C-strider: Type-Aware Heap Traversal for C

Karla Saur, Michael Hicks, and Jeffrey S. Foster

Dept. of Computer Science, University of Maryland, A.V. Williams Building, College Park, MD 20742

SUMMARY

Researchers have proposed many tools and techniques that work by traversing the heap, including checkpointing systems, heap profilers, heap assertion checkers, and dynamic software updating systems. Yet building a heap traversal for C remains difficult, and to our knowledge extant services have used their own application-specific traversals. This paper presents C-strider, a framework for writing C heap traversals and transformations. Writing a basic C-strider service requires implementing only four callbacks; C-strider then generates a program-specific traversal that invokes the callbacks as each heap location is visited. Critically, C-strider is type aware—it tracks types as it walks the heap, so every callback is supplied with the exact type of the associated location. We used C-strider to implement heap serialization, dynamic software updating, heap checking, and profiling, and then applied the resulting traversals to several programs. We found C-strider requires little programmer effort, and the resulting services are efficient and effective. Copyright © 2014 John Wiley & Sons, Ltd.

KEY WORDS: programming tools; dynamic analysis; run-time systems; heap traversal

1. INTRODUCTION

Researchers have developed many compelling application services that work by traversing the heap pointer graph of a program, such as checkpointing [1, 2], profiling [3], dynamic software updating [4], OS kernel integrity monitoring [5], and data-structure assertion checking [6, 7, 8, 9, 10] and repair [11]. When implemented for programs written in a language like Java, these services can piggyback on a tracing garbage collector. But supporting programs in C, which lacks both a garbage collector and the necessary run-time type information, has to date required a heroic effort to build a one-off C heap traversal.

In this paper, we present C-strider, a general framework for writing services that traverse and/or transform a C program’s heap. Figure 1 depicts C-strider’s architecture. Given an input C program prog.c, C-strider generates a program-specific traversal (prog_stride.c) that walks the heap starting from any location given its type. As heap locations are visited, the traversal invokes one of a small set of callbacks supplied by the user (in prog_specific.c). In almost every case, these callbacks simply delegate to program-independent service code (in service.c), e.g., to implement serialization or dynamic updating. The service code in turn invokes the C-strider run-time library, libstride.a, for services such as bookkeeping to ensure locations are visited just once.

Thus, C-strider provides a clean separation between two reusable pieces of code: the program-specific traversal (which can be used for many services in the same program) and the service-specific code (which can be used by many different programs); but it can be arbitrarily customized...
for particular program/service pairs, if needed. C-strider’s API has been designed to be easy to use: Section 2 presents a complete, portable serialization and deserialization service (e.g., for checkpointing).

A key feature of C-strider is that it is type aware—C-strider knows the type $t$ of each heap location visited, and passes a representation of $t$ to the callback functions, which can vary their behavior depending on the type. C-strider’s type information is exact, which is critical for many services. For example, a serialization service must know the precise size of objects, and must know precisely whether an object is a pointer and what it points to. As such, conservative garbage collection [12] is not a suitable starting place because its identification of pointers is approximate, and it does not know the types of non-pointer values. We observe that in practice, there is nearly always a statically well-typed pointer/struct chain to any location. Accordingly, C-strider analyzes the program and statically generates a traversal that follows well-typed paths to reach most memory, invoking callbacks and passing them type information along the way. The traversal employs a work queue serviced by one or more threads. (Section 3 describes C-strider’s traversal.)

However, most C programs lack some type information needed to describe the whole heap. For example, C-strider cannot traverse unions or void* pointers without knowing which arm of the union is valid or what the actual pointed-to type is, respectively. C-strider allows the programmer to fill in such missing type information in two ways. First, C-strider includes programmer annotations, in the style of Deputy [13] and Kitsune [4], to provide array length information and precise types for void* pointers in generic data structures. Second, C-strider permits programmers to customize the traversal by considering an object’s type specifically in the service code callbacks. For example, a callback can observe when it has reached a union embedded in a struct and use the values of other fields in that struct to determine how to interpret the union value. Our experience is that program-specific code in callbacks can be cleanly separated from program-agnostic service code. (Section 4 describes the annotations and other forms of customization.)

We used C-strider to implement four traversal-based services (details in Section 5), and used them with three different programs, memcached (a data caching service), redis (a key-value store), and snort (an intrusion detection system).

**Serialization and deserialization** This service, suitable for checkpointing, is implemented in just over 60 lines of code (LOC), and is carefully designed to be robust against changes to the program at load time, i.e., symbol relocations due to address-space layout randomization. Checkpointing is relatively fast; e.g., we could serialize and deserialize a memcached heap with 30,000 key-value pairs under 100ms.
Dynamic Software Updating

We modified Kitsune to implement its state transformation component using C-strider, requiring only 24 LOC. Taking advantage of C-strider’s ability to perform multi-threaded traversal, we could dynamically update redis with a heap of 100,000 key-value pairs in 38% of the time required by Kitsune.

Heap profiling

We developed a profiling service that counts the number and amount of memory consumed by objects of different types. The service required 65 LOC and generates an itemized log, which we used to better understand memory usage in our example programs.

Heap assertion checking

C-strider can be used to implement check heap assertions in the style of GC assertions [6]. We implemented several simple assertions such as ensuring that linked-lists are well formed, timestamps were not in the future, and that enum fields are valid.

Each application required some service-agnostic customization to provide missing type information, and in some cases some service-specific handling, usually to optimize performance. In all cases, traversal customizations required roughly 20 LOC, while type annotations depended on program size; e.g., for snort 143 annotations were required across its 215 KLOC, while for memcached only 6 annotations were required across its 4 KLOC.

C-strider has been carefully designed to implement heap traversal as a library employing entirely standard features of C. In particular, C-strider does not modify data structure representations (which could break important assumptions about object size and layout) and does not require custom compilation (which could inhibit compiler optimizations). C-strider also deliberately keeps its type annotation language simple for the common cases, with an escape hatch to C for reasoning about more complex type invariants (e.g., tagged unions); in our experience, C programs always have unusual, exceptional cases not captured by any sensible annotation system. Finally, C-strider’s traversal itself is easy to understand—as we will see in Section 3, the entire traversal is simple, written in a few tens of lines of code. Thus, we believe C-strider is accessible to a broad audience and adaptable to many uses.

To our knowledge, C-strider is the first general-purpose, type-aware heap traversal framework for C. We believe C-strider will prove useful for many applications in addition to the ones explored in this paper, and we plan to release C-strider under an open source license.

2. SERVICE PROGRAMMING IN C-STRIDER

This section focuses on how to implement a service with C-strider. As a running example, we use a serialization and deserialization service.

2.1. Implementing serialization

To implement a C-strider service, the programmer writes four callbacks that are invoked as the heap is traversed (as described in Section 3). By and large the service code is reusable across different programs, though sometimes, due to C’s weak type system, small customizations are needed, as discussed in Section 4. The type signatures of the callback functions are given in Figure 2. Each function has the same three arguments: argument in points to the program data being visited, argument out points to the result, and argument typ specifies the type of the argument.

Figure 2. Basic C-strider service API.
which is of type \( t \), the second argument. If this is a transformation service, argument out points to replacement memory, and writing out specifies how in’s data should be transformed (if at all). The out parameter is not needed for serialization, but we will use it for developing deserialization, below. We call services that do not use out traversal services. (For these the traversal code sets in and out to the same value.)

The first callback, \texttt{perfaction\_prim}, handles data of primitive type (e.g., \texttt{int}, \texttt{double}). The traversal calls this function for each global variable and each struct/union field that is a primitive type, indicating the specific primitive with \( \texttt{typ} \ t \). The code for serializing primitives using this function is given at the top of Figure 3, which does the obvious thing (here \( \texttt{ser\_fp} \) is the serialization file handle).

The second callback, \texttt{perfaction\_struct}, is called with the address of every \texttt{struct} reached during the traversal; the particular \texttt{struct} can be determined by examining \( t \). Oftentimes, as is the case here, this function does nothing other than return 1, indicating the traversal should continue by visiting each of the struct’s fields. Alternatively, the callback can return 0 to indicate traversal should not continue automatically. In this case, the function can call the \texttt{visit} library function, shown with the full C-strider API in Section 2.4, to selectively visit fields, as needed.

The third callback, \texttt{perfaction\_ptr}, is invoked for each pointer when it is first reached. Here the type of in is \texttt{void **}, rather than \texttt{void *}, because it is the address of a pointer. The code for serializing pointers is given in the middle of Figure 3. First, the code checks whether the pointer is null; if so, it writes null to the file. Otherwise, the code calls C-strider’s \texttt{lookup\_addr} function to check whether the pointer is to a global variable, i.e., one that has a symbol name. If a symbol name is returned, the code first writes 2 to the file as a token, which we assume is not a legal pointer, followed by the symbol name. Otherwise there is no symbol, so the code writes the pointer value itself. If the pointer is to a string, as determined by its type, the code writes the contents of the string; note that if we were to dereference the pointer and continue as usual, we would only write the first character of the string. Finally, if traversal should continue to the pointer contents, then we return 1; else we return 0 to prevent further traversal though this pointer.
The last callback, perfaction_ptr_mapped, is called for each pointer that has already been visited during the traversal, e.g., if the heap contains cycles. C-strider maintains a map from each visited *in* pointer to its *out* counterpart; this mapping is set when we return from perfaction_ptr, mapping the (original) value of *in* to the final value of *out*. When the traversal code considers a pointer to visit, it checks whether that pointer appears in the mapping table. If so, it calls perfaction_ptr_mapped; otherwise it calls perfaction_ptr. In the case of serialization, the perfaction_ptr_mapped function is quite similar to the perfaction_ptr function, except that it knows the pointer cannot be null, so only the else case of the above code is needed (but the code for writing strings is elided, since the string has already been written). No further traversal is needed for a mapped pointer, so nothing is (ever) returned.

**Example.** To see how all of this works, consider the code in Figure 4 which is assumed to be part of a larger program. Here we have defined a type dlist of doubly linked lists of integers, and have one global variable head that points to the head of the list, and another, phead, that points to head.

When the program reaches a programmer designated position (cf. the checkpoint() function in Figure 6), it will invoke serialization to checkpoint its state. Suppose, for our example, that when serialization begins the doubly linked list has two elements in it, at addresses a1 and a2. In the traversal, we will need to refer to various typ values. As we saw earlier some typ values (those corresponding to types statically present in the code) are predefined by C-strider. For example, TYPE_INT, TYPE_STRUCT_DLIST, and TYPE_PTR_STRUCT_DLIST are variables containing the types for int, struct dlist, and struct dlist *, respectively. For brevity, in what follows we refer to these as t_int, t_dlist, and t_dlist*, respectively. We also elide the out arguments, which are the same as in all cases (and are never used for serialization).

The traversal begins at the global variables. First, the variable head will be visited, which is a pointer to the first element of the list. As such, the traversal will call perfaction_ptr (&head, t_dlist), in Figure 3. Since the list is non-empty, this pointer is not NULL and we take the else branch at line 6. In the next line we see whether *in* (i.e., the contents of head) is a global variable; it is not—it is the pointer a1 to the first element of the list. As such we take the else branch on line 11. At this point we serialize the pointer itself, i.e., a1, on line 12, and then return 1 on line 15. The traversal then adds a1 to its mapping table (mapped to itself).

Next up is the first list element (pointed to by a1), so perfaction_struct(a1, TYPE_STRUCT_DLIST) is called. For serialization, this function returns 1 indicating that all fields should be visited. perfaction_ptr(&a1→next, t_dlist), is thus called which will follow the same steps as above to serialize the pointer to the next list element, a2; its target will also be marked for traversal and a2 will be added to the mapping table. Next the traversal considers a1→prev, but since this pointer is NULL, it will write and return 0 indicating that the pointer should not be traversed further (lines 5 and 17). Finally the traversal considers a1→x, calling perfaction_prim(&a1→x, t_int), and thus serializes the main value of the list element.

The traversal now considers a2, calling perfaction_struct which then prompts the traversal to visit each field. First visiting a2→next, it finds it null and so perfaction_ptr just writes 0. Then it visits a2→prev whose contents are a1. C-strider knows it has been visited already (it is in its mapping table), so perfaction_ptr_mapped(&a2→prev, t_dlist) is called and this function simply writes a1. Finally, &a2→x is serialized by calling perfaction_prim.

The traversal now turns to the other global variable and invokes perfaction_ptr(&phead,...). In this case execution will reach line 8 and will discover that phead’s contents are a pointer that

```c
27 struct dlist { struct dlist *next, *prev; int x; }
28 struct dlist *head = NULL;
29 struct dlist **phead = &head;
```

Figure 4. Example code to serialize/deserialize.
#### (a) Deserializing primitive values and pointers.

```c
void perfection_prim(void *in, typ t, void *out) {
    fread(out, sz, 1, ser_fp);
}

void * perfection_ptr (void **in, typ t, void **out) {
    void *ptr, *tgt;
    fread(&ptr, sizeof(void*), 1, ser_fp);
    if (ptr == (void*)2) { // pointer to a symbol
        char * symbol;
        deserialize_string (&symbol);
        *out = lookup_key(symbol);
        free(symbol);
    } else if (ptr == NULL) // null pointer
        *out = 0;
    else if ((tgt = find_mapping(ptr))) // seen it already
        *out = tgt;
    else if (typ == TYPE_POINTER_CHAR) {
        // a string
        deserialize_string ((char **)tgt);
        *out = tgt;
        add_mapping(ptr, tgt);
    } else {
        int sz_new = get_size(get_ptrtype(t));
        tgt = malloc(sz_new);
        *out = tgt;
        add_mapping(ptr, tgt);
        visit (tgt, get_ptrtype(t), tgt);
    }
    return 0;
}
```

#### (b) Helper functions.

```c
void deserialize_string (char **str) {
    int len;
    fread(&len, sizeof(int), 1, ser_fp);
    *str = (char*)malloc(len);
    fread(*str, len, 1, ser_fp);
}
```

Figure 5. Deserialization service functions.

对应于一个符号，即“head”。由于其写入标记值2，然后是名称“head”。它返回0，因为不再需要进一步的遍历。

### 2.2. Implementing deserialization

While serialization only reads the heap, deserialization modifies the heap during traversal: it begins by writing global variables and then continues to initialize the heap as it processes the serialization file. Thus, deserialization is what we call a transformation service. This service is a bit unusual because it traverses the heap as it creates it from reading data from the serialization; thus we will see that it must call API functions that add mappings and direct the traversal, whereas for serialization these tasks were handled automatically.

Figure 5a shows the deserialization code, which is symmetric to serialization. For primitives, the code determines the given type’s size and reads in that many bytes. For (non-mapped) pointers, `perfection_ptr` starts by reading the pointer itself from the file. What it reads in will be the actual address from the checkpointed program; but the address space of the current run of the program
may be different. Thus, the code first checks whether the pointer is 2; if so, it points to a symbol, whose name is then read and looked up to find its current address, which is assigned to \*out.

Otherwise, if the pointer is null, then null is written to \*out. Or, if this particular pointer was seen before—i.e., it was in the mapping table—then \*out is assigned to the value from the map. In other words, deserialization uses the mapping table to map addresses from the serialized file’s address space to those in the program’s address space. Otherwise, the code starting on line 46 handles a non-symbol, non-null pointer that has not been seen before. If it is a string, then the code reads in that string, and then adds a mapping to the mapping table between ptr (the address in the file) and tgt (the address of the same memory in the program). For non-strings, the code extracts the target type of the pointer and computes its size (if the pointer is to an array then the size of the target type will cover the entire array). Then it allocates memory for the target object, assigns the allocated pointer to \*out, and sets up the mapping. Finally, the code calls visit to manually direct to the traversal to the newly allocated memory; thus, the next read from the file will write into that memory. In all cases the perfaction_ptr function returns 0, since either no subsequent traversal is needed, or the traversal was manually specified.

For deserialization, perfaction_struct (not shown) always returns 1, i.e., the traversal reads the fields as usual in order. The perfaction_ptr_mapped function (also not shown) will never be called, because translation of pointers from the serialization file always happens through manual lookups of the table on line 44; if a pointer from the serialization file is seen again in deserialization, then there is no attempt to traverse it (perfaction_ptr returns 0).

Example. Let us reconsider our example, but now from the point of deserialization; i.e., we want to deserialize the contents of a file into the program from Figure 4. The traversal starts with the global variables, processing them in the same order as when they were serialized, beginning once again with a call to perfaction_ptr (\&head, \*dlist, \&head)—for global variables the out parameter is the same as the in parameter. The code reads in the pointer value \(a_1\) from the file and then works its way down to line 51, which allocates space for a new dlist element (e.g., at address \(a_1\)) and writes the pointer \(a_1'\) to the contents of head via \*out. Then it adds the mapping from \(a_1\) (the file’s address) to \(a_1'\) to the mapping table, and indicates on line 56 the traversal should continue with \(a_1'\). Notice that the function returns 0, which indicates that \*in should not be followed as would be the default; this is because in is not actually being used, as the data being considered are coming from the serialization file. The traversal continues with \(a_1'\), whose type is t_dlist (i.e., the result of get_ptrtype(t_dlist*)), which precipitates a call to perfaction_struct. This function just indicates that each of the fields of the struct should be visited.

Next, the traversal will call perfaction_ptr (\&\(a_1\) → next, t_dlist*, \&\(a_1\) → next), which follows the same path as for \&head. The process continues, allocating the second list element at \(a_2\) and adding a mapping from \(a_2\) to \(a_2'\) in the mapping table. Eventually we will reach \(a_2'\) → prev, which points to \(a_2'\). Whereas in serialization this results in a call to perfaction_ptr_mapped, for deserialization perfaction_ptr will be called instead, because the mapping table maps addresses from the file to current memory (e.g., \(a_1\) to \(a_1'\)), but the traversal is considering current memory (here, \(a_1'\)). The call to perfaction_ptr will end up at line 44 to set the pointer appropriately, and will return 0, saying this pointer should not be traversed further. Finally, phead is considered by perfaction_ptr, which ends up at line 36 since the code will read in the sentinel 2. As such it reads the symbol name “head” from the file, and then looks up that symbol to find the corresponding address, writing that address to \*out.

2.3. Using serialization for checkpointing

We briefly discuss how a program would use the (de)serialization service we just developed to perform periodic checkpointing. Suppose we have the program in Figure 6 which begins with the global variable declarations from Figure 4. When this program starts up, it would check whether the user has requested a restart at a checkpoint by calling do_serialize. If so, this code will call the entry point for deserialization and return 1, as such skipping the true branch of the conditional in main. This works by registering the deserialization perfaction_* callbacks with the C-strider.
int main(int argc, char **argv){
    if (!do_deserialize(argc, argv)) {
        /* initialize the doubly linked list */
        for (int i = 0; i < 5; i++) {
            struct list * n = calloc(1, sizeof(struct list));
            /* ... and code to set prev and next */
            if (head == NULL) { head = n; }
        }
        /* long–running loop for rest of the program... */
        while(1) {
            if (...) checkpoint();
            ... 
        }
    }
}

Figure 6. Example program employing serialization.

traversal service, and then starting the traversal, which when completed will have initialized the heap from the checkpoint. While the program is running it can call checkpoint to serialize the current state; this time we register the serialization callbacks before invoking the traversal. Note that checkpointing ignores local variables, so this code assumes all interesting state is global.

Stepping back, let us assess what we have accomplished. C-strider has allowed us to write a general-purpose serialization service for C programs in just a just over 60 lines of code. The serialization service developer writes his code once, and a user can use C-strider to generate a traversal for his program that serializes its data. And, of course, we can use C-strider to implement many services, not just serialization, as discussed in the rest of the paper.

2.4. Full C-strider API

We conclude this section by touching on the remainder of the API, given in full in Figure 7, that C-strider provides to service programmers writing their perfection code. More discussion about how these functions can be used to customize the generated traversal will be presented in the next section.

The first set of functions launches the traversal. C-strider must be initialized by calling init. Then the user either calls visit_all to traverse data reachable starting from the set of all global variables in the program, or calls visit to traverse data only from particular heap locations. The latter is also useful when the standard traversal cannot proceed as usual; e.g., in the deserialization code above, we used this to visit newly malloc’d memory at a given type.

The next set of functions perform lookups in the C-strider symbol table, mapping from symbol names to addresses or vice-versa.

The next two functions manipulate the mapping table used in transformation services. We saw both of these functions above, and they behave in the obvious way (note that find_mapping returns NULL when no mapping is present).

The next set of functions (beginning with get_ and is_ ) query attributes of run-time types. The particular set of queries shown here was derived from our experience building several services (Section 5); other applications could potentially require other accessors, which are easy to add.

Finally, the last set of functions (beginning with mktyp_) create new type representations from existing ones, in particular to make a type for a pointer to T given a T, similarly for an array type, and finally to instantiate a generic type with its type arguments. mktype_array is called once each time a (run-time) array is encountered during traversal. This allows a single syntactic array type to be instantiated at different lengths for different runtime array values. Generic types are a C-strider extension, discussed more in Section 4.1.

Notice that, since types are available at run-time and can be fully queried, it is not strictly necessary to have four perfection callbacks: in theory, perfection_prim, _struct, and _ptr could be combined into a single callback that would just test the type and behave appropriately. However, we
have found this particular grouping to be useful separation, because we often want to treat each of
those groups independently (e.g., as in Section 2).

3. TYPE-AWARE TRAVERSAL

Now that we have seen C-strider from the service developer’s perspective, we can discuss the
generated traversal.

Recall from Section 2 that, in C-strider, each visited location has an associated type, which is an
instance of typ. Internally, a typ t is an integer key into a type table that maps types to needed type
information, including its size in bytes, number of elements (for arrays), the pointed-to type (for
pointers and arrays), and, for a generic type, the base type and the type arguments this type has been
instantiated with (if applicable); Section 4 says more about generics.

C-strider creates one unique key for each type appearing statically in the program text and assigns
it to a variable with a predictable name, e.g., TYPE_POINTER_CHAR from Figure 3; the developer
can then use these variables in the perfaction functions to refer to types as needed. C-strider also
assigns a unique key to each type alias, e.g., typedef int size_t would create a new type TYPE_size_t
with the same associated information as TYPE_INT. Distinguishing type aliases like this is useful
for certain applications. For example, we found that Redis includes a string type that is aliased to
char *, but is actually a pointer to the middle of a data structure; traversal needs to visit the structure
as a whole, and not just the string. Finally, new types created with mktyp functions are converted
to integer keys using a hash function, and those keys are either added dynamically to the type table
(if the type is new) or reused (if the type was already in the type table).

The overall traversal is orchestrated by two pieces of code: the visit function, shown in Figure 8,
which tests its typ argument, calls the appropriate perfaction function, and then continues traversal
(if appropriate); and a task queue that maintains a list of pointers to structs whose fields need to
be visited. One task queue per traversal thread is created when the user calls init. Next, the user

void visit (void *in, typ t, void *out) {
    if (is_prim(t)) {
        perfaction_prim(in, t, out);
    } else if (is_array(t)) {
        /* call visit on each element of the array */
    } else if (is_ptr(t)) {
        if (!in) return;
        void *lookup;
        if ((lookup = find_mapping(*(void **)in))) {
            perfaction_ptr_mapped(in, t, &lookup);
            *(void **)out = lookup;
        } else {
            int retc = perfaction_ptr(in, t, out);
            add_mapping(*(void **)in, *(void **)out);
            if (!retc || is_funptr(t)) return;
            typ t_p = get_ptrtype(t);
            visit (*(void **)in, t_p, *(void **)out);
        }
    } else if (is_struct(t)) {
        if (perfac tion_struct(in, t, out))
            enqueue(in, t, out);
    }
}

Figure 8. Traversing using type information.

calls visit to start visiting nodes and enqueuing tasks, which are loaded round-robin into the task queues. After the user has called visit on all desired roots, the user calls finish, and at this time the main thread and helper threads (if multi-threaded) process all remaining items in the queues until they are empty. Then C-strider tears down any traversal infrastructure, the helper threads exit, and then finish returns. (More on this below).

The body of visit is straightforward. For primitive types (ints, chars, etc) and other non-standard terminal primitive types (mutex_t, time_t, size_t ), the traversal calls perfaction_prim on line 82. If typ is an array type, the code recursively visits all of the array elements (code not shown). To do this, it extracts the length of the array from its type representation, and iterates over the array. Finally, if typ is a pointer type, the code returns immediately if the pointer is null (line 86). Otherwise, there are two cases, depending whether the pointer has been visited before (line 88).

If the pointer has been visited, the code calls perfaction_ptr(mapped, passing the address from the mapping as its last (out) parameter. The code passes &lookup because perfaction_ptr_mapped takes a void **, i.e., a pointer to a location containing the pointer of interest. The code then writes the mapped-to value from lookup to *out. In the default case, this updates *out with the value from the map. Perfaction_ptr_mapped may also have updated the value in lookup, in which case the assignment writes the new value.

If the pointer has not been visited, then the code calls perfaction_ptr and updates the mapping to record that the pointer has been visited. Then either traversal stops (if perfaction_ptr so indicated, or if t is a function pointer), or it continues at the pointed-to type (lines 96–97).

Finally, if the type is a struct, the traversal calls perfaction_struct. If that function indicates the traversal should continue, the code calls enqueue to indicate t’s fields should be visited in the future, and then returns.

Each task on the task queue includes the same parameters as visit, i.e., in, the struct type t, and out. When a task is dequeued, the code switches based on t and calls a custom visit function that C-strider generates for each struct in the program. For example, Figure 9 shows the traversal code for struct dlist. (Code slightly simplified for clarity.) This code simply calls visit on each of the pairs of fields of in and out and then returns.

The handling of enqueued tasks depends on the mode C-strider is used in. In single-threaded mode, when finish is called, the main thread repeatedly pulls tasks off the queue and calls visit,
Figure 9. Generated traversal code for \texttt{struct dlist}.

until the queue is empty. Using a queue enables C-strider to traverse cyclic data structures (which in C must always go through a \texttt{struct}) without requiring a deep stack. In parallel C-strider, tasks are pulled off the queue by worker threads that run concurrently, potentially enabling performance improvements (Section 3.2); the call to \texttt{finish} simply waits for the workers to complete.

Finally, notice that the \texttt{visit} function skips unions; we made this choice because C has no standardized mechanism for determining which arm of a union is active. Thus, for these cases the developer has to customize the traversal with program-specific code (Section 4).

3.1. Roots of the traversal

The \texttt{visit} function from Figure 8 can be called on any location to start traversal from that point onward. For example, using the code from Section 2 this could be used to serialize a particular data structure. The traversal can also be called on multiple roots in sequence, which conceptually results in a single traversal that completes when \texttt{finish} is called.

Certain applications, however, need to traverse essentially the entire heap. For example, full heap traversal is needed for dynamic software updating and for heap profiling. To support these cases, C-strider generates a custom traversal function for each global variable in the program (exactly analogous to the functions for visiting \texttt{struct} types), and generates a function \texttt{visit all} that calls all those functions.

Additionally, we provide API functions to register and deregister roots for a traversal, which can be used to traverse any local variables. The registration function takes a pointer to the local, along with its type. Then, when \texttt{visit all} is called, this root is traversed as well. Before returning from the function where the local variables are registered, the user must deregister the local variables.

3.2. Parallelized traversal

To speed up the traversal process, C-strider supports multi-threaded traversal. In this mode, C-strider launches $n-1$ worker threads, where $n$ is the number of processors, in addition to the main thread. The threads consume tasks that are generated by \texttt{enqueue}, starting with tasks for the roots, produced by C-strider’s main thread.

Work stealing. C-strider uses a work-stealing scheduler: Each thread has a local queue, and when it runs out of tasks it attempts to steal one from another queue [14]. Each queue is implemented as a large pre-allocated array and a single lock used for enqueue and dequeue operations. Each queue also maintains an exact number (using the lock) of in-flight tasks currently being processed. This is because even if all queues appear to be empty, if a thread is currently processing a task, it may create a subsequent enqueue from that task. Therefore, we are not done traversing until all queues are empty and no tasks are in-flight.

Some tasks require more work than others, so C-strider attempts to avoid task stealing as much as possible by keeping the queues balanced. C-strider keeps a rough estimate of the fullest (high) queue and the least full (low) queue. The high and low counts are estimated because, for performance, they are maintained separately from the queues without locks. When a thread calls \texttt{enqueue} with a task, it attempts to place the task on the queue with the estimated lowest task count using
pthread_mutex_lock. If the queue with the lowest count is locked, rather than wait the thread moves on to the next queue, continuing to try new queues until it succeeds. For dequeuing, a thread will drain its local queue first before attempting queue stealing. Then, the same strategy with pthread_mutex_trylock is used for dequeuing from the highest queue when the thread’s local queue is empty. This practice greatly minimizes lock contention as task stealing occurs primarily at the beginning and end of the traversal. We used mutrace [15] to ensure that our algorithm minimized mutex contention in the queues.

Another cause of unbalanced queues and hence work stealing is arrays containing a large number of elements. When we encounter an array with more entries than \( n \) traversal threads, we divide the array into \( n \) segments and split the array among the queues.

Recording visited memory. In a multi-threaded traversal, the pointer map needs to be thread-safe. Using a single lock to protect it would introduce significant contention, so we use a hash table implementation with bucket-by-bucket locking.

When running heap traversals on our subject programs, we found that resizing the number of buckets in the hash table has a very large cost, because it requires blocking all threads. Instead, we fixed the number of buckets in the hash table, which eliminates the cost for resizing but requires knowing a good bucket size in advance. In our case, we found that using \( 2^{15} \) buckets worked well for all our subject programs and experiments. Users can adjust this number as necessary, or switch to an expandable hash if performance is less critical.

Ending the traversal. In single-threaded heap traversal, traversal is complete when the main thread has an empty work queue. In multi-threaded traversal, traversal is complete when all queues are empty and no thread is currently doing work. Thus, C-strider only exits the traversal when the number of in-flight tasks and the queue sizes reach zero. When traversal is complete, each thread frees an equal share of the shared traversal-related hash map. When all worker threads have exited their dequeue loops after all queues are empty and no tasks are in-flight, traversal is complete.

4. CUSTOMIZING THE TRAVERSAL

For many C programs, the standard C types do not provide quite enough information to support type-accurate traversal. To support such programs, C-strider lets the programmer customize the traversal in two ways: adding type annotations to the program (Section 4.1), and writing program-specific code in perfection functions (Section 4.2).

4.1. Type annotations

C’s type system is not sufficiently expressive to describe many common programming idioms. For example, the type int * could describe a pointer to a single integer, or a pointer to an array of integers. C-strider permits programmers to express additional information as type annotations to remove some of the ambiguity. These annotations are borrowed from Kitsune [4] and inspired by Deputy [13]. Currently, C-strider supports two kinds of type annotations that encode the most common missing type information: lengths of arrays, and types of void *’s used in generic data structures. These annotations can decorate types of either struct fields or global variables, in which case they modify the generated traversal function for that struct or global, respectively. We illustrate the annotations by example.

Type annotations for arrays C-strider includes annotations T_PTRARRAY(S) and T_ARRAY(S) to decorate a pointer or array, respectively, with a length S, which may be a constant integer or an expression of the form self . f, where f is a field at the same level of the current struct. For example, Figure 10 shows an annotated struct type declaration and its corresponding traversal code. The annotation states that arr_head is a pointer to an array whose length is contained in the field arrlen.
typedef struct _dlist_arr {
    int arrlen;
    struct dlist ** T_PTRARRAY(self.arrlen) arr_head;
} dlist_arr;

void _visit_struct_dlist_arr (void *in, type t, void *out) {
    struct dlist_arr *in_a = in;
    struct dlist_arr *out_a = out;
    visit (&in_a->arrlen, TYPE_INT, &out_a->arrlen);
    visit (&in_a->arr_head,
        mktyp_ptr(mktyp_arr(in->arrlen,TYPE_STRUCT_dlist_PTR)),
        out_a->arr_head);
}

Figure 10. Length annotation and generated traversal code.

struct list {
    // class List<T>
    void T_VAR(@t) *val;
    // T val;
    struct T_INST(@t) *next;
    // List<T> next;
} T_FORALL(@t);

void _visit_struct_list (void *in, type t, void *out) {
    struct list *in_l = in;
    struct list *out_l = out;
    typ *args = get_generic_args(t);
    int num_args = get_num_gen_args(t);
    assert(num_args == 1);
    typ t0 = args[0];
    visit (&in_l->val, t0, &out_l->val);
    visit (&in_l->next,
        mktyp_instantiate(TYPE_STRUCT_list_PTR,num_args, args),
        &out_l->next);
}

Figure 11. Generic annotation and generated traversal code.

of the same structure (self). Notice that line 120 of the generated traversal code calls mktyp_arr to
generate an array type of the appropriate length, and then wraps it in a pointer type.

Type annotations for generics. C-strider includes several annotations to type void *’s in generic
data structures. Figure 11 shows a generic linked-list data structure and its corresponding traversal
code. Here, the struct list type is parameterized by type variable @t, introduced with T_FORALL.
The type variable is used with T_VAR to provide the actual type of val. Then the code uses T_INST
to instantiate the type of next. For comparison, the Java generic linked list equivalent is shown in
comments in the example. In the traversal code, the argument t is an instantiation of the generic
type. The calls on lines 130 and 131 get an array with the instantiated arguments and the length of
that array, respectively. The code then binds t0 to the first element of the array, i.e., whatever @t
is instantiated as. That type is used to visit val, and then next is visited at the type of struct list
instantiated with the same arguments (line 137).

4.2. Customization in perfection functions

In some cases, type annotations alone are insufficient to guide the traversal. For example, consider
the type at the top of Figure 12, which defines a struct whose u field is either int or a char *,
struct tagged_union {
    int tag;
    union { int x; char *c; } u; /* selected by tag */
};

int perfaction_struct (void *in, typ t, void *out) {
    /* Service-agnostic, Program-specific */
    if (type == TYPE_STRUCT_tagged_union) {
        struct tagged_union *in_u = in;
        struct tagged_union *out_u = out;
        visit (&in_u->tag, TYPE_INT, &out_u->tag);
        if (in_u->tag)
            visit (&in_u->u.x, TYPE_INT, &out_u->u.x);
        else
            visit (&in_u->u.c, TYPE_CHAR_PTR, &out_u->u.c);
        return 0;
    } else if (current_service == SERIALIZE) {
        /* Service-specific, Program-specific */
        if (t == TYPE)... { ... } /* Service-agnostic */
        else return serial_perfaction_struct (in, t, out);
    } else if (current_service == DESERIALIZE) {
        ...}
    return 1;
}

Figure 12. Customizing traversal in perfaction functions.

depending the tag field. To select the correct field of u to visit, C-strider needs to know how to interpret tag, but there is no standardized way to do this in C.

We have found that to solve this problem, we create a program-specific perfaction function that either handles traversal specially for cases where it’s needed; performs program- and service-specific actions if necessary; or delegates to the service-specific action for the default case.

We can use this program-specific code across all services, so the union will be handled in a way that is agnostic to any implemented service. The bottom of Figure 12 shows this pattern for our example. Here if perfaction_struct is called on tagged_union, then we visit the tag on line 149, and then switch on the tag and call visit on the appropriate sub-field, returning 0 on line 154 to indicate the default traversal (visit all fields) should not happen.

Otherwise, for any other struct type, we check which service we are running and perform either a customized action or the default service action. For example, on line 155, if current_service is SERIALIZE, we first perform service-specific, program-specific actions, e.g., we check for a particular type (line 157) to optimize serialization for it (not shown). Otherwise, we call the default action, serial_perfaction_struct , on line 159. Similar to serialization, the code for deserialization (or any other service) performs service-specific checks, and then the default action.

Section 5.1 includes more details and examples of customizing the traversal for service specific customization.

5. APPLICATIONS AND EXPERIMENTS

We used C-strider to develop four services: serialization; state transformation for dynamic software updating; heap profiling; and heap assertion checking. We implemented each service for a subset of three programs, listed on the left of Table I along with their version numbers and sizes. Memcached is a widely used, high-performance data caching system employed by sites such as Flickr and Youtube. Redis is a key-value database used by several high-traffic services, including Instagram.
and stackoverflow. Snort is a network intrusion detection system claiming millions of downloads and nearly 400,000 registered users.

This section describes the implementation of our services, characterizes the programmer effort required to write them, and provides some performance measurements. Measurements were conducted on a 32-bit, Intel Core i5-3320M at 2.60GHz, with 4 cores and 7.5GB mem, running Ubuntu 12.04. Medians were taken over 11 trials. Since C-strider adds no overhead to normal program execution (e.g., because it does not compile the program any differently), we consider the time from when the traversal-based service is requested until it completes and normal execution resumes.

5.1. Programmer effort

Table I tabulates three kinds activities involved in using C-strider: writing service code, customizing a particular program’s traversal, and customizing the traversal in a service-specific manner. All three cases involve relatively little code, much of which is a one-time effort that can be shared across different services and/or different programs.

Writing a service. Writing a C-strider service takes relatively little effort, in terms of lines of code. The parenthesized number at the top of the last four columns of Table I count the lines of service-specific (but program-independent) code, tabulated by service (dynamic software updating, serialization, profiling, and heap assertions). Details of the implementation of each service is given in the following subsections.

Customizing a program’s traversal. While some programs can use a service out of the box, nontrivial programs will require some customization. The first step is customizing the traversal for that program, in a service-independent manner, as described in Section 4. The third column of Table I counts the type annotations we added, and the fourth column counts the lines of program-specific _ perfaction_ code we wrote. Nearly all of code for all applications was for handling _ unions_, or union-like data structures. For example, 17 LOC for _redis_ customizes the traversal of its main database item structure which contains a flag that determines the type of a _void_* field in the structure. The other 9 LOC handles a structure containing a mask that determines whether or not other fields in the structure are valid.

Customizing the services. The lower right portion of the table counts the lines of code that are program- and service-specific. For example, we wrote 10 lines of code specific to _memcached_ for dynamic software updating. We write “n/a” where we did not implement a service for that program.

We implemented the program- and service-specific code using the pattern shown in Figure 13, which excerpts part of the _ perfaction struct_ function for _memcached_. This code switches based on the current service (lines 166 and 177). For each service, it then either performs type-specific actions (lines 167 and 178) or calls the service-specific action (lines 175 and 182). The _current_service_ flag is set through the entry point to the service, e.g., the call to _do serialize_ in Figure 6 sets the service to deserialization, and the call to _checkpoint_ sets the service to serialization (both calls initiate a full traversal after setting the flag).
5.2. Heap serialization

We now turn to the different services we implemented, starting with heap serialization and deserialization suitable for implementing checkpointing. The implementation of this service was presented in Section 2 (Figures 3 and 5). In addition to those 62 lines of code, we have an additional 16 lines of code that deal with file manipulation (opening, closing, and some wrapper functions around fread and fwrite for simplicity). We must use a single-threaded traversal for this application to ensure data is written to the serialization file in a deterministic order.

While implementing serialization for memcached, we developed one interesting performance optimization, shown in Figure 13. Here on line 180 we use a macro from memcached to get the size of a flexible array member (void * end[]) contained in struct item; the length of this member is the sum of several of the item’s fields. By tailoring the traversal here, we can write the entire item to disk at once rather than traversing each byte of the flexible array separately. We also wrote a similar function for deserialization. In total, we wrote 44 additional lines of peraction. * code to optimize the traversal of memcached’s key-value database structures, as shown in the sixth column of Table I. We performed very similar changes to optimize the redis object structure for serialization, totaling 46 additional lines.

Time required for serialization. We measured the time it takes to serialize and deserialize the key/value database items of redis and memcached. (We did not serialize the rest of the program, such as statistics or connected user info, as this information would be stale between program restarts.) Figure 14 shows how performance varies with the size of the heap, in terms of number of key-value pairs. The keys and values are approximately 10B in length each. The numbers are approximate because they consist of a string appended with an incrementing integer. The parts of the heap that we traverse for serializing 30K key-value pairs contains 5592KB of allocated data structures in redis and 1477KB for memcached. We see that serialization and deserialization take nearly the same amount of time for a given program. Overall, memcached traversal is faster; the reason is the performance optimization mentioned above, which lets C-strider write the key and value to disk as a single block. In contrast, redis stores its key and value in separate structures that must be traversed separately.

Redis itself provides a serialization tool, allowing the user to load/store entries from/to a file. The Redis serialization process uses a custom iterator to skip to only populated elements of the array, serializing 20K (~10B-key,~10B-value) pairs to disk in ~13ms. This is in contrast with C-strider, which must visit every entry of the database array (2^{15} slots for 20K database entries) and maintain
the mapping table, taking ~123 ms to serialize the same 20K entries. We suspect we could customize our traversal to apply similar optimizations but have not investigated further.

5.3. State transformation

C-strider grew out of our experience with Kitsune [4], a source-to-source compiler and run-time library that lets a running C program be updated with code fixes and feature additions without shutting it down. Such dynamic software updating (DSU) services are implemented by loading the new code (compiled to a shared object) into the C program, and then transforming the existing state (i.e., heap memory) to meet the expectations of the new code. For example, in the old version of the program a `struct foo` might have two fields, while in the new version it has three. The state transformation code must find all pointers to `struct foo` objects in the program, allocate new memory for those objects, initialize retained fields to the existing values and the new field to a new value, and then redirect the pointer to the new object after freeing the old one.

State transformation in C-strider. Kitsune state transformation can be implemented as a C-strider transformation service in 24 LOC. This service traverses and modifies the program heap (thus using both the in and out parameters of the `perfaction_` functions). Additional code was required to implement other elements of DSU, of course, e.g., for loading in the new code and managing updating timing. We made one generalization to the C-strider API to support updating: we defined two variants of `lookup_addr` and `lookup_key` (cf. Figure 7), one for the old version’s symbol table, and one for the new version. We also added a function `get_out_size` to get the corresponding size of the new version of a type.

The DSU `perfaction_` functions do one of several things:

- When reaching a location in the new program that must be initialized from the old program (such as a global variable in the new program’s data segment), the code simply `memcpy` over the appropriate bytes.
- When first reaching a pointer to a block of a type that has changed size, the code allocates a new block to hold the new, differently sized data, and adds a mapping from the old block...
The program-specific actions perform the actual state changes. Referring back to our example of `struct foo` at the top of this section, we would customize `perfaction` to look at type `t`, and if it is `TYPE_STRUCT_foo` we allocate new memory, initialize it with the retained values and the new one, and then write the result to `out`.

For the programs we considered, the data representations did not change between versions, so the only program-specific DSU code we wrote simply optimized the traversal. For example, for `memcached`, Figure 13 (line 167) shows how we cut off the traversal of `struct stats` since it contains only primitives that need not be traversed individually (when it is stored in a global variable, it must still be copied to the new code’s address space).

As mentioned briefly in Section 3.1, sometimes we also need to use local variables as roots of the traversal. Thus, C-strider includes support for registering and deregistering local variables, which stores pointers to local variables during the scope of a function and provides traversal for local variables. We registered 7 local variables for `memcached`.

**Time required for a dynamic update.** We measured the time it takes to deploy a dynamic update with single-threaded and multi-threaded traversal (4 threads). We also compare against the time for the same update with Kitsune.

Figure 15 shows how update times vary with heap size measured in terms of (~10B-key,~10B-value) pairs in the database for both `redis` (version 2.0.1 to 2.0.2) and also for `memcached` (version 1.2.2 to 1.2.3), using either one or four threads. For `redis` we see that the benefit of parallelism increases with heap size, whereas with `memcached` performance is flat. Investigating further, we
found that memcached has a very limited traversal because the majority of its heap can be left unmodified and thus not traversed at all.

Updating redis with 4 threads took an average of 68% of the time taken with a single thread, with the speedup improving as the number of key-value pairs increases. For redis we do not achieve perfect speedup as the majority of redis’ state is in linked lists, effectively serializing a large amount of the traversal.

C-strider’s implementation of dynamic updating performs better than Kitsune on redis. Kitsune’s traversal is more heavyweight, allocating several data structures with type information for each item traversed, and several additional data structures for each instance of generic types. Redis makes heavy use of generics, which results in the allocation of a total of 17.16MB worth of data structures when updating redis with 100,000 key-value pairs with Kitsune. C-strider stores type information and reuses it for each entry by looking it up in the type table, so it does not incur this overhead, and generally benefits from C-strider’s more streamlined approach.

Memcached updated with a parallelized traversal is slower than when using a single-threaded one, running at an average of 2.3ms slower than the original time across all trials, due to the overhead of using multiple threads. However, the update time for memcached is quite small to begin with.

5.4. Heap profiling

It is often useful to profile a program’s behavior when optimizing its performance. With C-strider, we implemented a type-aware heap profiler that is able to give accurate counts (numbers, and total bytes) of objects present in the heap. This information can be useful for, say, finding the leading sources of bloat. The implementation of our profiler is straightforward: the perfaction code simply keeps a hashtable that maps the type t of a visited item to a pair tracking the total count and size in bytes. We are aware of only one C-heap profiler that provides such fine-grained per-type information [16].

We used our profiler to improve dynamic update times (Figure 15). The profiler showed what structs had the most instances, which we then manually inspected. If the high-count structs did not have any fields that needed to be updated or traversed (such as a structure with only primitive values), we then wrote perfaction rules directing the traversal for the DSU service to return 0 when we reached them. For example, the high counts of uint64_t directed us to write a rule for struct stats in Figure 13, line 167.

Time required for profiling. Generating a full profile requires visiting every object in the heap, whereas for other services we can sometimes halt traversal early. Table II shows a summary of the results of profiling. The first column shows the program name and a footnote explaining the amount of state present at the time of traversal. The second column shows the total number of calls to perfaction_ptr, perfaction_struct, and perfaction_prim. The third column shows the amount of time it took to profile the heap using 4 threads, and the fourth column shows the semi-interquartile range (SIQR) for the measured times. The fifth column shows the number of unique types that were discovered in the traversal, and the final time is the sum in KB of all unique allocated structures traversed. The time for traversing all of a memcached heap with 4,000 key-value pairs is only slightly slower than serializing a memcached heap with 4,000 key-value pairs. The time for traversing all of the redis heap for 4,000 key-value pairs is significantly slower than serializing the redis heap with the same number of entries because we did not employ any optimizations for the key-value items.
and traversed all fields individually. Snort is by far the largest heap to traverse with 10,299,744 calls to the \texttt{perfaction} functions and 138,417KB of allocated structures. While Snort has a high traversal time overall, the ratio of traversal time to calls to \texttt{perfaction} is similar for all three applications: 0.43 \textmu s/call for Snort, 0.41 \textmu s/call for redis, and 0.60 \textmu s/call for memcached, which is higher because the smaller heap does not amortize the startup and teardown costs.

5.5. Heap assertion checking

We also used C-strider to implement a simple heap assertion checking system, inspired by Aftandilian and Guyer’s GC assertions \cite{6}, which employ the Java garbage collector as a traversal mechanism. Heap assertions are tightly tied to particular data structures, and with C-strider we can write an assertion specific to each type. For example, we could assert that a doubly linked list is well-formed by adding a check during traversal like:

\begin{verbatim}
if (type == TYPE_linked_list_PTR)
    assert(&in->next->prev == &in);
\end{verbatim}

This application has no service-specific code.

\textbf{Time required for heap assertions.} The amount of time required for heap assertions varies greatly on the size of the portion of the heap being traversed. For example, in memcached, the primary hashtable contains a series of doubly linked lists. Asserting the property above for these lists required essentially the same time as for serialization (Figure 14) as the traversal covers the same portion of the heap. For redis, we created a set of assertions checking that timestamps were not in the future, file descriptors were valid (using \texttt{fcntl}), database item \texttt{enum} fields were valid, and that linked list pointers were as expected. We traversed the entire redis heap, and the traversal times were similar to the update time shown in Figure 15.

For Snort, we implemented a checker that asserts that installed function pointers are only drawn from a whitelist, and have not been hijacked to perform malicious functionality \cite{5}. Additionally, we asserted that structures in the custom memory pool had valid pointers, that all of the rule list nodes were set to the appropriate mode, and that the rule lists were correctly formed. In total, we asserted the correctness of 1749 function pointers and 446 other heap items in 279.98ms.

6. RELATED WORK

There are several threads of related work.

\textbf{Kitsune.} The most closely related work is Kitsune \cite{4}, a dynamic software updating (DSU) system for C that directly inspired C-strider. There are several major advances that C-strider makes over Kitsune. First, C-strider is much more general than Kitsune, as it can implement a variety of services beyond updating. Second, in addition to type annotations, Kitsune uses a domain-specific language, \texttt{xfgen}, to customize the traversal, rather than customization via \texttt{perfaction} functions in C-strider. The reason for this difference is that Kitsune does not have a run-time representation of types. We find C-strider’s approach much simpler and adaptable: \texttt{xfgen} is both over-specific to DSU and hard to extend to new applications. Third, C-strider’s use of run-time types improves performance, because arrays and generics can be implementing by simply making a new type (with \texttt{mktyp}) and using the standard \texttt{visit} function. In contrast, Kitsune uses a very complex closure system to traverse arrays and generics, and the extra levels of indirection cause the slowdown we saw in Figure 15. Finally, C-strider supports parallel traversal, while Kitsune is single-threaded.

\textbf{Type-directed programming and annotations.} In this paper, we used annotations to convey the actual type of heap items to make type-safe traversal possible. These type annotations are inspired by Deputy \cite{17, 13} and Cyclone \cite{18}. Cyclone is a type-safe variant of C that also uses programmer-supplied annotations (in addition to an advanced type system, a flow analysis, and run-time checks).
Deputy uses dependent type annotations, e.g., to specify the tag field for a union or to identify existing pointer bounds information.

Hinze and Löh [19] compare various approaches to datatype-generic programming and generate code based on type definitions. C-strider also generates code over all of the datatypes in the C program’s heap, but the exact code generated differs depending which types the program defines.

**Garbage collection.** Conservative garbage collectors [12] also traverse the entire heap, treating everything that looks like a pointer as a pointer. In contrast, C-strider’s heap traversals must be exact rather than conservative. For example, an `int` that happens to have the same value as a function pointer could be serialized and deserialized incorrectly (because the function pointer could be at a different address when deserialized in another process, thereby incorrectly changing the `int` as well).

**Applications.** As mentioned in the introduction, there are several systems that implement the particular traversals we explored, but in an ad hoc way.

*Dynamic Software Updating.* Dynamic software updating (DSU) systems must traverse the heap to transfer state from one version of a program to another. We already compared to Kitsune above. Many other DSU systems [20, 21] also implement state transformation. Similar to state transformation, hot-swapping object instances [22] allows objects to be switched to another implementation while the system is running, seamless to the user. C-strider could be extended to perform a similar function during traversal.

*Serialization.* In this paper, we implemented serialization and deserialization for the heap. This could be extended to a general checkpoint-and-restart system [1, 2, 23] by keeping some additional information about registers, thread-local data, and the stack. One benefit of the C-strider approach, compared to full program checkpoint-and-restart, is that C-strider serialization and deserialization is under program control and can be used piecemeal on portions of the heap. For example, it could be used to serialize a particular data structure in lieu of coming up with a new file format for that data structure.

*Heap Profiling.* Heap profilers can be used to determine how much memory a program uses, locate memory leaks, and find functions that do large amounts of allocation. Many popular heap profilers focus on function allocation granularity [24, 25, 3] based on calls to `malloc` rather than providing type-level granularity like Mihalicza et. al [16] and C-strider.

*Heap assertions.* Several researchers have developed systems for checking heap assertions. GC assertions [6] piggyback on top of the Java garbage collector to check a wide range of heap properties. DEAL [7] implements a language for heap assertions that can express combinations of reachability conditions. QVM [8] also checks heap properties, but using *heap probes* to keep overhead low by controlling the frequency of the checking and by sampling. PHALANX [9] extends QVM by adding checks for reachability; PHALANX also parallelizes the queries. Dynamic shape analysis [10] summarizes the pointer relationships of data structures and reports errors when invariants are violated. These systems all run on top of a garbage collector and virtual machine. In contrast, C-strider’s heap assertion checking works on C, which has no garbage collector or VM.

7. CONCLUSION

We have presented C-strider, which is, to our knowledge, the first general-purpose, type-aware heap traversal framework for C. C-strider analyzes a target program and generates traversal code for it, which can be run serially or in parallel. The traversal invokes programmer-supplied callbacks as it visits different locations, passing along appropriate type information. These callbacks implement, or customize, a service. We have experimented with several services, including (de)serialization, dynamic software updating, heap profiling, and heap assertion checking. Where needed, the programmer can augment the standard C types with additional information about array sizes and container types using special annotations; for other cases, like tagged unions, programmers can write
arbitrary C code to customize the traversal itself. We found that writing services and customizing traversals with C-strider generally requires a small amount of code, and that performance is reasonable and scales appropriately with heap size.

REFERENCES