Static Analysis of C for Hybrid Type Checking

Zachary Ryan Anderson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-1
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-1.html

January 4, 2007

Copyright © 2007, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Static Analysis of C for Hybrid Type Checking

Zachary R. Anderson
zra@cs.berkeley.edu

University of California, Berkeley

Abstract. Hybrid type checking[5] is an approach to enforcing the well-typedness of programs that, where possible, uses static analysis to determine the types of expressions, and run-time checking when the precision of static analysis is insufficeint. This approach is useful for dependent type systems in which types are parameterized by run-time values of expressions. Deputy is a dependent type system for C that allows the user to describe bounded pointers, tagged unions, and null-terminated strings. Deputy runs in two phases. In the first phase, simple typing rules are applied. The typing rules prescribe the insertion of run-time checks for certain operations. In the second phase, static analysis is used to identify checks that must either always succeed or always fail. The former may safely be removed, and the latter signify typing errors. This report describes the second phase of Deputy.

1 Introduction

Hybrid type checking[5] is an approach to enforcing the well-typedness of programs. In a dependent type system, the type of an expression can be parametrized by state elements of the program. Static analysis can reveal approximations of the values of the state elements, and these approximations may give enough information to type check the program. However, where the static analysis lacks precision, it may not be known statically whether an expression is well-typed or not. In these cases, a hybrid type-checker will add run-time checks to the program. The run-time checks encode constraints on the values of state elements that must be satisfied in order for expressions to be well-typed. The run-time checks will inspect the values of the state elements and succeed if the constraints are satisfied, or fail otherwise.

1.1 Deputy

Types can be used to specify program invariants. Dependent types augment traditional types with the ability to describe invariants relating state elements of a program. Deputy[14, 4] is a dependent type system for C that allows the description of invariants for bounded pointers, tagged unions, and null-terminated strings.

The goal of the Deputy type system is to enforce memory safety for most extensions to large software systems such as Linux device drivers. This is achieved

through the use of dependent type annotations on traditional C types at the interfaces between the extensions and the core system. For example, Deputy is applied to device drivers by annotating function prototypes and structure definitions in the kernel header files.

For the purposes of this report, the Deputy type checker can be considered to run in two phases. In the first phase, simple typing rules are applied. The typing rules prescribe the insertion of run-time checks for certain operations. In the second phase, static analysis is used to identify checks that must either always succeed or always fail. The former may be safely removed, and the latter signify typing errors. Run-time checks that fall into neither of these two categories remain in the program, and may either succeed or fail when the program runs. This two phased approach is used so that the typing rules and their implementation can be kept simple and easy to reason about.

Below, the Deputy annotations are described in detail. It is also explained how the first phase of Deputy uses the annotations to insert run-time checks. Further, the different kinds of run-time checks needed by Deputy are described.

1.2 Deputy Annotations

In lieu of a full description of the Deputy type system, which can be found in [4], an overview of the dependent type annotations, and their meanings will be given.

First, Deputy allows users to specify the bounds of a buffer. This can be done with the following annotations on any pointer type: `safe`, `sentinel`, `count(n)`, and `bound(lo, hi)`. Here, `n`, `low`, and `hi` are expressions that can refer to variables and structure field names in the immediately enclosing scope. For example, annotations on local variables may refer to other locals, annotations on function parameters can refer to other parameters, and annotations on structure fields can refer to other fields of the same structure.

A `safe` annotation means that the pointer is null or points to a single element of the base type. A `sentinel` annotation means that a pointer may only be used for comparisons. A `count(n)` annotation means that the pointer may either be null or point to an array of at least `n` elements of the base type. A `bounds(lo, hi)` annotation means that the pointer may either be null or point into a region of memory with low address `lo` and where `hi` is the address of the first element beyond the end of the region.

The invariant is that pointers having these annotations must always either be null or within the stated bounds. Accordingly, the first phase of Deputy adds run-time checks to ensure that pointers are non-null at dereference, that the results of pointer arithmetic are in the range `[lo,hi)` and refer only to whole elements. Furthermore, the first phase of Deputy will add run-time checks to enforce the typing rules for coercions among annotated pointer types. For example, run-time checks will be added to ensure that a pointer for a smaller buffer is not coerced into a pointer for a larger buffer, and that pointers coerced to `safe` are null or point to at least one element.

```
int * safe find(int * count(len) buf,
                int len) {
  assert(buf ≠ NULL);
  assert(buf <= buf + len <= buf + len);
  int * sentinel end = buf + len;
  int * bound(cur,end) cur = buf;
  while (cur < end) {
    assert(cur ≠ NULL);
    assert(cur < end);
    if (*cur == 0) return cur;
    assert(cur <= cur + 1 <= end);
    cur++;
  }
  return NULL;
}
```

Fig. 1. A function for finding zero in an array of integers. The first phase of Deputy adds checks based on the annotated pointer types.

Figure 1 gives examples of the use of the buffer annotations. The `count(len)` annotation on `buf` indicates that the function only accepts buffers at least as long as `len`. The `safe` annotation indicates that the function only returns a null pointer or a pointer to at least one element. The `sentinel` annotation means that `end` may only be used for comparison, and the `bound(cur,end)` annotation means that `cur` may not point to elements at address `end` or larger. The first check is added because Deputy disallows arithmetic on null pointers. The second is added to ensure that the result of `buf + len` remains in bounds. Then, whenever `cur` is dereferenced, incremented, or returned, it is verified that there is at least one element remaining in the array. We explain later how many of these checks can be safely removed.

In addition to annotations for bounds on buffers, Deputy also provides an annotation for stipulating that somewhere beyond the upper bound given by one of the above annotations is a null element. This annotation is `nullterm`, and it is used in conjunction with the annotations above. For example, the type `char * count(0) nullterm` means that the pointer is a standard null-terminated string. More flexibility exists, though, since the bounds annotations can be used to indicated that the buffer has some minimum size.

The following checks are inserted for `nullterm` pointers. On a dereference, it is checked that the pointer is non-null. On an increment, if the result is not within the declared bounds, then it is checked that the result does not go beyond the null element. The `nullterm` annotation can safely be dropped without a run-time check.

Figure 2 shows an example of annotating the `strlcpy` function with Deputy annotations. From its specification, `dst` is required to have `size - 1` elements before its null-terminator, whereas all that is required of `src` is that it be null-

```
size_t strlcpy(char * count(size-1) nullterm dst,
               const char * count(0) nullterm src,
               size_t size);
```

Fig. 2. Examples of annotations for null-terminated strings.

terminated. In the implementation, when `dst` and `src` are incremented, checks are added to ensure that the pointers are non-null and that either the resulting pointer is in bounds, or that the result does not go beyond the null-terminator.

For tagged unions, Deputy includes an annotation that can be placed on a union field specifying what the value of the tag must be if that field is to be active. The tag must be a field of a structure enclosing the union in question. Taking the address of a tagged union field is forbidden.

When reading a union field, the first phase of Deputy adds a run-time check ensuring that the correct tag is selected. A tag can be changed only if the newly selected union field is null, and run-time checks are added to ensure this.

```
struct e1000_option {
  enum {RANGE, LIST} type;
  union {
      struct {
          int min, max;
      } range when(type == RANGE);
      struct {
          int nr;
          struct e1000_opt_list {...} *p;
      } list when(type == LIST);
  } arg;
};
```

Fig. 3. An example of the Deputy annotations for tagged unions. The tag is `type`, and either the `range` or `list` field of the union is "active" depending on its value.

Figure 3 demonstrates the use of Deputy's annotations for tagged unions. For program variables of type `struct e1000_option`, when the `type` field is `RANGE`, then the `range` field of the union may be accessed, and the `list` field may not be accessed. When `type` is `LIST`, the opposite is true. Additionally, `type` may only be modified when `arg` contains only zeroes.

Additionally Deputy also has features to enforce type safety around calls to memory allocators like `malloc`, and around functions like `memset` and `memcpy`. Also, if a user knows that a pointer argument to a function or a return value may never be null, then the pointer type can be annotated as `nonnull`. It will be shown later how `nonnull` annotations can be discovered automatically.

1.3 Deputy Run-time Checks

The first phase of Deputy inserts the following run-time checks to ensure that its invariants always hold:

- `CNonNull(p)` – Succeeds only when the pointer `p` is non-null. This check is used to ensure pointers are non-null when dereferenced.
- `CPtrArith(lo, hi, p, x, sz)` – Succeeds only when `lo <= p + (x * sz) <= hi` and `p + (x * sz)` does not overflow. This check is used to ensure that the result of pointer arithmetic is in bounds.
- `CPtrArithNT(lo, hi, p, x ,sz)` – Succeeds only when `lo <= p + (x * sz) <= hi + strlen(hi)` and `p + (x * sz)` does not overflow. This check is used to ensure that the result of pointer arithmetic on `nullterm` pointers is in bounds. The notation `strlen(hi)` indicates the number of bytes between `hi` and the null terminator.
- `CLeqInt(e1,e2)` – Succeeds only when `e1 <= e2`. This check is used to check coercions to `safe`, and to ensure that open arrays at the end of structures are allocated with enough space to satisfy the bound annotation on them. Here `e1` and `e2` are `unsigned int`.
- `CLeq(e1,e2)` – Succeeds only when `e1 <= e2`. This check is used to ensure that pointer reads and writes access in bounds memory.
- `CLeqNT(e1,e2)` – Succeeds only when `e1 <= e2 + sizeof(e2)`. This check is similar to `CLeq` except that it is for `nullterm` pointers.
- `CNullOrLeq(e1,e2,e3)` – Succeeds only when `e1` is null, or when `e2 <= e3`. This check is used to ensure that coercions between types with different bounds are safe.
- `CNullOrLeqNT(e1,e2)` – Succeeds only when `e1` is null, or when `e1 <= e2 + sizeof(e2)`. This is the `nullterm` version of `CNullOrLeq`.
- `CWriteNT(p, hi, x)` – Succeeds only when `(p == hi) => (x == 0)`, that is: if `p` is `hi`, then `x` must be zero. This check is used to ensure that the null-terminator of a `nullterm` buffer is not overwritten with a non-null value.
- `CNullUnionOrSelected(u, e)` – Succeeds only when `u` is a union field the memory for which has been zeroed out, or when `e` evaluates to true. Here `e` is an expression indicating that the correct tag for the union field is selected.

1.4 Overview

In the following sections we describe the second phase of Deputy, which reasons about these run-time checks. The design of the second phase, which from here on we will refer to as the "optimizer," is novel because of issues specific to the Deputy type system. The optimizer is arranged in a pipeline of phases. Earlier phases must be inexpensive due to the large number of run-time checks and temporary variables created by the first phase. When checks and dead code are eliminated, more precise analyses may then be run with more reasonable cost.

In addition to a novel overall design, some of the static analyses in support of optimization phases are novel. These include a lightweight octagon analysis,

two analyses for nullterm pointers, and an analysis for deducing when functions expect arguments to be non-null.

In section 2, we describe the optimization phases and the analyses that support them. Further, we present a discussion of the cost and benefit of each of the stages. In section 3, we present two different approaches to the overall design of the optimizer. One we built around our lightweight special purpose octagon analysis, and the other we built around Miné's general purpose library[8].

In section 4, we evaluate the optimizer in terms of the following metrics: the number of run-time checks remaining in the code, the performance of various benchmarks, and the run-time of the optimizer itself. Then, section 5 will discuss some related work, and section 6 will conclude and discuss possible avenues for future work.

Deputy is written in Ocaml using the CIL[12] library for parsing and analyzing C. The optimizer accounts for about 5000 of Deputy's 20,000 lines. This report will argue that in spite of its simplicity, the optimizer is very effective.

2 Optimizer Phases

The presentation of the optimization phases will be broken into three sections. First, we present flow-insensitive optimizations. These rely on reasoning about the syntactic form of checks. Second, we present flow sensitive optimizations. These include the octagon analyses, loop optimizations, and others. Finally, we present an inter-procedural analysis for deducing when functions assume that arguments and return values are non-null.

2.1 Flow-insensitive Optimizations

We use two kinds of flow-insensitive analysis. Both rely only on the syntactic form of the run-time checks. For this reason, we employ an expression canonicalizer that converts expressions to a form linear in the expressions that it cannot further decompose. Expressions that the canonicalizer does not decompose are multiplication unless by a constant, division, bit shifting unless by a constant, casts that affect value, and bit-wise operators. For example, `5*(x + y + 7)` would be canonicalized as `5*x + 5*y + 35`, but `5*(x/3 + y)` would only be reduced to `5*(x/3) + 5*y`.

For comparing the magnitude of expressions, it is important to know when a canonicalized expression will be positive, negative, or zero. In particular, we compare expressions by canonicalizing their difference. For example, to determine whether `e1 <= e2`. The expression `e2 - e1` is canonicalized. If each term of the difference is non-negative, it can be concluded that `e2` is at least as big as `e1`. A term is non-negative only if its factor is positive and the type of its expression is not signed. We assume that a two's complement system is used for machine arithmetic, and that the native compiler will generate instructions for unsigned comparisons when the C types of the operators are not signed.

The first flow-insensitive phase attempts to identify checks that can be safely eliminated, and checks that will always fail. It does so by reasoning about the checks as follows:

- `CNullOrLeq(e1,e2,e3)`, `CNullOrLeqNT(e1,e2,e3)` – If `e1` is the constant zero, or if the canonicalized expression `e3 - e2` is non-negative, then these checks can be removed. If `e3 - e2` is negative, then a compiler error is issued. The check might still succeed, but given where the Deputy type system prescribes the insertion of these checks, it is highly likely that there is a bug.
- `CLeq(e1,e2)`, `CLeqNT(e1,e2)`, `CLeqInt(e1,e2)` – If the canonicalized expression `e2 - e1` is non-negative, then these checks can be removed. If `e2 - e1` is negative, then a compiler error is issued.
- `CWriteNT(p, hi, e)` – If `e` is zero, or if the canonicalized expression `hi - (p + 1)` is non-negative, then this check can be eliminated.
- `CPtrArith(lo,hi,p,x,sz)`, `CPtrArithNT(lo,hi,p,x,sz)` – If the canonicalized expressions `p + x - lo` and `hi - (p + x)` are both non-negative, then this check can be removed. Both the upper and lower bounds are checked, so overflow is not a problem. Also, when handling pointer expressions, the expression canonicalizer ensures that all terms are in units of bytes, so the multiplication by `sz` is taken care of during canonicalization.
- `CNullUnionOrSelected(u,e)` – If `e` is the constant zero, then this check will always fail, and a compile time error is issued.

The second flow-insensitive phase is a peephole optimization. Certain combinations of checks can be folded into a smaller number of logically equivalent checks. The sequence `CPtrArith(lo,hi,p,x,sz); CLeq(p+x+1,hi)`, does three comparisons. It appears at certain kinds of array accesses. To avoid filling the type system with special cases, this case is dealt with in the optimizer. The two checks can be replaced by a new check called `CPtrArithAccess(lo,hi,p,x,sz)`, which ensures that `lo <= p + x * sz` and that `p + (x + 1) * sz <= hi`. This is an improvement because `CPtrArithAccess` does only two comparisons.

Also, the sequence `CNullOrLeq(e1,e2,e3); CNonNull(e2,e3)` is generated by the type system after calls to memory allocators, and does three comparisons. It can be replaced simply by `CLeq(e2,e3); CNonNull(e1)`, which does only two comparisons.

A Note about Overflow The checks `CPtrArith(lo,hi,p,x,sz)` and `CPtrArithNT(lo,hi,p,x,sz)` require that the address calculation `p + x` does not overflow. An important invariant here is that `[lo,hi)` is a valid range of memory. This means, among other things, that the expressions `lo` and `hi` are not the results of arithmetic overflow. If it is the case that they are not, and if we conclude using the method above that the canonicalized expressions `hi - (p + x)` and `(p + x) - lo` are both non-negative, then it is not possible that `(p + x)` could overflow since `hi` would have to include factors of both `p` and `x`, and `lo` would have to include at least a factor of `p`.

The invariant that `[lo,hi)` is a valid range is maintained by the checks at coercions. Such ranges "begin" valid because they are stack allocated, globals, or given by memory allocators, which Deputy instruments soundly. If `[lo,hi)` is a valid range, then `[lo',hi')` is a valid range when `lo <= lo' < hi' <= hi`. A similar argument can be made here about the sign of the canonicalized differences for these inequalities. Since overflow is handled soundly at coercions, it is also handled soundly at pointer arithmetic.

2.2 Flow-sensitive Optimizations

The flow-insensitive analyses are able to reason about many checks that can trivially be removed, and are able to identify some obvious bugs. Over the benchmarks used in section 4, the initial flow-insensitive phase removes 46% of the checks inserted by the first phase. The fist phase inserts many unnecessary checks because its implementation has been kept simple. Less transparent checks, and trickier bugs require reasoning aided by flow-sensitive analyses.

Analysis for `nullterm` Pointers The goal of this kind of analysis is to be able to better reason about the checks `CLeqNT` and `CPtrArithNT`. Recall that a pointer with type annotation `bound(lo,hi) nullterm` may safely be anywhere in `[lo, hi + strlen(hi)]`. Therefore, it is safe to increment such a pointer as long as the value referenced by the pointer is non-null. Knowing that a `nullterm` pointer is safe to increment allows us to remove many `CPtrArithNT` checks.

A forward dataflow analysis for determining when `nullterm` pointers may be incremented proceeds as follows. The abstract state of the program is a mapping from lvalues of `nullterm` type to the symbolic amount by which it is safe to increment them. The analysis is initialized by setting the state for the entry point to the empty mapping. Mappings are added to the state when the analysis encounters a branch such as `if( *p != 0 )` where `p` is a `nullterm` pointer. For this test when the branch is taken, the mapping `[p` → `1]` is added to the state. Also, if the program calls a `strlen`-like function on a `nullterm` pointer, then it is safe to increment that pointer by the resulting amount, so the argument pointer can be mapped to the lvalue in which the result is stored after the call.

At control flow join points, the meet of the incoming mappings is taken. In the current implementation the meet is defined as follows. Given two mappings, the meet of them has a mapping only if they both have exactly the same mapping.

A more precise formulation that is unimplemented is the following. Given two mappings, if one has a mapping for a particular lvalue, and the other does not, then the meet has no mapping. If both have a mapping for a particular lvalue, say `e1` for one and `e2` for the other, then the meet should contain a mapping to `min(e1,e2)`. This assumes that a separate analysis is able to tell which of `e1` and `e2` is smaller. In the absence of such an analysis, the meet would contain no mapping in this case. This more precise formulation of the meet remains unimplemented because inspection of our benchmarks indicates that the benefit would not be significant.

The results of this analysis can be used as follows. If the state at a check `CPtrArithNT(lo,hi,p,x,sz)` contains a mapping `[p → e]`, and it is known statically that `p + x <= e`, then the check can be removed.

A separate analysis tracks variables storing the length of null-terminated buffers. It is similar to the above analysis in that the abstract state is a mapping from `nullterm` lvalues to the variables storing the buffer length. Dataflow facts are added on calls to `strlen`-like functions, and the meet operation is similar.

The results of this analysis can be used as follows. If the state at a check `CLeqNT(e1,e2)` contains a mapping `[p → len]`, and it can be known statically that `e1 <= e2 + len`, then the check can be removed. If the relative magnitudes of the expressions are not known statically, then the check can be rewritten as the simpler check `CLeq(e1,e2 + len)`.

Intraprocedural Non-null analysis The goal of this analysis is to exploit checks for null-ness that the programmer has already written into the program. This information can be used to eliminate unnecessary calls to `CNonNull`, and to convert calls to `CNullOrLeq` into the simpler `CLeq`.

A forward dataflow analysis for determining what lvalues are non-null proceeds as follows. The abstract state of the program is the set of lvalues that are known to be non-null. We initialize the analysis by setting the state at the entry point to be the set containing the arguments to the function whose types have been annotated as `nonnull`. We add an lvalue to the set in three different cases. First, if a branch like `if( p )` is taken, or a branch like `if( !p )` is not taken, then on the appropriate branch, `p` can be added to the set. Second, after a call to `CNonNull(p)`, `p` can be added to the set. Finally, if the return type of a call has been annotated as `nonnull`, then the lvalue assigned to the call can be added to the set. At control flow join points, the meet operation is simply set intersection.

Octagon Analysis An octagon analysis tracks linear constraints among pairs of program variables. The representation of the abstract state varies according to implementation, however the Deputy optimizer relies on the following interface. It must be able to add facts and make queries of the form `x ± y <= c` where `x` and `y` are program variables.

The goal of this analysis is to reason about checks that compare the magnitude of expressions. These are `CPtrArith`, `CLeqInt`, `CLeq`, `CNullOrLeq`, and their `nullterm` variants. We have experimented with two different octagon analyses: a lightweight special-purpose analysis, and an analysis using an off-the-shelf library due to Miné [8]. We discuss the design of the optimizer around these two analyses in section 3, and we compare their performance in section 4.

Lightweight Analysis The special-purpose analysis is a forwards dataflow analysis that proceeds as follows. The abstract state of the program is a set of inequalities of the form `x + c <= y` where `x` and `y` are lvalues. The analysis is initialized by setting the state at the entry point to the empty set. There are three ways that facts can be added to the state: assignments, tests in the program, and Deputy checks. It should also be noted that for the purposes of

the analysis, we use a dummy lvalue for representing zero where necessary. It will be referred to as `zero` below.

The special purpose analysis reasons about three forms of assignment. First are assignments of the form `x = c` where `c` is an integer constant. Here we add the facts that `x - c <= zero`, and that `zero + c <= x`. Next, for assignments of the form `x = y` where `y` is an lvalue, all facts about `y` are added for `x`. Finally, for assignments of the form `x = y + c` where `y` is an lvalue and `c` is an integer constant, it is possible to add new facts when it is known statically that `y + c` does not overflow.

Since we make the conservative assumption that `x` and `y` may alias, it is not possible to add the facts that `x - c <= y` and `y + c <= x`. However, for each fact `l1 + n <= l2` in the state before the assignment, if `l1` is `zero`, and `l2` is `y`, then `zero + (n + c) <= x` can be added to the state. Also, when `x` and `y` are the same lvalue, and `l2` is `x`, then `l1 + (n + c) <= x` can be added to the state. Likewise, when `x` and `y` are the same, and `l1` is `x`, then `x + (n - c) <= l2` can be added to the state. The correctness of these additions can be seen by applying the Hoare rule for assignment. As suggested above, this analysis could be made more precise through the use of an alias analysis.

Facts from tests such as `if( e1 <= e2 )` are added along both branches. The negation of the test is added on the branch for which the test failed. Facts are extracted from the test by canonicalizing the difference of `e2 - e1`, and adding to the state that the difference is at least zero when the canonicalized difference is in terms of no more than two lvalues. A similar approach is taken with facts gathered from Deputy checks. At control flow join points the inequalities known to hold in both states are kept.

In order to determine what checks can be removed, we make queries that determine if, for example, `e1 <= e2`. As above, if the canonicalized difference is in terms of no more than two lvalues, say `x` and `y`, then we see if there are any inequalities in the current state relating `x` and `y` that are strong enough to prove the inequality. This lightweight analysis does not do any kind of transitive closure over the dataflow facts, but it will be seen that in combination with other transformations, this is not a problem.

Off-the-shelf Analysis The Analysis using Miné's library is also a forwards dataflow analysis. The difference from the lightweight analysis is that an external library is used to calculate state transitions, unions at control flow join points, widenings, and to make queries about what facts hold in each state.

We treat each unique lvalue in the function in question as a separate octagon variable, and make conservative assumptions about aliasing. Also, since each octagon operation is quadratic in the number of variables, we make use of the following optimization. Function lvalues are divided into families such that any two lvalues that appear in the same CIL instruction belong to the same family. The abstract state for the analysis is actually an octagon for each non-singleton family of lvalues. For several benchmarks this optimization produces an improvement of three to four orders of magnitude in the running time of the analysis. Also, because the library does a transitive closure over the inequalities

it is given, the transformations needed for the lightweight analysis are not as important.

Another Note about Overflow The argument about overflow used above can be extended to the cases in which we use the results of these more sophisticated anlyses to determine that canonicalized expressions are non-negative. The octagon analyses give information about the relative magnitudes of pairs of program lvalues. This information is gathered from tests that are already in the program. For example, the result of a canonicalized difference may be `(x - y)` where `x` and `y` are lvalues. If the program does not already contain tests at least as strong as `y <= x`, then it cannot be concluded that the difference is always non-negative. Further, if the canonicalized difference has more terms, then nothing is concluded.

Because our analyses are conservative, it is not necessary to reason about overflow in a more sophisticated way. This has both positive and negative consequences. The positive side is that the implementation remains simple. The negative side is that program bugs related to arithmetic overflow will not be caught statically.

Syntactic Check Propagation Not all checks can be reasoned about by our expression canonicalization and octagon analysis. However, checks like these may still be generated redundantly by the first stage of Deputy. We attempt to reason about checks like these syntactically using the following analysis.

This forward dataflow analysis is very similar to an available expressions analysis, except that instead of expressions, we are interested in Deputy checks. The abstract state of the program is a set of checks that are "available." A check is available at a program point if the check must be computed before reaching that point, and if there are no intervening modifications of the lvalues in the check. When a check is processed it is added to the set. At control flow join points, set intersection is used. At a check in the program, if a syntactically identical check is available, then the check can be eliminated.

Loop Analysis If a check inside of a loop is only in terms of lvalues that are loop invariant, then the check can be moved ahead of the loop. A standard reaching definitions analysis is used to identify checks containing only loop invariants. These are placed in a block guarded by the loop condition immediately preceding the loop. This is so that the checks are made only when the loop body is executed at least once.

Standard Utility Passes We also make use of some standard analyses to improve the performance of the optimizer itself and the resulting code. The first phase of Deputy creates many fresh variables that it uses for constructing its checks. When the checks are eliminated the created variables are no longer used. We do a number of dead code elimination passes. The dead code elimination is based on a reaching definitions analysis. Dead code elimination allows subsequent

analyses to be computed more efficiently since there is less code to analyze. We quantify these effects in the evaluation section.

We also use a symbolic evaluation transformation for a couple of reasons. First, it improves the effectiveness of the dead code elimination passes, but more importantly, it greatly improves the performance of our lightweight octagon analysis. As was mentioned earlier, the lightweight octagon analysis does not perform any kind of transitive closure on its state. We compensate for this by making a best effort to ensure that checks are always in terms of irreducible expressions. As will be seen in the evaluation section, this approach gives reasonable performance when compared with the off-the-shelf octagon library.

Soundness In order to be sound, we make conservative assumptions about aliasing. These conservative assumptions entail that all pointers may alias, and demand that the maintenance of dataflow facts across memory operations be done carefully. In all of the analyses described below, the situations in which dataflow facts must be killed are similar. On any write to memory, dataflow facts referring to global variables, variables whose addresses have been taken, and facts referring to the contents of memory must be killed. Further, in regards to function calls, the optimizer's reasoning is limited. Calls inserted by the first phase of Deputy are assumed to have no side effects on memory. Further, C library functions, like `strlen`, having the attribute `pure` are assumed to have no such side effects. It is assumed that any other call may modify memory arbitrarily, so again facts referring to globals, variables whose addresses have been taken, and facts referring to the contents of memory must be killed. Similar assumptions are made about inline assembly.

2.3 Inter-procedural Optimization

It is common for programmers to write functions that assume that parameters are non-null. It is also common for programmers to write functions for which it is impossible for the return value to be null. Making annotations for all of these cases can be tedious, so the Deputy optimizer includes a pass that deduces where these annotations can go. When a parameter is annotated with `nonnull`, the first phase of Deputy places the non-null check at the call site rather than inside of the callee. This exposes the checks to more opportunities for optimization.

Deducing which parameters can be annotated as `nonnull` is achieved through a backwards dataflow analysis similar to a standard "very busy" expressions analysis. A check is "very busy" at a program point if it must be calculated on every path following the point. If a check at least as strong as `CNonNull(p)` is very busy at the entry point of a function, and `p` is a parameter to the function, then the programmer has made the assumption that the function is only called with non-null `p`, and an annotation can be added to this effect.

Further, it can be deduced that the return type of a function can be annotated as `nonnull`. This analysis makes use of the available checks analysis described earlier. If at every statement `return p` in a function there is either:

1. A check available of the form `CNonNull(p)`,
2. `p` is of a `nonnull` type,
3. `p` is the result of pointer arithmetic, or
4. It has been deduced by the intra-procedural nullness analysis that `p` cannot be null.

then it is impossible for the function to return a null value. In this case the return type can be annotated `nonnull`.

It should also be noted that it is possible to discover richer pre- and post-conditions using this method. Mechanisms for exploiting these are currently under development.

3 Optimizer Design

Now that the optimization phases have been presented separately, their composition can be discussed. Two different designs will be presented. One will be built around the lightweight octagon analysis, and the other will be built around the off-the-shelf library. These will be referred to as LW and OTS, respectively.

3.1 The LW Optimizer

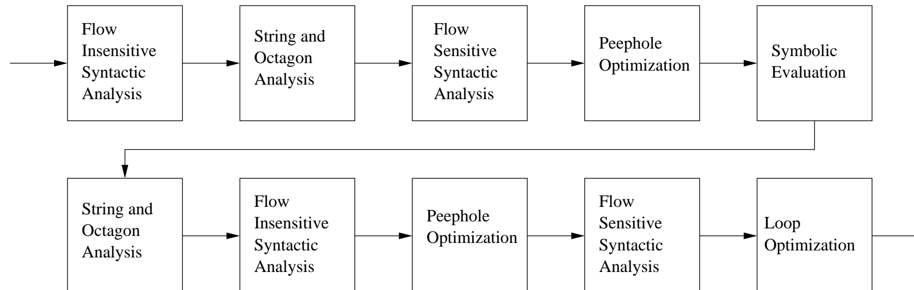

Fig. 4. The path that code takes through the LW optimizer

A flow diagram of the LW optimizer is shown in figure 4. The stage called "String and Octagon Analysis" performs the flow-sensitive string optimizations and the lightweight octagon analysis all at once. The other stages are named according to the phases described in the previous section.

The first flow-insensitive stage removes many checks that are easy to reason about. As discussed earlier, there are many checks here because one of the chief design goals for the first stage of Deputy was to keep the typing rules and implementation simple.

The second stage is made up of the flow sensitive string and octagon analyses. These analyses are relatively unsophisticated, and accumulate only a small number of facts. Therefore, the operations on the abstract state are efficient and fixed-points are reached quickly. For this reason, it is appropriate to run early in the optimizer when there are still many checks remaining in the code that can easily be seen to be redundant.

The third stage eliminates syntactically redundant checks that the second stage was unable to reason about. Before running the symbolic evaluator, a pass is made by the peephole optimizer. Results are similar as long as it is run once before symbolic evaluation, primarily to improve performance of later analyses, as the later peephole pass would optimize a superset of the checks improved by the first pass.

Now that much of the low hanging fruit has been picked off, reasoning about more subtle cases can be attempted. The symbolic evaluation operates on expressions inside of checks. Several of the passes examine checks syntactically with limited dataflow information. In many cases the information is so limited that expressions in checks contain terms not mentioned by the dataflow facts. When the check expressions are evaluated symbolically, more of them are in terms of the expressions and lvalues appearing in the facts gathered from dataflow analysis. This allows them to be reasoned about effectively with only a small number of dataflow facts.

After symbolic evaluation, the same passes are run again. They are followed by the pass for raising loop invariant checks out of loops. This phase is relatively expensive, so it is run last so that it operates on as little code as possible.

This approach has a couple advantages. First, all stages are either standard analyses and transformations used in compilers, or analyses that simply gather information from Deputy checks or checks that are already in the program. Second, since the analyses were kept simple, they only needed to be tuned to gather the information actually needed to reason about checks. This was an easy guide which allowed us to restrict their implementation to what was necessary.

3.2 The OTS Optimizer

A flow diagram of the OTS optimizer is shown in figure 5. Here, “Flow Insensitive Analysis” refers to both the syntactic analysis and the peephole optimizations, “String Analysis” refers to both of the string analyses, and “Octagon Analysis” refers to the analysis described in the previous section that uses the off-the-shelf octagon library. The other phases are named according to the phases described earlier.

The first few stages are as before except for the omission of the lightweight octagon analysis. Further, since the octagon analysis itself is expensive, it is important to minimize the amount of code on which it operates. Therefore, after the first few stages remove the easy-to-reason-about checks, a pass is made that eliminates dead code and unused variables.

Following this, the symbolic evaluation transformation allows the string analyses to reason about more checks, as before, and are followed by the flow insen-

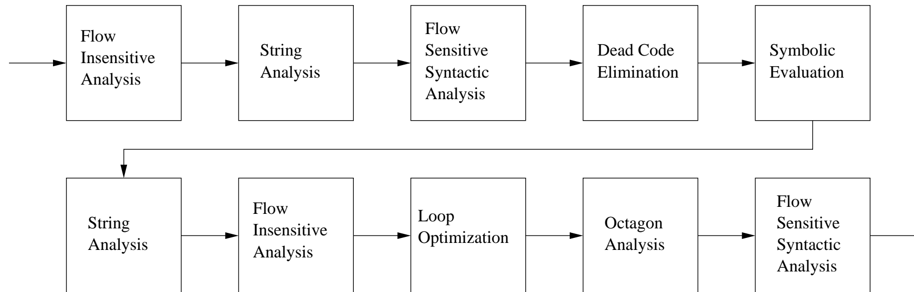

Fig. 5. The path that code takes through the OTS optimizer

sitive analyses and loop optimization. Now that the code is smaller, it is possible to run the octagon analysis somewhat efficiently. After that, a pass is made by the flow sensitive syntactic analysis to remove any redundant checks that the octagon analysis missed.

3.3 Interprocedural `nonnull` Optimization

The goal of the interprocedural optimizations is to reduce the number of times that an lvalue is checked for nullness. This is achieved through deducing what function parameters and return types can be annotated as `nonnull`. These annotations are discovered by doing the "very busy" check analysis described earlier. When instructed to do so, the LW optimizer includes this analysis after loop optimization. Then, annotated function prototypes can be emitted and merged with the original source so that the annotations can be used on each run with the extra analysis omitted.

The annotated function prototypes can be thought of as summaries of the functions. Iteration is achieved by doing subsequent "very busy" check analyses after re-running the intra-procedural non-null analysis. This method is capable of handling recursive and mutually recursive functions as well since it is no longer necessary to analyze a function once annotations have been discovered. Since the set of facts collected increases in size monotonically, and annotations can be placed in only a finite number of places, the analysis will terminate. In the current implementation, we rely on the first phast of Deputy to point out when a function with `nonnull` annotations is being assigned to a function pointer without the annotations. In this situation the annotations are erased and ignored both at call sites and in the function definition, i.e. the function is treated everywhere as unannotated. This situation could be handled more precisely by using a suitable subtyping relation for `nonnull` types, however this has not yet been implemented.

4 Evaluation

There are three metrics that can be used to judge the performance of the optimizer, and learn about the cost and benefits of its phases. These are:

1. The number of checks remaining after the optimizer runs.
2. The time it takes the optimizer to run.
3. The improvement in performance of benchmarks.

The number of checks remaining is a good indication of the effectiveness of the analyses and transformations. The runtime of the optimizer itself is a good indication of the cost of the analyses used to support optimization, and the performance of the benchmarks can indicate whether or not the expensive runtime checks are being effectively reasoned about.

The goal of the following experiments is to learn, first, how well each of the optimization phases performs under the above metrics, and also to compare the two different designs for the optimizer described in the previous section. The benchmarks used in the experiments are taken from SPEC[13], Mediabench[7], Ptrdist[1], and Olden[3]. The benchmarks used from these sources were selected because they were easy to convert to the Deputy type system, and because they exercise many of its features.

4.1 Overall Performance

Figure 6 shows the performance of the LW and OTS optimizers under the above metrics. All experiments were performed on a 2.67GHz Intel Core 2 Duo with 2GB of memory running on Linux 2.6.18 and compiled with gcc 4.1.1 and Deputy. The table shows the lines of benchmark code, the number of checks remaining, the time spent in optimization, and the percent slowdown of the benchmark when compared with compilation by `gcc` alone.

In summary, both leave a runtime check for every seven or eight lines of code, produce a slowdown that is generally under 25%, and process between 500 and 1500 lines of code in a second. On average, the OTS optimizer improves benchmark performance by a few percent, reduces the number of checks by about 5%, but runs about twice as slow as the LW optimizer.

A few of the benchmarks require explanation. Most notably, the `perimeter` and `array-incr` benchmarks that fall below the double line are interesting. The perimeter benchmark suffers a large slowdown because the runtime checks that the optimizer fails to remove prevent `gcc` from performing a tail-recursion optimization. These runtime checks are inserted to check the well-formedness of a deeply-nested tree data structure, and could possibly be removed with shape analysis. It should also be noted that the `perimeter` benchmark fails to check the return of `malloc` in a few cases. Appropriate checks are added here by Deputy, and should certainly not be optimized away.

The `array-incr` benchmark demonstrates a failure in both the lightweight and off-the-shelf octagon analyses. The deputized code appears in figure 7. The

Benchmark	Lines	Initial Checks	LW Optim Checks Left	LW Optim Optim Time	LW Optim Slowdown	OTS Optim Checks Left	OTS Optim Optim Time	OTS Optim Slowdown
SPEC								
go	29722	43385	4142	28.5s	17%	4072	45.1s	9%
gzip	8673	4890	558	2.3s	12%	533	9.3s	10%
li	9639	10961	2334	4.2s	50%	2152	9.7s	48%
Olden								
bh	1907	4613	347	2.1s	12%	264	4.8s	6%
bisort	684	331	17	0.1s	0%	15	0.6s	0%
em3d	585	306	48	0.3s	58%	45	1.3s	58%
health	717	505	35	0.3s	25%	35	1.1s	25%
mst	606	338	47	0.3s	7%	42	1.1s	7%
power	768	608	14	0.3s	0%	14	0.8s	0%
treeadd	377	79	7	0.2s	0%	7	0.8s	0%
tsp	565	446	25	0.2s	2%	25	0.7s	2%
Ptrdist								
anagram	635	20952	78	0.4s	4%	77	0.8s	4%
bc	7395	7433	1013	3.1s	23%	933	15.6s	16%
ft	1904	41194	69	0.5s	15%	68	1.2s	15%
ks	792	9476	46	0.3s	27%	44	0.8s	26%
yacr2	3976	3452	588	1.2s	98%	555	2.1s	74%
Mediabench								
adpcm	387	180	23	0.3s	12%	23	0.9s	12%
epic	3469	4923	688	2.6s	0%	657	6.4s	0%
total	72801	154080	10079			9561		
avg		8560			20%			17%
rate		0.5	7.2	1.5KLOC/s		7.6	.7KLOC/s	
perimeter	395		9	0.1s	340%	9	0.2s	340%
array-incr			1	0.03s	800%	1	0.03s	800%

Fig. 6. Deputy benchmarks. For each test we show the size of the benchmark, the number of checks inserted by the first phase of Deputy(including checks on global initializers), the number of checks remaining after optimization, the cost of optimization, and the slowdown of the benchmark for both the LW and OTS optimizers with respect to uninstrumented C code compiled by `gcc`.

```
1 int array[SIZE];
2 int main() {
3   int acc = 0, i = 0;
4   int *p = array;
5   while(i < SIZE) {
6     acc += *p;
7     CLeq(p + 1, array + SIZE);
8     p++; i++;
9   }
10  return acc;
11 }
```

Fig. 7. The array-incr benchmark. The remaining check is unnecessary, but evades analysis by an octagon analysis.

`CLeq` check is clearly unnecessary. As a sanity check to ensure that a misuse of the library was not the cause of imprecision, we ran Minè's reference implementation over this code after a translating it faithfully into the toy language on which it operates. It also failed to generate the information needed to reason about this check. On the other hand, it should be possible for a standard induction variable analysis to eliminate this check. It is not clear, however, that this will provide a significant advantage in the general case; the octagon analyses are able to eliminate many checks.

4.2 Cost and Benefit of Phases

Phase	Deputy Time	Checks Reduced	Speedup
Peephole	<1%	32%	2%
FI-Syntactic	4%	60%	55%
LW Octagon	8%	75%	56%
OTS Octagon	41%	77%	59%
FS-Syntactic	4%	16%	5%
Loop Analysis	7%	0%	8%
Dead Code Elim	5%	0%	<1%
Symbolic Eval	9%	35%	6%
Inter-nonnull	3%	<1%	<1%

Fig. 8. Cost and benefit of Deputy optimizer phases.

The table in figure 8 summarizes the cost and benefit of each of the optimization phases. The phase "FI-Syntactic" refers to the flow insensitive syntactic analysis. The phase "FS-Syntactic" refers to the flow sensitive syntactic analysis. The phase "Inter-nonnull" refers to the interprocedural nonnull analysis.

Further, the measurements for the intraprocedural nonnull analysis, string analysis, and lightweight octagon analysis are folded into a single row since they are implemented together, and difficult to extract from each other.

These figures were obtained by running the Deputy optimizer over the benchmarks above the double bar line in figure 6. The percent differences are obtained by comparing the baseline optimizer with one in which the phase in question has been disabled. The column labeled "% Deputy Time" shows what percent of the total Deputy runtime is spent in that optimization phase. The column labeled "Checks Reduced" refers to the percent by which the number of static checks in the program was reduced. The column labeled "Speedup" refers to the percent by which the performance of benchmarks was improved on average. These percentages are all with respect to the LW optimizer except for the OTS Octagon phase. Its effectiveness was measured by removing it from the OTS optimizer.

Each of the phases requires under 10% of the Deputy runtime. Even, the flow insensitive phases, which run over an inflated amount of code, do so very quickly. We have therefore confirmed the assertions about the performance of the phases made earlier. The table shows that earlier phases eliminate a large number of obviously needless checks. Symbolic evaluation allows reasoning about more difficult checks in the LW optimizer. The flow sensitive syntactic analysis removes duplicates that defy analysis by other stages.

Loop analysis is not as effective as other stages. A survey of the resulting code reveals that only a small number of `CNonNull` checks are being lifted out of loops, and as these checks are inexpensive to begin with–consisting only of a test and a jump–loop optimization only improves performance in a small number of cases. The interprocedural `nonnull` analysis was able to discover many types that could be annotated. It found over 50 annotations in the `li` benchmark. However, as mentioned, `CNonNull` is an inexpensive check, especially for a CPU with good branch prediction, so the performance improvement is not impressive.

4.3 Discussion

The table in figure 9 shows the breakdown of the remaining checks in the benchmarks above. The `CPtrArithAccess` checks nearly hold a majority. This is due to the `go` benchmark. It uses entries in global arrays to index into other global arrays. Without any interprocedural analysis, or detailed knowledge of the implementation of the benchmark, these checks are beyond the simple analyses presented here. When the `go` benchmark is left out, `CPtrArithAccess` checks only account for 17% of the remaining checks, and the plurality of remaining checks belongs to the `CNonNull` checks, which as mentioned before, are inexpensive.

4.4 Real Applications

After obsessing over micro-benchmarks, it is important to remember that when applied to real applications like Linux drivers, and TinyOS, the overhead of code optimized by these phases is generally reasonable. Several Linux device drivers,

Type	% of Those Remaining
`CSelected`	5.5%
`CPtrArithAccess`	47.0%
`CNullOrLeqNT`	<1%
`CLeq`	9.6%
`CLeqNT`	<1%
`CNotSelected`	7.9%
`CPtrArith`	4.4%
`CWriteNT`	<1%
`CPtrArithNT`	1.8%
`CNonNull`	19.1%
`CNullOrLeq`	4.0%

Fig. 9. Breakdown of the remaining checks.

and TinyOS modules, have been "deputized." Experiments in [14] and [4] show that the checks remaining after optimization have reasonable overhead. Results from Linux driver experiments are reproduced here.

Benchmark	Driver	Native Throughput	Deputy Throughput	Native CPU %	Deputy CPU %
TCP Receive	e1000	936Mb/s	936Mb/s	47.2	49.1 (+4%)
UDP Receive	e1000	20.9Mb/s	17.4Mb/s (-17%)	50.0	50.0
TCP Send	e1000	936Mb/s	936Mb/s	20.1	22.5 (+12%)
UDP Send	e1000	33.7Mb/s	30.0Mb/s (-11%)	45.5	50.0 (+9%)
TCP Receive	tg3	917Mb/s	905Mb/s (-1.3%)	25.4	27.4 (+8%)
TCP Send	tg3	913Mb/s	903Mb/s (-1.1%)	18.0	20.4 (+13%)
Untar	usb-storage	1.64MB/s	1.64MB/s	5.5	6.8 (+23%)
Aplay	emu10k1	n/a	n/a	9.10	9.64 (+6%)
Aplay	intel8x0	n/a	n/a	3.79	4.33 (+14%)
Xinit	nvidia	n/a	n/a	12.13	12.59 (+4%)

Fig. 10. Benchmarks measuring Deputy overhead. Utilization numbers are kernel CPU utilization.

Table 10 shows a comparison between native and deputized drivers. For network drivers, throughput and kernel CPU overhead were measured on a dual Xeon 2.4GHz machine. For the TCP experiments, 32KB packets were sent and received. For the UDP experiments, 16KB packets were sent and received. The `usb-storage` driver was tested by untarring the Linux source from a Flash drive back onto the drive itself. Sound drivers were tested by playing a sound, and the video driver was tested by setting up and tearing down an X Window session. More detailed explanations of these experiments can be found in the papers men-

tioned above. The important point for the optimizer is that overhead in terms of decreased throughput and increased CPU utilization is not burdensome.

5 Related Work

The Deputy type system and optimizer were key tools in the development of SafeDrive[14], a system for improved memory safety and recovery for Linux device drivers. The Deputy type system is described briefly there, but more thoroughly in an accompanying technical report[4]. Neither of these papers describe the optimizer in detail, which is the goal of this report.

There are other systems that perform static analysis guided by programmer annotations. Microsoft's SAL annotation language [6] is very similar to Deputy's. The ESPX checker attempts to find bugs statically based on these annotations. The Deputy optimizer is similar in that the static analysis can be used to find bugs. The difference is that, for Deputy, soundness must be maintained. For this reason, the primary focus of the Deputy optimizer is figuring out what run-time checks may be soundly removed rather than figuring out what operations will always cause(or lead to) an access violation.

CCured[9] is a whole program analysis that instruments pointers with bounds information. As in Deputy the bounds information may need to be checked at runtime. The CCured system includes an optimization phase that also attempts to reason about bounds checks, null checks, and others. The Deputy type system presents its optimizer with some different challenges than those seen by the CCured optimizer. The primary challenge is the sheer volume of runtime checks added by the type system. On the benchmarks presented here, deputized code achieves better performance than cured code, however CCured also enforces allocation safety, which Deputy does not.

ABCD[2] attempts to eliminate array bounds checks for Java at runtime. It uses a domain similar to the one used by the Deputy optimizer in the lightweight octagon analysis. Both ABCD and the Deputy optimizer must be fast, but for different reasons. ABCD must be fast because it is optimizing while the program is running, whereas the Deputy optimizer must be fast because the volume of checks is large.

There are also mechanisms for removing array bounds checks based on theorem proving and predicate abstraction[10, 11], but this approach is probably too heavyweight in the context of Deputy. By the time that our simpler analyses have run, the expense of more sophisticated techniques may outweigh the benefit.

6 Conclusions

This report presented a detailed description of the Deputy optimizer. There were three constraints on its design. First, it needed to be lightweight so that the large number of checks generated by static type checking could be handled. Second, in spite of its low complexity it had to eliminate enough spurious checks so that

the increased memory safety provided by the type system would come at a low cost. Finally, the analyses used needed to be strong enough to identify runtime checks that would always fail, i.e. to find bugs.

These constraints were satisfied through a series of lightweight syntactic analyses sometimes with the help of sparse dataflow information, and standard compiler transformations. Earlier analyses were fast and targeted at obvious checks so that the burden on later more sophisticated techniques would be less.

As evidence that the goals of the optimizer were accomplished, we reported the results of several benchmarks. They showed that although there is room for improvement in a few selected cases, overall performance is on par or better than optimization passes in similar tools. Further, a comparison was made with an analysis built around an off-the-shelf octagon library. In terms of benchmark performance, the LW optimizer is probably a better approach. Finally, performance of deputized code for real applications, namely Linux device drivers, were shown to have reasonable performance.

6.1 Future Work

The most direct route to the biggest improvement in the Deputy optimizer is probably through an alias analysis, and the calculation of procedure summaries. These could be used to maintain dataflow facts across procedure calls that are now lost needlessly.

Further, the "very busy check" analysis used to deduce `nullterm` annotations can be used to extract richer pre- and post-conditions for functions. These pre- and post-conditions could then be fed back into the first phase of Deputy to help it generate fewer checks. It is also easy to envision further uses for this information. It could be merged into annotation libraries, or used as a way to reduce the cost of modeling function calls in concolic execution tools.

References

1. Austin, T. M., Breach, S. E., and Sohi, G. S. Efficient detection of all pointer and array access errors. In *PLDI'94*.
2. Bodik, R., Gupta, R., and Sarkar, V. Abcd: eliminating array bounds checks on demand. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (2000).
3. Carlisle, M. C. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, June 1996.
4. Condit, J., Harren, M., Anderson, Z., Gay, D., and Necula, G. Dependent types for low-level programs. Tech. Rep. EECS-2006-129, UC Berkeley, 2006.
5. Flanagan, C. Hybrid type checking. In *POPL'06*.
6. Hackett, B., Das, M., Wang, D., and Yang, Z. Modular checking for buffer overflows in the large. Technical Report MSR-TR-2005-139, Microsoft Research, 2005.
7. Lee, C., Potkonjak, M., and Mangione-Smith, W. H. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture* (1997).

8. Miné, A. The octagon abstract domain. In *Higher-Order and Symbolic Computation* (2006).
9. Necula, G. C., Condit, J., Harren, M., McPeak, S., and Weimer, W. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems 27*, 3 (2005).
10. Necula, G. C., and Lee, P. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA* (1996).
11. Necula, G. C., and Lee, P. The design and implementation of a certifying compiler. *SIGPLAN Not.* (2004).
12. Necula, G. C., McPeak, S., and Weimer, W. CIL: Intermediate language and tools for the analysis of C programs. In *International Conference on Compiler Construction* (2002).
13. SPEC. Standard Performance Evaluation Corporation Benchmarks. *http://www.spec.org/osg/cpu95/CINT95* (July 1995).
14. Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., and Brewer, E. Safedrive: Safe and recoverable extensions using language-based techniques. In *OSDI'06* (2006).