Using Static Analysis to Reduce Dynamic Analysis Overhead [*]

Suan Hsi Yong (`suan@cs.wisc.edu`)
and Susan Horwitz (`horwitz@cs.wisc.edu`)
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA

Abstract. Dynamic analysis (instrumenting programs with code to detect and prevent errors during program execution) can be an effective approach to debugging, as well as preventing harm from being caused by malicious code. One problem with this approach is the runtime overhead introduced by the instrumentation. We define several techniques that involve using the results of static analysis to identify some cases where instrumentation can safely be removed. While we have designed the techniques with a specific dynamic analysis in mind (that used by the Runtime Type-Checking tool), the ideas may be of more general applicability.

Keywords: Static Analysis, Dynamic Debugging, Runtime Types.

1. Introduction

Languages like C and C++ that allow potentially unsafe operations such as pointer arithmetic, casting, and explicit memory management open the door to many difficult-to-detect errors. To identify such errors, a number of systems have been developed that involve dynamic analysis: instrumenting a program so that errors like out-of-bounds array indexes and bad pointer dereferences are detected when they occur during execution [2, 8, 15, 14, 10, 11]. In some cases, dynamic checks are even mandated by the language definition (e.g., Java guarantees that an exception will be thrown whenever an array index is out of bounds, or a bad cast is performed).

Naturally, the benefits of dynamic analysis have an associated cost: the instrumentation introduces a certain amount of runtime overhead.

[*] This work was supported in part by the National Science Foundation under grants CCR-9970707 and CCR-9987435.

This paper proposes several techniques for reducing the overhead by using static analysis to identify some cases in which instrumentation can safely be omitted. While the techniques were designed for one particular tool: the Runtime Type-Checking (RTC) tool [11], the ideas may be of more general applicability.

The remainder of the paper is organized as follows. Section 2 provides background on the RTC tool. Section 3 describes *type-safety-level analysis*, a flow-insensitive analysis to identify "type-safe" expressions for which instrumentation can be eliminated. Section 4 presents *redundant-check analysis*, a flow-sensitive analysis to identify redundant instrumentation that can safely be removed. Section 5 presents experimental results showing the performance improvements gained from these analyses. Section 6 describes *never-null-dereference analysis*, another dataflow analysis to improve RTC performance. Section 7 discusses related work, and Section 8 concludes.

2. The RTC Tool

The RTC (Runtime Type-Checking) tool instruments C programs so that the runtime type of every memory location is tracked during program execution, and inconsistent type uses cause warnings and errors to be reported. Whenever a value v is written into a location l, l's runtime type is updated with v's runtime type. Also, this runtime type is compared with l's declared type: if they do not match, a *warning* message is issued (a warning message is an indication that unusual behavior has been observed, and may be useful for diagnosing the root cause of a later error). Whenever the value in a location is used, its runtime type is checked, and if the type is inappropriate in the context in which the value is being used, an *error* message is issued; to avoid cascading error messages, the runtime type is set to the correct type after an error message is generated.

2.1. Motivating Examples

While a number of other tools have been proposed to detect out-of-bounds array accesses and bad pointer dereferences, the type-checking approach of the RTC tool can also detect more subtle errors involving type misuses.

EXAMPLE 1. *Bad Union Access:*

A very simple example of a logical error that manifests itself as a bad runtime type is writing into one field of a union and then reading from another field with a different type. This is illustrated by the following code fragment:

```
1. union U { int i; float f; } u;
2. u.i = 10;            /* write into u.i */
3. u.f = u.f + 1.5;     /* read from u.f */
```

In this example, an integer value is written into variable `u` (on line 2), and is subsequently read and used as a float (on line 3). The RTC tool would track at runtime the type of the data stored at the single location corresponding to both `u.i` and `u.f`. That type would be set to *int* after the assignment `u.i = 10` on line 2. On line 3, when the value in that location is read and used as a *float*, the RTC tool would report a type mismatch error, because the runtime type associated with that location is *int*.

EXAMPLE 2. *Simulated Inheritence:*

Another class of subtle errors comes from a programming style in which C programmers simulate classes and inheritance using structures [18]. For example, the following declarations might be used to simulate the declaration of a superclass `Base` and a subclass `Sub`:

```
struct Base { int a1; int *a2; };
struct Sub { int b1; int *b2; char b3; };
```

A function might be written to perform some operation on objects of the superclass:

```
void f ( struct Base *b ) {
   b->a1 = ...
   b->a2 = ...
}
```

and the function might be called with actual arguments either of type `struct Base *` or `struct Sub *`:

```
struct Base base;
struct Sub sub;
f(&base);
f(&sub);
```

The ANSI C standard guarantees that the first field of every structure is stored at offset 0, and that if two structures have a common initial sequence — an initial sequence of one or more fields with compatible types — then corresponding fields in that initial sequence are stored at the same offsets. Thus, in this example, fields `a1` and `b1` are both guaranteed to be at offset 0, and fields `a2` and `b2` are both guaranteed to be at the same offset. Therefore, while the second call, `f(&sub)`, would cause a compile-time warning (which could be averted with an appropriate type cast), it would cause neither a compile-time error nor a runtime error, and the assignments in function `f` would correctly set the values of `sub.b1` and `sub.b2`.

However, the programmer might forget the convention that `struct Sub` is supposed to be a subclass of `struct Base`, and while making changes to the code might change the type of one of the common fields, add a new field to `struct Base` without adding the same field to `struct Sub`, or add a new field to `struct Sub` before field `b2`. For example, suppose a new `int` field, `i1` is added to `struct Sub`:

```
struct Sub { int b1; int i1; int *b2; char b3; };
```

Now, when the second call to `f` is executed, the assignment `b->a2 = ...` would write into the `i1` field of `sub` rather than the `b2` field. The fact that the `b2` field is not correctly set by the call to `f`, or that the `i1` field is overwritten with an unintended value, will probably either lead to a runtime error later in the execution, or cause the program to produce incorrect output.

The tracking of runtime types performed by the RTC tool can help the programmer uncover the source of this logical error. The assignment `b->a2 = ...` causes `sub.i1` to be tagged with type *pointer*. A later use of `sub.i1` in a context that requires an *int* would result in an error message due to the mismatch between the required type (*int*) and the current runtime type (*pointer*).

Note that in this example, a tool like Purify [8] would not report any errors, because there are no bad pointer or array accesses: function `f` is not writing outside the bounds of its structure parameter, it just happens to be the wrong part of that structure from the programmer's point of view.

2.2. Tracking Types

The RTC tool associates with each memory location one of the following runtime types: *unallocated*, *uninitialized*, *pointer*, *zero*, *char*, *short*, *int*, *long*, *float*, *double*. The first two are used to tag unallocated and uninitialized memory respectively; pointers to different types are tagged with the same runtime type *pointer*, and so are treated as compatible with each other; the special *zero* type is used to tag memory locations that are assigned the literal 0, and is treated as being compatible with all other types. The remaining types correspond to C scalar types.[1] For aggregate objects (structures and arrays), the runtime type of each field/element is tracked separately; `typedef`s are resolved to their underlying basic type. The runtime types are stored in a "mirror" of the memory used by the program, with each byte of memory mapped to a four-bit nibble in the mirror (thus incurring a 50% space overhead).

The RTC tool has been implemented to handle all of ANSI C. Using Ckit[5] as its front end, it translates a given set of preprocessed C source

[1] The RTC tool actually represents C scalar types using a pair $\langle \sigma, \mathit{size} \rangle$, where σ is one of $\{\mathit{integral}, \mathit{real}\}$, and *size* is the size (in bytes) of the type. For example, on an x86 Linux, a `char` would be represented by $\langle \mathit{integral}, 1 \rangle$, and a `float` by $\langle \mathit{real}, 4 \rangle$.

This means, in fact, that *int* and *long* are not differentiated, and also that larger types like *long long* can also be represented. In this paper, these subtleties are ignored for simplicity.

files into instrumented C files. These are then compiled and linked with the RTC library, producing an executable that performs runtime type checking and reports error and warning messages.

The instrumentation phase is a source-to-source translation of the C program; it performs a syntax-directed transformation on the program's abstract-syntax tree to add calls to RTC library functions that track the runtime types. The following operations are performed by these library functions:

declare - a variable declaration is instrumented to set the runtime type in the variable's mirror to *uninitialized*. (Initially, the mirror for all memory is implicitly tagged *unallocated*.)

clear - when a variable is deallocated (at function return for stack variables, and during `free` for heap locations), the mirror for that location is tagged *unallocated*.

copy - an assignment statement is instrumented to copy the runtime type of the right-hand-side value into the mirror of the left-hand-side location; additionally, if the runtime type of the assigned value does not match the static type of the assignment, a warning message is issued.

verify - a use of a memory location x in the context of a type τ is instrumented to compare the runtime type in the mirror of x with τ. If the types are not compatible, an error message is issued, and the runtime type of x is corrected to τ (to prevent cascading error messages).

verify-pointer - a pointer dereference is instrumented to check whether the mirror of the pointer's target is *unallocated*. If it is, an error message is issued. This check detects dangling pointer dereferences, dereferences of certain stray pointers (those that point between or beyond allocated blocks), and also null-pointer dereferences (because the mirror of memory location 0 is tagged *unallocated*).

The tool is designed so that instrumented modules can be linked with uninstrumented ones. This flexibility is useful if, for example, a programmer only wants to debug one small component of a large program: they can instrument only the files of interest, and link them with the remaining uninstrumented object modules. A caveat when doing this, however, is that it may lead to spurious warning and error messages because the uninstrumented parts of the code do not maintain the necessary runtime type information for the memory locations they declare or use. For example, if a reference to a valid object declared in the uninstrumented portion of the program is passed to an instrumented function, the tool will consider that object unallocated, and may output a spurious error message if that object is referenced.

This problem extends, in general, to library modules. For example, the runtime types associated with the flow of values in a function like `memcpy`, the initialization of values from input in a function like `fgets`, and the types of the data in a static buffer returned by a function like `ctime` would not be captured. To handle these, we have created a collection of instrumented versions of common library functions that affect type flow. These are wrappers of the original functions, hand-written to perform the necessary tag-update operations in the RTC mirror to capture their type behavior.

Included among these instrumented library functions are memory-management functions. Each call to `malloc` (or one of its relatives) is replaced with a call to a wrapper version which, upon successfully allocating a block of memory, sets the mirror for that memory block to *uninitialized* (or *zero* for `calloc`). Similarly, the wrapper version of the `free` function resets the mirror to *unallocated*. The `malloc` wrapper also adds padding between allocated blocks to decrease the likelihood of a stray pointer jumping from one block to another (this is the approach used by Purify [8]).

The RTC tool was able to detect bugs in some SPEC 95 benchmarks (`go`, `ijpeg`), Solaris utilities (`nroff`, `col`, etc.), and Olden benchmarks (`health`, `voronoi`) [11]. Most of the errors were out-of-bounds array or pointer accesses. In the Solaris utilities, the out-of-bounds accesses resulted in program crashes; in the SPEC cases, the errors had no apparent effect on the execution, which made the errors difficult to detect without the use of a tool like RTC. In every case, the RTC tool was able to detect the out-of-bounds memory accesses because the type associated with the pointed-to memory was different from the expected type.

Finally, the RTC tool lends itself naturally to interactive debugging. When a warning or error message is issued, a signal (`SIGUSR1`) is sent, and can be intercepted by an interactive debugger like GDB [19]. The user can then examine memory locations, including the mirror, and make use of GDB's features to help track down the cause of an error.

A major shortcoming of the RTC tool is poor performance: in the worst case, an instrumented program ran 130 times slower than the non-instrumented version. This is because the RTC tool instruments *every* expression in the program and tracks the runtime type of *every* memory location used in the program. To reduce the overhead of the RTC instrumentation, we implemented two static analyses to identify and remove unnecessary instrumentation: *type-safety-level analysis*, and *redundant checks analysis*.

3. Type-Safety-Level Analysis

Our first static analysis is a flow-insensitive type-safety-level analysis that partitions the expressions in a program into levels of "type safety", so that certain classes of runtime instrumentation can be eliminated for expressions at certain type-safety levels.

For this description, we assume that the assignment statements in the input program have been normalized to the forms defined by the following context-free grammar:

$$\begin{array}{lcl}
\mathit{assign} & \Rightarrow & \mathit{lvalue} = \mathit{rvalue} \\
 & | & \mathit{lvalue} = (\tau)_{\mathit{cvt}}\mathit{rvalue} \\
 & | & \mathit{lvalue} = (\tau)_{\mathit{ext}}\mathit{rvalue} \\
 & | & \mathit{lvalue} = (\tau)_{\mathit{cpy}}\mathit{rvalue} \\
\mathit{lvalue} & \Rightarrow & \mathit{var} \mid *\mathit{var} \\
\mathit{rvalue} & \Rightarrow & \mathit{const} \mid \mathit{var} \mid *\mathit{var} \mid \&\mathit{var} \mid \mathit{var} \oplus \mathit{var}
\end{array}$$

where *const* is a constant, *var* is a variable, and $\oplus$ represents any C binary operator. This simplified language captures the essence of C assignment statements; details of how to handle other C constructs (e.g., structures, unions, and function calls) are omitted for brevity.

An assignment that involves an array index, such as $x = a[i]$, can be rewritten as $tmp = a + i; \; x = *tmp$.

Typecasts are divided into three forms. The first form, $(\tau)_{cvt}e$, is a typecast that involves a change in representation, and includes conversions (e.g., between integers and floating-point values) and truncation of data (e.g., when type-casting a `long int` into a `short int`). The second form, $(\tau)_{ext}e$, represents type-casts that extend data from a smaller type to a larger type with no change in the data bits (e.g., from a `short int` into a `long int`). The third form, $(\tau)_{cpy}e$, represents type-casts where there is no change in the form of the data, and includes casts between pointers and integers (of the same size). The difference between these forms that concerns us is that a conversion cast, $(\tau)_{cvt}e$, is treated by the RTC tool as a use of e that always returns a value of type τ, regardless of whether e's runtime type is compatible with its static type (if it is not, the RTC instrumentation issues an error message, then sets e's runtime type to its static type to avoid cascading error messages). Extension and copy casts $(\tau)_{ext}e$ and $(\tau)_{cpy}e$ are treated as bitwise copies, whose runtime type is the same as the runtime type of e.

3.1. Points-to Analysis

The type-safety-level analysis described below uses points-to analysis to account for possible aliasing in the program. Points-to analysis associates each pointer p with a points-to set, which is the set of variables that p *may* point to during program execution. For our implementation, we used Das's flow-insensitive points-to analysis [6], which is scalable and has good precision. Other (flow-insensitive) points-to analyses can also be used (e.g., [1, 22]). In general, better precision in the points-to analysis would enable the elimination of a greater amount of unnecessary instrumentation, thus potentially leading to greater improvement in the performance of the RTC-instrumented program.

3.2. Type-safety Levels

The main idea behind the analysis is to classify all *lvalue* expressions[2] in the program into the following type-safety levels:

- **safe** - An expression whose runtime type is guaranteed always to be compatible with its static type, and for which all instrumentation can be eliminated (assuming there are no uses of *uninitialized* values; Section 3.6 describes how we identify locations that may be uninitialized).
- **unsafe** - An expression whose runtime type may be incompatible with its static type. This includes expressions of the form `*p`, when the pointer `p` may be NULL, or may contain an invalid address. An *unsafe* expression must be fully instrumented.
- **tracked** - A location whose runtime type is always compatible with its static type, but which may be pointed to by a pointer p such that $*p$ is unsafe. A tracked location needs to have its runtime type initialized (to its static type) in the mirror, but instrumentation for verifying and copying its runtime type can be eliminated.

Figure 1 (a) presents an example code fragment to illustrate the intuition behind the approach. Since the approach is flow-insensitive, the order of the statements is ignored in the analysis.

Figure 1 (b) gives the type-safety levels we wish to identify for the expressions in the example program. The expressions `p0`, `p1` and `p2` are *safe* because they are only assigned pointer-typed values (recall that the RTC tool does not differentiate between pointer types, so the fact that `p1` is assigned both the address of an *int* variable and the address of a *float* variable is not important; also recall that the literal 0 is treated as being compatible with all types, including pointers).

The expressions `f`, `*p0`, `*p1`, and `*p2` are all *unsafe*. Variable `f` is *unsafe* because the assignment at line 13 could write an *int* value into

[2] An expression that represents a location in memory is called an *lvalue* expression. In the simplified language described in this paper, these are either of the form *var* or `*`*var*. In the full C version we have implemented, *lvalue* expressions also include expressions like structure members (*var.id*) and multi-level dereferences (`**`*var*).

(a)	Code	(b) Type-safety levels
1.	`int i;`	*safe*: `p0`, `p1`, `p2`
2.	`int *p0,*p1,*p2;`	*unsafe*: `f`, `*p0`, `*p1`, `*p2`
3.	`float f;`	*tracked*: `i`
		(c) Assignment Edges
4.	`i = 1;`	$i \overset{=}{\longleftarrow} \text{VALUE}_{int}$
5.	`f = 2.3;`	$f \overset{=}{\longleftarrow} \text{VALUE}_{float}$
6.	`p0 = 0;`	$p0 \overset{=}{\longleftarrow} \text{VALUE}_{zero}$
7.	`if(*p0 == 0)`	
8.	`  p1 = &i;`	$p1 \overset{=}{\longleftarrow} \text{VALUE}_{valid\text{-}ptr}$
9.	`else`	
10.	`  p1 = (int *) &f;`	$p1 \overset{=}{\longleftarrow} \text{VALUE}_{valid\text{-}ptr}$
11.	`p2 = p1 + i;`	$p2 \overset{=}{\longleftarrow} \oplus_{ptr} p1 \qquad p2 \overset{=}{\longleftarrow} \oplus_{ptr} i$
12.	`if(*p2 != 0)`	
13.	`  *p1 = 4;`	$*p1 \overset{=}{\longleftarrow} \text{VALUE}_{int}$

Figure 1. Type-safety example.

`f` via `*p1`. The expression `*p0` is *unsafe* because `p0` may be NULL (due to the assignment at line 6); `*p1` and `*p2` are *unsafe* because while they each have a static type of *int*, `*p1` may refer to a *float* (because of the assignment at line 10), and `*p2` may refer to an invalid address (as a result of the pointer arithmetic at line 11).

Finally, the expression `i` is *tracked*. Although `i` will always contain an `int` value, it may be pointed to by `p1`, and `*p1` is unsafe. This means that every use of `*p1` will be instrumented to check its runtime type, and so every location in `p1`'s points-to set — including `i` — must have its runtime type tracked in the mirror.

Within this framework, we can devise schemes of varying precision to determine the type-safety level of each expression. Using the following ordering,

$$\textit{unsafe} < \textit{tracked} < \textit{safe}$$

any scheme that classifies each expression at a level less than or equal to its true level is a safe approximation. For example, the unoptimized RTC tool corresponds to one extreme, where all expressions are considered *unsafe*. The next three sections describe an efficient flow-insensitive analysis to classify the type-safety level of expressions. The analysis works as follows:

Step 1: Build an *assignment graph* in which the nodes represent the expressions in the program, and the edges represent the flow of runtime types due to assignments.

Step 2: Compute a runtime-type attribute, *rt-type*, for each node in the graph.

Step 3: Compute the type-safety level for each *lvalue* node in the graph.

3.3. Step 1: Building the Assignment Graph

The first step of the analysis involves building an assignment graph that records the flow of runtime types among the expressions in the program. Each node in the assignment graph corresponds to an expression, and represents an "abstract object" of one of four forms: v, $*v$, $\oplus_\tau v$, and VALUE_τ. The v node represents a variable v; $*v$ represents a dereference; $\oplus_\tau v$ represents an arithmetic, logical, or bitwise operation on v (resulting in a value of static type τ); and VALUE_τ represents a value of type τ, e.g., from a constant expression. For both $\oplus_\tau v$, and VALUE_τ, τ will be either a scalar C type — *char*, *int*, *float*, etc. — or one of the following special types:

***valid-ptr*:** A pointer expression that is guaranteed to evaluate to a valid address (the address of an allocated memory location) has type *valid-ptr*. For example, the expression `&x` has type *valid-ptr*.

***pointer*:** A pointer expression that may evaluate to an invalid address (including NULL) has type *pointer*. For example, the expression `&x + k` has type *pointer*.

***zero*:** An expression that is guaranteed to evaluate to 0 has type *zero*. For example, the literal `0` has type *zero*.

expr	*AbsObj*(*expr*)	Notes
0	$\{\text{VALUE}_{zero}\}$	
C_τ	$\{\text{VALUE}_{\tau}\}$	C_τ is a non-zero constant of type τ
$\&y$	$\{\text{VALUE}_{valid\text{-}ptr}\}$	
y	$\{y\}$	
$*y$	$\{*y\}$	
$y \oplus z$	$\{\oplus_\tau y, \oplus_\tau z\}$	The expression $y \oplus z$ has static type τ

(a)

Assignment	Edge(s) in Graph
$e_1 = e_2$	$n_1 \xleftarrow{=} n_2$, $n_1 \in AbsObj(e_1)$, $n_2 \in AbsObj(e_2)$
$e_1 = (\tau)_{cpy} e_2$	$n_1 \xleftarrow{=} n_2$, $n_1 \in AbsObj(e_1)$, $n_2 \in AbsObj(e_2)$
$e_1 = (\tau)_{cvt} e_2$	$n_1 \xleftarrow{cvt} n_2$, $n_1 \in AbsObj(e_1)$, $n_2 \in AbsObj(e_2)$
$e_1 = (\tau)_{ext} e_2$	$n_1 \xleftarrow{ext} n_2$, $n_1 \in AbsObj(e_1)$, $n_2 \in AbsObj(e_2)$

(b)

Figure 2. Rules for initializing the assignment graph.

Nodes are connected by three kinds of (directed) assignment edges: *conversion edges* ($\xrightarrow{cvt}$), *extension edges* ($\xrightarrow{ext}$), and *copy edges* ($\xrightarrow{=}$). Conversion edges represent assignments with a right-hand side of the form $(\tau)_{cvt}e$, extension edges represent assignments with a right-hand side of the form $(\tau)_{ext}e$, and copy edges represent assignments that do not involve a type-cast, or that involve a type-cast of the form $(\tau)_{cpy}e$.

Figure 2(a) shows the mapping *AbsObj* from program expressions to abstract objects, while Figure 2(b) gives the rules for adding edges to the graph. For example, the assignment $x = y \oplus z$ adds to the graph two copy edges, $x \xleftarrow{=} \oplus_\tau y$ and $x \xleftarrow{=} \oplus_\tau z$, where τ is the static type of the expression $y \oplus z$.

Figure 1 (c) gives the assignment edges derived from the the example program.

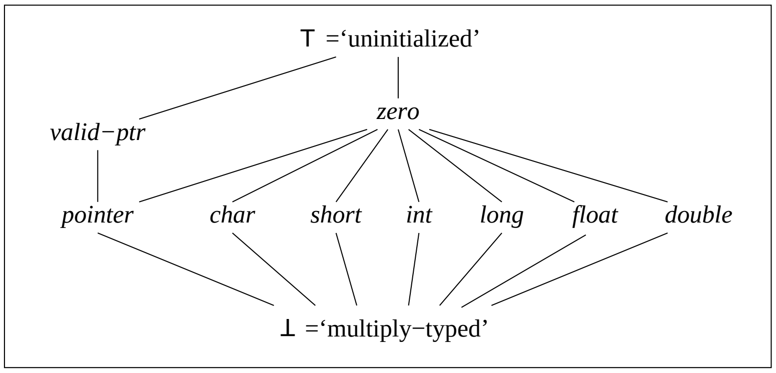


Figure 3. The lattice for *rt-type*.

3.4. Step 2: Computing Runtime Types

After building the assignment graph, the analysis computes a runtime-type attribute, *rt-type*(n), for each node n in the graph. The values of *rt-type* form the lattice shown in Figure 3. Intuitively, *rt-type*(n) summarizes the set of types that the expression corresponding to n might have at runtime. A *rt-type* of $\perp$ means that an expression could have more than one incompatible runtime type.

Figure 4 gives the constraints for computing *rt-type* for each node in the assignment graph. In the figure, *pt-set*(p) is the points-to set of p, while *static-type*(n) is the static type of the expression represented by n.

Rules T1 and T2 set the *rt-type* of each $\textsc{value}_\tau$ node and each $\oplus_\tau x$ node to be its static type (τ). For a $\oplus_\tau x$ node, the *rt-type* is τ because the value of an expression such as `x + y` is considered by the RTC to have a runtime type equal to the static type of the expression. Rule T3 constrains the *rt-type* of a $*p$ node to be no higher in the lattice than the type of each variable in p's points-to set. Rule T4 constrains the *rt-type* of the left-hand side of a conversion to be no higher than its static type; this is because the RTC tool always treats a conversion cast as having a runtime type equal to its static type. Rule T5 deals

	Condition	Inferred Constraint
T1.		$\mathit{rt\text{-}type}(\text{VALUE}_\tau) = \tau$
T2.		$\mathit{rt\text{-}type}(\oplus_\tau x) = \tau$
T3.	$x \in \mathit{pt\text{-}set}(p)$	$\mathit{rt\text{-}type}(*p) \sqsubseteq \mathit{rt\text{-}type}(x)$
T4.	$n_1 \xleftarrow{cvt} n_2$	$\mathit{rt\text{-}type}(n_1) \sqsubseteq \mathit{static\text{-}type}(n_1)$
T5.	$n_1 \xleftarrow{ext} n_2$	**if** $\mathit{rt\text{-}type}(n_2) == \mathit{static\text{-}type}(n_2)$ **then** $\mathit{rt\text{-}type}(n_1) \sqsubseteq \mathit{static\text{-}type}(n_1)$ **else** $\mathit{rt\text{-}type}(n_1) = \perp$
T6a.	$*p \xleftarrow{=} n_2$, $x \in \mathit{pt\text{-}set}(p)$	$\mathit{rt\text{-}type}(x) \sqsubseteq \mathit{rt\text{-}type}(n_2)$
T6b.	$n_1 \xleftarrow{=} n_2$, n_1 not of the form $*p$	$\mathit{rt\text{-}type}(n_1) \sqsubseteq \mathit{rt\text{-}type}(n_2)$

Figure 4. Rules for computing *rt-type*.

with extension edges: if the right-hand side of an extension assignment is well-typed, then the *rt-type* of the left-hand side is constrained to be no higher than its static type; otherwise, since the assignment will effectively be copying an aggregation of multiple RTC tags into the left-hand side, the *rt-type* of the left-hand side is set to $\perp$. Rules T6a and T6b handle assignment edges: if the left-hand side is a dereference of a pointer p (rule T6a), then the *rt-type* of each node in the points-to set of p can be no higher than the *rt-type* of the right-hand side; if the left-hand side is a variable v (rule T6b), then its *rt-type* can be no higher than the *rt-type* of the right-hand side.

The *rt-type* values are computed for the nodes in the assignment graph by solving the constraints generated from the above rules. This is done by building a directed graph with the same nodes as the assignment graph, and edges representing $\sqsubseteq$ constraints induced by rules T3, T6a, and T6b (and collapsing cycles for efficiency). Initially, all nodes are given *rt-type* $= \top$. Next, rules T1, T2, T4, and T5 are used to assign *rt-type* values to the nodes in the graph where these rules apply (rules T4 and T5 are used to assign $\mathit{static\text{-}type}(n_1)$ to the node that represents n_1). The graph is then traversed to propagate *rt-type* values along the $\sqsubseteq$ edges: for each node n, $\mathit{rt\text{-}type}(n)$ is assigned the meet of the old value of $\mathit{rt\text{-}type}(n)$ with the *rt-type* of all n' for which there is a constraint edge representing $n \sqsubseteq n'$. After propagation, we

Assignments	Assignment Edges	Inferred Constraints
`i = 1;`	$i \xleftarrow{=} \text{VALUE}_{int}$	$\mathit{rt\text{-}type}(i) \sqsubseteq \mathit{int}$
`f = 2.3;`	$f \xleftarrow{=} \text{VALUE}_{float}$	$\mathit{rt\text{-}type}(f) \sqsubseteq \mathit{float}$
`p0 = 0;`	$p0 \xleftarrow{=} \text{VALUE}_{zero}$	$\mathit{rt\text{-}type}(p0) \sqsubseteq \mathit{zero}$
`p1 = &i;`	$p1 \xleftarrow{=} \text{VALUE}_{valid\text{-}ptr}$	$\mathit{rt\text{-}type}(p1) \sqsubseteq \mathit{valid\text{-}ptr}$
`p1 =(int*)&f;`	$p1 \xleftarrow{=} \text{VALUE}_{valid\text{-}ptr}$	$\mathit{rt\text{-}type}(p1) \sqsubseteq \mathit{valid\text{-}ptr}$
`p2 = p1 + i;`	$p2 \xleftarrow{=} \oplus_{pointer}\ p1$	$\mathit{rt\text{-}type}(p2) \sqsubseteq \mathit{pointer}$
	$p2 \xleftarrow{=} \oplus_{pointer}\ i$	$\mathit{rt\text{-}type}(p2) \sqsubseteq \mathit{pointer}$
`*p1 = 4;`	$*p1 \xleftarrow{=} \text{VALUE}_{int}$	$\mathit{rt\text{-}type}(i) \sqsubseteq \mathit{int}$
	$(\mathit{pt\text{-}set}(p1) = \{i, f\})$	$\mathit{rt\text{-}type}(f) \sqsubseteq \mathit{int}$
		$\mathit{rt\text{-}type}(*p1) \sqsubseteq \mathit{rt\text{-}type}(i)$
		$\mathit{rt\text{-}type}(*p1) \sqsubseteq \mathit{rt\text{-}type}(f)$

Final *rt-type* values:	
$\mathit{rt\text{-}type}(p0) = \mathit{zero}$	$\mathit{rt\text{-}type}(i) = \mathit{int}$
$\mathit{rt\text{-}type}(p1) = \mathit{valid\text{-}ptr}$	$\mathit{rt\text{-}type}(f) = \bot$
$\mathit{rt\text{-}type}(p2) = \mathit{pointer}$	$\mathit{rt\text{-}type}(*p1) = \bot$

Figure 5. Computing *rt-type* for the example in Figure 1.

must revisit all extension edges (in the original assignment graph), and check against rule T5: if $\mathit{rt\text{-}type}(n_2) \neq \mathit{static\text{-}type}(n_2)$, and $\mathit{rt\text{-}type}(n_1)$ is not already $\bot$, then $\mathit{rt\text{-}type}(n_1)$ is set to $\bot$, and the graph must be traversed again to propagate this change. Since extension casts are rare in practice, this does not noticeably affect efficiency.

Figure 5 shows the assignment edges, inferred constraints, and final *rt-type* values for the example of Figure 1.

3.5. Step 3: Computing Type-safety Levels

Once the *rt-type* values are computed, each node of the graph is annotated with an attribute signifying its type-safety level — either *unsafe* or *tracked* — based on the rules given in Figure 6; after applying the rules, any node not annotated *unsafe* or *tracked* is considered *safe*.

Rule L1 annotates as *unsafe* any node whose *rt-type* is not compatible with its *static-type* (recall that in the lattice for *rt-type*, *valid-ptr* is compatible with *pointer*, and *zero* is compatible with all scalar types).

	Condition	Attribute
L1.	$\mathit{rt\text{-}type}(n) \not\supseteq \mathit{static\text{-}type}(n)$	n *unsafe*
L2.	$\mathit{static\text{-}type}(p) == \mathit{pointer}$, $\mathit{rt\text{-}type}(p) \neq \mathit{valid\text{-}ptr}$	$*p$ *unsafe*
L3.	$*p$ *unsafe*, $x \in \mathit{pt\text{-}set}(p)$, $x \neq \mathit{unsafe}$	x *tracked*

Figure 6. Rules for determining *type-safety* levels.

For Rule L2: a pointer p whose *rt-type* is not *valid-ptr* may contain an invalid address, therefore $*p$ must be instrumented to check its runtime type, and is thus annotated as *unsafe*. Rule L3 annotates as *tracked* any variable in the points-to set of a pointer p whose dereference node ($*p$) is *unsafe*.

Looking back at the example in Figure 5, Rule L1 makes f and $*p1$ *unsafe*, L2 makes $*p0$ and $*p2$ *unsafe*, and L3 makes i *tracked*. This leaves $p0$, $p1$ and $p2$ as *safe*.

Note that since *safe* variables are not instrumented, a safe variable v will always be tagged *unallocated* in the mirror. If the contents of v is accessed indirectly by an errant pointer p, the dereference expression $*p$ will have been annotated as *unsafe*, and thus will be instrumented with a *verify-pointer* operation. Since v is tagged *unallocated*, the check of $*p$ will trigger an "accessing unallocated memory" error. Thus, by identifying *safe* variables, the type-safety-level analysis not only lowers the overhead of the tool by eliminating unnecessary instrumentation, it also increases the likelihood of the RTC tool detecting an error, because it effectively tags more memory as *unallocated*.

3.6. May-be-uninitialized Analysis

The type-safety-level analysis described above does not account for uses of uninitialized data, which is an error that the RTC tool checks for. That is, by eliminating all instrumentation for *safe* and *tracked* locations, and by initializing *tracked* locations to their static types rather than to *uninitialized*, the RTC tool will no longer detect uses of uninitialized data in these locations. To address this problem, an additional flow-sensitive analysis is needed to find program points where instrumentation cannot be elided.

For a location x that is *safe* or *tracked*, the analysis finds instances of x where x may be uninitialized. This analysis is defined as a dataflow-analysis problem on a control-flow graph (CFG):

– The elements of the underlying lattice are sets of locations (variables or abstract locations representing heap objects).

– The analysis computes two sets for each CFG node n: $\mathit{Uninit}_{\mathit{in}}(n)$ and $\mathit{Uninit}_{\mathit{out}}(n)$, representing locations that *may* be uninitialized before and after n.

– The lattice meet is set union.

The dataflow transfer functions reflect how the RTC tool treats locations tagged as *uninitialized*. If y is *uninitialized*, an assignment $x = y$ is instrumented to copy the *uninitialized* tag into the mirror for x. But for a use of y, such as $x = y + 1$, if y is *uninitialized*, the RTC verify function will report an error, and set x's runtime type to its static type (to avoid cascading errors).

The dataflow transfer functions are as follows:

– At a node n that declares the variable x, or allocates the heap location x (e.g., via a call to `malloc`), $\mathit{Uninit}_{\mathit{out}}(n) = \mathit{Uninit}_{\mathit{in}}(n) \cup \{x\}$

– At a node n that is a direct assignment $x = e$ (where x is a variable),

  - if e is a variable y such that $y \in \mathit{Uninit}_{\mathit{in}}(n)$, then $\mathit{Uninit}_{\mathit{out}}(n) = \mathit{Uninit}_{\mathit{in}}(n) \cup \{x\}$
  - if e is a dereference $*q$ such that $\mathit{pt\text{-}set}(q) \cap \mathit{Uninit}_{\mathit{in}}(n) \neq \emptyset$, then $\mathit{Uninit}_{\mathit{out}}(n) = \mathit{Uninit}_{\mathit{in}}(n) \cup \{x\}$
  - otherwise, $\mathit{Uninit}_{\mathit{out}}(n) = \mathit{Uninit}_{\mathit{in}}(n) - \{x\}$

– At a node n that is an indirect assignment $*p = e$,

- if e is a variable y such that $y \in \mathit{Uninit}_{\mathit{in}}(n)$, then $\mathit{Uninit}_{\mathit{out}}(n) = \mathit{Uninit}_{\mathit{in}}(n) \cup \mathit{pt\text{-}set}(p)$
- if e is a dereference $*q$ such that $\mathit{pt\text{-}set}(q) \cap \mathit{Uninit}_{\mathit{in}}(n) \neq \emptyset$, then $\mathit{Uninit}_{\mathit{out}}(n) = \mathit{Uninit}_{\mathit{in}}(n) \cup \mathit{pt\text{-}set}(p)$
- otherwise, $\mathit{Uninit}_{\mathit{out}}(n) = \mathit{Uninit}_{\mathit{in}}(n)$

To account for function calls, which is an important aspect to consider in dataflow analysis, we augment the language used in our description to include simple function calls, `f()`, with no arguments or return values (arguments and return values can be modeled by assignments to globals). The dataflow transfer function for a function call is:

– At a node n containing a call `f()`,
  $\mathit{Uninit}_{\mathit{out}}(n) = \mathit{Uninit}_{\mathit{in}}(n) \cup \mathit{MayMod}(\mathtt{f})$

where $\mathit{MayMod}(\mathtt{f})$ is the set of locations that *may* be modified as a result of calling the function. Essentially, a call to `f` is treated as possibly assigning an *uninitialized* value to all locations in $\mathit{MayMod}(\mathtt{f})$.

After performing the may-be-uninitialized analysis, instrumentation is added to allow the RTC tool to detect uses of uninitialized data. A *tracked* or *safe* location x for which there is a use of x in a node n where $x \in \mathit{Uninit}_{\mathit{in}}(n)$ is treated as follows:

1. The declaration of x is instrumented with a *declare* operation that sets its type in the mirror to *uninitialized*.

2. For each node n that uses x, if $x \in \mathit{Uninit}_{\mathit{in}}(n)$, then the use of x is instrumented with a *verify* operation.

3. For each node n that defines x, if either $x \in \mathit{Uninit}_{\mathit{in}}(n)$ or $x \in \mathit{Uninit}_{\mathit{out}}(n)$, then the assignment is instrumented with a *copy* operation (this ensures that x's tag is set correctly for subsequent uses of x). The only definitions of x that need not be instrumented are those for which x is definitely not uninitialized both before and after the definition.

4. Redundant-check Analysis

When a location x is read many times with no intervening writes, a runtime check is only necessary for the first read of x. This is because if the first check of x reports a runtime error, the RTC tool will set its runtime type to its (statically) declared type to prevent cascading errors. A subsequent check of the same location, with no intervening writes to that location, is therefore redundant, and should be eliminated.

To identify redundant checks, we perform a dataflow analysis to keep track of *lvalue* expressions that have been checked. A check of an unsafe expression e is redundant at control-flow graph node n if every path from the CFG's *enter* node to n includes a node m such that:

- There is a use of e at node m (where e will be instrumented by the RTC tool to *verify* its runtime type), and
- No path from m to n changes the runtime type of e, changes the l-value of e, or deallocates location e.

The analysis is defined as a *Gen/Kill* dataflow problem:

- The elements of the underlying lattice are sets of unsafe *lvalue* expressions (either a variable v or a pointer dereference $*p$).
- The analysis computes two sets for each CFG node n: $\mathit{Checked}_{in}(n)$ and $\mathit{Checked}_{out}(n)$, representing expressions for which checks are redundant before and after n.
- The dataflow transfer function for each node n is of the form $\mathit{Checked}_{out}(n) = \mathit{Checked}_{in}(n) - \mathit{Kill}(n) \cup \mathit{Gen}(n)$
- The lattice meet is set intersection.

The set $\mathit{Gen}(n)$ for a node n includes the *lvalue* expression e if e is unsafe and there is a use of e at node n; that is, e will be instrumented to *verify* its runtime type. The set $\mathit{Kill}(n)$ for a node n is described by the conditions listed below.

A variable v is in $\mathit{Kill}(n)$ for a node n containing any of the following:

1. An assignment $v = e$, where e is an *unsafe* expression.

2. An assignment $*p = e$, where e is an *unsafe* expression, and v is in the points-to set of p.

3. A call `f`(), where v is in *MayMod*(`f`).

A dereference expression $*p$ is in *Kill*(n) for a node n containing any of the following:

4. An assignment $x = e$, where e is an *unsafe* expression, and x is in the points-to set of p.

5. An assignment $*q = e$, where e is an *unsafe* expression, and the points-to sets of p and q intersect.

6. *Any* assignment to p.

7. *Any* assignment to $*q$, where p is in the points-to set of q.

8. A call `f`(), where *MayMod*(`f`) $\cap$ *pt-set*(p) $\neq \emptyset$.

9. A call `free`(q), where the points-to sets of p and q intersect.

Note that r-value expressions of the form *const*, `&`x, or $e_1 \oplus e_2$ always evaluate to well-typed values, and are thus considered *safe* expressions. For condition 9, we treat calls to `free` differently from other function calls, to account for memory deallocation.

After performing this analysis, a use of an expression e at a node n need not be instrumented if e is in $\mathit{Checked}_{in}(n)$

5. Experiments

To evaluate the performance improvements of RTC-instrumented programs as a result of the static analyses described in this paper, we instrumented some programs from the SPECINT 95, SPECINT 2000, and Olden benchmark suites, as well as some of the programs used to evaluate Cyclone [10]. The programs were compiled with `gcc`, and executed on a 333MHz Pentium II running Linux.

Table I. Benchmark Characteristics and Performance

Program	Lines of Code (a)	Inst. RTC Routines: Static (b)	Inst. RTC Routines: Dyn $\times 10^6$ (c)	Exec time (d)	RTC slowd (e)	opt slowd (f)	imprv % (g)
Cyclone							
aes	1822	2304	662.27	5.36	21.80	8.56	60.7
cacm	340	182	330.46	4.01	27.01	6.87	74.6
cfrac	4218	10092	1241.63	8.28	56.49	30.79	45.5
matxmult	1377	109	463.33	4.76	6.50	3.45	46.9
ppm	1421	1378	390.17	3.06	35.20	26.27	25.4
tile	4880	4233	71.48	1.41	73.03	66.54	8.9
Olden							
bh	3200	1467	954.20	11.64	80.26	64.63	19.5
bisort	690	603	774.25	5.68	25.96	16.40	36.8
em3d	538	358	1065.55	11.97	6.82	4.89	28.2
health	706	573	184.49	5.62	7.25	5.77	20.4
mst	610	503	597.33	8.38	42.43	22.91	46.0
perimeter	472	474	120.65	1.30	41.85	18.46	55.9
power	867	1179	693.67	11.68	22.61	7.79	65.5
treeadd	375	151	170.39	1.84	23.83	14.07	40.9
SPEC95							
compress	3900	3095	3860.26	31.45	32.93	17.15	47.9
go	29629	48163	1773.64	13.81	53.36	16.02	70.0
li	7630	9406	955.60	4.57	74.21	64.65	12.9
SPEC2000							
bzip2	4650	4807	1239.41	7.01	41.64	28.59	31.3
gzip	8605	6686	962.19	6.12	54.24	25.76	52.5
mcf	2412	2234	163.70	1.74	15.75	11.63	26.2

Table I lists some characteristics of the benchmarks we used, and the runtime performance results. Column (a) gives the program size in lines of code. Columns (b) and (c) give the static and dynamic number of calls to RTC functions (the functions listed in Section 2.2) added by the RTC tool: the goal of the static analyses presented in this paper is to reduce these numbers.

Column (d) gives the execution time (in seconds) of the uninstrumented code. Columns (e) and (f) give the runtime slowdown factor of the RTC-instrumented programs, before and after optimizing with the

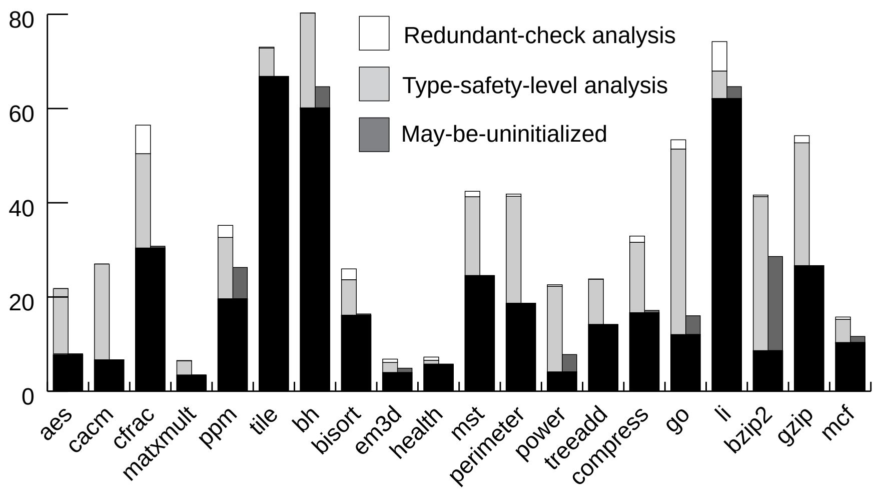


Figure 7. Execution time slowdown factor.

static analyses described in this paper; column (g) gives the percentage speedup gained from the optimization. On average, the unoptimized RTC-instrumented program (column (e)) ran 37 times slower than the uninstrumented executable, while the optimized version (column (f)) ran 23 times slower. On average, the optimizations improved execution time by 40.8% (column (g)).

Figure 7 graphs the slowdown factors, breaking down the effects of the various static analyses. The full height of the left-side bars represents the slowdown factor of the unoptimized RTC program (corresponding to Table I, column (e)). The two upper-left regions (white and light gray) represent the portion of runtime improvement due to the redundant-check and type-safety-level analyses respectively. The upper-right regions (dark gray) represent the slowdown due to instrumentation re-introduced by the may-be-uninitialized analysis. Thus, the height of the right-side bars (black plus dark gray) represents the slowdown factor of the optimized program (corresponding to column (f)). On average, the redundant-check analysis (white bar) improved performance by 3.6% (over the running time of the unoptimized RTC code); the type-safety-level analysis (light gray) improved performance by an additional 43.2%; and the may-be-uninitialized analysis (dark gray) subtracted 6.0% from these improvements.

Table II. Analysis Times (in seconds)

	Normal Compl	Type-Safety	May-be-Uninit	Redund-Chk.
cfrac	6	0.41	0.71	5.52
go	19	2.84	166.14	8.87
li	10	2.79	304.74	133.57

As an indicator of the efficiency of the different analyses, Table II gives the compilation and analysis times for the slower-compiling benchmarks; for the other benchmarks, all analyses took less than 1 second. Notice that the flow-insensitive type-safety-level analysis (the column labeled "type-safety"), which includes Das's points-to analysis, scales well, but the two flow-sensitive analyses are noticeably slower on larger programs. While some of this may be due to our suboptimal prototype implementation, the flow-sensitive analyses have a fundamentally higher complexity, so such behavior is expected.

Note also that if we allow the RTC tool to miss catching uses of uninitialized memory, we can skip the slow may-be-uninitialized analysis, and at the same time improve runtime performance (i.e., to the level of the black bars in Figure 7). The runtime overhead reintroduced by the may-be-uninitialized analysis is small overall but not insignificant. This is due in part to the fact that our implementation is intraprocedural; in particular, after a call to `f`, we conservatively consider everything in *MayMod*(`f`) as possibly uninitialized. An interprocedural implementation that builds a supergraph (which connects each call-site to the entry node of the called function), or incorporates context-sensitivity, is likely to be more precise, but will be slower.

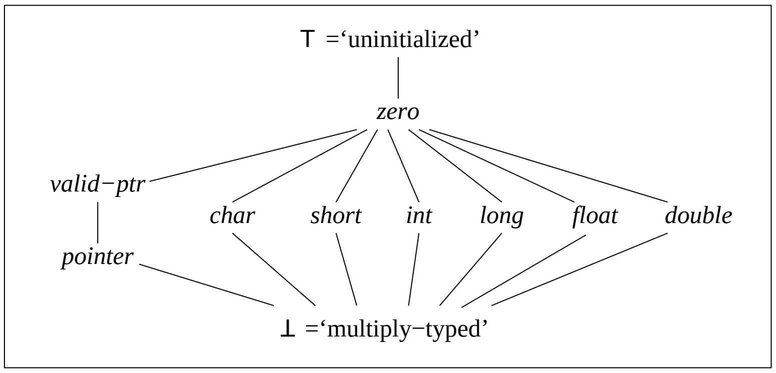
6. Future Work: Never-null-dereference Analysis

One shortcoming of the type-safety-level analysis presented in Section 3 is that if a pointer p is ever assigned a NULL value, then $*p$ is annotated as *unsafe* (and must thus be fully instrumented). This is because the NULL constant maps to the $\textsc{value}_{zero}$ object, whose *rt-type* is *zero*.

The assignment to p causes *rt-type*(p) to be constrained to be no higher in the lattice than *zero*. This prevents *rt-type*(p) from being *valid-ptr*; thus, by Rule L2 in Figure 6, **p* is annotated as *unsafe*. This does not cause the RTC tool to miss any errors or to report any spurious errors, but it adds to the runtime overhead by including some unnecessary instrumentation.

With a small change to the type-safety-level analysis and another flow-sensitive analysis, we can find some instances of $*p$ where p is guaranteed to contain either a valid pointer or a NULL value, and for each such instance, we can replace the *verify-pointer* instrumentation with a null-pointer check.

First, the *rt-type* lattice (in Figure 3) is modified so that the *valid-ptr* type is below the *zero* type, as follows:



Second, Rule L2 in Figure 6 is changed to:

	Condition	Attribute
L2.	*static-type*(p) == *pointer*, *rt-type*$(p) \not\sqsupseteq$ *valid-ptr*	$*p$ *unsafe*

With these modifications, for any pointer p that is only assigned valid pointer values or a NULL value, $*p$ will be *safe*. (Note that if a pointer may be assigned the result of pointer arithmetic or other invalid pointer values, then $*p$ will still be considered *unsafe*.)

Next, we perform another dataflow analysis to account for possible null-pointer dereferences of these pointers. This analysis is very similar to the *may-be-uninitialized* analysis described in Section 3.6. The lattice elements are sets of pointers, and the lattice meet is set union. The analysis computes the sets $\mathit{Null}_{\mathit{in}}(n)$ and $\mathit{Null}_{\mathit{out}}(n)$ for each node n, representing the sets of pointers that *may* be null. The transfer functions for the different kinds of node n are:

– $p =$ `NULL`: $\mathit{Null}_{\mathit{out}}(n) = \mathit{Null}_{\mathit{in}}(n) \cup \{p\}$

– $p = \&x$: $\mathit{Null}_{\mathit{out}}(n) = \mathit{Null}_{\mathit{in}}(n) - \{p\}$

– $p = q$:
  if $q \in \mathit{Null}_{\mathit{in}}(n)$, then $\mathit{Null}_{\mathit{out}}(n) = \mathit{Null}_{\mathit{in}}(n) \cup \{p\}$
  else, $\mathit{Null}_{\mathit{out}}(n) = \mathit{Null}_{\mathit{in}}(n) - \{p\}$

– $p = *q$:
  if $\mathit{pt\text{-}set}(q) \cap \mathit{Null}_{\mathit{in}}(n) \neq \emptyset$, then $\mathit{Null}_{\mathit{out}}(n) = \mathit{Null}_{\mathit{in}}(n) \cup \{p\}$
  else, $\mathit{Null}_{\mathit{out}}(n) = \mathit{Null}_{\mathit{in}}(n) - \{p\}$

– $*p = q$:
  if $q \in \mathit{Null}_{\mathit{in}}(n)$, then $\mathit{Null}_{\mathit{out}}(n) = \mathit{Null}_{\mathit{in}}(n) \cup \mathit{pt\text{-}set}(p)$
  else, $\mathit{Null}_{\mathit{out}}(n) = \mathit{Null}_{\mathit{in}}(n)$

– $*p = *q$:
  if $\mathit{pt\text{-}set}(q) \cap \mathit{Null}_{\mathit{in}}(n) \neq \emptyset$, then $\mathit{Null}_{\mathit{out}}(n) = \mathit{Null}_{\mathit{in}}(n) \cup \mathit{pt\text{-}set}(p)$
  else, $\mathit{Null}_{\mathit{out}}(n) = \mathit{Null}_{\mathit{in}}(n)$

Upon completion of the analysis, a *safe* dereference $*p$ at node n where $p \in \mathit{Null}_{\mathit{in}}(n)$ must be instrumented to perform a null-pointer check.

7. Related Work

Many approaches have been proposed and developed that instrument a program to track auxiliary information during program execution.

Purify [8] is a commercial product that has proven to be successful in detecting buffer overruns, memory leaks, and other errors at runtime. It instruments object code, which gives it an advantage of not requiring source code, but a disadvantage of being platform dependent. Further, lack of source code means techniques such as that proposed in this paper cannot be applied to improve its overhead of about 15× slowdown.

Valgrind [17] is a similar tool, but it interprets the executable binary on a "synthetic CPU", and thus incurs a high overhead (about 40× slowdown). Its `memcheck` component mirrors each byte of memory with additional information, similar to the RTC tool. But since it operates on executable binaries, the techniques described in this paper cannot be directly applied to improve its performance.

Insure++ [15] is another heavyweight debugging tool that detects common sources of program errors, like out-of-bounds array accesses and null-pointer dereferences. Like the RTC tool, it instruments the program at the source level, so it is possible that using techniques similar to those presented in this paper could improve its performance.

Austin *et al.* [2] describe the *Safe C* system, which tracks information about each pointer's referent, and uses this information to detect spatial (e.g., array out-of-bounds) and temporal (e.g., stale pointer dereference) access errors. They propose a compile-time optimization that is very similar to our redundant-check analysis. Patil *et al.* [16] describe a similar technique for checking spatial and temporal accesses called *guarding*, and propose a novel way to make this check more efficient by performing the tracking and checking of the auxiliary information in a *shadow process* on a separate processor.

Cyclone [10] and *CCured* [14] are two systems based on the C language that attempt to inject some level of safety while maintaining the low-level control of the language. The *Cyclone* language includes the definition of different kinds of pointers with different safety restrictions; "unsafe" pointer dereferences are instrumented with runtime checks (using "fat pointers" in a manner similar to *Safe-C*). To port an existing C program into *Cyclone*, the programmer must manually convert C pointers into the appropriate kind of Cyclone pointer to achieve optimal performance; an analysis to automatically classify pointers into the appropriate safety level, similar to the type-safety-level analysis proposed in this paper, would make it easier to port existing C code, and thus encourage greater use of this new language.

CCured also includes runtime checks for bad pointer dereferences. *CCured*'s checks are more limited than RTC checks: specifically, *CCured* focuses only on pointers, and does not differentiate non-pointer types. Furthermore, *CCured* can be too strict (i.e., certain valid program behavior, such as storing the address of a stack variable in a global variable, or storing a pointer value in an integer, casting it back, and dereferencing it, will cause a runtime check to fail).

To reduce the overhead of runtime checks, *CCured* uses a type-inference scheme to identify as many *safe* and *sequence* pointers as possible, thus minimizing the amount of instrumented operations. The goal of their type inference is thus similar to that of our type-safety-level analysis. However, their type-inference scheme is less precise than our proposed analysis: they effectively group points-to sets into equivalence classes (in the spirit of Steensgaard's points-to analysis [20]), while our analysis accounts for the directionality of assignments. Despite this, they were able to significantly improve the performance of instrumented *CCured* programs, from the unoptimized slowdown of 6-20 times, to between 1 and 2 times slowdown using the type-inference optimization.

The use of runtime checks to enforce safety properties, and techniques for eliminating unnecessary checks to improve performance, have been used in other programming languages and environments. Implementations of dynamically-typed languages like LISP and Scheme need to maintain runtime information to perform runtime type-checking as part of the language's semantics. To improve the performance of such a system, Henglein [9] proposes an efficient approach based on type inference. The Java language needs to perform potentially expensive runtime checks, such as array-bounds checks, to enforce safety properties guaranteed by the language. The elimination of redundant and unnecessary array-bounds checks in Java and other safe languages has been studied extensively [21, 13, 7, 3, 4, 12].

8. Conclusions

We have presented some techniques for reducing the runtime overhead of the instrumentation added to C programs by the Runtime Type-Checking tool. In Section 3, we defined a flow-insensitive type-safety-level analysis, which classifies each *lvalue* expression in the program as *safe*, *unsafe*, or *tracked*. In Section 4, we defined a dataflow analysis to identify redundant checks that can be eliminated. The results of these analyses, used in conjunction with the results of the may-be-uninitialized analysis defined in Section 3.6, improved the runtime performance of RTC by 40% on average, without sacrificing the ability of the RTC tool to detect errors.

Despite these improvements, the runtime overhead, averaging 23 times slower than the original program, remains relatively high. There is potential for developing other optimization techniques to improve performance.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
2. T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94), SIGPLAN Notices 29(6)*, pages 290–201, Orlando, FL, June 1994.
3. R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00), SIGPLAN Notices 35(5)*, pages 321–333, Vancouver, BC, June 2000.
4. W.-N. Chin, S.-C. Khoo, and D. N. Xu. Deriving pre-conditions for array bound check elimination. In *Proceedings of the Second Symposium on Programs as Data Objects, PADO 2001*, pages 2–24, Aarhus, Denmark, May 2001.
5. Ckit. http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ckit/.
6. M. Das. Unification-based pointer analysis with directional assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00), SIGPLAN Notices 35(5)*, pages 35–46, Vancouver, BC, June 2000.
7. R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.
8. R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, 1992.
9. F. Henglein. Global tagging optimization by type inference. In *LISP and Functional Programming*, pages 205–215, 1992.
10. T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.
11. A. Loginov, S. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *Lec. Notes in Comp. Sci.*, pages 217–232. Springer, Apr. 2001.
12. M. Lujan, J. R. Gurd, T. L. Freeman, and J. Miguel. Elimination of java array bounds checks in the presence of indirection. Technical Report CSPP-13, Department of Computer Science, University of Manchester, Feb. 2002.
13. V. Markstein, J. Cocke, and P. Markstein. Optimization of range checking. In *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices 17(6)*, pages 114–119, Boston, MA, June 1982.
14. G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages*, Portland, OR, Jan. 2002.

15. Parasoft. Insure++: An automatic runtime error detection tool, 2001. http://www.parasoft.com/insure/papers/tech.htm.

16. H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software–Practice and Experience*, 27(27):87–110, 1997.

17. J. Seward. The design and implementation of Valgrind. Technical report, http://developer.kde.org/˜sewardj/, 2000.

18. M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *Proc. of ESEC/FSE '99: Seventh European Softw. Eng. Conf. and Seventh ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, pages 180–198, Sept. 1999.

19. R. Stallman and R. Pesch. *Using GDB: A Guide to the GNU Source-Level Debugger*. July 1991.

20. B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.

21. N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *ACM Symposium on Principles of Programming Languages*, pages 132–143, Los Angeles, CA, Jan. 1977.

22. S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '90), SIGPLAN Notices 25(6)*, pages 91–103, Atlanta, GA, May 1999.