Efficient Detection of All Pointer and Array Access Errors*

Todd M. Austin Scott E. Breach Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
{austin, breach, sohi}@cs.wisc.edu

December 1, 1993

Abstract

In this paper, we present a pointer and array access checking technique that provides complete error coverage through a simple set of program transformations. Our technique, based on an extended safe pointer representation, has a number of novel aspects. Foremost, it is the first technique that detects all spatial and temporal access errors. Its use is not limited by the expressiveness of the language; that is, it can be applied successfully to compiled or interpreted languages with subscripted and mutable pointers, local references, and explicit and typeless dynamic storage management, *e.g.*, C. Because it is a source level transformation, it is amenable to both compile- and run-time optimization. Finally, its performance, even without compile-time optimization, is quite good. We implemented a prototype translator for the C language and analyzed the checking overheads of six non-trivial, pointer intensive programs. Execution overheads range from 130% to 540%; with text and data size overheads typically below 100%.

*© 1993 by Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. This work was supported by grants from the National Science Foundation (grant CCR-9303030) and Office of Naval Research (grant N00014-93-1-0465).

1 Introduction

It is not difficult to convince programmers (or employers of programmers) that programming errors are costly, both in terms of time and money. Memory access errors are particularly troublesome. A *memory access error* is any dereference of a pointer or subscripted array reference which reads or writes storage outside of the referent. This access can either be outside of the address bounds of the referent, causing a *spatial access error*, or outside of the lifetime of the referent, causing a *temporal access error*. Indexing past the end of an array is a typical example of a spatial access error. A typical temporal access error is assigning to a heap allocation after it has been freed.

Our own experiences as programmers as well as published evidence lead us to believe that memory access errors are an important class of errors to reliably detect. For example, in [MFS90], Miller *et. al.* injected random inputs (a.k.a "fuzz") into a number of Unix utilities. On systems from six different vendors, nearly all of the seemingly mature programs could be coaxed into dumping core. The most prevalent errors detected were memory access errors. In [SC91], Sullivan and Chillarege examined IBM MVS software error reports over a four year period. Nearly 50% of all reported software errors examined were due to pointer and array access errors. Furthermore, of these errors, 25% were temporal access errors – an error our checking methodology is particularly adept at catching.

Memory access errors are possible in languages with arrays, pointers, local references, or explicit dynamic storage management. Such errors are particularly difficult to detect and fix because:

- The effects of a memory access error may not manifest themselves except under exceptional conditions.
- The exceptional conditions which lead to the program error may be *very* difficult to reproduce.
- Once the error is reproduced, it may be very difficult to correlate the program error to the memory access error.

Consider the erroneous C function in Figure 1. This function can create a memory access error in the `return` statement expression. The function will reference the word immediately following the array referenced by the pointer `data` if the array does not contain the token. To fix this function, the `return` expression should be changed to `(i < count)`.

The function illustrates the three difficulties in finding and fixing memory access errors. First, `FindToken()` will only produce an incorrect result if the word following the array referenced by `data` contains the same value as `token` (or is inaccessible storage). This event is unlikely if the word contains an arbitrary value. Second, when (or if) `FindToken()` creates an incorrect result, it will be difficult to recreate during debugging. The programmer will have to condition the inputs of the program such that the word following the array referenced by `data` once again contains the same value as `token`. If the value of the illegally accessed word is independent of the value of `token`, the probability of success will be very low. Third, correlating the visible errors of the program to the incorrect actions of `FindToken()` may be very difficult. This connection may be very subtle and may not be visible for a long period of time.

In the vernacular of fault tolerant computing circles, debugging can be viewed as correlating the program fault to the program error. A program error is defined as an output of a program that is incorrect with respect to the specification of that program – this effect is what the users see. The program fault, on the other hand, is the initial incorrect condition (possibly many) that ultimately caused the error condition to occur. The primary goal of any good debugging environment is to provide a good

```
int FindToken(int *data, int count, int token) {
    int i = 0, *p = data;
    while ((i < count) && (*p != token)) {
        p++; i++;
    }
    return (*p == token);
}
```

Figure 1: A C function with a (spatial) memory access error.

correlation between faults and errors. Our checking methodology ensures that memory access errors are detected immediately, thus creating perfect correlation between the error and the fault, simply because they are one in the same![1]

Many execution environments do provide some level of protection against memory access errors. For example, in most Unix based systems, a store to the program text will cause the operating system to terminate execution of the program (usually with a core dump). Unix typically provides storage protection on a segment granularity – the segments are the program text, data, and stack. Other, more hostile environments such as MS-DOS, do not offer such luxuries, and stores to the program text may or may not manifest themselves as a program error. If a program error does occur, correlating it to a fault may be difficult, if not impossible.

Ideally, we would like the language execution environment to support memory access protection at the variable level, that is, an access to a variable should only be valid if the access is within the range (for both time and space) of the intended variable – all other accesses should immediately flag an error. We call any program that supports these execution semantics a *safe* program.

Our solution to the memory access error problem is simple and provides efficient and immediate detection of all memory access errors. We transform programs, at compile-time, to use an extended pointer representation which we call a *safe pointer*. A safe pointer contains the value of the pointer as well as *object attributes*. The object attributes describe the location, size and lifetime of the pointer referent. When a safe pointer value is created, either through the use of the reference operator (*e.g.*, '&' in C) or through explicit storage allocation, we attach to it the appropriate object attributes. As the value is manipulated, through the use of pointer operators, the object attributes are transferred to any new safe pointer values. Detecting a memory access error involves simply validating dereferences against the object attributes – if the access is within the space and time bounds of the object, it is permitted, otherwise an error is flagged and the access error is detected immediately.

We implemented a prototype source-to-source translator for the C language and examined the performance of six non-trivial programs. The performance is quite good. Instruction execution overheads range from 130% to 540%, and text and data size overheads are typically below 100%. We also benchmarked our prototype system against two commercially available tools that support memory access checking (Purify [HJ92] and CodeCenter [KLP88]) and found that our checking technique consistently uses less resources, even while providing better error coverage for memory access errors.

This paper is organized as follows. Section 2 introduces our extended safe pointer representation. Section 3 details the program transformations required to create safe programs, and in Section 4 we

[1] Hence, we can use the terms *memory access fault* and *memory access error* interchangeably under our checking methodology, as they are indeed the same.

```
typedef {
    <type> *value;
    <type> *base;
    unsigned size;
    enum {Heap=0, Local, Global} storageClass;
    int capability; /* plus FOREVER and NEVER */
} SafePtr<type>;
```

Figure 2: Safe pointer definition. This C-like type definition is parameterized by `<type>`, the type of the pointer referent.

discuss the translation and performance implications of providing complete error coverage. In Section 5, we present compile- and run-time optimization frameworks. Section 6 describes our prototype implementation and presents results of our analyses of six programs. Section 7 compares our checking technique to other published techniques. Section 8 concludes the paper.

2 Safe Pointers

To enforce access protection, we must extend the notion of a pointer value to include information about the referent. The idea is similar to tagged pointers used in many Lisp implementations [Lee91]. Figure 2 shows our safe pointer representation. The definitions of the contained fields follow:

value: The value of the safe pointer; it may contain any expressible address.

base and **size**: The base address of the referent and its size in bytes. In languages where pointers are immutable, `base` is redundant and may be omitted. With this information, we can detect all spatial access errors with a range check.

storageClass: The storage class of the allocation, either `Heap`, `Local`, or `Global`. Using this value, it is possible to detect errant storage deallocations, *e.g.*, it is illegal to free a global or local variable.[2]

capability: A capability to the referent. When dynamic variables are created, either through explicit storage allocation (*e.g.*, calls to `malloc()`) or through procedure invocations (*i.e.*, a procedure call creates the local variables in the stack frame of the procedure), a unique capability is issued to that storage allocation. The unique capability is also inserted into an associative store called the *capability store* and deleted from that store when the dynamic storage allocation is freed or when the procedure invocation returns (the exact mechanics of this process are discussed in a following section). Thus, the collection of capabilities in the capability store represent all active dynamic storage. Temporal access errors occur whenever a reference is made through a stale pointer, *i.e.*, a pointer which references storage whose capability is no longer in the capability store. Two capabilities are predefined. FOREVER is unique and always exists in the capability store; this capability is assigned to all global objects. NEVER is unique and never exists in the capability store; this capability can be assigned to invalid pointers to ensure any dereference causes an error.

[2] We cannot completely encode this information into the capability field. We need to discern local objects from heap objects so that errant frees can be detected. This distinction is not possible by simply examining the capability field, as both locals and heap objects have real capabilities. If the various storage classes have distinct locations in the address space, it may be possible to derive this value from the address in `base`.

The `value` attribute is the only safe pointer member that can be manipulated by the program source; all other members are inaccessible. `base` and `size` are the *spatial attributes.* `storageClass` and `capability` are the *temporal attributes.*

Safe pointers can exist in three states: *unsafe*, *invalid*, and *valid.* If the object attributes are incorrect, we say that the pointer has become *unsafe*; dereferencing this pointer may cause an undetected memory access error. It is the goal of this work and the impetus of the discussion in Section 4 to ensure that a safe pointer never becomes unsafe. If the safe pointer is not unsafe, it is either *invalid* or *valid*, depending on whether a dereference would flag an error. Languages with mutable pointers allow the program to legally create invalid pointers; for example, iterating a pointer across all the elements of an array exits the loop with the pointer pointing to the memory location following the last object. If the invalid pointer is never dereferenced, the program would not be in error. This behavior illustrates precisely why we only place error checks at dereferences; it is not illegal to have an invalid pointer – only to use it.

The initial value of a safe pointer, if not specified by an initialization expression, must be invalid. This condition ensures that a dereference before the initial assignment is detected. A simple way to invalidate a pointer value is to assign it the unique capability NEVER.

3 Program Transformations

Creating a safe program from its unsafe counterpart involves three transformations: pointer conversion, check insertion, and operator conversion. The first, pointer conversion, extends all pointer definitions and declarations to include space for object attributes. Check insertion instruments the program to detect all memory access errors. Operator conversion generates and maintains object attributes. In this section, we also describe the run-time support.

3.1 Pointer Conversion

All pointer definitions and declarations must be extended to include object attributes. To make this transformation transparent, the composite safe pointer must mimic the first class value semantics of scalar pointers. That is, when passed to a function, the safe pointer must be passed by value, and when operators are applied to a safe pointer, the result, if a pointer, must be a new safe pointer.

There is no need to add object attributes to array variables. Array variables (in the C sense) are merely address constants, and thus only exist as statically allocated objects or within structure definitions; as a result, the spatial attributes can be generated from the address constant and its type size, and the temporal attributes can be taken from the safe pointer to the containing object or derived from the array name.

3.2 Check Insertion

Assuming the safe pointer object attributes are correct (how to ensure this property is detailed in the following sections), complete safety for all pointer and array accesses is provided by inserting an access check before each pointer or array dereference.[3]

The dereference check first verifies that the referent is alive by performing an associative search for the

[3] We use the term *dereference* as a blanket term for any indirect access – either through application of the dereference operator (*e.g.*, '`*`' or '`->`' in C) or through indexing an array or pointer variable (*e.g.*, '[ ]' in C).

```
void ValidateAccess(<type> *addr) {
    if (storageClass != Global && !ValidCapability(capability))
        FlagTemporalError();
    if ((unsigned)addr - (unsigned)base > size-sizeof(<type>))
        FlagSpatialError();
    /* valid access!  */
}
```

Figure 3: Memory access check. This C-like function is parameterized by `<type>`, the type of the safe pointer referent. `FlagTemporalError()` performs system specific handling of a temporal access error, *e.g.*, force a core dump. `FlagSpatialError()` performs the same function, but for a spatial access error. The function `ValidCapability()` indicates whether or not the passed capability is currently active, *i.e.*, in the capability store.

referent's capability. If the referent has been freed, the capability would no longer exist in the capability store and the check would fail. Because capabilities are never re-used, the temporal check fails even if the storage has been reallocated. Once the storage is known to be alive, a bounds check is applied to verify that the entire extent of the access fits into the referent.

Our access check, shown in Figure 3, takes advantage of the wrap-around property of unsigned arithmetic to simplify the bounds check. If the accessed address is prior to the start of the array, the unsigned subtraction underflows and creates a very large number, causing the test to fail. The advantage of this expression over traditional bounds checks[4] is that it only requires one conditional branch to implement. This simplification reduces the additional control complexity introduced by dereference checks, which can result in better optimization results and better dynamic executions.

3.3 Operator Conversion

Pointer operators must interact properly with the composite safe pointer structure. When applied, they must reach into the safe pointer to access the pointer value. If the operator creates a new pointer value, it must include an *unmodified* copy of the pointer operand's object attributes.[5] For example, in the C statement `q = p + 6`, the application of the '+' operator on the pointer `p` creates a new safe pointer which is assigned to `q`. The new pointer value in `q` shares the same object attributes as `p`. Operators which manipulate pointer values never modify the copied object attributes because changing the value of the pointer does not change the attributes of the storage it references. This property holds even for pointers to aggregate structures. In this case, the object attributes refer to the entire aggregate.

The assignment operator requires special handling if the right hand side is a constant. Two common pointer constants are the NULL value and string constants (for C). If the assignment value is NULL, the NULL value can be replaced by an invalid safe pointer value, *e.g.*, one with the capability NEVER. For string constants, we can generate the needed object attributes at compile-time. If the right hand side of the assignment is a pointer expression, the resulting pointer value (and its object attributes) is copied to the pointer named on the left hand side of the assignment.

[4] Our check is functionally equivalent to:

```
(addr < base || addr > base+size-sizeof(<type>))
```

which requires two conditional branches (or extra instructions to combine the boolean terms).

[5] In C there exists one operator with two pointer operands, namely '-', which produces the difference between two pointers. The semantics of this operator imply that the object attributes of both operands should refer to the same object, so we can take the object attributes from either operand.

Expression	Access Path		
	Prefix	Prefix Type	Suffix
`a`	`a`	direct	–
`a.b`	`a`	direct	`b`
`a.b.c[4].d`	`a`	direct	`b.c[4].d`
`(**p)[3]`	`**p`	indirect	`[3]`
`(*p)->b`	`*p`	indirect	`b`
`w->x`	`w`	indirect	`x`
`w->x->y`	`w->x`	indirect	`y`
`w->x->y[3].z->c[4].b`	`w->x->y[3].z`	indirect	`c[4].b`

Table 1: Access path examples for C. In the above examples, the field `c` is an array variable, and `y` is a pointer.

Casting between pointer types does not require any special program transformations. Casting only alerts the compiler that future pointer arithmetic or dereferences of a particular pointer value should be made with respect to the new type size. Casting to a non-pointer type requires that the object attributes be dropped (if only pointers carry object attributes) and then the cast is carried out as defined by the language. Casting from a non-pointer type to a pointer type is problematic if non-pointer types do not carry object attributes. We address this problem in Section 4.

Handling of the reference operator, *e.g.*, the '`&`' operator in the C statement `q = &p->b[10]`, is slightly more complex as it must generate object attributes.

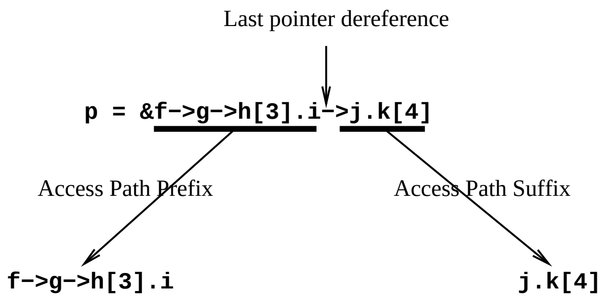
The reference operator is applied to an expression (`p->b[10]`, in our example) which names some storage. We call this expression the *access path.* The result of the operation is a new pointer value to the referent named by the expression.

To facilitate conversion, we decompose access paths into two parts, a prefix and suffix. The *access path prefix* is always non-empty and describes the sequence of variable names, dereferences, subscripts, field selectors, and pointer expressions leading to the memory object being referenced. It is from this prefix that we generate temporal attributes. The remaining part of the access path, the *access path suffix*, is composed of a sequence of field selectors and subscripts (on array variables only). The suffix describes what extent of the object is being referenced.

We further classify access paths as *direct* or *indirect.* A direct access path refers to an object in the global or local space by name. An indirect access path contains at least one pointer traversal.

Given a reference operator expression, we can parse the prefix by traversing the expression tree starting with the left-most, lowest precedence operator. The part of the expression up to but not including the last pointer traversal is the access path prefix, the remaining part of the expression is the access path suffix. If the access path does not contain any pointer traversals, the access path prefix is the name of the referenced variable. Table 1 shows a number of expressions and their decomposed access paths.

Temporal attributes are derived from the access path prefix. If the prefix is direct, the referenced object is either a global or a local variable. If global, we assign the capability FOREVER to the new safe pointer. If local, we assign the capability allocated to the local variable's stack frame (frame capability allocation is discussed in the following section). If the access path prefix is indirect, the temporal attributes are taken from the safe pointer named by the access path prefix.



```
p.value = &f->g->h[3].i->j.k[4]
p.base = f->g->h[3].i->j.k
p.size = sizeof(f->g->h[3].i->j.k)
p.storageClass = f->g->h[3].i.storageClass
p.capability = f->g->h[3].i.capability
```

Figure 4: Access path decomposition and safe pointer construction. In this example, `h` is a pointer, while `k` is an array variable.

To generate the spatial attributes for the reference, we start with the spatial attributes of the access path prefix. This value is either the address and size of the named variable if the access path prefix is direct, or the spatial attributes from safe pointer if it is indirect. Using these spatial attributes, we compute the actual base of the reference from the access path suffix, which describes the sub-object being referenced. Since all members of the referenced object (*i.e.*, the member of any contained structure) are of a known size, we can easily compute the offset into the object and its size at compile time. In the event the final term of the suffix is a subscript, we set the spatial attributes to the extent of the entire array. This technique allows the safe pointer to be subsequently manipulated to point to other members of the array.

Figure 4 shows an access path, its decomposition into the prefix and suffix, and the C statements required to construct the correct safe pointer value in `p`. The example also demonstrates the widening required if the final term is an array subscript.

3.4 Run-Time Support

The explicit storage allocation mechanism must be extended to create safe pointers. During allocation, a capability must be allocated for the storage, and any contained pointers must be invalidated. At deallocation, the capability given to the storage must be destroyed.

Figure 5 shows how this support would be provided for `malloc()`, the storage allocator provided under Unix. During allocation, `malloc()` generates a safe pointer using the size and location of the allocation request. The call to `NextCapability()` returns the next available and unused capability. `NextCapability()` can be implemented with an incrementing counter or a pseudo-random number generator. The capability is inserted into the capability store via the call to `InsertCapability()`. The call to `bzero()` clears the entire storage allocation. This action ensures that any pointers in the untyped allocation are initially invalid (assuming the storage class of `Heap` and capability NEVER are both assigned the value of 0). `calloc()` simply calls `malloc()` as in either case the storage is cleared before it is returned.

The implementation of `realloc()` is slightly more subtle. This function takes an existing storage allocation and resizes it to the requested size. The reallocated storage *may* move for any request, either

```
void *malloc(unsigned size) {
    void *p;
    p.base = p.value = unsafe_malloc(size);
    p.size = size;
    p.storageClass = Heap;
    p.capability = NextCapability();
    InsertCapability(p.capability);
    bzero(p.value, size); /* assuming capability NEVER == 0 */
    return p;
}

void *calloc(unsigned nelem, unsigned elsize) {
    return malloc(nelem*elsize);
}

void *realloc(void *p, unsigned size) {
    void *new;
    new = malloc(size);
    bcopy(p.base, new.base, min(size, p.size));
    free(p);
    return new;
}

void free(void *p) {
    if (p.storageClass != Heap)
        FlagNonHeapFree();
    if (!ValidCapability(p.capability))
        FlagDuplicateFree();
    if (p.value != p.base)
        FlagNonOriginalFree();
    DestroyCapability(p.capability);
    unsafe_free(p.value);
}
```

Figure 5: Safe malloc implementation with additional checking. InsertCapability(), ValidCapability(), and DestroyCapability() insert, locate, and delete capabilities, respectively. bzero() clears size bytes of memory starting at p.value. bcopy() copies min(size, p.size) bytes of storage from p.base to new.base. FlagDuplicateFree() is a system specific function which flags an error indicating that the program attempted to free a previously freed heap allocation. FlagNonHeapFree() flags an error indicating that the program attempted to free memory that is not in the heap. FlagNonOriginalFree() flags an error indicating that the program attempted to free memory without using a pointer to the head of the allocation. NextCapability() returns the next unique capability. unsafe_malloc() and unsafe_free() are interfaces to the system-defined storage allocator.

```
void Func(int a) {
    /* procedure prologue */
    unsigned frameCapability = NextCapability();
    InsertCapability(frameCapability);
    ZeroFramePointers(); /* assumes capability NEVER == 0 */
        .
        .
    /* procedure epilogue, common function exit point */
    DestroyCapability(frameCapability);
    return;
}
```

Figure 6: Function frame allocation and deallocation. `InsertCapability()` inserts a capability into the capability store. `ZeroFramePointers()` is a system specific function which clears all pointers in the newly allocated stack frame. `DestroyCapability()` deletes a capability from the capability store. `NextCapability()` returns the next unique capability.

larger or smaller. If moved, the contents of the new allocation will be unchanged up to the lesser of the new and old sizes. In our safe programming environment, we must move the storage in all cases, otherwise, there may exist safe pointers (which we cannot locate and change) whose object attributes have incorrect records of the referent size. If dereferenced, these pointers may flag errors even though the access was valid in the reallocated storage, or worse, the reallocation may have shrunk the referent, creating unsafe pointers whose referent sizes are too large. We can solve both these problems by always moving the storage. This action will force the program to update any old pointers to the previous allocation. Because the reallocated storage is allocated under a new capability, any stale pointers to the previous allocation will flag errors if dereferenced. We need not clear the remaining storage in the reallocation if it is larger, as the call to `malloc()` returns cleared storage.

At calls to `free()`, the capability of the allocation (contained in the safe pointer object attributes) is deleted from the capability store by the call to `DestroyCapability()`. Our implementation also verifies that the freed storage is indeed a heap allocation and a pointer to the head of the allocation (as this condition is required by `free()`).

The same allocation mechanism is applied to the dynamic storage allocated in procedure stack frames. When a function is invoked, a capability must be allocated for the entire frame if it contains any referenced locals. Any pointers contained in the frame must be set to an invalid state.

Figure 6 shows how this rewriting would be done for a C function. The function `ZeroFramePointers()` serves the same purpose as the call to `bzero()` in `malloc()`; it ensures that any pointers in the procedure stack frame are initially invalid by clearing the frame storage. Because stack frame allocations are strongly typed, `ZeroFramePointers()` could be replaced by NULL assignments to all the frame pointers.

If the language supports non-local jumps, *e.g.*, `longjmp()` in C, the run-time support must delete the frame capabilities of any elided function frames. This operation can be simply and portably implemented if the local capability space and heap capability space are kept disjoint, and function frame capabilities are allocated using an incrementing counter. The allocation of frame capabilities then becomes a depth-first numbering [ASU86] of the dynamic call graph. When a non-local jump occurs, all elided frame capabilities between the source frame and destination frame are deleted by removing all frame capabilities in the capability store that are *larger* than the frame capability of the destination frame. This mechanism only works if the source and destination frames are on the same call stack – this stipulation may not be true in all cases, *e.g.*, coroutine jumps.

```
struct {                    p                              q                    capability store
    char a;
    char b[100];                                                                     { }
} x, *p;             [x,x,x,x,NEVER]                                                 { }
char *q;             [1000,1000,101,Global,FOREVER] [x,x,x,x,NEVER]                   "
p = &x;                     "                      [x,x,x,x,NEVER]                   "
*p; /* no error */          "                              "                         "
q = &p->b[10];              "                      [1011,1001,100,Global,FOREVER]    "
q--;                        "                      [1010,1001,100,Global,FOREVER]    "
*q;                  [798,1000,101,Global,FOREVER]         "                         "
p -= 2;                     "                              "
*p; /* error!!! */                                         "
```

a)

```
                            p                              q                    capability store
                                                                                     { }
                     [x,x,x,x,NEVER]                                                { 1 }
char *p, *q;         [2000,2000,10,Heap,1]                                            "
p = malloc(10);             "                      [x,x,x,x,NEVER]                   "
q = p+6;                    "                      [x,x,x,x,NEVER]                  { }
*q; /* no error */          "                      [2006,2000,10,Heap,1]            { 2 }
free(p);             [2000,2000,10,Heap,2]                 "                         "
p = malloc(10);             "                              "
*q; /* error!!! */                                         "
                                                           "
```

b)

Figure 7: Memory access checking examples. Figure *a)* is an example of a spatial access error, Figure *b)* is an example of a temporal access error. Safe pointer values are specified as a 5-tuple with the following format: [*value*,*base*,*size*,*storageClass*,*capability*]. An occurrence of *x* indicates a don't care value.

The capability store is an associative memory containing the capabilities of all active memory. It can be implemented as a hash table with the capability as the hash key. Accesses to the capability store exhibit a great deal of temporal locality, so moving accessed elements to the head of the hash table bucket chains is likely to decrease average access time.

We close this section with two examples. Figure 7(a) shows a spatial access error, and Figure 7(b) demonstrates a temporal access error. Safe pointer values are specified as a 5-tuple with the following format: [*value*,*base*,*size*,*storageClass*,*capability*]. *x* indicates a don't care value. In the first example, a spatial access error is flagged when the program dereferences a safe pointer whose value is less than the base of the referent. In the second example, a stale pointer, `q`, is dereferenced. Even though the same storage has been reallocated to `p`, the capability originally assigned to `q` has been destroyed during the call to `free()`; thus, the temporal access error is detected.

4 Implications of Complete Error Coverage

Our safe programming technique can detect all memory access errors provided that the following conditions hold:

- Storage management must be apparent to the translator.
- The referents of all pointer constants must have a known location, size, and lifetime.
- The program must not manipulate the object attributes of any pointer value.

Our claim to complete error coverage must be limited to storage management controlled by the safe programming run-time system. If a program implements a domain specific allocator at the user level, some memory access errors, as viewed by the programmer, can be missed.

Consider, for example, a fixed size storage allocator. If a program relies heavily on a fixed size structure, storage requirements and allocation overheads can be greatly reduced by applying a fixed size allocation strategy. At the program level, the fixed size allocator calls the system allocator, *e.g.*, `malloc()` or `sbrk()`, to allocate a large memory allocation. The fixed size allocator then slices the system allocation into fixed size pieces with a zero overhead for each allocation. Under this scheme, our safe programming technique would ensure that no accesses to a fixed size allocations are outside of the space and time bounds of the block from which the fixed size allocation was derived. This imprecision occurs because the translator can not disambiguate the user level storage allocation actions from other pointer related program activities.

With some programmer intervention this problem can be solved. Any useful safe compiler implementation will have to include an application programmers interface, or API, through which systems programmers can construct and manipulate the object attributes of safe pointers. In the case of the fixed size storage allocator, the programmer would specify the base and size of the fixed size allocation. The storage class and capability would be generated from the safe pointer to the block from which the fixed size allocation was derived.

Without the second qualification, the compiler may not be able to generate correct object attributes for a pointer constant. For example, device driver code typically creates pointers to device buffers and registers by recasting an integer to a pointer value. The translator has no way of knowing the size and lifetime of the referent; thus, program safety cannot be maintained. In C, the only well defined pointer constants are NULL, strings, and functions.[6] For all other cases, this problem can be avoided by supplying the programmer with an API suitable for specifying the size and lifetime of problematic pointer constants.

The second qualification does not, however, preclude the use of recasts from non-pointer *variables* to pointer variables. To successfully support these operations, object attributes must be attached to all variables. In general, to provide complete safety, we need to attach object attributes to any storage that could hold a pointer value. It is our contention that most "well behaved" programs will only require pointer variables to carry object attributes.

The final qualification protects object attributes. If a program can arbitrarily manipulate the object attributes of a pointer value, then safety can always be subverted. For example, changing the storage class of a pointer from `Global` to `Heap` and then freeing the pointer would likely cause disastrous effects under our storage allocation scheme, and these effects would not be detected by our safe programming framework.

If object attributes are only attached to pointer values, the danger exists of manipulation through the use of recasts or unions. With a recast, it is possible to type storage in the referent first as a non-pointer value, manipulate the storage arbitrarily, and then recast the referent storage to a (possibly unsafe) pointer. Using a union, it is possible to create a pointer value under one field and then manipulate the object attributes of the pointer value through another overlaid, non-pointer field of the union.

[6] There are two ways we can integrate functions into our safe programming framework. If we assume that function pointers can only be assigned and dereferenced, they will not intermix with safe pointers and may remain simple pointers. The only check required at dereferences is a non-NULL check. If cast operators may be applied to function pointers, they can be represented as safe pointers with a storage class of `Function`. At dereferences of function pointers, we need to ensure the pointer value is a function pointer and has not been changed (*e.g.*, `value == base`); at non-function pointer dereferences, we must check that the pointer value is not a function pointer.

The only solution that we can conceive to prevent this kind of manipulation is to attach object attributes to each byte of allocated storage. For types larger than one byte, the object attributes would be copied to all other storage holding the allocation. In this way, any arbitrary overlaying of types would still not allow the object attributes to be manipulated at the program level.

In reality, we can provide a high margin of safety for "well behaved" programs by attaching object attributes only to pointer values. We consider a well behaved program to be one in which pointer values are never created from or manipulated as non-pointer values. If a program violates this rule intentionally (*e.g.*, through a recast), the safe compiler can make a conservative approximation as to the intended referent of the new pointer value and allow the pointer to access any live storage.[7] If the rule is broken unintentionally (*e.g.*, through incorrect use of a union), the error will likely be caught because it is difficult to manufacture, accidentally, an unsafe pointer.

5 Optimizing Dereference Checks

In the interest of performance, it may be possible to elide dereference checks and still provide complete program safety. If we can determine that the following invariant holds, the check may be elided.

> *A check at a dereference of pointer value v may be elided at program point p if the previous, equivalent check executed on v has not been invalidated by some program action.*

We can implement this check optimization either at run-time or at compile-time. Run-time check optimization has the advantage of being more flexible. We only need to execute the checks absolutely required to maintain program safety. However, the cost for this precision is extra safe pointer state which must copied, maintained, and checked at each dereference. Compile-time check optimization, on the other hand, is less flexible because we must constrain the decision to elide a check to all previous possible executions leading to a program point. The advantage of compile-time check optimization is that no additional overhead is required at run-time to determine if a check may be elided.

5.1 Run-Time Check Optimization

We have designed and implemented a framework for dynamically eliding spatial and temporal checks. Spatial checks have no side effects, thus we can employ memoization [FH88] (or function caching) to elide their evaluation. We store the operands to the last check in the safe pointer object attributes, which amounts to the effective address of the last dereference. At any dereference, the spatial check may be elided if the effective address since the last check has not changed. This test is shown in Figure 8 in the `if` statement surrounding the bounds check. It may be useful to memoize more than one set of operands. In our implementation, we memoize both the effective address of the last dereference, *i.e.*, use of '*', and the effective address of the last subscript operation, *i.e.*, use of '[ ]'. Changes in the former can be tracked with only a single "dirty" bit. Changes in the latter are tracked by retaining a copy of the last index applied to the pointer value.

To elide temporal checks, we keep a copy of a global counter, incremented when storage is deallocated, in the safe pointer. If this counter, which we call the *free counter*, has not changed since the last temporal check, the referent has not been freed and the temporal check can be safely elided. In our implementation,

[7] Note that in this case, we *cannot* try to bind the new pointer value to a live variable because the program may have manipulated the pointer value to point before of after the intended referent prior to recasting it to a non-pointer value.

```
void ValidateAccess(<type> *addr) {
    if (freeCount != currentFreeCount) {
        if (storageClass != Global && !ValidCapability(capability))
            FlagTemporalError();
        freeCount = currentFreeCount;
    }
    if (lastDerefAddr != addr) {
        if ((unsigned)addr - (unsigned)base > size-sizeof(<type>))
            FlagSpatialError();
        lastDerefAddr = addr;
    }
    /* valid access!  */
}
```

Figure 8: Memory access check with run-time check optimization. This C-like function is parameterized by `<type>`, the type of the safe pointer referent. `FlagTemporalError()` performs system specific handling of a temporal access error, *e.g.*, force a core dump. `FlagSpatialError()` performs the same function, but for a spatial access error. The function `ValidCapability()` indicates whether or not the passed capability is currently active, *i.e.*, in the capability store. The variable `currentFreeCount` is a global counter incremented each time storage is deallocated.

we do not increment the free counter when a procedure returns. Rather, we always perform temporal checks on pointers to locals. This strategy works very well in practice because procedure returns are quite frequent, while the use of local referents is generally infrequent.

5.2 Compile-Time Check Optimization

We have also designed (and are currently implementing) a compile-time optimization framework like those proposed by Asuru [Asu92] and Gupta [Gup90]. Our algorithm implements a forward data-flow framework similar to that used by common subexpression elimination [ASU86]. However, our algorithm extends previous work to include eliding of temporal error checks, and because of our simplified bounds check, there is no need to split the optimization into upper and lower bounds check elimination.

Our optimization algorithm is shown in Figure 9. The algorithm is run twice, once for optimization of spatial checks and again for temporal checks. The algorithm executes in three phases.

In the first phase, the algorithm seeds the data-flow analysis by approximating all *out* sets. For all blocks except the entry block, the value of $out[B_i]$ is set to all check expressions less those killed by the block B_i, *i.e.*, $U - kill[B_i]$. For the program entry block, B_1, we must assume that no checks are available, hence, $in[B_1]$ is set to empty and $out[B_1]$ is set to the checks generated in the entry block B_1.

In the second phase, the data-flow framework is solved to determine where check expressions reach in the program. For a check expression to reach a node B_i, it must be available at B_i for all executions, that is, it must be available in the *out* sets of all predecessors to block B_i. This requirement is precisely why the confluence operator is intersection. After the data-flow computation converges on a solution, *i.e.*, `change == false`, the set $in[B_i]$ contains all checks that reach block B_i.

In the third phase, the *in* sets are used to elide redundant checks. Checks may be elided wherever a lexically identical (or equivalent, if value numbering [RWZ88] or equality tests [AWZ88] are applied) check is available in the block (*i.e.*, the same check is in the *in* set of the block).

The defining feature for each analysis (spatial and temporal) is the specification of what constitutes a kill. A spatial check is killed by any assignment to a check operand, which includes assignment to the

Input: A flow graph G with blocks B with $gen[B_i]$ and $kill[B_i]$ computed for each block $B_i \in B$. $gen[B_i]$ is the set of check expressions generated in B_i. $kill[B_i]$ is the set of check expressions killed in B_i. The entry block is B_1.

Output: A flow graph G with redundant checks deleted.

Method: The following procedure is executed twice, once for spatial check optimization and again for temporal check optimization.

```
/* initialize $out$ sets */
$in[B_1] = \emptyset$;
$out[B_1] = gen[B_1]$;
$U = \bigcup_{B_i \forall B} gen[B_i]$;
for $B_i \in B - B_1$ do
    $out[B_i] = U - kill[B_i]$;

/* compute availability of checks, $in$ sets */
change = true;
while change do begin
    change = false;
    for $B \in B - B_1$ do begin
        $in[B_i] = \bigcap_{P \in Pred[B]} out[P]$;
        $oldout = out[B_i]$;
        $out[B_i] = gen[B_i] \cup (in[B_i] - kill[B_i])$;
        if $out[B_i] \neq oldout$ then
            change = true;
    end
end

/* elide redundant checks */
for $B_i \in B - B_1$ do begin
    for $c \in gen[B_i]$ do begin
        if $c \in in[B_i]$ then
            elide check $c$;
    end
end
```

Figure 9: Compile-time check optimization algorithm.

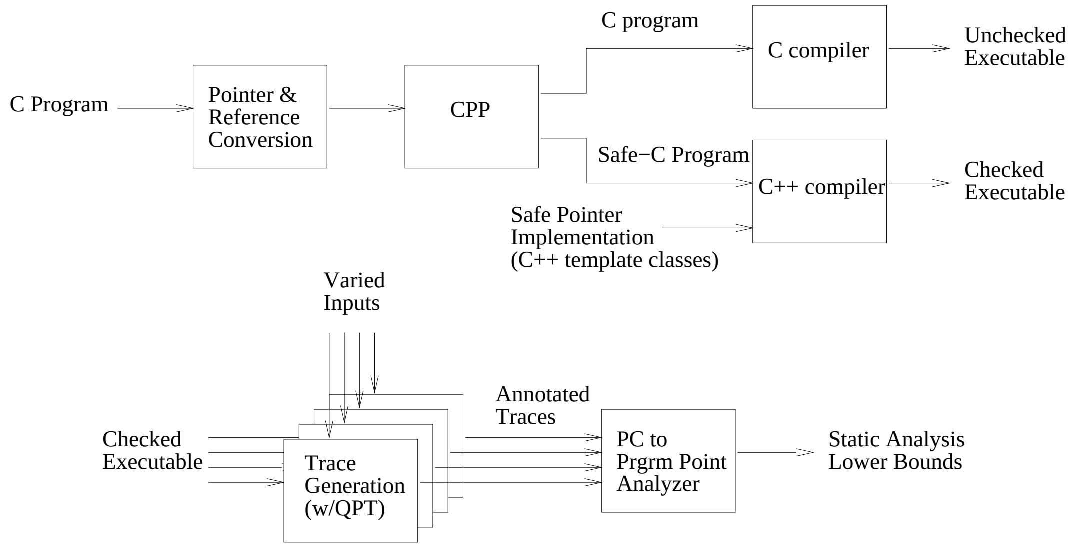


Figure 10: Experimental framework.

pointer variable or any of the operands of the index expression (if the pointer was indexed in the check expression). A temporal check is killed by any free of the referent storage. If the referent of a free can be determined to be different than the check referent (*e.g.*, through alias analysis), the free need not kill the check.

While performing these analyses, we must also be wary of kills that may occur through function calls or aliases. In either case, we must make a conservative approximation if insufficient information is available and assume that a kill does occur.

6 Experimental Evaluation

We evaluated our safe programming methodology by implementing a semi-automatic source-to-source translator and examining the run-time, code and data size overheads for six non-trivial programs. For each program, we analyzed its performance without optimization and with run-time resolved optimizations. We also generated lower bound statistics for the efficacy of compile-time optimization through the use of a trace analyzer.

6.1 Experimental Framework

Figure 10 shows our experimental framework. We translate C programs to their safe counterparts by first rewriting all pointer and array declarations, calls to `malloc()` and `free()`, and references (use of the '&' operator) to use our *Safe-C* macros. These macros, when passed through the C preprocessor (*CPP*), produce either the original C program or a Safe-C program. A Safe-C program has all pointer and array declarations changed to type parameterized C++ class declarations. Using operator overloading in the C++ class definition, we implement the extended safe pointer and array semantics as described in Section 3. Figure 11 shows a portion of the unoptimized safe pointer implementation.

All explicit storage allocation, *i.e.*, calls to `malloc()` and `free()`, call wrapper functions which create safe pointers from the standard library routines. Our `malloc()` implementation clears all allocated storage, so any contained pointers start in the invalid state. If a local in a function is used as a pointer

```
template <class Type>
class sp {
    /* safe pointer representation */
    Type *value;                /* native pointer */
    Type *base;                 /* base address of object */
    unsigned long size;         /* size of object in bytes */
    char storageClass;          /* type of allocation */
    unsigned short capability;  /* capability is always unique */

    /* constructor */
    sp(void) {
        value = NULL;
        base = NULL; size = 0;
        storageClass = None;
        capability = NEVER;
    }

    /* dereference */
    Type& operator*(void) {
        if (storageClass != Global && !ValidCapability(capability))
            FlagTemporalError();
        if ((unsigned)value - (unsigned)base > size-sizeof(Type))
            FlagSpatialError();
        return *value;
    }

    /* pointer addition */
    sp<Type> operator+(int addend) {
        sp<Type> p = *this;     /* no side-effect on *this */
        p.value = p.value + addend;
        return p;
    }

    /* pointer boolean equality test */
    int operator==(sp<Type>& p) {
        return value == p.value;
    }
}
```

Figure 11: A portion of the (unoptimized) safe pointer implementation.

Program	Instructions		Insts per Dereference		Class	Description
	Static	Dynamic	Static	Dynamic		
Anagram	10,096	19,457,876	106.3	7.6	S+,T-	anagram generator
Backprop	10,840	122,490,093	148.5	8.9	S+,T-	neural net trainer
GNU BC	19,508	12,290,996	15.5	7.6	S,T+	arbitrary precision calculator
Min-Span	11,976	13,337,918	48.7	5.9	S-,T+	min spanning tree computation
Partition	13,540	21,116,811	62.4	3.7	S,T-	graph partitioning tool
YACR-2	18,584	546,214,295	37.1	14.0	S+,T-	channel router

Table 2: Analyzed programs.

referent, we also rewrite the function to allocate a capability for the frame. Any pointer in the stack frame of a function is initialized to an invalid state in the constructor of the C++ safe pointer class. Application of the reference operator calls a function which creates a safe pointer from the decomposed access path.

We compute a lower bound on the number of checks required for compile-time optimization by modifying the safe pointer implementation to make superfluous stores to a global scratch pad array during dereferences. The location of the superfluous stores indicate whether or not a particular program point required a check at run-time. We employ our run-time resolved optimization scheme to determine if a check is required at run-time. The superfluous stores are tracked by an address trace analyzer which tabulates, by address, how many checks were executed. When the program terminates, we compute the total number of program points that did not execute *any* checks and the total dynamic checks elided at these program points.

These results form a lower bound on the number of static (in the code) and dynamic (executed at run-time) checks required for a compile-time optimized program. With true compile-time analysis, the actual number of checks required may be higher because 1) other inputs may require checks at program points that did not execute any checks, and 2) limitations in static analysis, *e.g.*, imprecisions due to program aliases, may force a compile-time optimizer to make conservative assumptions and add checks where they may not be needed. To increase the effectiveness of our lower bound study, we combined the results of four separate inputs.

Our lower bound results are not a strict lower bound. Other static analysis techniques, *e.g.*, range analysis [Har77] or program restructuring, could decrease the number of static checks required. However, for our proposed compile-time optimization framework without program restructuring, the lower bound results are a strict lower bound.

6.2 Analyzed Programs

We analyzed six programs, selected because each exhibits a high frequency of indirect references. Table 2 details the programs that we analyzed. For each, we show the code size (*Instructions/Static*), the number of instruction executed without checking (*Instructions/Dynamic*), the frequency of dereferences in the program text (*Insts per Dereference/Static*), and the dynamic frequency of dereferences executed (*Insts per Dereference/Dynamic*).

The *Class* column classifies each program according to its spatial and temporal complexity. The

Program		Optimization						
		None		Run-time Opt		Compile-time Opt (lower bound)		
		Static	Dynamic	Static	Dynamic (% Unopt)	Coverage	Static (% Unopt)	Dynamic (% Unopt)
Anagram	Spatial	95	2,554,106	95	1,000,414 (39%)	83%	46 (48%)	2,019,514 (79%)
	Temporal	95	2,554,106	95	0 (0%)	83%	0 (0%)	0 (0%)
Backprop	Spatial	73	13,730,020	73	9,157,590 (67%)	70%	40 (55%)	11,766,310 (83%)
	Temporal	73	13,730,020	73	0 (0%)	70%	0 (0%)	0 (0%)
GNU BC	Spatial	1,259	1,623,490	1,259	329,410 (20%)	34%	77 (6%)	363,568 (22%)
	Temporal	1,259	1,623,490	1,259	144,102 (9%)	34%	58 (5%)	499,364 (31%)
Min-Span	Spatial	246	2,268,757	246	547,369 (24%)	60%	7 (3%)	548,483 (24%)
	Temporal	246	2,268,757	246	72,494 (3%)	60%	22 (9%)	72,494 (3%)
Partition	Spatial	217	5,662,107	217	958,802 (17%)	76%	45 (21%)	1,034,748 (18%)
	Temporal	217	5,662,107	217	0 (0%)	76%	0 (0%)	0 (0%)
YACR-2	Spatial	501	38,995,428	501	33,476,773 (86%)	76%	213 (43%)	34,125,516 (88%)
	Temporal	501	38,995,428	501	0 (0%)	76%	0 (0%)	0 (0%)

Table 3: Checks required by optimization type.

spatial complexity, S, indicates the frequency of pointer arithmetic or indexing: either high (+), medium, or low (-). The temporal complexity, T, is an indicator of how often the program frees storage. If this factor is high (+), the program frees storage throughout execution, if low (-), the program never frees storage (or only at program completion).

All programs were compiled and executed on a DECstation 3100 using AT&T USL *cfront* version 3.0.1. The output of *cfront* (C code) was compiled using MIPS *cc* version 2.1 at optimization level '-O2'. All instruction counts were obtained with *QPT* [Lar93].

For all analyses, object attributes were only attached to pointer values. We used a 15 byte safe pointer (275% overhead) in the unoptimized case: 4 byte pointer value, 4 byte base, 4 byte size, a 1 byte storage class specifier, and a 2 byte capability. For run-time resolved optimization, we added a 1 byte dirty flag, a 4 byte last index, and a 2 byte free counter for a total size of 22 bytes (450% overhead). Due to a bug in the C++ compiler, we could not use `sizeof()` in the safe pointer implementation if the referent referred to itself; as a result, *BC*, *Min-Span*, and *Partition* all required the size of the referent to be stored in the safe pointer, which added a 4 byte overhead for these programs. There were no space overheads for array variables, as all required object attributes are known at compile-time. We only rewrote the actual program code, all system library routines remained unchecked. We did, however, perform *interface checking*. Whenever a system library is called, any pointer arguments are validated against the time and space bounds expected by the library routine. For example, if a call were made to `fread()`, the interface check would ensure that the destination of the read was live storage and that the entire length of the read operation would fit into the referent.

6.3 Results

Table 3 shows the number of program checks required by optimization type. *None* shows the total checks required with no optimization, both in the program text (*Static*) and at run-time (*Dynamic*). *Run-time Opt* shows the number of checks required under our run-time optimization scheme. And *Compile-time Opt* shows a lower bound on the number of checks required under our compile-time optimization

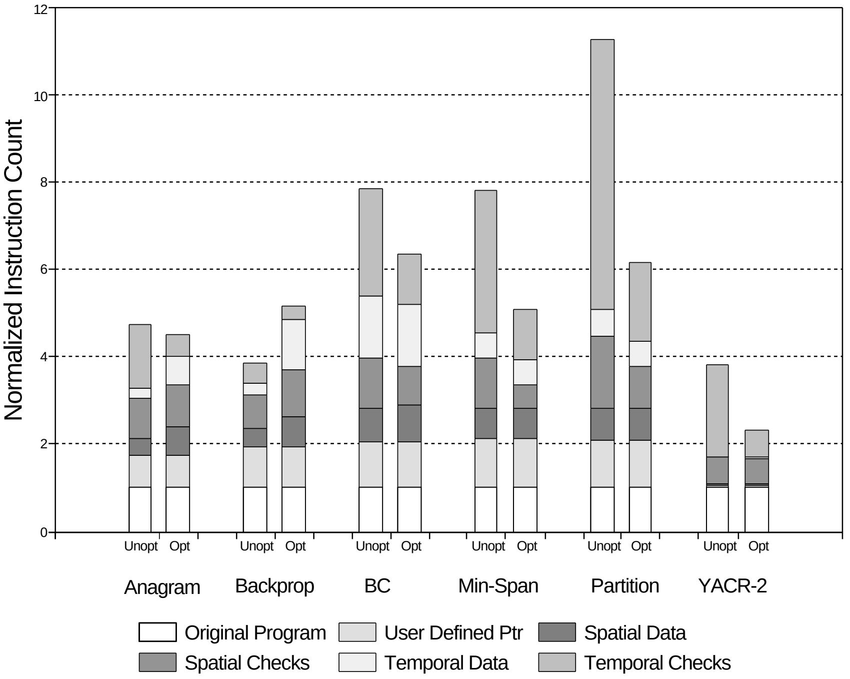


Figure 12: Execution overheads.

scheme. *Coverage* indicates the fraction of dereferences in the program source that were executed at least once.

Figure 12 shows the execution overheads for the analyzed programs. The *Unopt* columns show total dynamic instruction counts for executions with no optimization, and the *Opt* columns show instruction counts with run-time check optimization.

For the run-time optimized executions, the normalized instruction counts range from 2.3 (*YACR-2*) to 6.4 (*BC*). This overhead reflects program performance without any compile-time optimization. While this performance degradation will likely be acceptable for the development cycle of short or medium length program executions, it may still be prohibitively expensive for very long running programs, and it is certainly too costly a price to pay for in-field instrumentation of a program. Examining more closely the breakdown of the execution overheads yields much insight into how the performance of our checking methodology could be improved.

For each program, we break down the overhead costs into five categories. *Original Program* is the instruction count for the unchecked program, always normalized to one.

User Defined Ptr is the cost in our framework for implementing all pointers as structures at the user level. We measure this cost by running the program using safe pointers, but without any object attributes or checks. The primary factors affecting performance here are increased loads, stores, and function calls. The first factor is due to the MIPS *cc* compiler's handling of structure variables; once wrapped in a structure, the field variables are no longer eligible for register allocation. When data within a safe pointer is manipulated it is first loaded from the structure, used and then possibly stored back if changed. If it is used again, it is again loaded from the structure. Wrapping attributes around pointers also increases the costs of procedure call parameter passing. MIPS *cc* compiler passes most scalar arguments through registers; however, composite structures are always passed through memory (on the stack). The second major factor affecting performance is an increased number of function calls. The AT&T C++ compiler simplifies complex expressions created during template instantiation by extracting portions of the expression into static functions. This cost is only a side-effect of our implementation.

Spatial Data is the cost of maintaining and copying spatial object attributes. For the optimized executions, this overhead includes the cost of maintaining the pointer dirty bits and previous index values. *Spatial Checks* is the cost of performing spatial checks. *Temporal Data* is the cost of maintaining and copying temporal object attributes. For the optimized executions, this overhead includes the cost of maintaining the additional counter variable. *Temporal Checks* is the cost of performing temporal checks.

For *BC*, *Min-Span*, and *Partition*, run-time check optimization paid off with a slightly lower execution cost for spatial checking. For *Anagram*, *Backprop*, and *YACR-2*, adding run-time checks resulted in a higher cost for spatial access checking; and in the case of *Backprop*, a higher overall execution overhead.

These programs demonstrate the trade-offs involved in providing run-time check optimization. Run-time optimization adds the extra overhead of copying, maintaining, and checking the extra safe pointer state. If this added overhead, plus the overhead of the required checks, is greater than doing all the checks, there is no advantage to run-time check optimization. With faster checks, compile-time optimization, and spatially complex programs, this trade-off becomes even more acute.

Since *Anagram*, *Backprop*, and *YACR-2* must execute many of their checks, they do not benefit from the run-time optimizations. For *YACR-2*, the effects are much less pronounced because dereferences are much less frequent (as shown in Table 2). We predict that compile-time analysis will be ineffective for most pointer and array intensive programs, as they are either spatially complex or rely heavily on dynamic storage, two properties which reduce the effectiveness of compile-time spatial check optimization.

The second effect to observe when comparing the optimized to unoptimized execution costs is that the greatest benefit of run-time check optimization always comes from eliding temporal checks. In fact, adding run-time optimization for temporal checks caused a significant decrease in all execution overheads except *Backprop*. There are two aspects to this result. First, temporal checks are very expensive (requiring an associative search), so eliding one has a great performance advantage. Second, our run-time check optimization of temporal checks is very effective. Temporal checks are rarely required, even for *BC* and *Min-Span*, both of which free storage often. In the case of *Backprop*, adding run-time optimization for temporal checks resulted in an increased execution overhead. *Backprop* has only one dynamic object, an array, so temporal checking is relatively cheap without any optimization (the capability is always at the head of the hash bucket chain). In this case, the cost of maintaining the extra storage required for the free counter outweighs the cost of executing all temporal checks.

Our lower bound analyses (shown in Table 3) also suggest that few program points require temporal checks; after inspecting the code, it is apparent that compile-time analysis (even when not inter-procedural) for eliding temporal checks will be very effective on these programs. Few of the dominating loops and procedures contain procedure calls or calls to `free()`.

Adding checking code reduces the effectiveness of many traditional compiler optimizations. We inline all check code except for calls to `ValidCapability()` and `abort()`. These functions are both externally defined, so the compiler must make conservative assumptions as to what actions they take. This conservative approximation has the effect of limiting the effectiveness of many optimizations such as invariant code motion, register allocation, copy propagation, and common subexpression elimination. Neither of these functions produce any side-effects for normal executions. Hence, better compiler integration, *i.e.*, providing a special channel of communication between the safe program generator and the compiler optimizer, would certainly increase the performance of our safe executions.[8]

Text size overheads are shown in Figure 13. All checking code, except the capability routines and what the C++ compiler extracts for expression simplification, is inlined into the original program text. Surprisingly, the text overheads are quite small; 35% to 300% for the unoptimized executables and 41% to 340% for the run-time optimized programs. The text sizes for the run-time optimized programs are larger due to additional code required for maintaining, copying, and checking the extra object attributes. As shown by comparing Table 2 and Figure 13, there is a strong correlation between static dereference density and the resulting text overhead.

The data size overheads, shown in Figure 14, are measured as the total size of initialized (`.data`) and uninitialized (`.bss`) data segments plus the size of the heap segment when the program terminates execution. The data size overheads on the stack were not measured. All programs, except *Min-Span*, have data size overhead below 100%. *Backprop* has the lowest overhead (less than 5%) because most of its storage is large global arrays which do not require any object attributes. *Min-Span* has the highest overhead (330%), which stems from the high density of pointers in its heap allocations, most of which contain eight pointers and three integers. Some of the run-time optimized programs have slightly larger overheads due to the additional object attributes.

To summarize the main points of our results:

- Execution overheads, even without compile-time optimization, are low enough to make our methodology useful during program development. However, the overheads are not likely low enough that programmers would release software with checking enabled. We are currently exploring the use of compile-time optimization and better compiler integration as means of increasing the performance of our approach.

- The largest contributing factors to execution overhead are 1) safe pointer structures are not register allocated, and 2) many traditional optimizations fail with the addition of checks. Other performance losses are attributed to the C++ compiler simplifying expressions through the use of static functions, and, due to a bug in the C++ compiler, the need to include the type size of the referent in the object attributes. None of these difficulties are without recourse, however. Better integration between the safe compiler and the optimizer could fix most problems.

- Dynamically eliding spatial checks is generally ineffective, primarily because maintaining the extra state, and checking it, quickly outweighs the cost of executing all checks. Our spatial check is very cheap to execute, and spatially complex programs tend to execute most of the checks anyway.

- Temporal checks, on the other hand, are very expensive to perform and are rarely required, so run-time optimization shows to be beneficial in most cases.

- The text and data size overhead are generally quite low. The text overheads for all programs with run-time optimization, range from 41% to 340%, with all but two below 100%. Data overheads range from 5% to 330%, with all but one below 100%. Run-time optimized executions have slightly larger text and data sizes.

[8] Many compilers, *e.g.* GNU *gcc*, already understand the special semantics of `abort()` and use this inter-procedural information to improve optimizations. We should be able to achieve the same results for `ValidCapability()`.

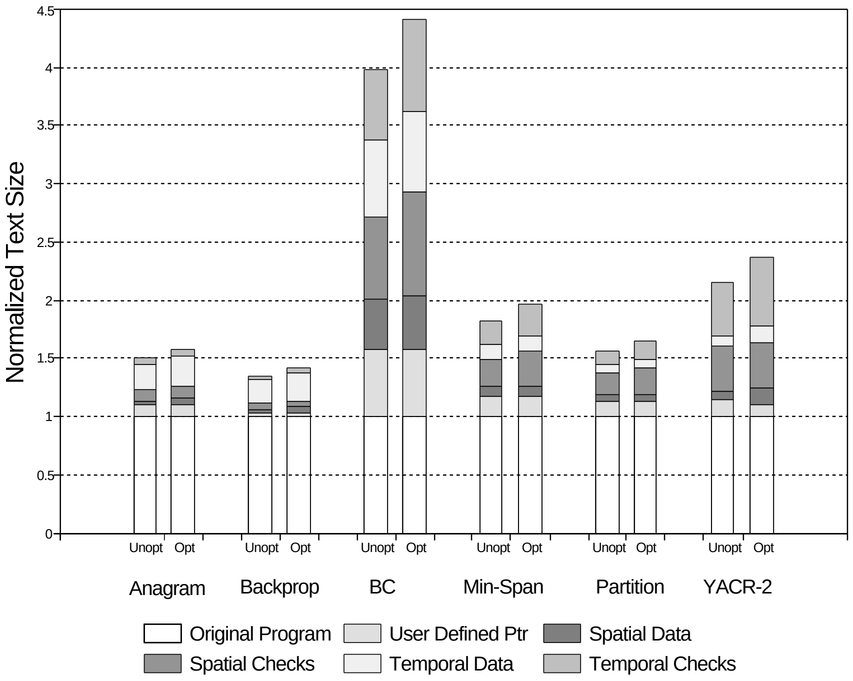


Figure 13: Text overheads.

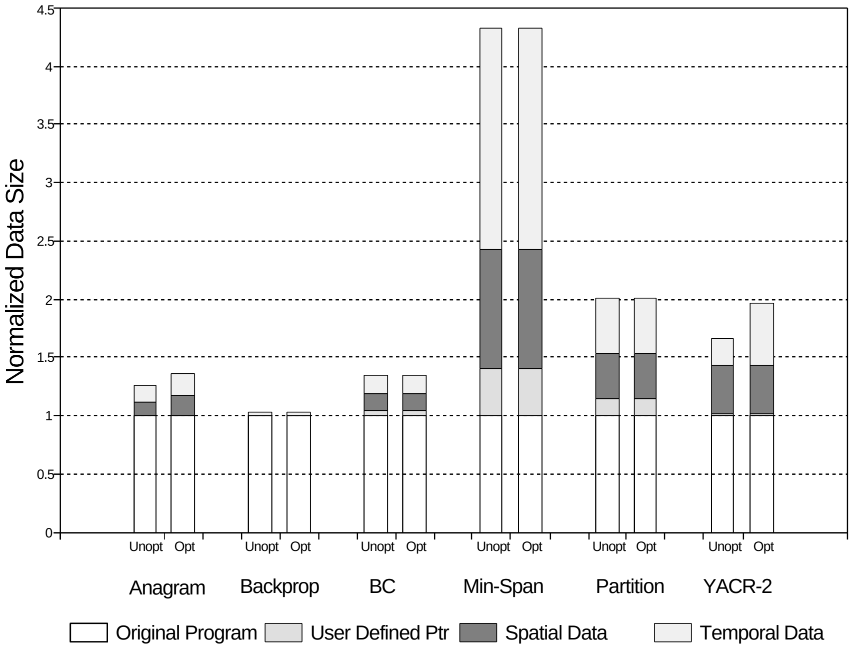


Figure 14: Data overheads.

7 Related Work

Our first attempt at creating a safe programming environment for C employed reference chaining. The technique is similar to that used by many "smart pointer" implementations [EP91, Gin92]. The idea is to insert any pointer which is generated either through use of explicit storage allocation, *e.g.*, `malloc()`, the reference operator ('`&`'), or assignment into a reference chain rooted at the referent. When a pointer value is destroyed, *e.g.*, through assignment, storage deallocation, or at procedure returns, the pointer is removed from the reference chain. This technique has a number of useful properties. First, it is possible to ensure temporal safety by destroying all pointer values when a referent is freed – just march down the reference chain assigning NULL to all pointers. Second, if a destructed pointer value is the last value in the referent's reference chain, a storage leak has occurred and it is detected immediately. Unfortunately, this technique cannot be made to work reliably in C. It is relatively easy for the programmer to subvert the checking mechanism through recasting and type-less calls to `free()`. Storage leak detection also fails in the presence of circular references. The safe programming technique described in this paper is significantly more reliable because its correctness does not rely on tracking pointer values.

Some researchers have recently proposed providing complete program safety through limiting the constructs allowed in the language. The main thrust of this work is the design of languages that support garbage collection reliably and portably. For example, in [ED93], a safe subset of C++ is defined. The safe subset does not permit any invalid pointers to be created; this restriction, for example, precludes the use of any explicit pointer arithmetic. If requested, the compiler can enforce safety in a module by ensuring that the programmer does not use any intrinsically unsafe operations. The safe subset also requires some checking, but much less than our checking technique requires. Languages which can easily be made totally safe have existed for a long time; for example, many FORTRAN implementations provide complete safety through range checking [Cor87]. However, these languages tend to be less expressive than intrinsically unsafe languages such as C or C++. We felt that it was important not to restrict the expressiveness available to the programmer. Our checking technique is not limited by the language upon which it is applied, it can be applied successfully to compiled or interpreted languages with subscripted and mutable pointers, local references, and explicit and type-less dynamic storage management.

Table 4 details our work (*Safe-C*) and five other published systems that support memory access checking.

Hastings' and Joyce's *Purify* [HJ92] is a commercially available memory access checking tool. It is particularly easy to use because it does not require program source – all semantic changes to the program are applied to the object code. *Purify* supports both spatial and temporal access error checking to heap storage *only* through the use of a memory state map which is consulted at each load and store the program executes. *Purify* also provides uninitialized read detection, and storage leak detection through a conservative collector [Boe93, BW88]. Spatial access errors are detected by bracketing both ends of any heap allocation with a "red zone". These zones are marked in the memory state map as inaccessible. When a load or store touches a red zone a memory access error is flagged. Temporal access errors are detected by setting the memory state of freed storage to inaccessible. *Purify* cannot detect all memory access errors. For example, accessing past the end of an array into the region of the next variable, or accessing freed storage that has been reallocated cannot be detected. These limitations occur because *Purify* does not determine the intended referent of memory accesses – it can only verify that the accessed storage is active. Our checking technique, on the other hand, can detect all memory access errors because it tracks not only the state of storage, but also the intended referents of all pointer values. To increase the

Name	Environment	Method	Error Model		
			Spatial Checks?	Temporal Checks?	Extensions
Safe-C	C/C++	source-to-source translation	yes*	yes*	errant free's
Purify [HJ92]	object files	object code translation	yes limited to heap	yes limited to heap	errant free's, uninitialized reads, storage leaks
RTCC [Ste92]	C	safe compiler	yes*	no	
CodeCenter [KLP88]	C/C++	interpreter	yes*	yes	errant free's, uninitialized reads, type checking, arithmetic errors, etc.
Integral C [Ros87]	C	safe compiler	yes*	no	
UW-Pascal [FL80]	Pascal	safe compiler	yes*	yes	errant free's, union type checking, arithmetic faults, etc.

Table 4: Comparison of memory access checking work. Entries with an asterisk (*) indicate that the method detects all errors for that particular error class.

effectiveness of temporal error checking, *Purify ages* the heap, holding freed storage in the heap free list longer than needed. This aging increases the storage requirements of programs that use the heap. The primary disadvantage of our technique over that used by *Purify* is that we require source code before any checking can be implemented; thus, source code is required if libraries are to be checked. Our technique is also not portable across languages, that is, a given implementation must be tailored for a specific language. However, our technique is quite portable across different platforms, especially if implemented as a source-to-source translator. Although *Purify* is portable across languages (on a given platform), it is not portable across platforms.

Steffen's *RTCC* [Ste92] extends the functionality of the C language compiler PCC to include spatial error checking. *RTCC* attaches object attributes to pointers in a fashion similar to our technique; it does not, however, detect temporal access errors, nor does it explore the use of check optimization. Our checking technique finds both spatial and temporal access errors, and incorporates run-time and compile-time optimizations through which access checks can be elided. In the implementation of *RTCC* the issue of interfacing to library and system calls is addressed through *encapsulation*; Steffen also augmented *sdb* to provide users with transparent debugging support.

CodeCenter [KLP88] is an interpreted C language environment. The checking provided is very rich – it detects many memory access errors as well as provides dynamic type checking (*i.e.*, the type of the last store to memory must match the type of subsequent loads), uninitialized read detection, errant free detection, and other useful checks. The published information describing *CodeCenter* is somewhat ambiguous as to how it implements memory access checking. Object attributes (namely, type and size) are attached to all storage when it is initialized. If a reference is made to storage, it appears that the base and size attributes, associated with the referent storage, are also attached to the pointer value. Using this information, *CodeCenter* provides complete coverage for spatial access errors. However, it does not employ a capability based temporal checking scheme, so it is (sometimes) possible to access freed storage after it has been reallocated for another purpose. Temporal access checking can also fail for pointer references to local variables. Because our checking technique employs a capability based scheme, it never misses temporal access errors. The primary disadvantage of *CodeCenter* is its resource requirements. Since programs run in an interpreter, the execution overheads may discourage its use, and in the case of long running programs, may preclude its use. Due to our use of compile-time instrumentation, resource requirements are significantly lower. Compile-time instrumentation also allows us to employ static check optimizations.

Integral C [Ros87] is an integrated programming environment for the C language. The user interface is very similar to *CodeCenter*, however, internally it does not employ an interpreter. As the user updates code, it is incrementally compiled (at function granularity) to machine code. Like *RTCC*, *Integral C* attaches only base and bound attributes to pointer values, thus it can only detect spatial access errors.

Fischer and LeBlanc's *UW-Pascal* compiler [FL80] supports both temporal and spatial error checking. However, the lack of mutable pointers and dynamically sized arrays makes access checking much easier. While *UW-Pascal* detects all spatial access errors, temporal access errors may not be detected if storage is reallocated. Use of our checking technique is not limited by the expressiveness of the language; that is, it can be applied successfully to compiled or interpreted languages with subscripted and mutable pointers, local references, unions, and explicit and typeless dynamic storage management.

A closely related area of work, which can benefit from our safe programming technique, is storage leak detection. A *storage leak* is any storage to which the program can no longer generate a name. These leaks occur when the last accessible pointer to a heap object is overwritten. Without the ability to generate a name to the heap object, it cannot be freed, hence it has "leaked" out of the heap.

For languages like C and C++, leak detection is commonly implemented with a *conservative collector* [Boe93, BW88]. A conservative collector sweeps memory looking for unreferenced storage. Because it is difficult to know where all pointers are located, the collector makes the conservative assumption that all program accessible (non-heap) storage contains pointers. It then uses a traditional mark and sweep collection method. While effective, this method has some drawbacks. First, storage leak detection is not immediate, it is usually applied only when the programmer demands it or when the program completes execution. Thus, for it to be useful, some dynamic information, like a partial call chain, must be kept with allocations, in order for the programmer to deduce the circumstances under which the storage leak occurred. Second, the conservative pointer assumption can cause non-pointer values to be mistaken as pointer values which seem to reference heap storage. These false hits can hide a storage leak. The problem is aggravated by large storage allocations because it is more likely that non-pointer values inadvertently reference them; unfortunately, it is these large allocation leaks that we would most like to find. Third, if the program hides pointers, for example, by encoding type information in the upper bits of a pointer, or does not keep all pointers within the bounds of memory allocations, the collector may regard heap storage as a leak when it is still in use.

These false leaks cannot occur under our checking scheme because the `base` field always holds a pointer to the head of the allocation, and the program cannot manipulate this value. We can also address the problem of false hits, that is, non-pointer values which appear to reference heap storage, by applying safe pointer invariants to possible references. One trivial test is to ensure that both the capability and the free counter values of the possible reference are valid. If an incrementing counter is used for each, each value should be less than the current counter value. To summarize, using a conservative collector to detect storage leaks with our safe programming technique makes the process intrinsically more reliable by eliminating false leaks and reducing the possibility of false hits.

Zorn and Hilfinger's *mprof* takes a notably different (and clever) approach to detecting storage leaks [ZH88]. During the analyzed program's execution, *mprof* maintains a table of partial call chains, each with a count of how many `malloc()`'s and `free()`'s have occurred to storage whose call chains terminated with that sequence. Detecting storage leaks then involves adjusting the appropriate counts at calls to `malloc()` and `free()`. At a `malloc()`, the current call chain is used to increment the appropriate `malloc()` counter. At a `free()`, a hidden pointer in the header of the freed allocation is used to increment the corresponding `free()` counter. At program termination, detection of storage leaks involves reporting the partial call chains whose `malloc()` and `free()` counts differ. Unlike conservative collection, their technique does not suffer from false hits, that is, a true storage leak will always be detected. The primary disadvantage of their technique (over conservative collection) is that leak diagnostics may only be gathered after execution completes, and many programs do not deallocate storage needed until program termination (*e.g.*, in C, the call to `exit()` will ensure that all the program's resources are reclaimed). This behavior can yield many (arguably) false leaks. *Mprof* also provides a wealth of other information useful for optimizing a program's memory usage.

8 Conclusions

In this paper, we presented a pointer and array access checking technique that provides complete error coverage through a simple set of program transformations. Our technique, based on an extended safe pointer representation, has a number of novel aspects. It is the first technique that detects all spatial and temporal access errors. Its use is not limited by the expressiveness of the language; that is, it can be applied successfully to compiled or interpreted languages with subscripted and mutable pointers, local references, unions, and explicit and type-less dynamic storage management. We showed the transformations required in the context of the C language, and also developed run-time and compile-time check optimization frameworks. Finally, we described our prototype implementation, and used it to analyze the execution, text and data size overheads of six non-trivial, pointer intensive C programs. We showed that performance with only run-time resolved optimizations was quite good. For all six programs, instruction execution overheads ranged from 130% to 540%, with text and data size overheads typically below 100%. The primary factors to performance degradation in safe programs are the lack of safe pointer register allocation and ineffective optimization in the presence of check functions. We see the solution to these problems as better integration between the safe compiler and the code generator.

Our prototype implementation, while successful at showing the viability of our compile-time safe programming methods, leaves many unanswered questions. We close this paper with a few that we are addressing with the development of our fully automatic, optimizing *Safe-C* compiler:

- *Efficiency*: Can our safe programming method be made efficient enough for in-production software releases? Will compile-time optimization suffice? Can a hybrid solution be made to work? Will inter-procedural analysis be required? How can a safe compiler best support "loosely" typed operations, such as recasting a non-pointer value to a pointer value?

- *Usability*: What is the best way to integrate safe programming techniques into a development environment? Into the compiler? As a source-to-source translator? Should libraries be versioned, *i.e.*, a safe and unsafe version, or will interface checking suffice? What API can sufficiently support user developed storage managers and systems programming?

- *Extendibility*: How can the error model be extended to include detection of other important memory-related errors such as storage leaks, uninitialized reads, parallel programming races, as well as others?

Acknowledgements

This work stems from a class project two of the authors (Austin and Breach) did in an advanced compiler course taught by Jim Larus. We thank him for the many ideas he has injected into this work. Numerous discussions with Tom Ball, Alain Kägi, Alvy Lebeck, Harish Patil, and T.N. Vijaykumar also helped form this paper. We thank them as well. Finally, thanks to Mary Baker, Hans-Juergen Boehm, John Ellis, Mark Sullivan, Mark Weiser, and Ben Zorn for useful comments and directing us to relevant references. The first author wishes to thanks his wife and son for their endless support while the author was shifting his priorities to lesser things.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[Asu92] Jonathan M. Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, 1(2):109–118, June 1992.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, CA, January 1988.

[Boe93] Hans-Juergen Boehm. Space efficient conservative garbage collection. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, 28(6):197–204, June 1993.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18(9):807–820, September 1988.

[Cor87] MIPS Computer Corporation. MIPS Fortran 77 compiler (f77) user's guide, 1987.

[ED93] John R. Ellis and David L. Detlefs. Safe, efficient garbage collection for C++. Technical Report 102, DEC Systems Research Center, June 1993.

[EP91] D. R. Edelson and I. Pohl. Smart pointers: They're smart but they're not pointers. *Proceedings of the 1991 Usenix C++ Conference*, April 1991.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley Publishing Company, 1988.

[FL80] Charles N. Fischer and Richard J. LeBlanc. The implementation of run-time diagnostics in Pascal. *IEEE Transactions on Software Engineering*, SE-6(4):313–319, 1980.

[Gin92] Andrew Ginter. Design alternatives for a cooperative garbage collector for the C++ programming language. Technical Report 91/417/01, Department of Computer Science, University of Calgary, 1992.

[Gup90] Rajiv Gupta. A fresh look at optimizing array bound checking. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 272–282, June 1990.

[Har77] W. Harrison. Compiler analysis of the value ranges of variables. *IEEE Transactions on Software Engineering*, 3(3):243–250, 1977.

[HJ92] Reed Hastings and Bob Joyce. Purify: fast detection of memory leaks and access errors. *Proceedings of the Winter Usenix Conference*, 1992.

[KLP88] Stephen Kaufer, Russel Lopez, and Sesha Pratap. Saber-C: an interpreter-based programming environment for the C language. *Proceedings of the Summer Usenix Conference*, pages 161–171, 1988.

[Lar93] James R. Larus. Efficient program tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[Lee91] Peter Lee, editor. *Topics in Advanced Language Implementation.* The MIT Press, Cambridge, MA, 1991.

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of Unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[Ros87] Graham Ross. Integral C – a practical environment for C programming. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SIGPLAN Notices)*, pages 42–48. Association for Computing Machinery, January 1987.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, CA, January 1988.

[SC91] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability – a study of field failures in operating systems. *Digest of the 21st International Symposium on Fault Tolerant Computing*, pages 2–9, June 1991.

[Ste92] Joseph L. Steffen. Adding run-time checking to the Portable C Compiler. *Software – Practice and Experience*, 22(4):305–316, 1992.

[ZH88] Benjamin Zorn and Paul Hilfinger. A memory allocation profiler for C and Lisp programs. *Proceedings of the Summer Usenix Conference*, pages 223–237, 1988.