

Precise Garbage Collection for C

(submission for double-blind review)

Abstract

Xyzzy is a source-to-source transformation for C programs that enables precise garbage collection, where *precise* means that integers are not confused with pointers, and the liveness of a pointer is apparent at the source level. Precise GC is mainly useful for long-running programs and programs that interact with untrusted components. In particular, we have successfully deployed precise GC in the C implementation of a language run-time system that was originally designed to use conservative GC. We also report on our experience in transforming parts of the Linux kernel to use precise GC instead of manual memory management.

1. GC in C

Automatic memory management simplifies most implementation tasks, and among programming languages currently in widespread use, most automate memory management through garbage collection (GC). The C programming language is a notable exception, leaving memory management largely in the hands of the programmer. For some tasks, especially relatively low-level tasks, explicit control over memory management is useful and important. Even so, C is used for many implementation tasks that would benefit from automatic memory management.

A popular approach for adding GC to C is to use *conservative GC* [8]. A conservative GC assumes that any word in a register, on the stack, in a static variable, or in a reachable allocated object is a potential pointer that counts toward the further reachability of objects. A conservative GC is not completely “conservative” in the sense of retaining any object whose address might be computed eventually, because the C language allows arbitrary references to be synthesized, which would mean that no object could ever be collected. Nevertheless, the approximation to “conservative” in a conservative GC works well in practice, because C programs rarely synthesize pointers out of thin air.

A conservative GC is typically implemented as a library, which makes using conservative GC relatively easy. In many cases, calls to `malloc` can be replaced with `GC_malloc`, and calls to `free` can be turned into no-ops. The resulting program usually runs about as well as before, but without the ongoing maintenance burden of manual memory management. In our experience, however, conservative GC works poorly well for long-running programs, such as a web server, a programming environment, or an operating system kernel. For such programs, conservative GC can trigger unbounded memory use due to linked lists [7] that manage threads

and continuations; this problem is usually due to liveness imprecision [19], rather than type imprecision. Furthermore, the programs are susceptible to memory-exhaustion attack from malicious code (e.g., user programs or untrusted servlets) that might be otherwise restricted through a sandbox [33].

In this paper, we report our design and experience with a GC for C that is less conservative. Our *precise GC* counts references toward reachability only for words that correspond to values whose static type is a pointer type, and only for words that correspond to variables that are in scope. Of course, this GC is not completely “precise” in the sense of retaining only objects that will be referenced later in execution, but it is precise in the sense that a program can be written to reliably drop all references to a given object; no integers or dead registers will accidentally or maliciously retain the object.

Precise GC requires that a C program obeys many restrictions—the same ones that apply to using conservative GC, and more—or the program can crash. For example, to use precise GC with a C program, the program must never extract a pointer from a variable typed as `long`, at least not if a collection may have occurred since the value was stored in the variable. Fortunately, these extra constraints are satisfied by normal C programming practices. Furthermore, our implementation relies on fewer assumptions about the C compiler and target architecture than conservative GC, because it transforms the original program to explicitly cooperate with the GC.

Our original motivation for precise GC in C was to support a run-time system for a high-level language, XYZ Scheme. The XYZ Scheme run-time system is implemented in C, and it was originally designed to use conservative GC. Switching to precise GC has solved many memory-use problems for software that is built on XYZ Scheme:

- Users of the Qzqzqzq programming environment, which runs on top of XYZ Scheme, were forced to restart at least every day when XYZ Scheme used conservative GC. Even students writing tiny programs would have to restart after several hours. Users of Qzqzqzq with precise GC no longer restart Qzqzqzq to free memory. More concretely, running the current version of Qzqzqzq within Qzqzqzq using conservative GC accumulates around 40MB on every iteration, while memory use with precise GC is stable at around 100MB.
- When building bytecode and documentation for the current XYZ Scheme distribution, peak memory use is about 200MB using precise GC. With conservative GC, peak use is around 700MB—and the gap continues to widen as the documentation grows more complex.
- With conservative GC, a mail client built on XYZ Scheme would have to be restarted every couple of days, or immediately if after opening a large attachment. With precise GC, the mail client does not run out of memory, and it routinely runs for weeks at a time.

[Copyright notice will appear here once ‘preprint’ option is removed.]

More generally, the switch to precise GC allows XYZ Scheme to provide programmers with the usual guarantees about memory use, such as being safe for space [1, 10].

Based on our positive experience with precise GC in XYZ Scheme, we have experimented further by testing its use on other C programs—even applying it to significant subsystems of the Linux kernel. Our *Xyzy* tool automates the conversion of C source programs to cooperate with precise GC. *Xyzy* handles many C programs automatically, and it can handle others with varying levels of programmer intervention.

The contributions of this paper are twofold. First, we describe the design and implementation of *Xyzy*'s conversion of C to cooperate with a precise GC, including the assumptions and heuristics that have proven effective on a spectrum of programs. Second, we provide a report on applying *Xyzy* to the XYZ Scheme run-time system, other user applications, and the Linux kernel.

The rest of this paper is organized as follows. Section 2 describes the semantic assumptions of precise GC with respect to C programs. Section 3 describes the transformation of C programs that enables them to use precise GC at run time. Section 4 details our experience using and benchmarking precise GC. Section 5 compares to related work.

2. C Semantics and Precise GC

The key constraint on a C program to use precise GC is that static types in the program source must match the program's run-time behavior, at least to the degree that the types distinguish pointers from non-pointers. The implementation of precise GC, meanwhile, is responsible for handling `union` types and the fact that the static type of a C variable or expression often provides an incomplete description of its run-time shape. For example, `void*` is often used as a generic pointer type, where the shape of the referenced data is unspecified. Thus, the concerns for precise GC in the source program can be divided into two parts: those that relate to separating live pointers from other values, and those that relate to connecting allocation with the shape of the allocated data.

2.1 Pointers vs. Non-pointers

For GC to work, all live objects must be reachable via pointer chains starting from roots, which include static variables and local variables with pointer types. For precise GC, the values of variables and expressions typed as non-pointers must never be used as pointers to GC-allocated objects, because objects are not considered to be reachable through such references. In addition, to support GC strategies that relocate allocated objects, *Xyzy* disallows references through non-pointer types even if the same object is reachable through a pointer-typed reference. Memory can be allocated by `malloc` and other system-supplied (non-GC) allocators, but the allocated memory is treated by the GC as an array of non-pointers.

Object-referencing invariants must hold whenever a collection is possible, but some code (such as a library function) might safely break them temporarily between collections. Library functions that are opaque to the GC remain safe as long as they do not invoke callbacks or save pointers. Custom allocators, in contrast, typically break these invariants permanently, because they allocate objects of statically unspecified shape in a block of primitively allocated memory. In Section 2.3, we discuss other ways in which realistic C programs sometimes break these invariants and how those problems can be addressed.

Some constraints on the C program depend on the specific GC implementation that is used with *Xyzy*-converted code. The GC may support *interior pointers*, i.e., pointers into the middle of a GC-allocated object, including to the end of the object. In practice, we find that interior pointers are common in arbitrary C

programs, while programs specifically written to work with precise GC (such as the XYZ Scheme runtime system) can avoid them to help improve GC performance. Note that a pointer just past the end of an array is legal according to the C standard, and *Xyzy* considers such pointers to be valid interior pointers.

The GC may also allow a pointer-typed location to refer to an address that is not a GC-allocated object. In practice, we find that references that point outside of GC-allocation regions should be allowed always, so that pointers produced by a non-GC allocator can be mingled with pointers to GC-allocated objects. GC implementations that relocate objects require that a pointer-typed object either refer to a GC-allocated object or refer outside the region of GC-allocated objects; otherwise, an arbitrary value might happen to overlap with the GC-allocation region and get “moved” by a collection. Pointer-valued local variables need not be initialized explicitly to satisfy this requirement, because *Xyzy* inserts initialization as needed.

A C `union` type can give a single memory location both pointer and non-pointer types. *Xyzy* requires that only one variant of the union is conceptually “active” at any time, as determined by the most recent assignment into the union or address-taking of a union variant, and accesses of the union must use only the active variant. *Xyzy* tracks the active variant as determined by assignments, address operations, and structure copying through `memcpy`.

2.2 Allocation Shapes

To dynamically track the shapes of referenced data, all allocation points must be apparent, and the type of the data being allocated—at least to the degree that it contains pointers versus non-pointers—must be apparent. More generally, *Xyzy* requires that the types of data allocated by a C program fit into four categories: atomic (non-pointer) data, arrays of pointers, single structures, and arrays of structures. Single structures and arrays of structures are separated for performance reasons, since the former are simpler and more common. Similarly, pointer arrays and atomic arrays could be treated as structure arrays, each containing only a pointer or a non-pointer, but they are common enough to be handled directly.

A combination of type and category is determined from each allocation expression, which must use a recognized allocation function—typically `malloc`, `calloc`, and `realloc`, but our tool supports a configurable set.¹ The argument to the allocation function must be computed via `sizeof`, except when allocating atomic data. Roughly, the size expression is expected to be one of the following:

<code>sizeof(t)</code>	allocates a single <i>t</i> structure
<code>sizeof(t) * e</code>	allocates an array of <i>t</i> structures
<code>sizeof(t*) * e</code>	allocates a pointer array
<code>e</code>	allocates an atomic block

If *t* is a pointer type, then `sizeof(t)` is treated like `sizeof(t*)` by allocating a pointer array. Similarly, if *t* is a structure type that has no pointer fields, then `sizeof(t)` can be treated as just a number to allocate an atomic block. The interesting case is when *t* is a structure type that contains a mixture of pointers and non-pointers; a single structure or array of structures is allocated, depending on whether the `sizeof` expression is multiplied by another expression.

More generally, an allocation size can be computed with operations other than just `sizeof` and `*`, and the `sizeof` sub-expression need not appear on the left-hand side of a multiplication. When *Xyzy* sees a multiplication, it checks both operands; if one or the other (but not both) indicates a single structure or structure array,

¹To keep a call to `malloc` intact, so that it continues to allocate non-GC memory in the converted program, wrap the call in a function whose implementation is withheld from *Xyzy*.

then the multiplication expression indicates a structure array. A division, left-shift, or right-shift operation also indicates a structure array when the first argument indicates a single structure or structure array. For an addition or subtraction expression, the operands are checked separately, and the result with the higher precedence applies to the overall expression, where the precedence order is (least to greatest) atomic block, single structure, pointer array, and structure array.

For example, the code

```
int ** x = malloc(sizeof(int*) * 5);
```

would be considered a pointer to an array of pointers, because `int*` in `sizeof(int*)` is a pointer, and `5` is atomic. Allocators that return arrays of objects, such as `calloc`, return an array of the resulting type (i.e. `calloc(sizeof(int), 2)` is the same as `malloc(sizeof(int)*2)`).

2.3 Violating Assumptions

A programmer can easily construct C code that does not satisfy Xyzzy's assumptions but that works in a non-GC setting. In this section, we point out some specific issues and relate our experience with realistic programs.

Pointer Manipulation Pointer arithmetic can shift an address so that it does not refer to an allocated object, and then further arithmetic can shift the address back. If a collection can occur between the arithmetic operations, then a reachability assumption of our GC has been violated. The following pathological code illustrates this problem, assuming that `work` might trigger a collection. It uses pointer arithmetic to treat an array as indexed from 1024 to 2047, instead of 0 to 1023:

```
int *p = GC_malloc(sizeof(int) * 1024);
p -= 1024;
work(p);
p[1024];
```

Programmers rarely create code like this, but we did encounter it in one Spec2000 benchmark. Such pointer manipulations trigger unspecified behavior with Xyzzy.

Other potential kinds pointer manipulation (that we did not encounter) include saving pointers to disk and later loading and dereferencing them, or using exclusive-or operations to collapse pointers in a doubly-linked list. Again, those pointer manipulations would trigger unspecified behavior with Xyzzy.

A programmer might more reasonably use pointer arithmetic to compute a position within an allocated object or to increment or decrement a pointer in a loop. Such code works fine with Xyzzy, since interior pointers are allowed.

Pointers as Integers and Integers as Pointers For various reasons, some C programs store pointers as integers or vice-versa. In many cases, passing a pointer/integer as an integer/pointer corresponds to an implicit `union`. For example, some functions support a kind of closure argument by accepting a function pointer combined with a data pointer that is supplied back to the function; a programmer who needs to pair a function with an integer may simply pass the integer as a pointer instead of allocating space to hold the integer. In other cases, such as in the main Ruby implementation [28], pointers are represented as integers to facilitate tests on the bits that form an address.

Programs that mix pointers and integers in this way can work with conservative GC, but they generally do not work with precise GC. (Although precise GC can work with explicit `unions`, assuming an implicit `union` on every word of data is not practical.) For a GC implementation that does not relocate allocated objects, storing an integer in a pointer-typed location does not lead to a crash, but it

may cause an object that should be freed to be retained. The problem is worse for a GC implementation that relocates objects, since relocation could change a value stored in a pointer-typed location that is actually used as an integer. Of course, storing a pointer in an integer-typed location can lead to a crash with any precise GC implementation.

We have no work-around for programs that use integer locations to store pointer values. The Ruby implementation is the only program that we attempted to convert to precise GC where we found such a mismatch, and in that case, we gave up on conversion.

Unconverted Libraries Libraries that are not instrumented to support precise GC often can be used within an application using precise GC. Indeed, any useful application that runs on a stock operating system must at least use system functions, and most programs rely on standard libraries that do not cooperate with GC.

Since unconverted libraries do not directly call GC allocation functions, collections are normally not possible during a library function call. The fact that pointers passed to the function are effectively hidden, then, does not break Xyzzy assumptions in a way that matters. Two possibilities can break assumptions in a significant way: a library function might keep a pointer that it was given and try to use the pointer in later calls, and a library function might invoke a callback that uses precise GC before the library function returns. These issues are essentially the same as for the foreign-function interface of a high-level language with GC.

As an example of the first case, a function might take a string pointer and store it in a table that is used by future function calls. Since programmers are generally responsible for managing memory, functions that keep pointers must be documented as doing so, which means that the cases where functions create trouble for precise GC are clear, at least at the level of documentation. Furthermore, since manual memory management is so easy to get wrong, library APIs tend to avoid relying on memory management by the caller; a library function that accepts a string will typically copy the string content if it needs to be preserved for later use by the library.

In some cases, a library function cannot avoid relying on the caller to handle allocation. For example, library may record a mapping from string names to objects, and the library cannot in general make copies of objects. Such cases are not handled automatically by our tools. Instead, a programmer must manually use the GC's *immobile boxes*, which are explicitly allocated and freed, but which can refer to GC-allocated objects (unlike memory allocated by a non-GC `malloc`).

When a library invokes callbacks into code that uses GC, then any objects originally passed to the function may have moved by the time the callback returns. Library functions that invoke callbacks must be treated in a similar way to functions that store pointers, where the return from the callback is analogous to calling a second library function that uses values stored by the earlier call.

3. Transformation

The transformation from an unmodified C program to one that can use precise GC requires the following five conversions: insert code into functions to track local variables [18], generate traversal routines for structures that contain pointers [16], replace allocation sites with calls to GC allocators, track unions, and identify static/global variables.

3.1 Track Local Variables

Figure 1 illustrates how Xyzzy exposes stack variables to the collector through a per-function transformation. Variables that have pointer type and structures with pointer-typed fields have their address added to a stack-allocated linked list that shadows the C stack. The `GC_set_stack_frame()` function sets the current GC frame

```

// ORIGINAL
int cheeseburger(int *x) {
    add_cheese(x);
    return x[17];
}

// TRANSFORMED
int cheeseburger(int *x) {
    void* gc_stack_frame[3];
    /* chain to previous frame: */
    last_stack_frame = GC_last_stack_frame();
    gc_stack_frame[0] = last_stack_frame;
    /* number of elements + shape category: */
    gc_stack_frame[1] = (1 << 2) + 0;
    /* variable address: */
    gc_stack_frame[2] = &x;
    /* install frame: */
    GC_set_stack_frame(gc_stack_frame);

    add_cheese(x);

    /* restore old GC frame */
    GC_set_stack_frame(last_stack_frame);
    return x[17];
}

```

Figure 1. Stack-registration conversion

that the collector will traverse, while `GC_last_stack_frame()` returns it. These functions are normally inlined during compilation.

The example in Figure 1 has a single local variable in the pointer category. If the function included additional pointer variables, they would be added to the same frame. If the function includes variables that are arrays of pointers, structures, or arrays of structures, they would be added to additional frames, one for each shape category.

For each category, the general shape of the frame is as follows:

```

frame[0] = last_gc_frame;
frame[1] = (number_of_elements << 2) + type;
frame[2] = &variable1;
frame[3] = &variable2;
...

```

Traversing the stack during the mark phase of a garbage collection consists of walking the GC frames and applying the correct mark and repair functions to each pointer in the frame.

A non-local jump via `longjmp` or `setcontext` prevents the normal stack unwinding from occurring on return, which leaves the GC shadow stack in an inconsistent state relative to the program stack. To counter this, after any call to `setjmp` or `getcontext`, Xyzy re-establishes the GC stack by re-installing the enclosing function’s GC frame.

A local variable is registered with the collector only if its value is needed across a call that potentially triggers a GC. For example, the transformation of the function

```

int feed() {
    int *x;
    x = create_burger(75);
    return cheeseburger(x);
}

```

does not introduce a GC frame. Although the local variable `x` is a pointer, it acts only as a temporary location to get the result from `create_burger` into `cheeseburger`, and no GC is possible

during the transition. Since our transformation does not register a location for `x` with the GC, the C compiler is therefore able to use a register for `x`, instead of allocating a stack location for it.

Even when a variable is live across a function call, the function does not necessarily lead to a GC. For example, the transformation in Figure 1 is needed only when `add_cheese` allocates. Prior to transforming any functions, our tools can perform a call-graph analysis to determine which functions lead to allocation, so we can identify functions that do not call any allocators and therefore do not need to create GC frames. This analysis and optimization can dramatically reduce the overhead of GC on small, frequently called functions that compute without allocation (as demonstrated in the benchmarks of Section 4.2).

Nested blocks with local variables can be treated like nested functions, where `break`, `continue`, and `goto` statements must be converted (similar to `return`) to correctly remove a nested GC frame. Most of our experiments have used that approach, but the transformation that we use on the Linux kernel simply lifts local variables to the beginning of the function (and this lifting is performed automatically by CIL [5], which we use to parse C source). The variable-lifting implementation treats the entire function body as the live range for each local variable.

3.2 Traversal Routines

For each structure and array type that is declared in the original program, if the structure or array contains pointers, Xyzy generates a *mark* and *repair* function for the type. These functions are stored in a `gc_tag_struct` structure, and when an object is allocated, the relevant `gc_tag_struct` structure is associated with the allocated object.

The mark function is used to traverse a structure or array during the mark phase of a collection, while the repair function is used—only with GC implementations that relocate objects—to update pointers when objects are moved. For example, given the following struct:

```

struct a {
    int * x;
    int * y;
}

```

A mark function is generated to call `GC_mark` on each pointer within an instance of the struct:

```

void gc_mark_struct_a(void * x_){
    struct a * tmp = (struct a *) x_;
    GC_mark(tmp->x);
    GC_mark(tmp->y);
}

```

Similarly, a repair function is generated to call `GC_repair` on the address of each pointer within the struct:

```

void gc_repair_struct_a(void * x_){
    struct a * tmp = (struct a *) x_;
    GC_repair(&tmp->x);
    GC_repair(&tmp->y);
}

```

Xyzy normally generates mark and repair automatically when transforming C code to cooperate with GC, but a programmer can optionally annotate the original code to provide hand-implemented mark and repair functions. Hand-implemented mark and repair functions are useful when a data structure does not fit into one of the four categories that the GC transformation can recognize (see Section 2.2) or when unions are better handled by using existing information in the data-type instead of adding code to track the active variant (as discussed in Section 3.4).

3.3 Allocation Conversion

The transformation to support precise GC must detect allocations and replace them with GC allocations, in the process associating tags (for mark and repair routines) with the allocated data. For some types of data, a generic tag such as `gc_atomic_tag` can be used. For example, the allocations

```
p = (int *)malloc(sizeof(int)*n);
a = (int **)malloc(sizeof(int*)*m);
f = (bun *)malloc(sizeof(bun));
```

are converted to

```
p = (int *)GC_malloc(sizeof(int)*n, gc_atomic_tag);
a = (int **)GC_malloc(sizeof(int*)*m, gc_array_tag);
f = (bun *)GC_malloc(sizeof(bun), gc_bun_tag);
```

where `gc_bun_tag` is generated from the declaration of the `bun` type.

The tag is computed from the expression that contains `sizeof`. The type argument of the `sizeof` operator could identify an atomic type, a pointer type, or a structure type, which selects respectively `gc_atomic_tag`, `gc_array_tag`, or a specific tag for the structure.

3.4 Unions

When a type contains a union of pointer and non-pointer types, then the mark and repair routines need to follow and update a pointer variant only when it is active. The active variant of a union is tracked by storing an extra byte outside of the object whenever a field of the union is assigned or its address is taken. The automatically generated mark and repair routines consult the byte to determine whether to follow or repair the pointer variant.

For example, given the declaration

```
union {
    int i;
    int *p;
} a;
```

then the sequence

```
a.i = 1;
iwork(&a);
a.p = q;
pwork(&a);
```

is converted to

```
a.i = 1;
GC_autotag_union(&a, 0);
iwork(&a);
a.p = q;
GC_autotag_union(&a, 1);
pwork(&a);
```

In addition, active-variant information must be copied between unions on struct assignment and struct copying via `memcpy`.

For a union that is allocated on the stack, the active-variant information is also recorded on the stack, and the connection between the location and its active variant is recorded in a GC frame. For a union that is allocated in the heap (by the GC), its active variant is recorded by associating an array of bytes to all memory locations, setting the appropriate byte when a union variant is activated.

3.5 Static/Global Variables

In addition to converting functions to register local variables, Xyzyz locates all static/global variables and registers them in initialization functions that are called when the program starts. Each variable's location is registered with a tag that provides mark and repair functions and (if necessary) union variant tracking.

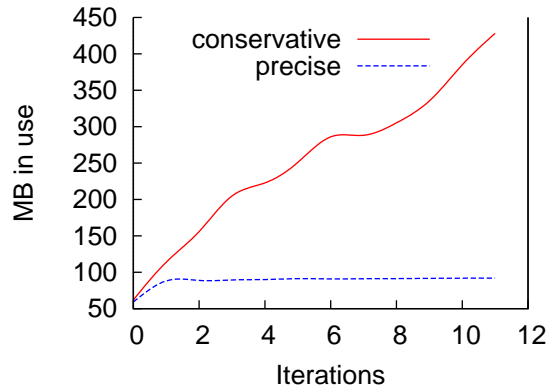


Figure 2. Running Qzqzqzq inside Qzqzqzq

4. Implementation and Experience

We have produced three implementations of Xyzyz. The first is used within the XYZ Scheme run-time system, and is fairly specific to that code base. The second was developed to test our approach to precise GC on arbitrary C applications. The third was developed to apply precise GC to the Linux kernel.

4.1 Precise GC for XYZ Scheme

Our exploration of precise GC in C started as a way to improve the memory performance of the Qzqzqzq programming environment, which is built on XYZ Scheme. Although Qzqzqzq is implemented in Scheme, the XYZ Scheme run-time system—which provides the core language, threading, and GUI constructs—is implemented in C and C++. The run-time system was originally implemented with conservative GC, because that choice allowed us to quickly combine a Scheme interpreter with an existing GUI library. Also, relying on GC for new C and C++ code let us avoid problems with manual memory management.

For a while, our investment in the C implementation grew rapidly, and conservative GC seemed to work well enough. As Qzqzqzq grew ever larger and more complex, however, the imprecision of conservative GC interfered with the way that we wanted to structure the programs. Threads became particularly troublesome; Qzqzqzq is designed to run untrusted programs within the same virtual machine as the programming environment itself, which simplifies communication for debugging, and we have developed sandboxing techniques and language constructs within XYZ Scheme to keep debugged code separate from the environment. These techniques, however, rely on reliable garbage collection of deadlocked threads and other scheduling entities. Unfortunately, since threads often refer to each other in various ways, they end up forming the kind of linked lists that Boehm warns against [7]. That is, retaining an extra thread due to GC imprecision can end up holding onto many extra threads, each of which refers to lots of other code and data, and so on. We attempted to help the GC by manually breaking links within the run-time system, but since threads capture stacks and registers, conservative GC does not provide enough guarantees to reliably break links. The end result is that running a program repeatedly within Qzqzqzq would cause memory use to grow without bound.

Since we were stuck with a huge investment in C code for XYZ Scheme, we created an ad hoc tool to transform our C and C++ code to support precise GC. This conversion strategy let us work on the GC problem while the C code continued to evolve to meet other needs. Eventually, we were able to replace conservative GC with precise GC in the default XYZ Scheme build, and that

	fastest (msec)	conservative GC / mutator only	precise GC / mutator only
cpstack	441	4.48 / 1.56	1 / 0.79
ctak	4916	1.38 / 0.63	1 / 0.94
dderiv	614	4.58 / 1.45	1 / 0.87
deriv	546	4.81 / 1.52	1 / 0.87
div	526	6.28 / 2.00	1 / 0.74
dynamic2	1856	1 / 0.64	1.05 / 0.64
earley	511	1.79 / 0.58	1 / 0.51
fft	2246	1.30 / 1.30	1 / 0.87
graphs	709	3.03 / 1.17	1 / 0.84
lattice2	6280	1 / 0.90	1.08 / 1.08
maze2	619	1.30 / 0.89	1 / 0.95
mazefun	894	1.77 / 1.06	1 / 0.94
nboyer	6497	1.66 / 0.66	1 / 0.47
nestedloop	727	1 / 1	1.16 / 1.16
nfa	352	1 / 1	1.20 / 1.20
nqueens	460	2.58 / 1.19	1 / 0.93
nucleic2	399	2.70 / 1.43	1 / 0.88
paraffins	351	3.31 / 0.98	1 / 0.45
puzzle	279	1 / 1	1.02 / 1.02
sboyer	13387	1.15 / 0.91	1 / 0.89
scheme2	470	1.16 / 0.87	1 / 0.96
tak	441	1 / 1.14	1 / 1
takl	869	1 / 1	1.28 / 1.28
takr	1592	1 / 1	1.04 / 1.04
triangle	140	1 / 1	1.01 / 1.01

Figure 3. XYZ Scheme benchmarks, normalized to fastest Intel Core Duo 2GHz, 1.25 GB, OS X 10.5.6, XYZ Scheme 4.1.4

change immediately solved a large class of memory-use problems. In particular, repeatedly running `Qzqzqzq` within itself maintains a steady state for any number of runs. Figure 2 illustrates graphically.

The precise GC implementation for XYZ Scheme works only when the C/C++ code follows a certain constrained style, which the XYZ Scheme maintainers must follow. Memory management uses a hybrid copying–compacting collector, where objects are allocated into a nursery and surviving objects are copied into an older generation, which is itself periodically compacted. This GC implementation does not allow interior pointers, except for large blocks of memory that are allocated specifically with interior-pointer support. Disallowing interior pointers reduces the GC overhead, but increases the burden on maintainers of the C/C++ code to not create interior pointers.

For typical non-allocating tasks, XYZ Scheme with precise GC performs 10%–20% slower than the conservative GC variant, mostly due to the overhead of registering stack-based pointers and other cooperation with the GC. Programs that allocate significantly, however, tend to run faster with precise GC, due to its lower allocation overhead and faster generational-collection cycles. Consequently, most XYZ Scheme programs run faster with precise GC while also using less memory.

The table in Figure 3 shows the run times of XYZ Scheme on several standard Scheme benchmarks, comparing Boehm’s conservative GC [8] to precise GC. (Note that these are different Scheme benchmarks running in a single C-implemented run-time system, as opposed to different C programs. From the perspective of testing precise GC, we are effectively varying the inputs and keeping the program constant.) The numbers are normalized to the fastest run of each benchmark, including the time required for garbage collection; each cell in the table shows total run time followed by the run time not counting garbage collection. The times in the table are for XYZ Scheme using a just-in-time (JIT) native-code compiler, but the relative results are similar when the JIT is disabled.

	conservative GC	precise GC	no opt
164.zip	1.03	1.09	1.31
175.vpr	0.98	0.96	1.02
176.gcc	-	1.34	1.61
179.art	1.00	0.94	0.94
181.mcf	1.00	1.00	1.15
183.quake	1.00	0.99	1.02
186.crafty	0.99	1.02	1.09
186.amp	0.90	0.96	0.96
197.parser	1.44	5.35	6.89
197.parser*	1.44	3.52	5.22
254.gap	1.00	2.39	2.46
256.bzip2	1.01	0.99	0.99
300.twolf	0.95	0.88	0.89

Figure 4. Xyzzy benchmarks, normalized to original Intel Pentium 4 1.8 GHz, 256 MB, FreeBSD 4.11-STABLE

The table shows that benchmarks run slower with conservative GC when collection is a significant part of the benchmark; that is, when the second number for the conservative GC column is much less than the first, then both numbers tend to be larger than the corresponding numbers in the precise GC column. These results demonstrate how both allocation and collection tend to be faster in our precise GC implementation. For benchmarks that spend relatively little time collecting, however, the conservative GC implementation can be up to 20% faster than precise GC.

Thus, the benchmarks illustrate that we pay a price in base performance when building on a C infrastructure with precise GC compared to using conservative GC. In the context of a run-time system for a functional language, however, allocation is common and guarantees about space usage important, so that the cost in base performance is worthwhile.

4.2 Precise GC for User Applications in C

Yyy’s dissertation [...] describes an implementation of Xyzzy for converting arbitrary C code to work with precise GC. Unlike the tool used with XYZ Scheme—where the maintainers of the C code initially used conservative GC and are willing to write new code in a slightly constrained style to better support GC—this implementation is intended for use on “legacy” C code that was designed and implemented with manual memory allocation. Xyzzy thus provides a general migration path from manual memory management in C, and also lets us measure the cost and benefits of precise GC compared to manual memory management.

An important component of Yyy’s Xyzzy is its graphical user interface (GUI) for exploring and confirming allocation and reference points within a C program. The GUI steps through each point in a program that appears to be an allocation, and it asks the user to confirm Xyzzy’s inference of the type of allocation that is performed. The GUI also asks the user to confirm Xyzzy’s analysis of datatype definitions and how they relate to needed mark and repair procedures. Finally, the GUI lets a user confirm Xyzzy’s tracking of unions, and it encourages the user to explain how existing fields in a structure can be used to select the active union variant instead of relying on separate tracking. Of course, Xyzzy also offers a non-GUI mode, in case a programmer believes that the tool’s automatic inferences will be completely correct, as is often the case.

Figure 4 summarizes the results of using Xyzzy on several Spec2000 benchmarks, reporting the run time relative to the original program after replacing allocation with Boehm’s conservative GC [8] or Xyzzy’s precise GC. The “no opt” column of the table reports the run time when using precise GC with Xyzzy’s interprocedural analysis disabled (so that each function is transformed with the assumption that any function call can trigger a collection).

The 197.parser benchmark is of special interest, since its base performance relies on a custom memory allocator. Both the conservative and precise GC variants of this benchmark replace the custom allocator with the precise GC allocator, but in the variant of the benchmark marked with an asterisk, custom mark and repair routines allow Xyzzy to avoid tracking the active variant of a union. Even so, the precise GC variant is considerably slower than the conservative GC variant. Xyzzy’s optimizer is able to speed the program considerably by avoiding unnecessary registrations of local variables with the GC. Furthermore, hand tuning of the Xyzzy output—to avoid some remaining registrations—brought performance of precise GC in line with conservative GC.

The 254.gap benchmark also runs much more slowly with precise GC than conservative GC. In this case, Xyzzy’s optimization provided relatively little improvement, probably because the benchmark uses function pointers extensively, which weakens Xyzzy’s analysis. More significantly, profiling suggests that conversions for precise GC cause addresses of local variables to be taken inside tight loops, handicapping compiler optimizations.

Yyy’s dissertation provides an extensive report on the process and difficulty of converting the benchmarks for precise GC. To summarize, Xyzzy’s inference can convert all of the benchmark programs automatically, and confirming the inference through Xyzzy’s GUI took no more than a few minutes for each benchmark. Yyy’s experiments also confirm that memory usage in both the conservative and precise GC variants of the benchmark track memory usage of the original program.

Overall, experiments with Xyzzy suggest that precise GC is a viable alternative to manual memory management for a typical C program. However, it is not especially beneficial for short-running programs without untrusted components. For a domain where precise GC can offer useful guarantees compared to conservative GC, we turn our attention to an OS kernel.

4.3 Precise GC for Linux

Memory management in an operating system (OS) is difficult for several reasons: memory leaks are harmful, since an OS should be able to go for months between reboots; memory corruption, for example due to use of freed storage, is extremely undesirable, since it can lead to security violations; since many operating systems are used in throughput-oriented applications, such as database servers, memory allocation protocols are typically finely tuned, and therefore difficult to understand, debug, and modify; and modern operating systems tend to be monolithic, meaning that many millions of lines of code run in a single address space, lacking any significant isolation from kernel components that leak memory. These problems result in operating systems that are insecure, unreliable, and hard to maintain. Indeed, it is not uncommon to periodically reboot a system that is known to suffer from memory management problems. Engler et al. [14] found a number of memory management bugs in Linux—and certainly left others undiscovered.

High-level languages with automatic memory management have been used to implement operating systems before, including SPIN [6], Lisp machines [9], JavaOS [31], and Singularity [21]. The performance implications of GC in such settings, however, are mixed with other details of a safe programming model that are potentially costly. Our approach to precise GC in C offers a way to test the effectiveness and cost of GC in an OS kernel without have to re-write a stock, C-implemented kernel. In particular, we have applied our precise GC to the core of the Linux kernel.

Neither the XYZ Scheme tool nor the improved Xyzzy was up to the task of converting Linux code, which uses many gcc extensions of C. Since parsing was the main obstacle at first, we opted to use the CIL parsing and analysis engine [5] for a new implementation. The resulting tool is a 2300 line OCaml program

	base time (sec)	precise GC	no opt
175.vpr	250.08	1.03	1.16
164.gzip	68.33	1.01	1.21
197.parser	444.71	4.45	13.83
300.twolf	787.45	1.01	1.04
456.hmm	1034.70	1.18	1.36
464.h264	2466.33	1.33	1.73
lame(mp3)	90.139	0.99	1.12

Figure 5. CIL-based Xyzzy benchmarks. normalized to base time Intel Pentium 1.8GHz, 768 MB, Linux 2.6.22.4

that can process hundreds of kilobytes of C programs in less than a second. Otherwise, the tool uses essentially the same inference and conversion algorithms as the second Xyzzy.

As an initial test, we applied our new tool to several Spec2000 benchmarks and some other programs. Figure 5 shows the relative run time of the benchmarks converted to precise GC. Although only half of the benchmarks are the same as tested with the earlier version of Xyzzy, the overall results are consistent with the earlier results (as listed in Figure 4).

Transforming the Linux kernel presented interesting additional challenges for our tool (besides merely parsing gcc syntax). The Linux kernel is about 95% C code and rarely violates the constraints set out in Section 2, but the kernel includes some assembly code, and it occasionally violates our constraints. Also, Linux uses many different allocators, which complicates the transformation of allocation sites.

Partial Transformation Altering the entire kernel to properly use a precise collector would probably require man-years of effort. We simplified the problem in several ways:

- We did not convert the entire kernel. Instead, we concentrated conversion on the ext3 filesystem code, the kernel filesystem interface, and the IPV4 network stack, because they are well modularized and easy to benchmark. To ensure that the kernel references to these subsystems were visible to the GC, we also converted parts of the scheduler. In total, we transformed 74,975 lines of C source (386,511 after pre-processing), and we modified 85 lines by hand. Since we transformed only part of the kernel, we disabled Xyzzy’s interprocedural analysis.
- The GC never moves allocated objects. This simplification allows the kernel to store references in places that the GC does not traverse—especially in unconverted parts of the kernel—as long as the reference also exists in a place that the GC traverses.
- The GC allocates physical memory only, so that it can support kernel allocations that require contiguous physical memory. This choice effectively disables the virtual memory system for kernel data, and it could be restored by using separate allocation techniques for physical and virtual memory.
- We consider only execution on a uniprocessor, and we disable thread pre-emption within the kernel (which is the default configuration for Linux).

The main conversion task for the filesystem and network code took only a couple of hours. Expanding the set of converted files and correctly converting allocation sites, however, took much longer; the debugging and testing process stretched out to a couple of months of part-time work. We performed the conversion by running the transformation tool on a set of source files, recompiled the kernel for User Mode Linux [32], and ran the resulting kernel in a debugger. When a crash occurred, we were usually able to track it down to an allocator that was not correctly converted to a GC allocator.

Given this approach to partial conversion, we are not confident that memory is always traced correctly in our prototype. The partially converted kernel runs well enough to host a XYZ Scheme build, run the Apache and XYZ Scheme web servers, and perform other basic tasks for hours at a time. We believe that the prototype works well enough to provide meaningful performance results, but our conversion effort is a lower bound (well below the real cost!) of the effort needed for a correct conversion.

Allocation Much of the kernel’s code is no different from application code, and Xyzzy is able to insert appropriate GC stack frames and create most traversal routines. The kernel uses many different kinds of allocators, however, that are difficult to convert to GC allocations automatically. In particular, `kmem_cache_alloc` allocates an object of a predetermined size and then passes the object to an arbitrary function, usually for initialization purposes, before returning the value. We replaced by hand all calls to `kmem_cache_alloc` with `kmalloc` and a call to the initialization function. The filesystem code used an allocator, `alloc_fdmem`, which allocates memory from high memory (using `vmalloc`) if a large block of memory is required; we replaced this allocator with a call to `kmalloc` as well.

Some kernel data structures required hand coded traversal routines, because the structure could not be replaced with a garbage collector allocator. A `dentry_hash` structure is allocated with `vmalloc`, which our collector currently doesn’t handle, so each `dentry_hash` instance is registered with the GC as a root. Another example is loadable modules, which are not allocated at all, but instead are mapped on top of the memory layout of the data from the module file. Fortunately, modules do not point to any data structures that need to be traversed in the partial conversion, so new traversal routines were not created for them.

The kernel includes a lockless datastructure that is shared between processors by using a read-copy-update (RCU) object [29] that is attached to the data structure of interest. Although RCU objects exist for multiprocessing, they are relevant to memory tracing even on a uniprocessor. Each processor maintains its own copy of the RCU object, and when the last processor relinquishes control of the object, a predefined callback is executed, much like a finalizer. These finalizers hold on to references of objects that are unreferenced by the rest of the system, while the RCU objects are referenced only by a special cache that is maintained by the processor itself. We wrote explicit mark routines for RCU objects to detect and handle these cases.

Stack Handling One notable difference between the kernel and a normal application is the size of the stack. Each process has a corresponding kernel stack that is preallocated a fixed size, typically 8K on a 32-bit machine. Our experiments started with the user-mode Linux architecture, which uses a 4K stack. This turned out to be too small, because the GC frames add enough overhead to breach the 4K limit. After increasing the stack size to 8K, our transformation could successfully run without corrupting memory. The normal settings for an x86 build of Linux use an 8K stack, so this was not an issue when we tested on real hardware.

A single-threaded user application has only one stack, but a kernel associates a stack with each process. Our converted kernel creates a shadow GC stack for each process. We added a field to the process structure, `struct task`, called `gc_stack`, and we defined `GC_set_stack` to set this field in the current running process. During the kernel initialization, there is no current process, so we added an initialization stack and an additional global variable that records whether to use the initialization stack or the current processes stack.

GC Implementation One crucial aspect of integrating GC with Linux is how to initialize the GC’s data structures and register

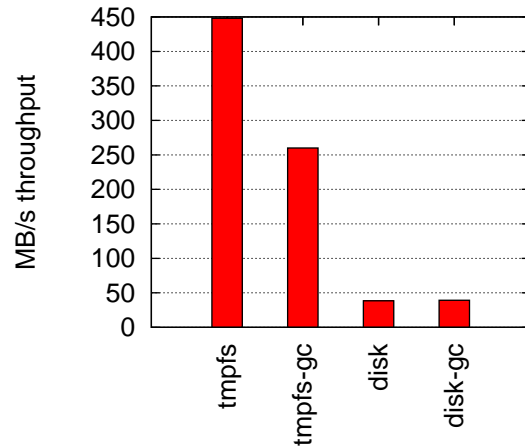


Figure 6. Running dd with an ext3 filesystem

root pointers. If GC initialization precedes virtual-memory initialization, then some data structures could not be allocated. If GC initialization is too late, then earlier parts of the kernel cannot use GC allocators. We added GC initialization immediately after virtual-memory initialization. Kernel initialization steps that precede virtual-memory initialize use boot memory, which the GC ignores.

The converted Linux kernel invokes the GC at a constant frequency, which lets us disable collections during interrupts. A GC that runs in parallel with the kernel would work better, because the kernel is not executing all the time, but we opted not to build a concurrent GC.

Interrupts are allowed to preempt processes, and they could potentially violate the invariants maintained by the GC shadow stack. However, because interrupts cannot force a garbage collection to occur, the interrupt return will restore the shadow stack to a consistent state.

To map each page in memory to a GC record for its meta-data, we use a pagemap structure in Linux, which provides a direct map between the first 896 megabytes of virtual memory and physical memory. Specifically, we store a pointer to the GC meta-data inside the page frame structure.

Performance Figure 6 shows the throughput of running dd under several configuration. The “tmpfs” bars show dd on an ext3 filesystem mounted inside an already-mounted tmpfs filesystem, and the run with GC is suffixed with “-gc.” In these cases, CPU is the bottleneck, and the results show a loss in throughput due to support for GC. We also tested a transformed kernel with collections disabled, but there was no measurable difference, which indicates that the overhead is entirely in instrumentation to cooperate with the GC. The bars prefixed with “disk” show the throughput of dd on a physical partition formatted with the ext3 filesystem. In this case, where disk is the bottleneck, no performance is lost.

Figures 7 through 9 show results from running the `dbench` test suite on a kernel with GC. The `dbench` filesystem benchmark suite is designed to run various filesystem operations and measure the duration of each operation. Figure 7 shows the amount of memory reclaimed by each collection during the benchmark run, which demonstrates that the benchmark generates considerable allocation work. Figure 8 shows the worst-case duration that `dbench` experienced while performing I/O. The spikes correspond to collections, although a spike did not occur for every collection. Finally, Figure 9 plots the time taken for a full collection. On average, the GC required 30ms to traverse the entire object hierarchy and free

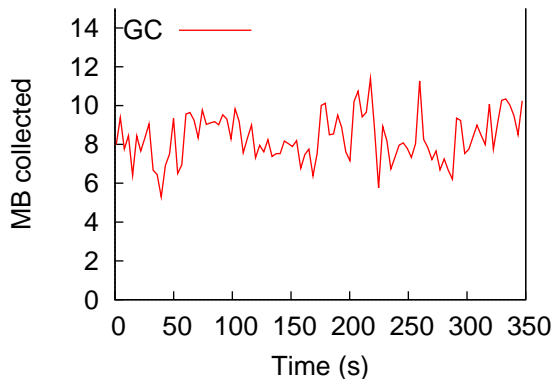


Figure 7. Memory collected during dbench; each data point represents a collection

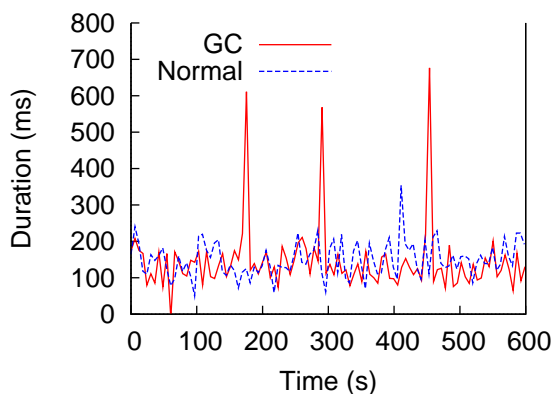


Figure 8. Duration of dbench filesystem operations with and without GC

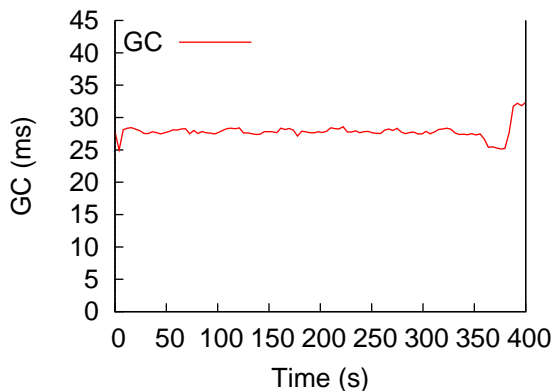


Figure 9. Time required for each collection during dbench

unused memory. We used the `rdtsc` (read timestamp counter) instruction to capture these timings, because the normal timing facilities that Linux offers do not work when interrupts are disabled.

5. Related Work

When adding a garbage collector to a C program, most programmers choose a non-moving, conservative collector [8]. These collectors easily link into C programs, as they require no further knowledge from the C runtime and do not move objects during collection. Using virtual memory, some conservative collectors provide a limited form of copying collector, with the copying hidden from the underlying C code [27], but such collectors are rare. Both types of conservative systems suffer from memory fragmentation [27] and memory leaks [7], arising from two common inference errors. First, the collector may mistakenly assume some value is a system root, which requires any object it reaches be saved. Second, the collector may assume that a numeric value in the heap is actually a pointer; considered as a pointer, such a number may reference a valid heap location, thus incorrectly marking that location as reachable. For most applications, conservative identification of roots is a more significant problem than misidentification of integers as pointers [19].

Our approach to precise GC for C is similar to Henderson’s approach [18] for precise collection in C code generated by the Mercury compiler. In particular, the shadow stack that we use for GC frame is the same as in Henderson’s system. Baker et al. [3] demonstrated an improvement to Henderson’s linked stack-frame technique by putting stack-variable addresses in a separate array, instead of embedding frames in the C stack, but also by using C++ exceptions to take pointer references only when a collection occurs. Jung et al. [23] use a similar approach; local variables are updated upon return from a function that performed a garbage collection. Compared to all of these systems, the main difference in our work is that Xyzy converts C code written by humans, instead of code generated by a compiler, so it must accommodate a wider variety of source code.

HnxGC [20] supports precise GC in C++ programs where the programmer uses special annotations and templates to enable GC allocation and reclamation. HnxGC also does not work on arbitrary code, and instead requires a programmer to use the HnxGC API. Other “smart pointer” libraries for C++ work in a similar way.

Xyzy assumes that a C program to transform obeys various constraints, but another approach to automatic memory management in C is to enforce constraints through a type system or sound analysis. Dharjati et al. enforce strong types for C (among other restrictions), and then soundly check the code for memory errors [11]. Although their approach works given the restrictions on the language, it still does not test for some kinds of memory errors. For example, it does not check that a program does not reference any deallocated objects. Cyclone [22], CCured [25], and Managed C++ [24] more aggressively follow the same path, leading to a language that is no longer C, though it may resemble C, and therefore requires porting the original program.

Another way to make memory management more reliable in C is to use tools that detect memory usage errors, instead of using automatic memory management [2, 12, 13, 15, 17, 30]. These approaches combine heuristics, programmer annotations (in some cases), and unsound analyses to find some set of situations that may cause a memory error. Although these tools somewhat assist the programmer, they all emit false positives. Further, they may miss situations leading to memory errors. Thus, they still potentially require considerable effort on the part of the programmer. Given the presence or lack of these tools, other researchers suggest methods for finding the source of memory leaks by hand [4, 26].

6. Conclusions

Automatic memory management offers many benefits for C code, and conservative GC and reference counting are fine choices for many programs. Those memory-management techniques are a poor match, however, for long-running programs that have complex reference patterns and that host extensions or untrusted code. For example, the main Ruby implementation uses conservative GC, Perl 5 uses reference counting with weak references, and Python uses conservative GC; programmers who stress these systems frequently encounter problems with memory use.

We have presented precise GC for C as a practical alternative, specifically as implemented in the Xyzy source-to-source transformer and related infrastructure. On the one hand, precise GC incurs mutator overhead, and it requires some programmer effort when the type structure of a program is not readily apparent. On the other hand, precise GC eliminates important classes of space leaks, and it enables shorter collection times. Precise C is a clear improvement over conservative GC for XYZ Scheme in terms of memory use, and most XYZ Scheme programs run faster with precise GC. We can whole-heartedly recommend precise GC for implementors of C-based language run-time systems who have previously relied on conservative GC or reference counting.

Emboldened by success in XYZ Scheme, we have also applied precise GC to parts of the Linux kernel. In this environment, the benefits of precise GC are less clear, for two reasons. First, the user/kernel interface of an OS tends to be much narrower than a high-level language interface, directly preventing user-mode code from causing space leaks in the kernel. Second, low-level concurrency and hardware-level interactions can create problems for a precise collector. While precise GC is less obviously desirable in this setting, our experiments suggest that it is at least viable.

References

- [1] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. Technical report, University of Wisconsin-Madison, December 1993.
- [3] J. Baker, A. Cunei, F. Pizlo, and J. Vitek. Accurate garbage collection in uncooperative environments with lazy pointer stacks. In *Proc. International Conference on Compiler Construction*, 2007.
- [4] S. J. Beaty. A technique for tracing memory leaks in C++. *SIGPLAN OOPS Mess.*, 5(3):17–26, June 1994.
- [5] Berkeley. CIL. <http://manju.cs.berkeley.edu/cil/>, 2005.
- [6] B. N. Bershad, S. Savage, P. Paradyak, E. G. Sizer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proc. ACM Symp. on Operating Systems Principles*, pages 267–284, Dec. 1995.
- [7] H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 93–100, 2002.
- [8] H.-J. Boehm. Space efficient conservative garbage collection. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 197–206, 2003.
- [9] H. Bromley. *Lisp Lore: A Guide to Programming the Lisp Machine*. Kluwer Academic Publishers, 1986.
- [10] W. D. Clinger. Proper tail recursion and space efficiency. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 174–185, 1998.
- [11] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proc. Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 69–80, 2003.
- [12] C. Ding and Y. Zhong. Compiler-directed run-time monitoring of program data access. In *Proc. of the Workshop on Memory System Performance*, pages 1–12. ACM Press, 2002.
- [13] N. Dor, M. Rodeh, and M. Sagiv. Detecting memory errors via static pointer analysis (preliminary experience). In *Proc. Workshop on Program Analysis for Software Tools and Engineering*, pages 27–34. ACM Press, 1998.
- [14] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. ACM Symp. on Operating Systems Design and Implementation*, 2000.
- [15] D. Evans. Static detection of dynamic memory errors. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 44–53. ACM Press, 1996.
- [16] B. Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 165–176, 1991.
- [17] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 168–181. ACM Press, 2003.
- [18] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proc. ACM International Symposium on Memory Management*, pages 150–156. ACM Press, 2002.
- [19] M. Hirzel, A. Diwan, and J. Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Trans. Program. Lang. Syst.*, 24(6):593–624, 2002.
- [20] HnxGC. <http://hnxgc.harnixworld.com/>.
- [21] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.
- [22] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. USENIX Annual Technical Conference*, pages 275–288, 2002.
- [23] D.-H. Jung, S.-H. Bae, J. Lee, S.-M. Moon, and J. Park. Supporting precise garbage collection in Java bytecode-to-C ahead-of-time compiler for embedded systems. In *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 35–42, 2006.
- [24] Microsoft. *Managed Extensions for C++ Programming*, 2004.
- [25] G. C. Necula, S. McPeak, and W. Weimer. Cured: type-safe retrofitting of legacy code. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 128–139, 2002.
- [26] S. M. Pike, B. W. Weide, and J. E. Hollingsworth. Checkmate: cornering C++ dynamic memory errors with checked pointers. In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pages 352–356. ACM Press, 2000.
- [27] G. Rodriguez-Rivera, M. Spertus, and C. Fiterman. A non-fragmenting non-moving, garbage collector. In *Proc. ACM International Symposium on Memory Management*, pages 79–85. ACM Press, 1998.
- [28] Ruby, 2008. <http://www.ruby-lang.org/>.
- [29] J. D. Slingwine and P. E. McKenney, 1995. Patent No. 5,442,758.
- [30] J. Sparud. Fixing some space leaks without a garbage collector. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 117–122. ACM Press, 1993.
- [31] Sun Microsystems, Inc. JavaOS: A standalone Java environment, 1997. <http://www.javasoft.com/products/javaos/-javaos.white.html>.
- [32] User Mode Linux, 2008. <http://user-mode-linux.sourceforge.net/>.
- [33] A. Wick and M. Flatt. Memory accounting without partitions. In *Proc. ACM International Symposium on Memory Management*, 2004.