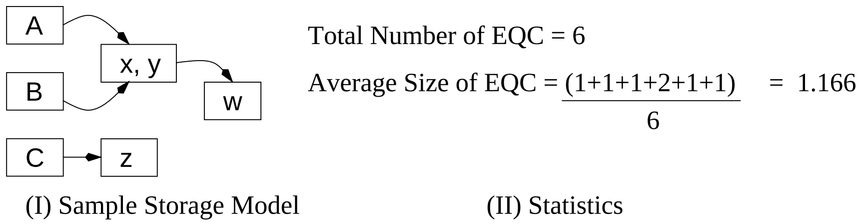


Figure 22: Fast Points-to Statistics example

From table 5 we can see how too conservative approaches are taking place on benchmarks with reported high value on Maximum EQC like the case of li and espresso where for benchmarks like anagram and c4 the maximum size associated with an EQC is fairly good which results in quite accurate points-to information.

Also, we can appreciate that the average size of all EQC remains within the range of 1.00 and 1.08 which is a close approximation to an optimal size of 1.00 for each equivalence class.

Benchmark	Total EQC	Max. Size EQC	Avg. Size EQC
c4	117	2	1.02
rubik	96	2	1.04
unroll	138	2	1.00
alvinn	74	2	1.01
anagram	171	2	1.01
circuit	514	5	1.01
compress	272	5	1.01
travel	177	4	1.02
vlsi	212	2	1.00
bc	2456	33	1.01
nas-appsp	6524	2	1.00
ft	273	2	1.00
ks	225	4	1.02
yacr2	1056	3	1.00
ear	978	15	1.01
sc	9171	32	1.00
li	2824	232	1.08
espresso	9693	32	1.00
eqntott	840	8	1.01

Table 5: Fast Points-to Statistics

We can deduce that for benchmarks that reported large equivalent classes but still a fairly good average size for all EQC, has a storage model where only few EQC are large.

We can classify further on the results reported by our fast points-to analysis by detecting what type of target is associated with each of the pointer variables. This classification can give us some insight on the general shape of the storage model that was built for each benchmark.

A first classification of our points-to information can be to determine how many pointers have their target on the stack and how many of them have their target on the heap. It also gives us more detail on knowing how many pointers are references to function bodies. This classification is given in table 6. For each listed benchmark we count:

Total Pointers: This quantity corresponds to the total number of pointers found in the storage model of the benchmark. This is, the total number of distinct variables that have the REF pointer different from BOTTOM.

Heap Pointers: Some variables are pointers to the heap. This number represents the number of variables that points to a heap location in the storage model.

Heap Locations: Each time there is a dynamic allocation, the storage model is modified by adding a new equivalence class that represents the newly allocated memory location. This number reflects the amount of ECR's that were created for this purpose.

Benchmark	Function Pointers	Heap Pointers	Heap Locations
c4	1	0	0
livc	72	0	0
cq	1	0	0
lharc	5	1180	3
config	2	1	1
pgm3	1	69	3
sim	0	359	1
anagram	2	85	2
compress	2	4	1
travel	0	33	3
vlsi	0	126	5
bc	11	2473	31
ft	0	161	5
ks	0	113	5
yacr2	0	857	26
ear	0	756	25
sc	13	9020	1
li	4	2902	6
espresso	38	9638	175
eqntott	11	747	6

Table 6: Pointer Classifications with many-heap representation

Function Pointers: This quantity corresponds to the number of variables that sometime in the program points-to a function. In other words, this number corresponds to the number of variables which REF pointer belongs to the same equivalence class than a function declaration.

We have mentioned before that our fast points-to analysis can be tuned to represent the memory locations that refer to the heap in two different ways. From this capability arises another interesting statistic, which is the static count of heap locations obtained by our analysis in the case of selecting the many-heap representation. This quantity is also shown in table 6.

We can categorize even further on the type of pointers found by our analysis. In table 7 we distinguish further among the stack pointers found by our analysis by counting the number of locations that can be de-referenced. We show a list of items that describes the shape of the storage model built for a set of benchmarks. The meaning of each item of table 7 is defined as follows:

1 Loc: Counts the number of variables that are a one location pointer, and this location is not the heap.

2 Loc: Counts the number of variables that are pointers to two locations, not including the heap.

3 Loc: Counts the number of variables that are pointers to 3 locations, not including the heap.

4-10 Loc: Counts the number of variables that points to more than 3 locations but less than 10 locations, not including the heap.

>10 Loc: Counts the number of variables that points to more than 10 locations, not including the heap.

a heap Loc: Counts the number of variables that points to at least one heap location.

Total: Is the total number of variable pointers.

Benchmark	1 Loc	2 Loc	3 Loc	4-10 Loc	>10 Loc	a heap Loc	Total
c4	100	2	1	5	3	0	111
nas-appsp	4939	9	9	54	1489	0	6500
unroll	123	1	1	5	1	0	131
alvinn	59	1	1	5	1	0	67
cq	749	1	1	5	211	0	967
travel	137	1	1	5	1	33	178
rubik	48	3	3	18	17	0	89
lanczos	50	1	1	11	73	0	136
compress	166	1	1	24	70	4	266
vlsi	72	1	1	5	1	126	206
ft	101	1	1	5	1	161	270
ks	99	1	1	6	5	113	225
yacr2	186	2	1	5	13	857	1064
ear	39	1	1	2	152	756	951
bc	32	0	0	0	3	2473	2508
li	139	1	1	1	13	2902	3057
espresso	168	0	2	3	73	9638	9884
eqntott	71	2	1	10	8	747	839

Table 7: Pointer Classification by locations

From the results collected in table 7 it becomes clear that many pointers fall into the category of 1 Loc, which is a good prediction for the use of our analysis. Other listed benchmarks, such as li, bc indicate that most of the pointers fall into the heap category. Benchmarks like ft and ks indicates that the pointers can generally de-reference high number of variables; while the benchmark nas-appsp delights us with a very good number of 1 Loc pointers.

4.3 Applications of Points-to Analysis

From theory it is clear to to see that the storage model will change either by resetting some equivalence class members or by adding new classes to the model.

The flow-insensitive analysis needs to keep track on every possible pointer information it generates, but it is not able to kill information simply because the analysis is independent from the order in which the statements are executed, which means that the same storage model will be built, even if we reorganize the program statements.

Therefore, the updates of the storage model only adds up features but never deletes. Figure 23 shows how this is a drawback in points-to information accuracy, since the storage model indicates that variable `b` points to `c`, which would never occur in the program. The flow-insensitive points-to analysis maintains a notion of the behavior of the program's storage by keeping track on what is done on each different memory location.

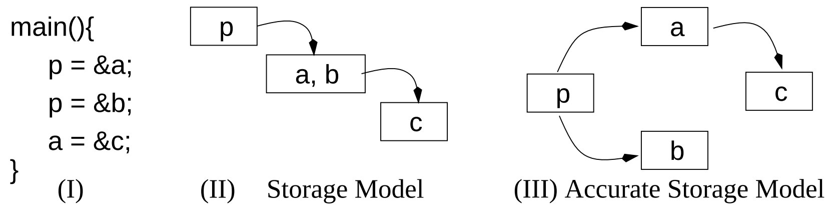


Figure 23: Update of the Storage Model

It is clear that for each single valued variable there is an associated memory location. But for compound variables such as **arrays**, **unions** or **structures** there are a set of related fields, which in terms of our storage model, translates into a set of different memory locations that are to be associated to a single variable name.

For simplicity on the rules used by the fast points-to analysis algorithm; the algorithm makes no distinction on whether a variable is single (a single-valued variable) or compound (associated with multiple values). And hence, different fields of a **structure** or a **union** will be viewed by the storage model as one memory location. Also, **arrays** will be viewed as a same memory location without distinguishing any of it subcomponents. This simplification creates more conservative approaches on the storage model since it does not make a distinction on subcomponents for compound variables.

Figure 24 shows how this also reflects on a drawback in the accuracy of points-to information, since the storage model indicates that variable `y` points to variable `z`, once again this information implies a restriction for a non existing fact of the program.

Experiments on different benchmarks demonstrate that these drawback are a considerable price to pay when using the fast points-to analysis. Using read/write sets analysis we are able to observe a decrement on the levels of accuracy present on the points-to information results obtained with FP analysis against the levels of accuracy present on those obtained with the CSP analysis.

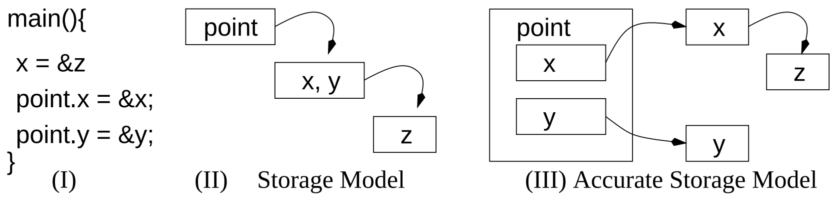


Figure 24: Compound Variables in the Storage Model.

In table 8 we show the statistics obtained when we used the results of fast points-to analysis to generate the call graph of the program and to perform the read/write sets analysis, on a set of benchmarks.

The items shown in table 8 are:

Ind. Ref: Total number of indirect references counted during the fast read/write sets analysis. This implies the number of times read/write sets employed the points-to information to generate the corresponding sets.

Misses by V: Misses on Variables. This number counts the number of times the analysis requires exact information on subcomponents of a compound variable, but the points-to sets only has conservative information to offer.

Avg. P.Sets: The average size of pointer sets (EQC) counted for each indirect reference. This number corresponds to the average sum over the size of the points-to sets that are de-referenced in the program.

Max. EQC Size: Is the maximum size found for an EQC of the Storage Model.

func.ptr: Is the number of times a function pointer is used by the call graph analysis to generate a node.

In table 8 we observe how the drawback produced by considering compound variables as single chunks of memory without differentiating among their fields represents a major loss in the overall points-to information accuracy. Some cases listed in table 8 report that around half or more of the indirect references found in the program requires more specific information than the one provided by fast points-to analysis, reflecting an addition to the count of misses by field.

Also we can observe that when indirect references hit mostly large points-to sets the average size of EQC increases. This indicates a highly conservative approach by allowing the formation of large EQC. Which in many cases, is related to the inability of making any distinction among different fields for compound variables. And hence, even if two pointers are associated to different fields of compound variable, the analysis will consider them as pointers to the same memory location and join any other EQC that independently may be associated with them.

Benchmark	Ind. Ref	Avg. EQC Size	func.ptr	Misses by V.
c4	46	1.00	2	15
nas-appsp	1627	1.21	0	0
livc	25	2.96	0	0
cq	961	1.00	1	1
lharc	468	14.67	12	398
config	86	1.11	0	45
pgm3	88	2.03	0	63
sim	320	2.46	0	241
anagram	72	1.66	0	40
compress	356	1.29	0	210
travel	56	1.26	0	9
vlsi	163	2.22	0	134
bc	3385	37.24	27	3304
ft	236	2.38	0	182
ks	150	2.30	0	95
yacr2	351	17.31	0	319
ear	430	5.19	0	175
sc	3656	16.30	0	3470
li	1508	60.67	714	1458
espresso	5769	157.45	56	5371
eqntott	785	4.49	29	682

Table 8: Use of Points-to Information

Considering the loss of accuracy in the points-to information previously discussed; it becomes interesting to compare the results obtained by read/write sets with accurate CSP analysis and fast read write sets with fast points-to analysis. Table 9 shows a quantitive approach to measure the accuracy loss on a set of benchmarks using the read/write sets analysis with fast points-to and read/write sets analysis with accurate CSP analysis. For these results we use the many-heap representation.

The items of table 9 are the following:

Definitely Read: The average size for all the counted read sets. A read set is counted only if it is associated with a parameter in a function call statement, or it is associated with a modify expression involving an indirect reference on the right-hand side.

- FP: uses points-to information generated by fast points-to information.
- CSP: uses points-to information generated by accurate CSP information.

Definitely Write: The average size for all the counted write sets. A write set is counted only if it is associated with a modify expression involving an indirect reference on the left-hand side.

- FP: uses points-to information generated by fast points-to information.
- CSP: uses points-to information generated by accurate CSP information.

Possibly Read: The average size for all the counted may-read sets. A may-read set is counted only if it is associated with a parameter in a function call statement, or it is associated with a modify expression involving an indirect reference on the right-hand side.

- FP: uses points-to information generated by fast points-to information. A conditional count is issued for referenced variables that are defined outside the current function, this is to simulate the behavior of CSP invisible variables.
- CSP: uses points-to information generated by accurate CSP information.

Possibly Write: The average size for all the counted may-write sets. A may-write set is counted only if it is associated with a modify expression involving an indirect reference on the left-hand side.

- FP: uses points-to information generated by fast points-to information. A conditional count is issued for referenced variables that are defined outside the current function, this is to simulate the behavior of CSP invisible variables.
- CSP: uses points-to information generated by accurate CSP information.

Total Read: Is the sum of Definitely Read and Possibly Read corresponding to each case CSP and FP.

Total Write: Is the sum of Definitely Write and Possibly Write corresponding to each case CSP and FP.

In order to analyze the results of table 9 we first describe a property associated with the results obtained using accurate CSP analysis. Accurate CSP analysis is able to distinguish between points-to tuples that are Definite and points-to tuples that are Possible. When it has been determined that a points-to tuple (or alias pair) will occur in the program regardless from any possible execution path that may have the program at runtime, the points-to tuple is said to be a definite alias pair. While in other cases, if an points-to tuple may hardly occur for a specific (but possible) execution path on runtime; then the points-to tuple (or alias pair) is said to be a possible alias pair.

From table 9 we can argue that, in general terms, the following issues holds for the read/write sets overall behavior:

- The Definitely Read sets

  The results show a tendency to decrease the average size when using the information generated by the fast points-to analysis. We can attribute this behavior to the inability of categorizing the points-to information as definite results, and hence all pointers information that contribute with the read set when using CSP will no longer be able to do so for the case in which it uses FP.

Benchmark Name	Definitely				Possibly				Total			
	Read		Write		Read		Write		Read		Write	
	FP	CSP	FP	CSP	FP	CSP	FP	CSP	FP	CSP	FP	CSP
backprop	1.66	1.83	0.00	1.00	2.03	2.92	1.00	1.00	3.70	4.75	1.00	2.00
anagram	1.24	1.74	0.00	0.00	2.43	2.20	2.33	1.00	3.68	3.94	2.33	1.00
circuit	1.24	1.41	0.00	1.00	7.19	1.22	7.26	1.00	8.43	2.64	7.26	2.00
ear	1.62	2.36	0.00	1.00	8.69	2.60	3.63	1.00	10.31	4.96	3.63	2.00
em3d	1.38	1.38	0.00	0.00	4.71	1.50	7.00	1.00	6.09	2.88	7.00	1.00
ft	1.50	1.72	0.00	1.00	2.73	1.81	3.00	1.03	4.24	3.54	3.00	2.03
graphics	1.39	1.35	0.00	1.00	6.61	1.55	8.48	1.00	8.00	2.90	8.48	2.00
health	1.21	1.24	0.00	0.00	3.86	1.33	4.00	1.00	5.08	2.57	4.00	1.00
ks	1.34	1.54	0.00	1.00	2.94	2.20	2.86	1.19	4.29	3.74	2.86	2.19
nbody	1.44	1.87	0.00	0.00	6.18	2.81	6.00	1.23	7.63	4.69	6.00	1.23
pug	1.46	1.40	1.00	1.00	8.75	1.64	7.78	1.02	10.21	3.05	8.78	2.02
specear	1.58	2.34	0.00	1.00	8.63	2.66	3.50	1.00	10.22	5.01	3.50	2.00
vor	1.33	1.64	0.00	1.00	9.53	1.16	10.05	1.05	10.87	2.81	10.05	2.05
yacr2	1.82	2.59	1.00	1.00	13.49	3.02	18.70	1.00	15.32	5.61	19.70	2.00

Table 9: Comparing the Read/Write Sets using CSP and FP (many-heap locations).

- The Definitely Write sets

  The results obtained with fast read/write sets on the set benchmarks will tends to remain close to the average size number obtained when using accurate CSP analysis. These numbers will be 1 or 0. We can appreciate that each time a 1 appears on the FP side, then there must be a 1 at the CSP side. Once again, the general behavior can be attributed to the lack of definite sets, since all pointer information provided by FP will never be considered as definite information.

- The Possibly Read sets

  The results show a tendency to increase in average size. In the general case we find an increment of the average size, since pointers information will fall into possible sets as a result of the lack of definite information discussed above. This increment is highly marked for the benchmarks yacr2 and ear.

- The Possibly Write sets

  The results show a tendency to increase in average size. Once again we can observe how for benchmarks like yacr2 and vor. In particular for yacr2 there is a considerable increment of the average size from 1 to 18.70, meaning very imprecise results are achieved.

- The Total Read sets

  In general the read sets for the FP case considerably increases in average size compared to the CSP case. Nevertheless, some benchmarks such as backprop, anagram, ft, and ks will not report much difference between the average sizes for FP and CSP. These differences will generally fall into an acceptable range.

- The Total Write sets

  Increments are highly noticed for the average size of the write sets using FP. The write sets for FP will suffer higher increments in size, than those reported by the read sets. This is an indicator of the imprecision obtained when using the FP version.

The overall view for the FP results is that although we obtain imprecise results, many of them can be seen as to fall into acceptable ranges which can be used for further analysis. The benefit of using FP is that with a very inexpensive procedure we are able to collect acceptable results.

4.4 A Combined Strategy

Up to now our experiments have indicated how the drawbacks of loss in accuracy may influence the application of fast points-to analysis to generate points-to information. Some may consider this issue as not such a tolerable prize to pay for the gain of speedup in processing time discussed in section 4.1. But, again we raise the issue on large benchmarks where the cost of performing accurate CSP analysis turns out to be unsuitable in many cases.

Dealing with large size benchmarks it is reasonable to search for some strategy that includes both the benefits of performing fast points-to analysis as well as accurate CSP analysis. In other words, we would like to overcome the accuracy problem by gaining some territory on a program analysis with the use of fast points-to analysis first and then accurate CSP analysis whenever required.

It is relatively fast for the fast points-to analysis to determine whenever a function does or does not modify the storage model, this is, whether a function changes the pointer information of the entire program.

A combined strategy will save us processing time by using fast points-to as a short-cut on finding exactly where in a program is the information that needs further consideration in order to achieve reasonable accurate results.

Using fast points-to we can determine in a the program the parts that directly modify the pointer information from parts that only uses pointer information but do not change it. And in order to achieve accurate results we can set a trace for the accurate CSP analysis to process further on only these selected parts.

A first attempt for combining strategies is by allowing fast points-to to select a set of functions that requires a more precise analysis. For example consider classifying a program functions as Selected or Excluded. Where a function is considered as Selected when a statement of that program performs a modification on the storage model by updating pointer information, otherwise a function falls into the category of Excluded. A function is selected, if the there was a indirect reference resolution in the definitely and possibly written sets of the function, this is whenever statement of the `*x = a` exists in the function body.

In table 10 we show for a set of benchmarks the number of functions contained by the program, as well as the number of functions that are selected by the fast points-to analysis and the number of the remaining functions. Once the read/write sets for all functions have been collected, this selection process is done by marking functions that include variables that are pointers in their associated write and may write sets. A function is selected if it was previously marked.

Benchmark	Total Functions	Selected Functions	Excluded Functions
c4	13	10	3
lharc	91	88	3
bintree	17	13	4
chomp	22	21	1
cq	40	34	6
livc	82	81	1
puzzle	17	15	2
stanford	48	42	6
eqntott	63	59	4
espresso	80	79	1
whetstone	7	6	1
ft	42	38	4
yacr2	58	54	4
ear	95	84	11
sc	151	149	2
li	358	354	4
ks	13	12	1
compress	17	16	1

Table 10: Using Fast Points-to analysis to select functions

In general terms we can argue that fast points-to analysis is able to exclude a fair number of functions from the program that only uses pointer information but never updates it. Other factors like the size of a function is of importance and may influence on the decision of re-processing with accurate CSP analysis on selected functions, this is reasonable to think given that one may encounter small size functions that require a tolerable amount processing time for accurate results.

5 Related Work

Our major concern was the collection of points-to information in a short and relatively fast processing time for large programs. To achieve this goal our approach uses a flow-insensitive analysis. Our fast points-to analysis is an implementation of the technique described by Steensgaard in [10].

Context-sensitive analysis will generally take longer to obtain the points-to information. But, the results will be more accurate than the ones obtained through the flow-insensitive version. The improvement on processing time is accompanied by more conservative points-to information. To overcome, this important factor, Steensgaard presents an extension to his techniques for improving the accuracy of the resulting points-to information. This extension targets the compound variables with a more precise representation and is fully described in [9].

The importance of accurate information for interprocedural analysis is discussed in [7]; this work refers to an interprocedural side effects algorithm that studies the possible modifications induced by pointer-related data flow information.

Other techniques concerning the accuracy improvement of the flow insensitive points-to analysis are described by Shapiro and Horwitz in [8]. In their work they also evaluate the trade-offs between context-sensitive and flow-insensitive versions of points-to analysis.

A different approach that combines the flow-insensitive and flow-sensitive analysis into a program decomposition methodology is described in [11]. This combined strategy is based on the application of the different analysis method onto separate parts of the program. This approach results in a fast method for obtaining comparable results to those obtained solely by the flow-sensitive method.

6 Conclusions and Future Work

We have presented fast points-to analysis, which is an implementation of Steensgaard's algorithm for flow-insensitive points-to analysis. Our empirical results have shown the almost linear time complexity obtained with the application of the fast points-to analysis. We also discuss the use of points-to information on the read/write sets analysis. We include fast read/write sets, which is a version of the read/write sets analysis that uses the points-to information generated by the fast points-to analysis. An experimental comparison between the flow-insensitive and the context-sensitive versions of points-to analysis is made through the use of the corresponding different versions of the read/write sets analysis. Experimental results have shown an estimate for the accuracy loss obtained while using the points-to information from the fast points-to analysis. This estimate indicates the use of a potentially conservative approach for handling pointer information, nevertheless these estimates also show that for practical reasons the overall loss in accuracy may not be as costly as to disregard the application of the fast points-to analysis. Specifically, large programs will benefit from the advantage of having points-to information with a conveniently low processing time.

Future work relies on the possibility of obtaining more precise and accurate points-to information. This objective can be reached if we allow to keep independent abstract locations within each partition of a compound variable. A different handling will be required for the storage model, each abstract location will then need to maintain a representation for the notion of "fields" of a variable. This internal handling will avoid the imprecision that have been made by the fast points-to analysis for variables that are structures or arrays. In [9] Steensgaard describes the extension of his algorithm to include the proper handling for compound statements. We have shown the benefits of using points-to information with read/write sets analysis. Many other mid-level program analysis can also take advantage of the alternative fast points-to analysis

As studied in section 4.3, a combined strategy between the flow-insensitive and the context-sensitive versions of points-to analysis, could open the possibility of creating a balance between the existing trade-offs of both versions.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1994.

[3] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, Montréal, Québec, July 1993.

[4] 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, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices,* 29(6), June 1994.

[5] Laurie J. Hendren, Guang R. Gao, Chandrika Mukerji, and Bhama Sridharan. Introducing McCAT—The McGill Compiler-Architecture Testbed. ACAPS Technical Memo 27, School of Computer Science, McGill University, Montréal, Québec, September 1991.

[6] Laurie J. Hendren and Bhama Sridharan. The SIMPLE AST - McCAT compiler. ACAPS Technical Note 36, School of Computer Science, McGill University, Montréal, Québec, October 1992.

[7] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. *Laboratory of Computer Science Research Technical Report, Number LCSR-TR-201*, 1993.

[8] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages*, 1997.

[9] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. *In Proceedings of the 1996 International Conference on Compiler Construction*, pages 136–150, 1996.

[10] Bjarne Steensgaard. Points-to analysis in almost linear time. *In Proceedings of the Twentythird Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[11] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step towards practical analyses. *Proceedings of the 4th Symposium on the Foundations of Software Engineering*, 1996.

A Implementations Details

In order to be able to apply the Fast Points-To Analysis, the McCAT compiler has been equipped with the following option flags:

`-ffast-alias` this flag activates the application of the fast points-to analysis.

`-ffast-rw-sets` this flag activates the Fast Read Write Sets Analysis; this analysis uses the points-to information generated by the Fast Points-To Analysis.

`-feq-1-heap` this flag sets the heap representation to be only one heap. This is, it will create only one abstract location to represent memory on the heap. By default, Fast Points-to will use the Many Heap representation.

`-df` this flag is for useful for inquiring on the points-to information generated by Fast Points-To Analysis. The points-to information available from the storage model is added to each statement in the C dump file.

The new files added to the McCAT directory that are involved with the Fast Points-To Analysis are listed in table 4.1.

File Name	Description
alias/eqc.c	contains the abstract location operations used to generate points-to information
alias/eqc.h	contains the abstract locations data structure
alias/eq_dump.c	contains dump information of the points-to information
alias/eq_rules.c	contains the basic rules that shape the storage model
rw-sets/eq_rw.c	contains the procedure used to perform intraprocedural FRWSets
rw-sets/eq_wl.c	contains the procedure used to perform interprocedural FRWSets
rw-sets/eq_wl.h	contains the data structure for the worklist
cgraph/eq_cg.c	contains the procedure to build the call graph
cgraph/eq_cg.h	contains the data structures for the call graph

Table 11: List of new files on McCAT

Modifications include some updates on existing McCAT files, these are:

- rw-sets/rw-dump.c
- rw-sets/vs-dump.c
- main/mymain.c
- c-dump/dh.c
- front-end/toplev.c