SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–19
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

SafeType: Detecting Type Violations for Type-Based Alias Analysis of C

Iain Ireland[2], José Nelson Amaral[1*], Raúl Silvera[3], Shimin Cui[2]

[1] *University of Alberta*
[2] *IBM Canada Software Laboratory*
[3] *Google Inc.*

SUMMARY

To improve the ability of compilers to determine alias relations in a program, the C standard restricts the types of expressions that may access objects in memory. In practice, however, many existing C programs do not conform to these restrictions, making type-based alias analysis unsound for those programs. As a result, type-based alias analysis is frequently disabled.
Existing approaches for verifying type safety exist within larger frameworks designed to verify overall memory safety, requiring both static analysis and runtime checks. This paper describes the motivation for analyzing the safety of type-based alias analysis independently; presents SafeType, a purely static approach to detection of violations of the C standard's restrictions on memory accesses; describes an implementation of SafeType in the IBM XL C compiler, with flow- and context-sensitive queries to handle variables with type `void *`; evaluates that implementation, showing that it scales to programs with hundreds of thousands of lines of code; and uses SafeType to identify a previously unreported violation in the `470.lbm` benchmark in SPEC CPU2006.
Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

1. INTRODUCTION

An alias analysis is a static program analysis that is used at compile time to determine whether expressions may refer, at runtime, to the same memory location. One form of alias analysis is type-based alias analysis. Type-based alias analysis relies on the idea that memory references with different types should generally not alias. To formalize this intuition, the C standard imposes type-based restrictions on memory access [1]. Objects in memory have types; if an object is accessed using an expression with a type that does not conform to the restrictions, the behaviour of the program is undefined. An optimizing compiler is therefore permitted to assume that accesses to memory that violate these type-based restrictions do not exist, and conclude that memory references of different types do not alias. This information can be used to enable more aggressive code transformations to improve the performance of the compiled code.

When applied to programs that violate the C standard's type-based restrictions, type-based alias analysis is unsafe. Its use may unintentionally alter program semantics, leading to the introduction of difficult-to-diagnose bugs. One example comes from the `176.gcc`

*Correspondence to: E-mail: jamaral@ualberta.ca benchmark in SPEC CPU1995. In one function, `176.gcc` clears the contents of a structure by casting the structure to an array of `int` and assigning 0 to each element of the array. Because an array of `int` cannot be legally aliased to an arbitrary structure, a series of seemingly valid transformations may leave uninitialized data in the structure[2].

Unfortunately, in practice, many C programs do not conform to the C standard's type-based restrictions on memory access, and their users still expect a compiler to generate functional code. In some cases, the existence and identity of violations is known. For example, in the SPEC CPU2006 benchmark suite, the list of known portability issues for the `400.perlbench` benchmark includes a warning about violations of ANSI aliasing rules. Similarly, prior to Python 3, the reference implementation of Python used a representation of Python objects that violated ANSI aliasing rules [3]. In both of these cases, the existence of violations was known. However, there are other cases in which the presence of violations is uncertain. Violations may occur due to programmer error, or ignorance of the restrictions. Violations may also occur in legacy programs written prior to the creation of the C standard. In many cases, the effort required to identify non-compliant code and rewrite those programs to be compliant is prohibitive. This obstacle is particularly relevant for the largest and most important code-bases.

Bugs introduced by unsafe type-based alias analysis can be difficult to diagnose and correct. As a result, type-based alias analysis is frequently turned off. This situation is unfortunate. Even in non-standard-compliant programs where type-based alias analysis is unsafe, there remain many aliasing situations where type-based alias could provide useful information. Furthermore, SafeType assists in the creation of in the creation of standard-compliant code, which is more portable and easier to maintain an debug. It is therefore useful to be able to automatically identify code that may violate the C standard's type-based restrictions. If code that violates the standard can be isolated, it becomes possible to make informed decisions about when to use type-based alias analysis. Furthermore, it also becomes possible to identify the code changes that must be made to ensure that type-based alias analysis can be safely applied to a program.

Approaches to this problem exist that guarantee the overall memory safety of C, including as one component of this guarantee the absence of run-time type violations[4][5]. However, these approaches impose overhead on execution by requiring the insertion of runtime checks. For some use cases, this trade-off of performance for safety is acceptable. However, for type-based alias analysis in particular, this trade-off is unsatisfying. The purpose of aliasing information is to give an optimizing compiler the information it requires to make more aggressive transformations and improve performance. Verifying the safety of type-based alias analysis through the use of runtime checks, therefore, exchanges performance for less performance. It is therefore desirable to have a purely static analysis for verifying the safety of type-based alias analysis that imposes no runtime overhead.

Surprisingly, no such analysis is explored in the literature. We therefore present SafeType, a purely static approach to detection of violations of the C standard's restrictions on memory accesses. SafeType attempts to preserve the invariant that each pointer in memory is either a null pointer or points to an object of a type that the pointer is permitted to access. In cases where SafeType cannot prove that the invariant is preserved, instead of inserting runtime checks, SafeType emits a warning to the programmer. Programs without warnings can be safely compiled using type-based alias analysis, while programs with warnings require the programmer to manually examine some cases. SafeType can also be used as a diagnostic tool to speed the process of debugging in cases where the programmer suspects type-based alias analysis may be at fault. To minimize the number of warnings, SafeType includes a fully flow- and context-sensitive analysis of variables with type `void *`.

Section 2 gives background information on type-based alias analysis. Section 3 presents a formal definition of the problem of identifying violations, and introduces a static analysis, SafeType, that soundly identifies such violations. Section 4 completes the presentation of SafeType, describing its handling of void pointers. Section 5 evaluates a prototype implementation of SafeType, demonstrates that it scales to programs with hundreds of thousands of lines of code, and identifies a previously unreported violation of the C standard's type-based restrictions on memory access in the `470.lbm` benchmark from SPEC CPU2006. Section 6 describes related work on points-to analysis and safety analysis for C. Section 7 concludes the paper.

2. BACKGROUND

Although the idea of type-based alias analysis has been mentioned in passing for decades [6], the first publication to present an algorithm or evaluate the benefits of type-based alias analysis was an implementation for Modula-3 by Diwan *et al.* [7]. They present three versions of the analysis. The first version is purely type-based: two memory references may alias if and only if they have the same declared type, or if the declared type of one is a subtype of the declared type of the other. The second version improves the precision of the analysis using additional high-level information. The third version uses a flow-insensitive algorithm to exclude aliases between a type T and a subtype S of T unless a statement assigns a reference of subtype S to a reference of type T. Each of the three versions is evaluated as the only alias analysis in a Modula-3 compiler.

The first implementation of type-based alias analysis for C to be described in the literature is by Reinig for the DEC C and DIGITAL C++ compilers [2], The DEC implementation operates on each function independently and creates a set of `effects classes` and associated `effects signatures` that allow the compiler to efficiently represent both type-based alias analysis and structural aliasing. Like Diwan's analysis, this analysis is flow-insensitive, context-insensitive, and field-sensitive.

Although no paper was published, gcc introduced type-based alias analysis in the same time period as the DEC compiler [8]. In gcc's implementation, alias information produced by the C front end based on the type assigned to an object by the programmer was propagated into the optimization passes. This information was used in conjunction with a pre-existing "base address" alias analysis, which eliminated a different set of impossible alias pairs. The combination of the two analyses was more precise than either analysis independently.

Type-based alias analysis has gone on to become a common and important feature of production C compilers. For example, it is present in IBM's XL C compiler, the Intel Itanium compiler [9], and, as of April 2011, LLVM's Clang compiler [10].

Type-based alias analysis is also used for other languages, such as Java [11][12] and C++ [13]. Lhoták shows that type-based alias analysis improves both the precision and the efficiency of points-to analysis in Java [14]. The type safety guarantees of some of these languages, including Java, ensure that type-based alias analysis is always sound.

The most important evaluation of the benefits of type-based alias analysis comes from Ghiya [9]. In the context of the Intel Itanium compiler, Ghiya evaluates seven complementary strategies for memory disambiguation: disambiguation of direct references; a simple base-offset analysis; an array data-dependence analysis; an intraprocedural version of Andersen's points-to analysis modified to be field-based; a global address-taken analysis; an interprocedural version of Andersen's analysis; and a type-based alias analysis. Each of these techniques is applied to the twelve C/C++ benchmarks from the SPEC CINT2000 benchmark suite. Ghiya determines that type-based alias analysis "pays off" for five of the twelve benchmarks, eliminating alias pairs that no combination of the other six methods can disambiguate. In particular, type-based alias analysis is important for disambiguating pointers against scalar objects. While acknowledging that type-based alias analysis is potentially unsafe, Ghiya concludes that a suite of disambiguation techniques – including type-based alias analysis – is the optimal approach.

3. SAFETYPE: AN ANALYSIS TO ENABLE TYPE-BASED ALIAS ANALYSIS

SafeType is a static analysis that examines each statement in a program to identify code that makes type-based alias analysis unsafe. At each statement, SafeType generates a set of constraints. In the general case, these constraints can be evaluated immediately. When dealing with variables of type `void *`, a flow-sensitive approach is required. SafeType uses on-demand queries to attain flow sensitivity, and uses summary functions to attain context-sensitivity. This section introduces the underlying principles of SafeType, and describes the flow-insensitive component of SafeType.

3.1. Assumptions

The goal of SafeType is to statically detect violations of the C standard's restrictions on memory accesses that might make type-based alias analysis unsafe. To this end, SafeType assumes the absence of three types of illegal memory operations: dereferences of undefined pointers, dereferences of freed memory, and array-bounds violations. In the presence of these errors, SafeType may inaccurately conclude that two memory references do not alias. We make this decision for three reasons. First: in the presence of these illegal memory operations, almost any memory references may potentially alias. It is therefore impossible in the general case to prove the absence of illegal memory operations in unmodified C programs without introducing runtime overhead, which contradicts the purpose of type-based alias analysis. Second, the C standard categorizes these illegal memory operations as undefined behaviour, giving explicit permission to make precisely the required assumption. Third, a large number of tools exist to detect illegal memory operations, and programmers have experience in detecting and correcting them. No tools exist to detect the violations that SafeType targets, and programmers have much less experience identifying and correcting bugs introduced by unsafe type-based alias analysis. Thus, by assuming the absence of illegal memory operations, SafeType is more effective, more tractable, and more precisely focused on the problem it aims to solve.

3.2. Preliminaries

In what follows, τ ranges over types, P ranges over programs, f ranges over functions, v ranges over variables, e ranges over expressions, l ranges over lvalues, o ranges over objects, s ranges over statements, and $\mathcal{S}$ ranges over program states.

The C standard defines an *object* as a region of data storage in the execution environment. The contents of an object represent a *value* when interpreted as having a particular type. An *lvalue* is an expression that refers to such an object. An *access* is an execution-time action reading or modifying the value of an object in memory. A *modification* is an access that assigns a new value to an object (including cases where the new value being stored is the same as the old value). A *read* is an access that uses the current value of an object.

An expression in a program is a memory reference if its execution accesses memory. In an assignment $v_1 = v_2$, there are two memory references: one that reads v_2, and one that modifies v_1. Two memory references are *aliased* if they may refer to the same memory location.

The C standard defines the *effective type* of an object in memory as the declared type of that object, if it exists. Objects allocated on the heap have no declared type. For such objects, the effective type is set when a value is stored into that object through an lvalue with a type that is not a character type.

The C standard imposes restrictions on the types of lvalue expressions that may access objects in memory. These restrictions, found in paragraph 7 of section 6.5 of the standard [1], can be characterized as follows:

Definition 1
- Except where noted below, an lvalue expression of type τ may only access an object in memory with the same type τ.

- Type qualifiers (`volatile, const`) are ignored when determining whether two types τ_1 and τ_2 are the same.
- The signedness of integer types is ignored when determining whether two types τ_1 and τ_2 are the same.
- By definition, an access to an lvalue with an aggregate type (an array, a struct, or a union) accesses the memory of the members of that aggregate. This is known as *structural aliasing*. Thus, an access to an lvalue with an aggregate type τ_{aggr} is permitted to access an object in memory with a type τ_{member} if τ_{aggr} includes a member of type τ_{member}, either directly or recursively as a member of a subaggregate.
- To allow for bytewise manipulations of memory, an lvalue expression with type `char` is permitted to access memory of any type.

Definition 2
Define the can-access relation $\rhd$ such that $\tau_l \rhd \tau_o$ iff an lvalue l of type τ_l is permitted to access an object in memory o of type τ_o according to these restrictions.

Definition 3
Define $\mathsf{safe}(l)$ to be true for an lvalue l if l is permitted to access the value of the object to which it refers.

These restrictions provide the foundation for existing implementations of type-based alias analysis. Any lvalue of type τ_l can be assumed not to alias any object of type τ_o unless $\tau_l \rhd \tau_o$.

3.3. Inputs

In addition to the abstract syntax tree of a program P, SafeType requires data-flow information for each function in P. For a function f, let $S(f)$ be the set of statements in f, augmented with a dummy statement s_0 representing the entry point of f. For a use of a variable v in a statement $s \in \mathcal{S}(f)$, the use-def function $\mathcal{D}(v, s)$ is the subset of statements in $S(f)$ that define a value for v such that this value may reach s without an intervening assignment to v. These definitions may occur as the result of an explicit assignment, or as the result of a side-effect of a function call. They may also represent the pre-existing value of v at the entry point of the function. The implementation of SafeType takes advantage of the factored single static assignment (SSA) representation generated by the XL compiler. In principle, the use of factored SSA is not necessary. The required information could be obtained from another form of SSA, from def-use and use-def chains, or by traversals of the abstract syntax tree.

SafeType also requires the construction of the static call graph of P. The call graph for a program P is a directed multi-graph $C_P = (V, E, \mathit{Call})$, where V is the set of functions, E is the set of call edges, and Call is the set of call sites within those functions. Each edge $(f, c, g) \in E$ represents a possible call from a function call statement $c \in \mathit{Call}$ in function f to function g, where $f, g \in V$. Cycles in the call graph represent function recursion; a self-recursive function will be represented by an edge (f, c, f). Multiple edges may exist from a function f to a function g if f calls g from multiple distinct call sites. Multiple edges may also exist from a call site c in a function f if the call site may call more than one function (due to the use of function pointers).

Let T_C be the set of C types present in a program. Let T_{P_f} be the set of placeholder types present in a function f, as defined in Section 4.2. For a function f, let $T_f = T_C \cup T_{P_f}$ be the set of types present in f. Let the power set of T_f be ordered such that for two sets $A, B \in 2^{T_f}$, $A \leq B$ iff $A \supset B$. The result is a lattice $\mathcal{L}(f)$ bounded by the null set ($\top$) and by T_f itself ($\bot$). Each element of this lattice represents a subset of T_f. The lattice for a program with two C types τ_1 and τ_2 and a placeholder type π can be seen in Figure 1.

3.4. Intuition

Type based alias analysis is safe so long as the restrictions in Definition 1 are respected for each memory access. Memory accesses in C are represented by lvalues; thus, type-based

$$\begin{tikzcd}
 & \top \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\{\tau_1\} \arrow[d, dash] \arrow[dr, dash] & \{\tau_2\} \arrow[dl, dash] \arrow[dr, dash] & \{\pi\} \arrow[dl, dash] \arrow[d, dash] \\
\{\tau_1, \tau_2\} \arrow[dr, dash] & \{\tau_1, \pi\} \arrow[d, dash] & \{\tau_2, \pi\} \arrow[dl, dash] \\
 & \bot &
\end{tikzcd}$$

Figure 1. Type Lattice

```
lval  ::=  id
        |  lval.id
        |  * lval
```

Figure 2. Syntax of lvalues

alias analysis is safe for a program P if $\mathsf{safe}(l)$ is true for every lvalue l accessed by P. The syntax of lvalues is shown in Figure 2. There are three cases: identifiers, field accesses, and pointer dereferences. (Language constructs such as array accesses and the `->` operator are syntactic sugar for these basic operations.) For identifiers, $\mathsf{safe}(l)$ is trivially true. The object in memory accessed by the expression `v` will always have the same effective type as the declared type of `v`, by definition.[†] By the same token, for field accesses, $\mathsf{safe}(v.f)$ is true iff $\mathsf{safe}(v)$ is true.

The possibility of type mismatches comes from pointer dereferences. Although the expression `v` is safe for every variable v, the expression `*v` is only safe if `v` currently points to an object it is permitted to access. Thus, a program is safe unless it dereferences a pointer pointing to an object of an incompatible type. To determine whether or not a program violates the C standard's type-based restrictions on memory, it is sufficient to determine whether such a pointer is ever dereferenced. In turn, the presence or absence of such violations in a program is sufficient to determine whether the application of type-based alias analysis is sound.

It is infeasible at compile time to determine every possible value that may be taken on by a dereferenced pointer in a program. SafeType approaches the problem from the opposite direction; it attempts to verify that no unsafe pointers are created in the first place. Consider an expression `*`e in a program P. If the dereference of e is unsafe, then e must fall into one of two cases:

1. e contains a type-cast expression that creates an unsafe pointer. For example, if `i` is a variable with type `int`, the expression `(double *) &i` creates a pointer that cannot be legally dereferenced.
2. e loads a pointer value from memory that points to an object of an incompatible type.

The underlying principle of these cases is that unsafe pointers do not appear out of nowhere. They must be created at some point. If no unsafe pointer is ever created, then no unsafe pointer can ever be dereferenced. Thus, if SafeType can verify that each individual statement in a program preserves the invariant that each non-null, non-void pointer can be

[†]Although it may not be safe to dereference a pointer `v`, it is always safe to load the value of that pointer.

safely dereferenced, then it can conclude that at each dereference `*`e in P, $\mathsf{safe}(*e)$ is true. More formally:

Definition 4
Given a state of program execution $\mathcal{S}$, the safe-memory property $\mathsf{safe\text{-}mem}(\mathcal{S})$ is true if and only if for every live pointer variable v, $\mathsf{safe}(*v)$ is true – that is, for every object o in memory whose effective type is a pointer type $*\tau_{\text{ptr}}$, either o is a null pointer, or o is a void pointer, or the value of o is the address of an object in memory with effective type τ_{obj} such that $\tau_{\text{ptr}} \triangleright \tau_{\text{obj}}$. By definition, $\mathsf{safe\text{-}mem}(\mathcal{S}_0)$ is true for the initial state of a program $\mathcal{S}_0$.

For a statement s, let $\mathcal{S}_s$ and $\mathcal{S}'_s$ be the program state immediately prior to and immediately following the execution of s, respectively. SafeType attempts to verify for all s that $\mathsf{safe\text{-}mem}(\mathcal{S}_s) \implies \mathsf{safe\text{-}mem}(\mathcal{S}'_s)$. This verification takes the form of an analysis of assignment statements, and is described in Section 3.5. Each assignment statement can be examined independently to determine whether or not it preserves the safe-mem invariant, making the bulk of this analysis flow- and context-insensitive. However, precisely determining the effective type of an expression containing variables of type `void *` requires flow- and context-sensitive queries, which are described in Section 4.

One additional benefit of this focus on the creation rather than the dereference of unsafe pointers is that, in the presence of violations, it more accurately identifies the responsible statements in the source code. When a bug causes a program to unexpectedly dereference a null pointer, the dereference itself is rarely the source of the bug. It is more useful to identify the statement that caused the pointer to be null. The same is true when detecting violations of the C standard's restrictions on memory access.

3.5. The SafeType Analysis

Section 3.4 justifies an approach in which each assignment statement is inspected to determine whether it may create a pointer in memory that cannot be legally dereferenced. SafeType examines four types of assignment statements. Each statement generates one or more constraints.

- **Explicit assignments:** for an lvalue l of type τ_l and an expression e of type τ_e, an assignment of the form $l = e$ generates the constraint $\tau_l \triangleright \tau_e$.
- **Function calls:** for a function f with return type τ_R and parameter types $\{\tau_{f_1}, \ldots, \tau_{f_n}\}$ and a set of expressions $\{e_1, \ldots, e_n\}$ of types $\{\tau_{e_1}, \ldots, \tau_{e_n}\}$, a function call of the form $f(e_1, \ldots, e_n)$ generates the set of constraints $\{\tau_{f_i} \triangleright \tau_{e_i} \mid 1 \leq i \leq n\}$.
- **Return statements:** for an expression e of type τ_e evaluated in a function f with return type τ_R, a return statement of the form `return` e generates the constraint $\tau_R \triangleright \tau_e$.
- **Type casts:** for a type τ_c and an expression e of type τ_e, a typecast of the form (τ_c) e generates the constraint $\tau_c \triangleright \tau_e$.

Each constraint is evaluated according to the definition of $\triangleright$ in Section 3.2. When a void pointer exists on the right-hand side of an assignment, the effective type of the value being assigned requires flow-sensitive analysis to determine, and its type in the constraint is represented by a placeholder type. Constraints involving only regular C types are evaluated immediately. The evaluation of constraints involving placeholder types must be delayed, as described in Section 4.2. For any constraint that is not satisfied, a warning is produced to indicate that the assignment may make type-based alias analysis unsound. This warning includes the location of the statement in the source code, and the two types being compared.

These warnings can then be used by the programmer to correct the source code. In many cases, this can be accomplished with little effort by changing the type of a variable to `char` or by declaring a union. By making use of the exceptions that are provided by the C standard, these simple changes can convey the necessary information to a standard-compliant type-based alias analysis without requiring a complete overhaul of the program.

Listing 1. Non-compliant

```
1  void *vp;
2  int i;
3  double *dp;
4
5  vp = &i;
6  dp = (double *) vp;
7  *dp = 1.0;
```

Listing 2. Compliant

```
1  void *vp;
2  double d;
3  double *dp;
4
5  vp = &d;
6  dp = (double *) vp;
7  *dp = 1.0;
```

Figure 3. Void Pointers

4. DEALING WITH VOID POINTERS

This section explains the need for a flow-sensitive analysis to correctly handle void pointers, and describes that analysis. Section 4.2 extends that analysis to be fully context-sensitive.

4.1. Flow-Sensitive Analysis

The preceding discussion implicitly assumes that it is possible to assign an effective type to each expression. However, this assumption is not true for variables of type `void *`. A void pointer can never be dereferenced without casting it to a non-void type, but is permitted to point to any object in memory. This flexibility creates a problem. Consider the example in Listing 1. The code includes three assignments: the assignment of `&i` to `vp`, the cast of `vp` to `double *`, and the assignment of the cast expression to `dp`. This code violates the C standard. After line 6, `dp` points to `i`. When `dp` is dereferenced on line 7, an object of type `int` is accessed through an lvalue of type `double`, which violates the rules laid out in Definition 1. Conversely, in Listing 2, `dp` points to d, with type `double`, and the dereference in line 7 complies with the C standard.

As described above, the safe-mem invariant is maintained by ensuring that each pointer always points to an object that it is permitted to access. On its own, this approach is insufficient for void pointers, because they can point to any address and can never be dereferenced. As can be seen in Listings 1 and 2, the type of a void pointer, at the point at which it is cast to a non-void pointer, is a flow-sensitive property depending on previous assignments. It is therefore necessary, in the specific case of void pointers, to extend SafeType to make it flow-sensitive. This accurate treatment of void pointers is particularly important when dealing with custom memory management routines.

Flow-sensitivity is attained in SafeType using on-demand flow-sensitive queries. Consider an expression e that uses a variable v with declared type τ. If $\tau =$ `void *`, then a flow-sensitive query is necessary to determine the actual type of the object to which v points.

Definition 5
Given a variable v and a definition d of that variable (such that $d \in \mathcal{D}(v, s)$ for some statement s), the types function $\mathcal{T}$ is defined such that $\mathcal{T}(v, d)$ is a lattice element representing the possible types of the value assigned to v by definition d.

This section describes the implementation of $\mathcal{T}$ in the case where d is an assignment statement. Section 4.2 describes the implementation of $\mathcal{T}$ in the case where d is a function call statement or a function entry point.

Definition 6
Given a variable v and a statement s, the flow-sensitive query function $\mathcal{F}(v, s)$ is defined as follows:

$$\mathcal{F}(v, s) = \bigvee_{d \in \mathcal{D}(v,s)} \mathcal{T}(v, d)$$

The definition of $\mathcal{T}(v, d)$ is straightforward for assignment statements. Given a statement $v = e$, if e has type τ_e, then $\mathcal{T}(v, v = e) = \{\tau_e\}$. If e also uses a void pointer, additional flow-sensitive queries may be necessary to determine τ_e. SafeType uses a worklist algorithm and caches results to enable flow-sensitive queries to be performed efficiently.

4.2. Context-Sensitive Analysis

The definition of $\mathcal{T}$ in Section 4.1 is sufficient for void pointers that are defined and used entirely within a single function. A flow-sensitive query $\mathcal{F}(v, s)$ for a variable v of type `void *` used in a statement s may require interprocedural data-flow in two situations:

1. A call site defines v, and that definition reaches s.
2. v is live upon entry to the function, and the entry definition reaches s.

To handle these cases, SafeType uses two ideas: placeholder types, and summary functions. Placeholder types are used to represent the unknown types of void pointer variables which are live upon function entry. Each placeholder type is defined in the context of a single function. For a function f, placeholder types are used in two cases:

1. A formal parameter v_p has type `void *`. For any call to f, the type of the object to which v_p points depends on the expression assigned to that parameter at the call site.
2. A global variable v_g used in a function has type `void *`. For any call to f, the type of the object to which v_g points depends on the value of that global variable at the point immediately preceding the call site.

Each function f has a set of placeholder types T_{P_f}. Let V_p be the set of formal parameters of f with type `void *`. Let V_g be the set of global variables with type `void *` that are used or defined in f, or in a function called by f. For each variable $v \in V_g \cup V_p$, the placeholder type $\pi_v \in T_{P_f}$ represents the type of v for an arbitrary call to f.

Summary functions represent the effects of a function call at a call site. These effects include changes to the effective type of global variables, which may be necessary for flow-sensitive queries. Summary functions also contain delayed constraints that must be verified but cannot be evaluated without information from the calling context. Unlike points-to analysis, which depends on the value of each pointer and is not amenable to succinct summarization [15], SafeType concerns itself with types. As a result, it can represent each function in a compact form, and use that summary at each individual call site to efficiently achieve full context-sensitivity. For a function f, the summary function $T(f)$ is a function $A \to (\tau_R, C, E)$, where:

- A is an assignment of types to the placeholder types in f, such that for each input variable $v \in V_g \cup V_p$, $A(v)$ is a type τ_{in_v}.
- τ_R is the return type of F.
- C is a set of delayed constraints of the form $\tau_1 \triangleright \tau_2$, where τ_2 includes at least one placeholder type $\pi \in T_{P_f}$.
- E is a map representing the exit types of each global variable used or defined in f, such that for each input variable $v \in V_g$, $E(v)$ is a type τ_{out_v}.

For a function f, the summary function $T(f)$ is constructed while performing the analysis of f:

- **Return type:** If $S_R = \{$`return` $e_1, \ldots,$ `return` $e_n\}$ is the set of return statements in f, and the type of each e_i is τ_i, then

$$\tau_R = \bigvee_{s \in S_R} \tau_i$$

- **Delayed Constraints:** If a constraint $\tau_1 \triangleright \tau_2$ is generated such that τ_2 is a placeholder type, it cannot be evaluated outside a particular calling context. Instead, the constraint is added to C. It is subsequently evaluated at each call site of f independently.

- **Exit types:** For each variable v in V_g, $E(v) = \mathcal{F}(v, s_{\text{exit}})$, where s_{exit} is a dummy node representing the exit of f.

Each of the three outputs of a summary function may include one or more placeholder types. Where this is the case, each placeholder type is replaced with the actual type of the corresponding input argument at the call site. The type of a formal parameter is available directly from the call statement. When the type of a global variable is required, a flow-sensitive query is performed for that variable, starting at the call site. This enables delayed comparisons to be correctly evaluated, and also allows for the return type or exit types of a called function to depend on the inputs to that function at the call site.

The use of summary functions makes it possible to define the types function $\mathcal{T}$ for function call statements and function entry points. Given a statement $f(e_1, \ldots, e_n)$ which defines a variable v, if the type of each e_i is τ_i and $T(f) = (\tau_R, C, E)$, then $\mathcal{T}(v, f(e_1, \ldots, e_n)) = E(v)$. For a function entry point d that defines v, $\mathcal{T}(v, d) = \pi_v$.

To ensure that the summary function for a called function f is available at its call site in each calling function g, SafeType uses a post-order traversal of the call graph to generate the order in which each function is analyzed. However, self-recursive or mutually recursive functions create cycles in the call graph and make a post-order traversal impossible. To solve this problem, SafeType collapses strongly connected components of the call graph into a single node, and uses a post-order traversal on the modified call graph (which is cycle-free).

Within strongly connected components, summary functions may have cyclic dependencies. SafeType therefore uses a worklist algorithm to find a fixed point for the summary functions. This algorithm works as follows. Initially, each summary function is initialized to be blank: $\tau_R = \top, C = \emptyset$, and for all $v \in V_g$, $E(v) = \top$. Each function in the strongly connected component is analyzed. Whenever the analysis of a function causes its summary function to change, the callers of that function are inserted into the worklist for reanalysis. This continues until the worklist is empty, meaning that a fixed point has been reached.

Termination is addressed in three ways. First: define the size of a summary function $T(f)$ as the sum of the number of possible return types in τ_R, the number of delayed comparisons in C, and the total number of possible exit types in E. The size of a summary function is monotically increasing over the course of the analysis: return types and exit types may increase in size, and new delayed comparisons may be added, but existing types and comparisons are never removed. If T_f is the (finite) set of types present in f and T_{P_f} is the (finite) set of placeholder types present in f, then the maximum size of a summary function $T(f)$ is reached when the return type $\tau_R = \bot$, $E(v) = \bot$ for all $v \in V_g$, and a delayed constraint $\tau_1 \triangleright \tau_2$ exists for each $\tau_1 \in T_f$, $\tau_2 \in T_{P_f}$. Because the size of each summary function monotonically increases towards a finite maximum, termination is guaranteed. Second: to ensure termination in practice, an iteration count is maintained. If the analysis of a strongly connected component exceeds a set number of iterations (which scales with the size of the strongly connected component), a warning is produced. Each summary function in the strongly connected component is then conservatively approximated such that $\tau_R = \bot$, $E(v) = \bot$ for all $v \in V_g$, and $C = \emptyset$. Third: as described in Section 5.3, it is empirically rare that this algorithm causes a function to be analyzed more than twice. Although the conservative approximation described above could cause spurious warnings to appear at each call site of an approximated function, in practice the iteration limit is simply precautionary and has no impact on the results of the analysis.

4.3. Complexity

The complexity of the non-flow-sensitive, non-context-sensitive component of SafeType is linear in the size of the program. It examines each statement once and performs an amount of work proportional to the size of the abstract syntax tree for that statement.

The complexity of the flow-sensitive component of SafeType is implementation-dependent. It is dominated by the time spent finding the definitions for each query variable.

The complexity of the context-sensitive component is more complicated to characterize. As described in Section 4.2, recursive functions must be analyzed until their summary functions reach a fixed point. Each function must be analyzed at least once. If the analysis of a function F updates the summary function for F, then the callers of F must be reanalyzed using the more precise summary function. Thus, after the first analysis of F, F will be reanalyzed only if the previous analysis of F updated the summary function for F and (directly or indirectly) caused an update to a callee of F. For a strongly connected component of the call graph containing $O(n)$ functions, each of which has $O(m)$ arguments, the pathological worst-case behaviour of the fixed point computation is $O(nmt)$ reanalyses of each function, where t is the number of types in the program. This pathological behaviour is, however, not present in real-world programs.

SafeType is most effective when performed as a whole-program analysis. The prototype implementation of SafeType discussed in Section 5 uses a whole-program analysis. When whole-program analysis is not possible or desirable, a single-compilation-unit implementation of SafeType must use summary functions that make conservative approximations about the effect functions of called functions in other compilation units.

4.4. Example

Consider the following (highly contrived) code:

Listing 3: Context-sensitive Example

```
 1  void *g_in, *g_out;
 2
 3  void *foo(void *p_in, void *cond) {
 4   g_out = p_in;
 5   int i = *(int *) cond;
 6   if (i)
 7    return p_in;
 8   else
 9    return g_in;
10  }
```

- `foo` may return either `p_in` or `g_in`. The return type of `foo` is therefore the lattice element $\pi_{\texttt{p\_in}} \vee \pi_{\texttt{g\_in}}$. The values of these placeholder types will vary between call sites of `foo`.
- The only global variable defined in `foo` is `g_out`. The exit type of `g_out` is $E(\texttt{g\_out}) = \pi_{\texttt{p\_in}}$. The global variable `g_in` is used, but not defined, in `foo`. Its exit type is $E(\texttt{g\_in}) = \pi_{\texttt{g\_in}}$.
- There is one delayed constraint in `foo`, arising in line 5: $\texttt{int *} \triangleright \pi_{\texttt{cond}}$. At each call site of `foo`, that constraint must be evaluated to verify the safety of casting `cond` to an `int *`.

The combination of the return type, the exit types, and the delayed constraints of `foo` are sufficient to create a summary function that precisely characterizes the behaviour of `foo`. Using that summary function, the effects of a call to `foo` can be determined for any number of call sites without requiring any further analysis of `foo`.

5. EXPERIMENTAL EVALUATION

This experimental evaluation uses a 64-bit, 32-processor, 2.3 GHz POWER 5 machine with 640 GB of memory running AIX 6.1.7.15. SafeType is evaluated on each of the C benchmarks from SPEC CPU2006. Additionally, SafeType is evaluated on Python 2.7.5 and GNU Emacs 24.3 to increase the number of large programs in the test set. Other large programs were considered, but were incompatible either with the XL compiler (for example, the Linux

Listing 4. Violation in 470.lbm

```
1  double grid[N];
2  int flag = 1 << M;
3  {
4    int* const _aux_ =
5      ((int*) ((void*) (&(grid[...]))));
6    (*_aux_) |= flag;
7  }
```

kernel) or with the POWER architecture (for example, WINE). Lines of code are counted using the open-source tool CLOC. For `emacs` and `python`, lines of code written in Emacs Lisp and Python, respectively, are subtracted from the overall total, because SafeType is only applicable to the portion of a program written in C. Although the principles of SafeType should apply to C++ programs, an extension would be required to deal with the entire C++ type system. In particular, SafeType is not equipped to verify the safety of downcasts from a base type to a derived type.

During compilation, compiler flags are chosen to disable other optimizations and minimize the amount of work done outside SafeType. Although this affects the reported results by increasing the relative percentage of compilation time spent performing the SafeType analysis, the disabling of other optimizations better reflects the expected use case of SafeType. To speed development, programs are typically compiled and tested with a minimal set of optimizations. After a program has been written and tested, higher levels of optimization are used to create a production version of the program. SafeType is best used during the transition from development to optimization, to verify that it is safe to use type-based alias analysis to enable the transformations that are turned on at higher levels of optimization. The actual compilation of the optimized version of a program should not use SafeType. Instead, the optimized compilation will benefit from being able to safely use a type-based alias analysis to enable other optimizations.

5.1. Results

This section discusses a selection of the possible violations identified by SafeType, including both true and false positives. True positives are statements reported as potential violations that actually violate the C standard's type-based restrictions on memory access. These are the statements that SafeType is designed to detect. If a program does not have any true positives, it is safe to use a type-based alias analysis while compiling that program. False positives are assignments that are flagged as potentially unsafe, but that are actually safe.

lbm: SafeType identifies a true type violation in `470.lbm` that, to our knowledge, has not been previously reported. The header file `lbm_1d_array.h` defines a macro `SET_FLAG`. A use of that macro expands to the code in Listing 4. The variable `grid` is an array of `double`. The address of a member of `grid` is taken in line 5. That pointer is cast, first to `void *`, then to `int *`. Finally, the new `int *` value is used to set a flag bit. That use constitutes a modification of a `double` object in memory by an lvalue of type `int`; as described in Section 3, such a modification is a violation of the C standard. Although we are not aware of a compiler that takes advantage of the incorrect invariant generated by a type-based alias analysis in this case, a sufficiently aggressive optimizing compiler could theoretically eliminate or reorder uses of `SET_FLAG` in a way that does not preserve the semantics of the program.

Python and Perl: The implementation of Python objects in versions of Python prior to 3.0 violated the C standard's restrictions on memory access, and the internal data structures representing Perl's variables are accessed in such a way as to violate aliasing rules in `400.perlbench`. SafeType correctly identifies these violations. These violations typically take the form of a struct of one type being cast to a struct of another type. In addition to the

Listing 5. False Positive in 400.perl

```
 1  struct SV {
 2      void* sv_any;        /* pointer to something */
 3      U32   sv_refcnt;     /* how many references to us */
 4      U32   sv_flags;      /* what we are */
 5  };
 6
 7  struct SV sv;
 8  ...
 9  if (sv->sv_flags == ...) {
10    ...
11  }
```

Listing 6. False Positive in 403.gcc

```
1  struct basic_block_def {
2    ...
3    struct edge_def *succ;
4    ...
5  };
6
7  struct basic_block_def *block, *head;
8  block->succ = (struct edge_def *) head;
```

large number of true positives, `400.perlbench` also includes an example of an important type of false positive. As seen in Listing 5, `400.perlbench` uses tagged void pointers to represent scalar values. The `SV` struct includes a void pointer (`sv_any`) that represents a value, and a tag (`sv_flags`) to indicate the type of the variable being pointed to. The value of `sv_flags` is used in control-flow statements to determine the type of `sv_any`, as in line 9 of Listing 5. This code construct causes difficulties for SafeType. As a result of the test in line 9, the block beginning at line 10 may be able to safely cast `sv_any` to a particular type. However, to determine if such a cast is safe, SafeType would have to examine the condition in line 9, and then prove the existence of an invariant relationship between that condition and the type of the object pointed to by `sv_any`. In the general case, computing such invariants is infeasible. As a result, it is necessary for SafeType to conservatively emit a warning for a cast that occurs within the block beginning on line 10.

gcc: gcc contains another example of a false positive. Listing 6 demonstrates the situation. In `cfg.c`, gcc manages a pool of structs representing basic blocks. In the functions `expunge_block_nocompact` and `alloc_block`, unused basic blocks are added to and removed from a linked list. To save memory, gcc uses a pointer variable that already exists as a member of the `basic_block` struct to store the links in the linked list. However, the member in question is defined as a pointer to a different type of struct, the `edge_def` struct, which represents an edge between basic blocks. SafeType identifies the assignment in line 8 of Listing 6 as a violation. However, because the value of block-¿succ is never dereferenced, no actual violation occurs.

5.2. Missing Type Information

As described in Section 3.3, SafeType requires data-flow and call-graph information to accurately perform flow- and context-sensitive analysis. In our prototype implementation, this information is not available until the optimization phase of the compilation, after the source code has already been converted into an internal representation. This internal representation is language-independent, and does not preserve all of the type information available in the source code. This impedes the ability of SafeType to determine the declared

Table I. Violations Detected by SafeType

Name	**Total**	**# of True Positives**	**# of False Positives Safe Type**	**in Prototype**
400.perlbench	52938	3463	166	49309
401.bzip2	14	0	0	14
403.gcc	8380	0	2	8380
429.mcf	13	0	0	13
433.milc	180	0	0	180
445.gobmk	5737	0	0	5737
456.hmmer	53	0	0	53
458.sjeng	325	0	0	325
462.libquantum	0	0	0	0
464.h264ref	2723	0	0	2723
470.lbm	57	30	0	27
482.sphinx3	1153	0	0	1153
emacs	7167	0	0	7167
python	16554	9089	0	7465

type of some symbols representing pointers. To remain sound, it is therefore necessary for this implementation of SafeType to conservatively emit a warning in cases where the missing type information is necessary to prove safety.

Manual examination of the output indicates that these additional warnings are a limitation of the prototype implementation, rather than the underlying SafeType analysis. A mature implementation of SafeType could eliminate these spurious warnings by ensuring that the front end of the compiler preserves precise type information for every symbol, and communicates that information to the SafeType analysis.

Table I shows the number of false positives that are due to the SafeType analysis and the ones that are reported in the prototype. The majority of these can be easily identified with an automated post-processing script, a few required manual inspection to determine that they were false positives. The frequency of these implementation-specific false positives is significantly higher than the frequency of warnings produced by the SafeType algorithm itself. Overall, on the benchmarks studied, over 80% of the emitted warnings were due to the limitations of the prototype. Of the remaining warnings, 98% occurred in `python` and `400.perlbench`, benchmarks already known to violate aliasing rules. Examination of these warnings resulted in the discovery of the false positives in `400.perlbench` described above. The remainder appear to be true positives, although it is likely that several false positives were missed. The remaining warnings occurred in `403.gcc` and `470.lbm`, as described above. It is likely that a mature implementation of the SafeType algorithm would have emitted additional warnings that were masked by the limitations of the prototype implementation. However, based on our exploration of these results, we cautiously predict that the number of false positives emitted by a mature implementation of SafeType would be manageable.

5.3. Performance

Table II shows the time necessary to run SafeType on each benchmark. Column **LoC** is the number of lines of code in each benchmark. Column **Total** is the time taken to compile the benchmark. The following two columns show the time spent inside SafeType: first as an absolute value, and then as a percentage of the total compilation time. As mentioned above, compilation time is measured for a compilation with optimizations turned off. With less compilation time spent on other optimizations, the percentage of compilation time spent inside SafeType is higher.

Table II. Performance of SafeType

Name	**LoC**	**Total (s)**	**SafeType (s)**	**%**
400.perlbench	138480	3939.0	3819.9	96.9
401.bzip2	5882	10.6	1.7	15.9
403.gcc	387777	650.9	304.8	46.8
429.mcf	1669	2.7	0.1	3.5
433.milc	9746	15.5	1.1	7.2
445.gobmk	157792	109.6	25.8	23.5
456.hmmer	20793	29.1	1.8	6.1
458.sjeng	10630	13.8	1.0	7.2
462.libquantum	2732	4.2	0.2	4.3
464.h264ref	36162	74.8	20.7	27.7
470.lbm	1006	2.9	0.15	5.1
482.sphinx3	13240	22.2	1.6	13.9
emacs	326779	338.2	90.1	26.7
python	514345	390.8	263.9	67.5

Table III. Timing Breakdown

	Flow-Sensitive			**Context-Sensitive**	
Name	**Data (%)**	**FSQuery (%)**	**Other (%)**	**Rec Time (%)**	**Rec Freq (%)**
400.perlbench	0.4	95.3	4.3	99.6	81.3
401.bzip2	59.8	21.0	19.2	0.0	0.0
403.gcc	22.3	38.6	39.1	74.3	75.5
429.mcf	38.6	10.9	50.5	2.0	2.8
433.milc	33.6	28.2	38.2	0.0	0.0
445.gobmk	18.7	32.0	49.3	76.3	80.9
456.hmmer	38.1	20.4	41.5	3.8	2.9
458.sjeng	27.0	18.6	54.4	10.0	4.4
462.libquantum	43.7	10.9	45.4	9.8	11.3
464.h264ref	20.9	50.1	29.0	2.5	6.7
470.lbm	24.1	23.1	52.8	0.0	0.0
482.sphinx3	24.3	11.9	63.8	1.6	2.9
emacs	16.3	13.8	69.9	75.9	70.3
python	7.7	55.8	36.5	91.8	82.4

For the majority of programs in Table II, performance is entirely reasonable. The clear outlier is `400.perlbench`, which spends 96.9% of its compilation time inside SafeType. As described below, this appears to be caused by inefficiencies in the analysis of calls through function pointers. There are several reasons to believe that the performance of `400.perlbench` is not a problem for SafeType as a whole. First, these measurements must be taken in the context of an unoptimized prototype. There exists significant low-hanging that a more performance-oriented implementation could use to reduce execution time. Second, `400.perlbench` is not a good representative of the class of programs for which SafeType is expected to be useful. SafeType is best used on programs with few or no known violations, to identify whether changes must be made to make type-based alias analysis safe. It is less important to be able to identify each of the many known violations in `400.perlbench`. The two benchmarks that spend the greatest percentage of their compilation time executing SafeType are `400.perlbench` and `python`, the benchmarks that were already known to be unsafe for type-based alias analysis. Third, as described above, SafeType is not intended to be used during every compilation. Because SafeType is used relatively infrequently, even a large slowdown is not prohibitive.

Table III categorizes the time spent inside SafeType in two ways. The first three columns divide the time spent inside SafeType into three categories that demonstrate the behaviour of the flow-sensitive analysis. **Data** is the percentage of the execution time of SafeType spent building the data structures necessary to analyze data flow. In a normal compilation, the time spent building these data structures would be amortized over each analysis that uses them. However, as described above, no other analysis using these data structures was enabled during the evaluation. **FSQuery** is the percentage of the execution time of SafeType spent performing flow-sensitive queries as described in Section 4.1. **Other** is the percentage of the execution time of SafeType that does not fall into one of the previous two categories.

The remaining columns of Table III illuminate the behaviour of the context-sensitive analysis. **Rec Time** is the percentage of the execution time of SafeType spent analyzing recursive functions: that is, functions that are part of a strongly connected component in the call graph. This category includes both self-recursive and mutually recursive functions. As a point of comparison, **Rec Freq** is the percentage of functions in the program that are recursive, measured as a static count.

More than half of all recursive functions are analyzed a single time. In the majority of test programs, no function must be analyzed more than twice. The benchmark `445.gobmk` includes three functions that must be analyzed three times each. `emacs` includes a number of functions that must be analyzed three times, and two that must be analyzed four times each. `gcc` includes functions that must be analyzed up to five times. In each of these cases, the functions that must be analyzed more than twice are part of large mutually recursive cycles. Amortized across each of the functions in a strongly connected component of the call graph, the average number of iterations per function necessary to find a fixed point in these cases is always two or less.

This is not true for `400.perlbench` or `python`. Each of these benchmarks has a significant number of functions that must be analyzed six or more times: 329 for `python`, and 182 for `400.perlbench`. As before, higher numbers of iterations are substantially less common than lower numbers. However, `400.perlbench` includes a number of functions that require many iterations to find a fixed point, including one function that requires 33 iterations to find a fixed point. Regardless, amortized over an entire strongly connected component, the average number of iterations per function necessary to find a fixed point in `400.perlbench` or `python` never surpasses 3.5.

Empirically, functions that require a large number of iterations occur in strongly connected components of the call graph that are large and contain a large number of calls through function pointers. Profiling shows that the majority of time that is spent analyzing these functions is spent performing flow-sensitive queries to determine the effective types of global variables at call sites. For example, Python uses a number of global variables of type void* to implement global locks; at the call site for each function that touches a lock, or calls a function that touches a lock, the current effective type for the variable representing that lock must be determined. Further work may be able to improve the performance of the analysis. However, even the current prototype scales to programs of hundreds of thousands of lines of code.

6. RELATED WORK

A closely related idea to alias analysis is points-to analysis. Where an alias analysis determines if two memory references point to the same location, a points-to analysis approximates, for each pointer in the program, the set of locations to which it could point at runtime [16]. Thus, a points-to analysis can be used to determine aliasing information, although the converse is not in general true. Points-to analysis is a rich area of research [17][18][16][19][20][21][22] [23] [15] [24].

SafeType itself, however, is neither an alias analysis nor a points-to analysis. It is an analysis designed to verify the safety of alias analysis. Much work has been done on the question of analyzing the safety of C programs. The most relevant comparisons for SafeType are analyses that aim to detect illegal memory operations: dereferences of undefined pointers, dereferenced of freed memory, and array-bounds violations.

Valgrind is an open-source instrumentation framework for building dynamic analysis tools [25]. Memcheck, the default Valgrind tool, instruments each memory access to detect a variety of common errors, including (among others) invalid pointer dereference and the use of uninitialized values. However, Memcheck is not designed to detect memory accesses through pointers of incompatible type. The addition of this instrumentation leads to a 10-50 times slowdown in program execution. Thus, Valgrind is useful as a developer's tool to detect bugs, rather than being used to run production code.

Cyclone is a type-safe programming language derived from C [26]. Cyclone aims to preserve the low-level control of data representation and resource management available in C, while eliminating safety violations such as incorrect type-casts, buffer overruns, dangling pointer dereferences, and memory leaks. This safety is accomplished by dividing memory into regions and requiring that pointer variables be annotated with information about the region into which they point. Empirically, porting C code into Cyclone requires alterations in approximately 8% of the lines of code in a program. Compared to the original C program, a network or I/O bound Cyclone application runs with little to no overhead, but compute-intensive applications may be three times slower than the C version. This overhead comes from bounds-checks inserted into the code, as well as garbage collection and fat pointers.

CCured is a program transformation system that adds type-safety guarantees to existing C programs. CCured extends the type system of C to separate pointer types based on their usage; it then attempts to use static analysis to verify that programs adhere to that type system. Run-time checks are inserted in cases where static analysis is insufficient: these checks include null checks, bounds checks, and type checks. CCured replaces manual memory management with a garbage collector and requires the programmer to annotate custom allocators. CCured also requires wrappers around calls to external library functions to preserve the metadata associated with runtime checks. Like Cyclone, CCured imposes little overhead on network- and I/O-bound applications, but causes a slowdown of 5 to 87% on most compute-intensive benchmarks, increasing to a ten times slowdown in pathological cases.

SafeCode is a compilation strategy for C programs that uses static analysis and run-time checking to ensure the accuracy of the points-to graph, call graph, and the available type information [4]. SafeCode uses *automatic pool allocation* to partition heap memory into fine-grained pools, while retaining explicit memory management (rather than introducing a garbage collector). SafeCode requires no code modifications and imposes less than 30% overhead. Unlike Cyclone and CCured, SafeCode does not guarantee the absence of dangling pointer references.

In comparison to each of the above, SafeType limits its scope to consider exclusively type violations. In doing so, SafeType does not introduce any runtime checks, does not impose any overhead on execution, and does not require any modifications to existing code. The previous approaches are useful in cases where users wish to exchange performance for memory safety. However, the primary use of type information is in alias analysis, which in turn is used by various transformations to improve performance. Thus, it is useful to have a specialized analysis that a programmer can use to verify the safety of type-based alias analysis without incurring a performance penalty. Certain aspects of the static analysis of CCured and SafeCode resemble the flow-insensitive component of SafeType. However, they do not have any static analogue to SafeType's flow-sensitive, context-sensitive approach to pointers of type `void *`. Instead, CCured and SafeCode verify the safety of such pointers using runtime checks.

Unlike languages such as C++ and Java, C does not include a concept of inheritance, and the C standard does not provide any support for subtypes or supertypes. Despite this, many C programmers simulate inheritance in C using typecasts. In many cases, this situation can be analyzed by extending the C type system to include *physical subtyping* [27][28]. The version of SafeType presented in this paper restricts itself to the type system described in the C standard. However, it would be straightforward to extend SafeType to handle alternative type systems for C by changing the definition of $\triangleright$ used in Section 3.5.

7. CONCLUSION

Type-based alias analysis is a useful tool for enabling and improving the precision of program transformations in optimizing compilers. However, type-based alias analysis is unsafe when applied to programs that violate the C standard's type-based restrictions on memory access. As a result, programs are frequently compiled without the benefits of type-based alias analysis.

SafeType is a sound and scalable static analysis that identifies violations of the C standard's type-based restrictions on memory access. To do so, SafeType verifies that each non-null pointer in the program can be safely dereferenced. This verification uses a flow-insensitive analysis in the general case, augmented with flow-sensitive queries to safely analyze pointer variables of type `void *`. These flow-sensitive queries are made context-sensitive through the use of precise, compact summary functions.

A prototype implementation of SafeType in the IBM XL C compiler was created and evaluated. This prototype implementation demonstrates the utility of the SafeType approach by identifying previously unreported violations in the `470.lbm` benchmark of SPEC CPU2006. The prototype implementation scales to programs with hundreds of thousands of lines of code.

The description of the SafeType approach in this paper, along with the discoveries made with the performance evaluation of the prototype implementation, are a solid basis for a robust and precise analysis that will enable the safe use of type-based alias analysis in commercial compilers.

8. ACKNOWLEDGEMENTS

This work was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Collaborative Research and Development (CRD) grant, by the Alberta Innovates Technology Futures organization, and by the IBM Canada Centre for Advanced Studies.

REFERENCES

1. ISO 9899. Programming languages – C 2000.
2. Reinig AG. Alias analysis in the DEC C and DIGITAL C++ compilers. *Digital Technical Journal* 1998; **10**(1):48–57.
3. Loewis M. ANSI Strict Aliasing and Python. `http://mail.python.org/pipermail/python-dev/2003-July/036909.html` 2003. [Online; accessed 10-November-2013].
4. Dhurjati D, Kowshik S, Adve V. SAFECode: enforcing alias analysis for weakly typed languages. *Programming Language Design and Implementation (PLDI)*, Ottawa, Ontario, Canada, 2006; 144–157.
5. Necula GC, Condit J, Harren M, McPeak S, Weimer W. CCured: type-safe retrofitting of legacy software. *Transactions on Programming Languages and Systems (TOPLAS)* May 2005; **27**(3):477–526.
6. Chase DR, Wegman M, Zadeck FK. Analysis of pointers and structures. *Programming Language Design and Implementation (PLDI)*, White Plains, New York, USA, 1990; 296–310.

7. Diwan A, McKinley KS, Moss JEB. Type-based alias analysis. *Programming Language Design and Implementation (PLDI)*, Montreal, Quebec, Canada, 1998; 106–117.
8. Alias Analysis. `http://gcc.gnu.org/news/alias.html` 1998. [Online; accessed 10-November-2013].
9. Ghiya R. On the importance of points-to analysis and other memory disambiguation methods for C programs. *Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, USA, 2001; 47–58.
10. LLVM 2.9 Release Notes. `http://llvm.org/releases/2.9/docs/ReleaseNotes.html` 2011. [Online; accessed 10-November-2013].
11. Sundaresan V, Hendren L, Razafimahefa C, Vallée-Rai R, Lam P, Gagnon E, Godin C. Practical virtual method call resolution for Java. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Minneapolis, Minnesota, United States, 2000; 264–280.
12. Bravenboer M, Smaragdakis Y. Strictly declarative specification of sophisticated points-to analyses. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, USA, 2009; 243–262.
13. Bacon DF, Sweeney PF. Fast static analysis of C++ virtual function calls. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, San Jose, California, USA, 1996; 324–341.
14. Lhoták O, Hendren L. Scaling java points-to analysis using spark. *Compiler Construction (CC)*, Warsaw, Poland, 2003; 153–169.
15. Wilson RP, Lam MS. Efficient context-sensitive pointer analysis for C programs. *Programming Language Design and Implementation (PLDI)*, La Jolla, California, USA, 1990; 1–12.
16. Emami M, Ghiya R, Hendren LJ. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Programming Language Design and Implementation (PLDI)*, Orlando, Florida, USA, 1994; 242–256.
17. Hind M. Pointer analysis: haven't we solved this problem yet? *Program Analysis for Software Tools and Engineering (PASTE)*, Snowbird, Utah, United States, 2001; 54–61.
18. Andersen L. Program analysis and specialization for the C programming language. PhD Thesis, DIKU, University of Copenhagen May 1994.
19. Hardekopf B, Lin C. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. *Programming Language Design and Implementation (PLDI)*, San Diego, California, USA, 2007; 290–299.
20. Hardekopf B, Lin C. Flow-sensitive pointer analysis for millions of lines of code. *Code Generation and Optimization (CGO)*, Chamonix, France, 2011; 289–298.
21. Landi W, Ryder BG. A safe approximate algorithm for interprocedural aliasing. *Programming Language Design and Implementation (PLDI)*, San Francisco, California, USA, 1992; 235–248.
22. Steensgaard B. Points-to analysis in almost linear time. *Principles of Programming Languages (POPL)*, POPL '96, St. Petersburg Beach, Florida, USA, 1996; 32–41.
23. Whaley J, Lam MS. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *Programming Language Design and Implementation (PLDI)*, Washington DC, USA, 2004; 131–144.
24. Zhu J, Calman S. Symbolic pointer analysis revisited. *Programming Language Design and Implementation (PLDI)*, Washington, DC, USA, 2004; 145–157.
25. Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *Programming Language Design and Implementation (PLDI)*, San Diego, California, USA, 2007; 89–100.
26. Grossman D, Morrisett G, Jim T, Hicks M, Wang Y, Cheney J. Region-based memory management in Cyclone. *Programming Language Design and Implementation (PLDI)*, Berlin, Germany, 2002; 282–293.
27. Chandra S, Reps T. Physical type checking for c. *Program Analysis for Software Tools and Engineering (PASTE)*, Toulouse, France, 1999; 66–75.
28. Siff M, Chandra S, Ball T, Kunchithapadam K, Reps T. Coping with type casts in c. *Foundations of Software Engineering (FSE)*, Toulouse, France, 1999; 180–198.