SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

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.

Received . . .

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, including 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 run-time type information, has to date required a heroic effort.

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 that implement a program-independent *service* (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. C-strider's API has been designed to be easy to use: Section 2 illustrates how to use C-strider to build a 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

*Correspondence to: {ksaur,mwh,jfoster}@cs.umd.edu

Contract/grant sponsor: NSF; contract/grant number: CCF-0910530, CCF-1116740

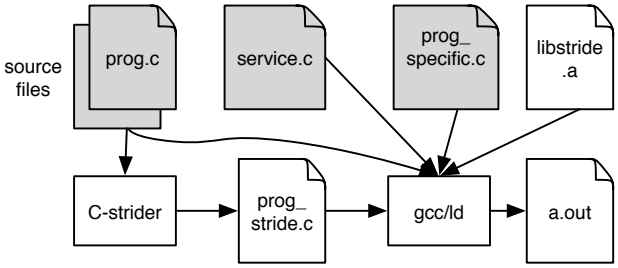

Figure 1. C-strider architecture.

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 is (mostly) breadth-first and 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. The programmer can fill in such missing type information in two ways. One way is programmer annotations, in the style of Deputy [13] and Kitsune [4], which can provide array length information and precise types for **void***** pointers in generic data structures. The other is by customizing the traversal to consider 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. This program-specific code (in prog_specific.c) is implemented by proxying the callbacks and delegating to the service code to perform the underlying service. Thus, C-strider provides a clean separation between two reusable pieces of code: the *program-specific* traversal (the annotations, the generated traversal, and the code in prog_specific.c, all of which can be used for many services in the same program) and the *service-specific* code (in service.c, and which can be used by many different programs). In addition, the code can be arbitrarily customized for particular program/service pairs, if needed. (Section 4 describes the annotations and other forms of program-specific customization.)

We used C-strider to implement four traversal-based services (details in Section 6), 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, e.g., symbol relocations due to address-space layout randomization. Checkpointing is relatively fast; we could serialize and deserialize a memcached heap with 30,000 key-value pairs in under 100ms.

Dynamic Software Updating We modified Kitsune, a dynamic software updating system for C, 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.

Prepared using speauth.cls
DOI: 10.1002/spe

```
struct traversal {
  /* Processing primitive values. (int, float, etc.) */
  void (*perfaction_prim)(void *in, typ t, void *out);

  /* Processing struct values. Return 1 to traverse fields in the struct, else return 0. */
  int (* perfaction_struct )(void *in, typ t, void *out);

  /* Pointer processing; first visit. Return 1 to follow the pointer, else return 0. */
  int (* perfaction_ptr )(void **in, typ t, void **out);

  /* Previously encountered pointer processing. */
  void (*perfaction_ptr_mapped)(void **in, typ t, void **out);
};
```

Figure 2. Basic C-strider service API.

Heap assertion checking C-strider can be used to 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 are not in the future, and **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. For example, 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 the ability to drop back 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. DEVELOPING SERVICES WITH C-STRIDER

This section focuses on the design of C-strider from the developer's perspective. The key challenge in C-strider's design is trading off simplicity—we want the API to be easy to use—and expressivity—we need it to be useful in many situations. We will illustrate our design by showing how to use C-strider to build a serialization and deserialization service.

2.1. C-strider API

C-strider traversals aim to visit and transform elements of a program's heap. There are three kinds of elements: *primitives* like integers, enumerated types, etc.; *aggregates* like structs, which are single objects that contain multiple elements; and *pointers* to other elements. C-strider's API asks a service developer to write three main callback functions, corresponding to the three kinds of heap elements.[†]

[†] Arrays are another kind of aggregate element; these are handled automatically by the traversal, as discussed in Section 3.

Prepared using speauth.cls
DOI: 10.1002/spe

```
1  void serial_prim(void *in, typ t, void *out){
2    fwrite(in, get_size(t), 1, ser_fp);
3  }
4  int  serial_struct (void *in, typ t, void *out){
5    return 1;
6  }
7  int  serial_ptr (void **in, typ t, void **out){
8    if (*in == 0)
9        uintptr_t  zero = 0;
10       fwrite(&zero, sizeof( uintptr_t ), 1, ser_fp);
11   else {
12     char *symbol;
13     if ((symbol = lookup_addr(*in))) {
14       uintptr_t  two = 2;
15       fwrite(&two, sizeof( uintptr_t ), 1, ser_fp);
16       serial_string (symbol);
17     } else {
18       serial_prim ((void *)in, t);
19       if (t == TYPE_PTR_CHAR)
20          serial_string ((char *)*in);
21       else return 1;
22   } }
23   return 0;
24 }
25
26
27
28
29
30
31
32
33
34 // helper function, not part of API callbacks
35 void  serial_string (char *str){
36   /* assumes str != NULL */
37   int len = strlen(str)+1;
38   fwrite(&len, sizeof(int), 1, ser_fp);
39   fwrite(str, len, 1, ser_fp);
40 }
```

(a) Serialization

```
41 void deserial_prim(void *in, typ t, void *out){
42   fread(out, get_size(t), 1, ser_fp);
43 }
44 int  deserial_struct (void *in, typ t, void *out){
45   return 1;
46 }
47 int  deserial_ptr (void **in, typ t, void **out){
48  void *ptr, *tgt;
49  fread(&ptr, sizeof(void*), 1, ser_fp);
50  if (ptr == (void*)2) { // pointer to symbol
51    char * symbol;
52   deserial_string (&symbol);
53    *out = lookup_key(symbol);
54    free(symbol);
55  }
56  else if (ptr == NULL) // null pointer
57    *out = 0;
58  else if ((tgt = find_mapping(ptr))) // revisit
59    *out = tgt;
60  else // haven't seen it; read the data
61    if (typ == TYPE_PTR_CHAR) { // a string
62      deserial_string ((char **)&tgt);
63      *out = tgt;
64      add_mapping(ptr, tgt);
65    } else {
66      int sz_new = get_size(get_ptrtype(t));
67      tgt = malloc(sz_new);
68      *out = tgt;
69      add_mapping(ptr, tgt);
70       visit (tgt, get_ptrtype(typ), tgt);
71    }
72  return 0;
73 }
74 // helper function, not part of API callbacks
75 void  deserial_string (char **str) {
76  int len;
77  fread(&len, sizeof(int), 1, ser_fp);
78  *str = (char *)malloc(len);
79  fread(*str, len, 1, ser_fp);
80 }
```

(b) Deserialization

Figure 3. An example service: serialization and deserialization.

The names and type signatures these three callbacks are listed in Figure 2; we will discuss the last callback below. Pointers to the callbacks reside inside **struct** traversal, which is passed as a parameter to initialize C-strider.

Each callback has the same three arguments. The first argument, in, points to the program data being visited, which has a type represented by t, the second argument. The last argument's use depends on whether the programmer is developing a *transformation* service that modifies memory locations of heap objects, or a *traversal* service that does not. A traversal service like serialization only has one set of traversal roots, and simply ignores out. For a transformation service, like deserialization, out points to replacement memory such that writing out specifies how in's data should be transformed.

DOI: 10.1002/spe

2.2. *Implementing serialization*

To explain these three callbacks in more depth, we refer to the code in Figure 3a, which implements a serialization service. This service is a traversal service, so out is unused. The functions are passed as callbacks to C-strider in an instance of **struct** traversal from Figure 2.

The first callback, serial_prim (assigned as a callback for perfaction_prim), handles data of primitive type (e.g., **int**, **double**). The traversal calls this function with the address of each global variable and each struct/union field that is a primitive type, indicating the specific type of primitive with typ t. The code for serializing primitives using this function is given at the top of Figure 3a. It does the obvious thing: it uses API function get_size to compute the number of bytes for type t, and writes the data via the serialization file handle, ser_fp.

The second callback, serial_struct (assigned as a callback for perfaction_struct), is called with the address of every **struct** reached during the traversal. The particular **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 which case the function can call the visit library function (explained in Section 2.5) to selectively visit fields.

The third callback, serial_ptr (assigned as a callback for perfaction_ptr), is invoked for each pointer when it is first reached. Here the type of in is **void** **, rather than **void** *, because it is the address of a pointer. For serialization, the callback checks whether the pointer is null; if so, it writes null to the file. Otherwise, it calls C-strider's 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, 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.

As presented so far, the API has one limitation: If there are cycles among pointerful data structures in the heap, traversal may loop forever. Rather than force the programmer to add ad hoc logic to avoid this case, C-strider instead includes support for tracking previously visited pointers. Thus, the C-strider API includes a fourth callback, perfaction_ptr_mapped, which is called for each pointer that is reached but has been visited previously during the traversal. 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 corresponding serial_ptr_mapped callback function is not shown because it is quite similar to the serial_ptr function; the difference is that it knows the pointer cannot be null, so only the **else** case of the above code is needed, and 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.

2.3. *Implementing deserialization*

While serialization is a traversal service that only reads the heap, deserialization is a transformation service that modifies the heap during traversal. It begins by writing global variables and then continues to initialize the heap as it processes the serialization file. Deserialization is unusual because the input heap comes from prior program run, as captured in the serialization file. Thus, we will see that deserialization must call API functions that add mappings, read from this file, and direct the traversal. A more typical transformation service, like dynamic software updating (Section 6.3) or hot swapping [14], would simply traverse the existing heap while transforming it, requiring no extra programmer assistance.

Figure 3b shows the deserialization callbacks, which are passed to C-strider in an instance of **struct** traversal. The first callback, deserial_prim, is just like serialization, but reads data instead of

DOI: 10.1002/spe

```
81 t1 *global;
82 t2 *other_global;
83
84 int main(int argc, char **argv){
85   if (!do_deserialize(argc,argv)){
86     /* do standard initialization */
87   }
88   /* long-running loop for
89            rest of the program... */
90   while(1){
91     if (...) checkpoint();
92     ...
93   }
94 }
```

```
95 int do_deserialize(int argc, char **argv){
96   if ((char)argv[1][0] == 'D'){
97     ser = fopen("ser.txt", "rb");
98     init (&deserial_funs, 0);
99     visit (&global, TYPE_t1, &global);
100    finish ();
101    fclose(ser);
102    return 1;
103  }
104  return 0;
105 }
106 void checkpoint(void){
107   ser = fopen("ser.txt", "wb");
108   init (&serial_funs, 0);
109   visit (&global, TYPE_t1, &global);
110   finish ();
111   fclose(ser);
112 }
```

Figure 4. Example program employing checkpointing.

writes it. deserial_struct returns 1, which means struct fields are traversed as usual, i.e., in the same order they were serialized.

The callback deserial_ptr inverts the logic from the serialization case, allocating new memory as needed and updating the mapping. It starts by reading the pointer itself from the file. Notice this is the address from the checkpointed program. Since the address space of the current run of the program may be different, the traversal uses the mapping table to map the checkpointed run's addresses to the equivalent ones in the current run's address space. There are several cases. 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. If this particular pointer was seen before—i.e., it was previously read from the file and therefore is in the mapping table—then *out is assigned to the value from the map, which is the corresponding address in the current run of the program.

Otherwise, the code starting on line 60 handles a non-symbol, non-null pointer that has not been seen before. If it is a string, the code reads in that string, and then adds a mapping to 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 deserial_ptr function returns 0, since either no subsequent traversal is needed, or the traversal was manually triggered by calling visit .

The deserial_ptr_mapped function (not shown) will never be called, because translation of pointers from the serialization file always happens through manual lookups of the table on line 58. If a pointer from the serialization file is seen again in deserialization, then there is no attempt to traverse it (deserial_ptr returns 0).

2.4. *Using serialization for checkpointing*

Given the callback definitions, C-strider provides, or generates, the code that actually traverses the various elements of the heap, invoking the callbacks as it goes. To use a C-strider service, therefore, the programmer then needs to invoke the traversal at the appropriate place in their program.

Using the (de)serialization service is simple. Consider Figure 4, which uses this service for checkpoint and restart. Assume the data to be (de)serialized is stored in some variable global of type t1. When the program begins, it calls do_deserialize on line 85 to check whether the user has requested to restart at a checkpoint. If not, do_deserialize simply returns 0, and initialization

DOI: 10.1002/spe

```
// Directing the traversal
void init(struct traversal * funs, int parallel);
void visit(void *in, typ t, void *out);
void visit_all(void);
void register_root(void * in, typ t);
void deregister_root(void * in);
void finish(void);
// Functions for querying the symbol table
char *lookup_addr(void *addr);
void *lookup_key(const char *key);
// Functions for manipulating the pointer
// mapping table
void add_mapping(void *in, void *out);
void *find_mapping(void *in);
```

```
// Functions for manipulating type information
int is_prim(typ t);
int is_ptr(typ t);
int is_funptr(typ t);
int is_array(typ t);
int get_size(typ t);
int get_num_array_elems(typ t);
int get_ptrtype(typ t);
// Constructors
typ mktyp_ptr(typ t);
typ mktyp_arr(int arrlen, typ t);
// Generics
int get_maintype(typ t);
typ *get_generic_args(typ t);
int get_num_gen_args(typ t);
typ mktyp_instantiate(typ t, int nargs, typ *args);
```

Figure 5. C-strider API.

continues as usual. Otherwise, the do_deserialize function invokes the deserialization traversal to initialize memory. First, C-strider must be initialized by calling init (line 98), which takes two parameters: a struct specifying which traversal functions to be used (having type **struct** traversal from Figure 2), and a flag specifying whether the traversal should be run in single-threaded mode, guaranteeing an in-order traversal (0), or parallel mode with multiple threads (1). In this example, for deserialization, the first init parameter is a **struct** traversal with the function pointers set to the C-strider API functions from Figure 3b, and the second parameter is 0 to signify a single threaded traversal, as (de)serialization relies on an in-order traversal to preserve the order of the heap data. The init call also initializes some auxiliary data structures that C-strider uses to perform the traversal.

Next, the programmer selects which objects of the heap to traverse. One option is to call visit_all, a function generated by C-strider that traverses all objects reachable from global variables. Alternatively, as we do here (line 99), the programmer can call visit to traverse starting from a specific root. This is useful when a full heap traversal is not appropriate, e.g., here where we do not want to traverse other_global. Deserialization is a transformation service, but as mentioned earlier, it is an unusual one: the source heap is read from the serialization file, while the target heap is that of the current program. As such, we must specify out as &global. For in, we also use &global but this ends up not being important—as we can see from Figure 3b, the in argument is actually ignored by the deserialization code. (A more typical transformation service is discussed in Section 6.3.)

Lastly, the programmer calls finish, which tells C-strider to clean up the auxiliary data structures it uses for the traversal. At this point, all data has been deserialized, so the program closes the file, and do_deserialize returns 1 on line 102, thereby skipping from-scratch initialization.

The main program now proceeds, entering a long-running loop. Intermittently it calls checkpoint to serialize the current state. The function checkpoint (line 91) is similar to do_deserialize, except now the programmer uses the serialization functions, which write to the serialization file rather than read from it. Because serialization is a traversal service, we specify the in argument to visit as &global; the third argument is unimportant since serialization will ignore it, but we specify it as &global to be compatible with the expectations of automatically generated code (cf. Figure 7 in the next Section). For traversal services such as serialization, a call to visit_all will automatically set out equal to in for all elements of the traversal.

2.5. *Full C-strider API*

We now wrap up our discussion of C-strider's basic design by discussing the remainder of the API, given in full in Figure 5. Section 4 will discuss in more detail how these functions can be used to customize the generated traversal.

DOI: 10.1002/spe

Starting in the upper-left, the first set of functions controls the initialization (init), starting points (visit or visit_all), and conclusion of the traversal (finish), as briefly explained in Section 2.4. Figure 4 showed how the visit function can be called on any location to start traversal from that point onward. Certain applications, however, need to traverse the entire heap. For example, full heap traversal is needed for dynamic software updating and for heap profiling. To support these cases, C-strider provides a function visit_all , which traverses the entire heap automatically by calling a list of generated custom traversal functions for each global variable in the program. Additionally, the programmer can use register_root and deregister_root to add and remove, respectively, additional locations for visit_all to visit. For example, we use this to traverse certain local variables in dynamic software updating, discussed in Section 6.

The next two functions, lookup_addr and lookup_key, perform lookups in the C-strider symbol table, mapping from symbol names to addresses or vice-versa. The last two functions on the left, both of which we saw above, manipulate the mapping table used in transformation services. While we could have left it up to the user to implement these features on an ad hoc basis, we incorporated them into C-strider because we found them useful for a range of services.

Finally, as we saw in the serialization and deserialization code, one of the major features of C-strider is that it is type aware, which not only allows the traversal to be precise, but also allows service code to adjust its behavior by type. For example, strings of TYPE_PTR_CHAR are serialized differently than other pointer types. In general, C-strider generates a set of type names for those types that are statically present in the code, e.g., TYPE_INT, TYPE_PTR_INT. User-defined types also available at run time, e.g., TYPE_STRUCT_DLIST for a program that includes **struct** dlist .

The functions on the right of the figure manipulate types. The first group of functions manipulate standard types, e.g., determining whether they are primitives (is_prim) or pointers (is_ptr), calculating their size (get_size, get_num_array_elems), and returning the pointed-to type (get_ptrtype). The particular set of queries shown here was derived from our experience building several services (Section 6); other applications could potentially require other accessors, which are easy to add.

The next two functions, mktyp_ptr and mktyp_array, create new pointer or array types, respectively, at run time. We use these to maintain type information for arrays and other pointed-to blocks whose length is only determined at run time. We will see an example in Section 4. The last set of functions lets us create and query generic types, e.g., parametrically typed linked lists. Section 4 discusses generics in detail.

Notice that, since types are available at run-time and can be fully queried, it is not strictly necessary to have four perfaction callbacks. In theory, perfaction_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, because we often want to treat each of those groups independently (e.g., as in serialization and deserialization).

3. TYPE-AWARE TRAVERSAL

Section 2 showed C-strider from the service developer's perspective. This section provides details on the inner workings of the traversal, in two parts. First, we discuss the main components of the traversal, which are the visit function, which is called for each element of the heap, and the *task queue*, which organizes the work of the traversal. Second, we explain how the visit function relies on several bits of C-strider-generated code, in particular a table of type representations, a set of struct-specific visit functions, and finally a visit_all function, which can be used to initiate a heap traversal from the program's roots.

3.1. Main components of the traversal

To perform the traversal, C-strider must determine which perfaction_ function to call with which addresses, and it must keep track of those addresses it has visited and the addresses to visit next. The overall traversal is orchestrated by two pieces of code: the visit function, shown in Figure 6, which

DOI: 10.1002/spe

```
113 void visit (void *in, typ t, void *out){
114   if (is_prim(t)){
115     perfaction_prim(in, t, out);
116   } else if (is_ptr(t)){
117     if (!in) return;
118     void *lookup;
119     if ((lookup = find_mapping(*(void**)in))) {
120       perfaction_ptr_mapped(in, t, &lookup);
121       *(void**)out = lookup;
122     }
123     else{
124       int retc = perfaction_ptr(in, t, out);
125       add_mapping(*(void**)in, *(void**)out);
126       if (!retc || is_funptr(t)) return;
127       typ t_p = get_ptrtype(t);
128       visit (*(void **)in, t_p, *(void **)out);
129     }
130   } else if (is_struct(t)){
131     if (perfaction_struct(in, t, out))
132       enqueue(in, t, out);
133   } else if (is_array(t)){
134     /* enqueue task that calls visit on each array element */
135 } }
```

Figure 6. Traversing using type information.

calls the appropriate perfaction_ function based on an object's type; and a task queue that contains pointers to **struct**s or arrays whose elements need to be visited.

Traversal visit function. The visit function is the workhorse of the traversal. It is invoked for each visited heap object, starting with the roots. As arguments, visit is given a pointer to the heap object to consider, a representation of that object's type (expressed as an object of type typ), and a pointer to the corresponding object in the transformed heap. We defer discussion of the implementation of type representations to the end of this section.

The body of visit is straightforward. For primitive types (**int**s, **char**s, etc) and other non-standard terminal primitive types (mutex_t, time_t, size_t), the traversal calls perfaction_prim on line 115, which in turn calls the user-provided callback from **struct** traversal, passing it a pointer to the primitive being traversed.

If typ is a pointer type, the traversal returns immediately if the pointer is null (line 117). Otherwise, there are two cases, depending whether the pointer has been visited before (line 119).

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.

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 127–128).

If the type is a **struct**, the traversal calls perfaction_struct. If that function indicates the traversal should continue, the code calls enqueue, which creates a task that will visit t's fields and adds it to the queue.

Finally, if typ is an array type, the traversal likewise enqueues a task to recursively visit each of the array's elements. When performing a multi-threaded traversal, this code will divide this task into sub-tasks covering sub-ranges of the array; the number of tasks depends on the size of the array and the number of processors on the machine.

DOI: 10.1002/spe

```
struct dlist { struct dlist * next, *prev; int x; }
struct dlist *head = ...;
struct dlist **phead = &head;
```

```
139 void _visit_struct_dlist (void *in, typ t, void *out) {
140   struct dlist_i *in_l = in;
141   struct dlist_o *out_l = out;
142
143   visit (&in_l→next, TYPE_dlist_PTR, &out_l→next));
144   visit (&in_l→prev, TYPE_dlist_PTR, &out_l→prev));
145   visit (&in_l→x, TYPE_INT, &out_l→x);
146 }
```

Figure 7. Generated traversal code for **struct** dlist .

Note that the 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).

The task queue. The task queue consists of traversal work that remains to be done. It contains two types of tasks: structs whose fields need to be visited, and arrays whose elements need to be visited. The traversal in the former case is defined by a custom visit function that C-strider generates for each **struct** in the program (as described in Section 3.2). The traversal in the latter case is simply a loop over the specified range of the array. Either way, the traversal function simply calls visit on each of the elements of in and out and then returns.

Single and multi-threaded traversal modes. The handling of enqueued tasks depends on the mode C-strider is used in. In single-threaded mode, the main thread repeatedly pulls tasks off the queue and calls visit , until the queue is empty, forming a breadth-first search of the enqueued tasks. Using a queue enables C-strider to traverse cyclic data structures (which in C must always go through a **struct**) without requiring a deep stack.

To speed up the traversal process, C-strider supports multi-threaded traversal. In this mode, C-strider launches $n-1$ worker threads in addition to the main thread, where n is the number of processors. The threads consume tasks that are generated by enqueue, which are initially produced by C-strider's main thread.

C-strider uses a work-stealing scheduler: Each thread has a local queue (created initially at the call to init), and when it runs out of tasks it attempts to steal one from another queue [15]. 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 enqueue new tasks. Therefore, we are not done traversing until all queues are empty and no tasks are in-flight.

In a multi-threaded traversal, the map of visited locations 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.

A traversal completes once the service calls finish . At this point, the main thread and helper threads (if multi-threaded) process all remaining items in the task queues until they are empty. Once complete, temporary data structures are freed and the helper threads exit.

In C-strider we have made a specific design choice to only create tasks for structs and arrays. As an alternative design, we could have enqueued tasks for every element in the heap, including pointers and primitives. However, the cost of making a task, queuing it, dequeuing it, and executing it, would dwarf the cost of just executing the task directly, despite the lost opportunity for parallelism. We find that our current design generally balances overhead with opportunities for parallelism.

DOI: 10.1002/spe

3.2. Generated code

In addition to the visit function and queues, C-strider relies on per-program generated code to perform the traversal, including type representations, a set of traversal functions for each **struct** in a program, and code to register globals to be visited by visit_all.

Type representations and the type table C-strider traversals use *type representations* that describe each type in a program. Such type representations cover both built-in types (like **int** and **char**) and user-defined types. Type representations have type typ, and are used by both C-strider library functions like visit and the perfaction callback functions provided by the service developer with **struct** traversal.

A typ t is represented as an integer that indexes a generated *type table*. This table maps types to relevant information about them like their size in bytes, the 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 was instantiated with (if applicable); Section 4 says more about generics.

C-strider creates a unique typ for each type appearing (statically) in the program text and gives it a predictable variable name, e.g., TYPE_PTR_CHAR from Figure 3a. A service developer can refer to such names in the perfaction functions. Types are extracted from the source program by an analysis implemented in CIL [16]. CIL takes as input preprocessed C source code and outputs information about the types of all variables and functions of the program.

C-strider also assigns a distinct typ to each type alias, e.g., **typedef int** size_t in the source would yield 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. Thus, we can write the traversal so that when it visits this particular named type, it visits the structure as a whole and not just the string.

A traversal can generate type representations on the fly, using the mktyp_ functions (cf. Table 5). If the run-time generated type matches one already in the type table generated statically, then mktyp_X will return the associated index. However, if the type did not appear in the program text then it will not be in the table already. In this case, a new entry is added to the table dynamically, and the index to that element is returned.

Traversing structs. As seen in the previous section, the visit function directs the traversal by handling the pointers and calling the appropriate perfaction_ functions. To traverse inside a struct, C-strider must call visit for each field of the struct. To assist with this, in addition to generating the type table, C-strider generates a traversal function for each struct and each field within the struct. For example, Figure 7 shows the traversal code for a doubly linked list **struct** dlist, defined at the top of the figure. (Code slightly simplified for clarity.) Each generated traversal function is named by prepending _visit to the name of the struct. The parameters **void** *in and **void** * out are the addresses of the struct being traversed, and parameter typ t is used in the case of generics, as described in Section 4.1. C-strider generates code to assign the parameter addresses to the type of struct being traversed (shown on lines 140 and 141) to allow the memory to be accessed properly as struct fields. Finally, C-strider generates code to traverse each field of the struct (shown on lines 143 - 145) by getting the type information for each field from CIL. C-strider inserts the corresponding type name generated according to the pattern used when generating type representations (e.g., TYPE_dlist_PTR for **struct** dlist *) and generates a call to visit for each field of the struct.

visit_all. During compilation, C-strider gathers a list of all non-static global variables of a program using CIL. When init is called, C-strider uses the list of non-static globals to populate hash tables mapping symbol name to address (for lookup_key from Figure 5), and address to symbol name (for lookup_addr). After initialization, if the user then calls visit_all, C-strider iterates through all entries in the lookup_key table, combines each entry with some auxiliary type information, and uses that information to call visit on each global. After visit_all calls visit on all globals in the

DOI: 10.1002/spe

```
typedef struct _dlist_arr {
  int arrlen;
  struct dlist ** T_PTRARRAY(self.arrlen) arr_head;
} dlist_arr ;
```

```
151 void _visit_struct_dlist_arr (void *in, type t, void *out) {
152   struct dlist_arr *in_a = in;
153   struct dlist_arr *out_a = out;
154    visit (&in_a→arrlen, TYPE_INT, &out_a→arrlen);
155    visit (&in_a→arr_head,
156      mktyp_ptr(mktyp_arr(in_a→arrlen,TYPE_STRUCT_dlist_PTR)),
157      out_a→arr_head);
158 }
```

Figure 8. Length annotation and generated traversal code.

hash table, it then calls visit on all roots the programmer has manually added with register_root. C-strider does not process static global variables, as discussed in Section 5.

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 perfaction 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. Depending on how an array is declared, C-strider may or may not be able to statically determine its length. If an array is declared with static length information such as **struct** dlist array_x[6], C-strider can determine that it needs to process 6 elements for array_x. However, when a user declares an array with dynamic length, such as **struct** dlist ** arr_head, C-strider cannot statically determine the length of the array and needs additional information to determine how to process arr_head. 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 8 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 of the same structure (self). Notice that line 156 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.

If the user does not annotate an array, C-strider will presume it is a singleton. From the example in Figure 8, the code **struct** dlist ** arr_head; would be treated by C-strider as a single instance

DOI: 10.1002/spe

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

```
163 void _visit_struct_list (void *in, type t, void *out) {
164   struct list *in_l = in;
165   struct list *out_l = out;
166   typ *args = get_generic_args(t);
167   int num_args = get_num_gen_args(t);
168   assert(num_args == 1);
169   typ t0 = args[0];
170
171    visit (&in_l→val, t0, &out_l→val);
172    visit (&in_l→next,
173      mktyp_instantiate(TYPE_STRUCT_list_PTR,num_args, args),
174      &out_l→next);
175 }
```

Figure 9. Generic annotation and generated traversal code.

of a "pointer to a **struct** dlist pointer," rather than a "pointer to array of **struct** dlist pointer" with the annotation.

Type annotations for generics. Because C-strider uses static type information to generate the heap traversal code, **void** *'s in a program present a roadblock to the traversal generation process as **void** *'s provide no type information. C-strider includes several annotations to type **void** *'s in generic data structures. Figure 9 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 166 and 167 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 173).

If the user does not add an annotation to a **void** *, C-strider prints out a warning during the type generation process, reminding the programmer to add an annotation if necessary. If no annotations are added to a **void** *, C-strider does not traverse the pointer as no type information is available.

4.2. Customization in perfaction functions

In some cases, type annotations are insufficient to guide the traversal. For example, consider the type at the top of Figure 10, which defines a **struct** whose u field is either **int** or **char**, depending the tag field being either 1 or 0, respectively. To select the correct field of u to visit, C-strider needs to know how to interpret tag.

To solve this and other problems, the programmer can write program-specific perfaction functions. Such functions (contained in file prog_specific.c in Figure 1) can assist C-strider in performing the traversal in a service-agnostic manner, per our example. Customizations can also adjust a service's functioning based on the particular program. For example, we might know that our program has an array of structs that all reference each other. Rather than traverse the entire array, we could customize serialization to write the whole array at once, and thereby avoid further traversing of the structs. When no customization is needed, the programmer can just delegate to the relevant service code.

DOI: 10.1002/spe

```
struct tagged_union {
    int tag; /* 1 selects u.x; 0 selects u.c */
    union{ int x; char c; } u; /* selected by tag */
};
```

```
180 int current_service;
181 #define DESERIALIZE 0
182 #define SERIALIZE 1
183
184 struct traversal deser_prog_funs = {
185   .perfaction_prim = &deserial_prim,
186   . perfaction_struct = &s_d_prog_struct,
187   . perfaction_ptr = &deserial_ptr,
188   .perfaction_ptr_mapped =
189       &deserial_ptr_mapped
190 };
191
192 int do_deserialize(int argc, char **argv){
193   if ((char)argv[1][0] == 'D'){
194     current_service = DESERIALIZE;
195     ser = fopen("ser.txt", "rb");
196     init (&deser_prog_funs, 0);
197     ...
198   }
199   return 0;
200 }
```

(a) Custom deserialization initialization code

```
201 int s_d_prog_struct(void *in, typ t, void *out){
202   /* Service-agnostic, Program-specific */
203   if (type == TYPE_STRUCT_tagged_union) {
204     struct tagged_union *in_u = in;
205     struct tagged_union *out_u = out;
206     visit (&in_u->tag, TYPE_INT, &out_u->tag);
207     if (in_u->tag)
208       visit (&in_u->u.x, TYPE_INT, &out_u->u.x);
209     else
210       visit (&in_u->u.c, TYPE_CHAR, &out_u->u.c);
211     return 0;
212
213   /* Service-specific */
214   } else if (current_service == DESERIALIZE){
215       /* Program-specific */
216       if (t == TYPE_...) {...}
217       /* Program-agnostic */
218       else return deserial_struct (in,t,out);
219   } else if (current_service == SERIALIZE) {
220     ...
221     return serial_struct (in, type, out);
222   }
223   return 1;
224 }
```

(b) Custom (de)serialization struct callback

Figure 10. Customizing traversal in perfaction functions.

Figure 10 provides a modified example of deserialization customized to handle our example tagged union. On line 180, **int** current_service keeps track of the current service, either SERIALIZE (1) or DESERIALIZE (0); we define this flag because both services will share code (though we focus on the code for deserialization, since serialization is similar). On line 184, **struct** traversal deser_prog_funs sets up the callback functions for the new set of program-specific deserialization functions. For perfaction_struct we specify a program-specific function, s_d_prog_struct (defined on the right side of the figure), while for the rest we use the standard functions from Figure 3b. The function do_deserialize (line 192) is similar to the previous version (Figure 4, line 95), with the addition of setting current_service on line 194 and passing deser_prog_funs to init on line 196. The checkpoint function (not shown) is similar to the version on line 106, with the addition of setting current_service = SERIALIZE and passing ser_prog_funs to init (which are initialized analogously).

The right side of Figure 10 shows the program-specific implementation of s_d_prog_struct, which acts as a wrapper around serial_struct and deserial_struct. The first part of the function is program-specific, but service-agnostic: If s_d_prog_struct is called on **struct** tagged_union, then we visit the tag on line 206, and then switch on the tag and call visit on the appropriate sub-field, returning 0 on line 211 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 214, if current_service is DESERIALIZE, we first perform service-specific, program-specific actions, e.g., we check for a particular type (line 216) to optimize serialization for it (not shown). Otherwise, on line 218 we call the default action, deserial_struct, which is a wrapper for the reusable service-specific code from Figure 3b, line 44.

DOI: 10.1002/spe

Similar to deserialization, the code for serialization (or any other service) performs service-specific checks, and then the default action.

Section 6.1 includes more details and examples of customizing the traversal for service specific customization.

5. LIMITATIONS

While C-strider is general and flexible, it does have some limitations, which we discuss here.

A general problem is that by being a generic framework, C-strider's performance might suffer compared to a purpose-built service for a particular application. For example, C-strider's serialization/deserialization service will, by default, serialize an array by serializing each of its elements individually. But if those elements contain no pointers, then it would be faster to simply write the entire array directly (similarly to how C-strider serializes strings). C-strider also uses generated type representations which might not otherwise be needed for a purpose-built service. That said, C-strider does allow programmers can customize a service to their program, to improve performance.

C-strider is intended traverse the heap, but it may come across pointers that refer to other parts of memory, such as stack-allocated variables. C-strider does not reliably provide information to a service developer about whether a pointer is to the heap or not, but in many cases this can be determined simply by looking at the pointer's address; e.g., on Linux, the heap is in "low" memory and the stack is in "high" memory. If necessary, the programmer could write a perfaction_ rule based on the memory address of the item being traversed, instructing C-strider to perform some custom action (such as printing an alert) if the traversed location is an unexpected memory location.

In principle, C-strider could be used to produce a type-aware garbage collector for C programs. Doing so would be challenging in practice, however. For one thing, C-strider would need assistance in finding all of the roots of the heap, which includes pointers on the stack, thread-local storage, etc. At the moment, this would require the programmer to explicitly register and deregister these roots (using register_root and deregister_root) when they come into and go out of scope. There is also an issue with roots that are static variables, discussed below, and with any roots that might be stored by external libraries. Some of these problems could be overcome through a source-to-source transformation that registers and deregisters roots automatically, similar to the one proposed for Kitsune [4].

Another limitation of C-strider is that it cannot properly traverse variables declared as bit fields, because C does not allow taking the address of bit fields. For example, C-strider does not generate traversal code for a structure with a element field of **unsigned int** valid : 1; or else the generated code visit (&in→valid ... , would not compile. In our experiments, we modified the program code to not have bit fields.

C-strider generates the visit_all function by analyzing all of the source files in a program, and generating a single function that registers its global roots. This function will not have access to **static** variables in other files. The programmer either will need to make variables non-static, or only initiate traversals (using visit) on variables that are in-scope.

6. 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.

DOI: 10.1002/spe

Prog.	LOC	Traversal T_*	Traversal Cust.	DSU (24)	s/d (78)	prof (65)	asrt (0)
memcd 1.2.3	~4K	6	0	17	44	9	20
redis 2.0.2	~13K	45	26	19	46	11	36
snort 2.9.2	~215K	143	21	n/a	n/a	4	67

Table I. Programmer Effort (measured in LOC)

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.

6.1. 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 we wrote was for handling **union**s or union-like data structures. For example, of the 26 LOC for the custom traversal for *redis*, 17 LOC 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 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 17 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 11, which excerpts part of the perfaction_struct function for *memcached*. This code switches based on the current service (lines 226 and 237). For each service, it then either performs type-specific actions (lines 227 and 238) or calls the service-specific action (lines 235 and 242). The current_service flag is set through the entry point to the service, e.g., the call to do_serialize in Figure 4 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).

6.2. 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

DOI: 10.1002/spe

```
225 int perfaction_struct (void *in, typ t, void *out){
226   if (current_service == DYNUPDATE){
227     if (t == TYPE_STRUCT_stats){
228       if (lookup_addr(out) != NULL){
229         int in_sz = get_size(t);
230         int out_sz = get_out_size(t);
231         assert(in_sz == out_sz);
232         memcpy(out, in, in_sz);
233       }
234       return 0;
235     } else return update_struct(in, t, out);
236   }
237   if (current_service == SERIALIZE){
238     if (t == TYPE_item){
239       struct _stritem_cpy * it = in;
240       int len_total = ITEM_ntotal(it);
241       // write len_total and other struct fields to disk
242     } else return serial_struct (in, t, out);
243 }}
```

Figure 11. perfaction code for *memcached*.

presented in Section 2 (Figures 3a and 3b). In addition to those 69 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 11. Here on line 240 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 perfaction_* 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 information, as this information would be stale between program restarts.) Figure 12 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. (We say "approximately" 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 5,592KB of allocated data structures in *redis* and 1,477KB 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.

DOI: 10.1002/spe

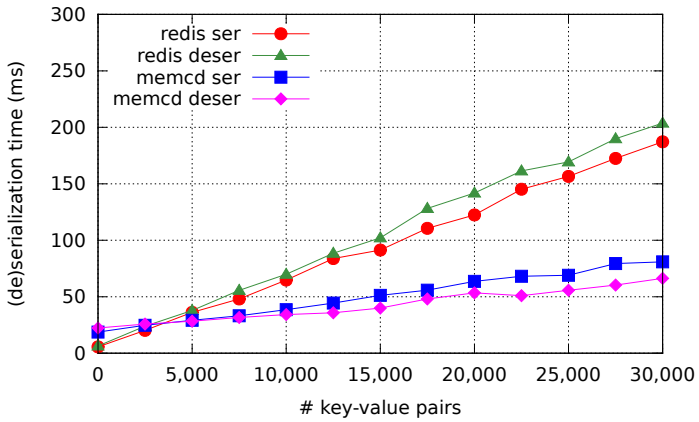

Figure 12. Serialization times (color plot).

6.3. *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 with the old version of the program as the in set of roots and the new version of the program as the out set of roots). Additional code (outside of C-strider) was required to implement other elements of DSU, 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 5), one for the old version's symbol table, and one for the new version. We also needed to track the size information for the new type's size, so we added a function get_out_size to get the corresponding size of the new version of a type for the DSU case. (Notice this is a DSU-specific function because it needs to understand types from a different program version.)

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 memcpys the appropriate bytes over.
- 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 to the new block to C-strider's hash map of visited locations. The program-specific action is then called to initialize the new block appropriately.
- When *re*-visiting a pointer to a malloc'd block for a type that has changed size, the code looks up the corresponding new block address and overwrites the old pointer with it (similarly to Figure 3b, line 58).
- When visiting a pointer corresponding to a known symbol, the code updates it with a pointer to the matching symbol from the new program version (similarly to Figure 3b, line 53).

DOI: 10.1002/spe

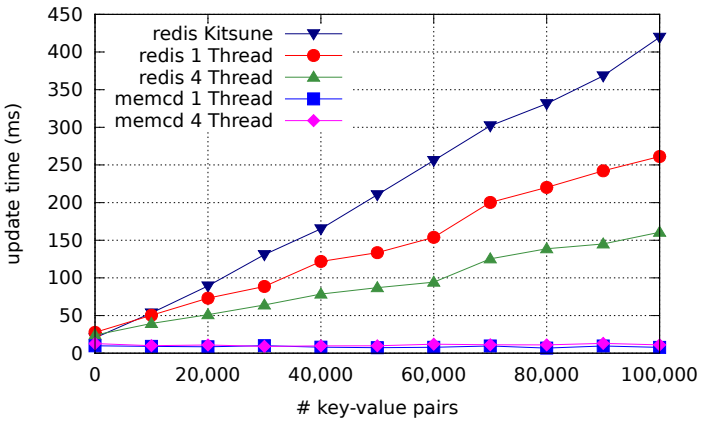

Figure 13. DSU state transformation times (color plot).

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_struct 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 11 (line 227) 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 2.5, C-strider allows local variables to be added as roots of the traversal. We used this feature to support updates to memcached—in particular, we inserted code to register local variables of the main function. These variables store information that the user supplies as command line options. Registering these local variables ensures that the command line option state is propagated to the next version when we call visit_all at the end of the main function. In total, 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 13 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. Rather than create and allocate generic type information for each generic instance like 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 additional overhead.

Prepared using speauth.cls
DOI: 10.1002/spe

Program	Total calls to perf	Trav. (ms)	siqr (ms)	Uniq types	Alloc'ed (KB)
memcd 1.2.3*	81,844	49.36	(7.14)	65	360
redis 2.0.2*	494,389	203.21	(16.71)	92	2,189
snort 2.9.2+	10,299,744	4412.51	(194.71)	1,019	138,417

*For 4,000 entries +For traffic at 30 pkts/s

Table II. C-strider Profiling Heap Data

Memcached updated with a parallel 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.

6.4. 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 hash table 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 prior C-heap profiler that provides such fine-grained per-type information [17].

We used our profiler to improve dynamic update times (Figure 13). The profiler showed what **struct**s had the most instances, which we then manually inspected. If the high-count **struct**s 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 11, line 227.

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 perfaction_ functions and 138,417KB of allocated structures. While *Snort* has a high traversal time overall, the ratio of traversal time to calls to perfaction_* is similar for all three applications: 0.43 μs/call for *Snort*, 0.41μs/call for *redis*, and 0.60μs/call for *memcached*, which is higher because the smaller heap does not amortize the startup and teardown costs.

6.5. Heap assertion checking

Finally, we also used C-strider to implement a simple heap assertion checking system, inspired by Aftandilian and Guyer's GC assertions [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:

DOI: 10.1002/spe

```
if (type == TYPE_linked_list_PTR)
    assert(&in→next→prev == &in);
```

This application has no service-specific code.

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 hash table contains a series of doubly linked lists. Asserting the property above for these lists required essentially the same time as for serialization (Figure 12) 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 fcntl), database item **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 13.

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 [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 1,749 function pointers and 446 other heap objects in 279.98ms.

7. RELATED WORK

There are several threads of related work.

Kitsune. The most closely related work is Kitsune [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, *xfgen*, to customize the traversal, rather than customization via 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: *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 implemented by simply making a new type (with mktyp_) and using the standard 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 13. Finally, C-strider supports parallel traversal, while Kitsune is single-threaded.

Type-directed programming and annotations. In this paper, we used annotations to convey the actual type of heap objects to make type-safe traversal possible. These type annotations are inspired by Deputy [18, 13] and Cyclone [19]. 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 [20] 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, if we "conservatively" treat an **int** as a function pointer that happens to have the same value, we would serialize and deserialize it incorrectly (because the function pointer could be at a different address when deserialized in another process).

DOI: 10.1002/spe

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 [21, 22] also implement state transformation. Similar to state transformation, hot-swapping object instances [23] 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, 24] 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 [25, 26, 3] based on calls to malloc rather than providing type-level granularity like Mihalicza et. al [17] 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.

8. 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

1. Roman E. A survey of checkpoint/restart implementations. *Technical Report*, Lawrence Berkeley National Laboratory, Tech 2002.
2. Litzkow M, Tannenbaum T, Basney J, Livny M. Checkpoint and migration of UNIX processes in the Condor distributed processing system. *Technical Report UW-CS-TR-1346*, University of Wisconsin - Madison Computer Sciences Department April 1997.
3. Google Developer Tools. Heap profiler. http://google-perftools.googlecode.com/svn/trunk/doc/heapprofile.html.

DOI: 10.1002/spe

4. Hayden CM, Saur K, Smith EK, Hicks M, Foster JS. Kitsune: Efficient, general-purpose dynamic software updating for c. *ACM Trans. Program. Lang. Syst.* Oct 2014; **36**(4):13:1–13:38, doi:10.1145/2629460. URL http://doi.acm.org/10.1145/2629460.
5. Petroni NL Jr, Hicks M. Automated detection of persistent kernel control-flow attacks. *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, ACM: New York, NY, USA, 2007; 103–115, doi:10.1145/1315245.1315260. URL http://doi.acm.org/10.1145/1315245.1315260.
6. Aftandilian EE, Guyer SZ. Gc assertions: Using the garbage collector to check heap properties. *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, ACM: New York, NY, USA, 2009; 235–244, doi:10.1145/1542476.1542503. URL http://doi.acm.org/10.1145/1542476.1542503.
7. Reichenbach C, Immerman N, Smaragdakis Y, Aftandilian EE, Guyer SZ. What can the gc compute efficiently?: A language for heap assertions at gc time. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, ACM: New York, NY, USA, 2010; 256–269, doi:10.1145/1869459.1869482. URL http://doi.acm.org/10.1145/1869459.1869482.
8. Arnold M, Vechev M, Yahav E. Qvm: An efficient runtime for detecting defects in deployed systems 2008; :143–162doi:10.1145/1449764.1449776. URL http://doi.acm.org/10.1145/1449764.1449776.
9. Vechev M, Yahav E, Yorsh G. Phalanx: Parallel checking of expressive heap assertions. *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, ACM: New York, NY, USA, 2010; 41–50, doi:10.1145/1806651.1806658. URL http://doi.acm.org/10.1145/1806651.1806658.
10. Jump M, McKinley KS. Dynamic shape analysis via degree metrics. *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, ACM: New York, NY, USA, 2009; 119–128, doi:10.1145/1542431.1542449. URL http://doi.acm.org/10.1145/1542431.1542449.
11. Demsky B, Rinard M. Data structure repair using goal-directed reasoning. *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, ACM: New York, NY, USA, 2005; 176–185, doi:10.1145/1062455.1062499. URL http://doi.acm.org/10.1145/1062455.1062499.
12. Boehm HJ, Weiser M. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.* Sep 1988; **18**(9):807–820, doi:10.1002/spe.4380180902. URL http://dx.doi.org/10.1002/spe.4380180902.
13. Condit J, Harren M, Anderson Z, Gay D, Necula GC. Dependent types for low-level programming. *Proceedings of the 16th European Conference on Programming*, ESOP'07, Springer-Verlag: Berlin, Heidelberg, 2007; 520–535. URL http://dl.acm.org/citation.cfm?id=1762174.1762221.
14. Soules C, Appavoo J, Hui K, Silva DD, Ganger G, Krieger O, Stumm M, Wisniewski R, Auslander M, Ostrowski M, *et al.*. System support for online reconfiguration. 2003.
15. Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. *J. ACM* Sep 1999; **46**(5):720–748, doi:10.1145/324133.324234. URL http://doi.acm.org/10.1145/324133.324234.
16. Necula GC, McPeak S, Rahul SP, Weimer W. Cil: Intermediate language and tools for analysis and transformation of c programs. *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, Springer-Verlag: London, UK, UK, 2002; 213–228.
17. Mihalicza J, Porkoláb Z, Gabor A. Type-preserving heap profiler for C++. *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, IEEE Computer Society: Washington, DC, USA, 2011; 457–466, doi:10.1109/ICSM.2011.6080813. URL http://dx.doi.org/10.1109/ICSM.2011.6080813.
18. Zhou F, Condit J, Anderson Z, Bagrak I, Ennals R, Harren M, Necula G, Brewer E. Safedrive: Safe and recoverable extensions using language-based techniques. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, USENIX Association: Berkeley, CA, USA, 2006; 45–60. URL http://dl.acm.org/citation.cfm?id=1298455.1298461.
19. Jim T, Morrisett JG, Grossman D, Hicks MW, Cheney J, Wang Y. Cyclone: A safe dialect of C 2002; :275–288URL http://dl.acm.org/citation.cfm?id=647057.713871.
20. Hinze R, Löh A. Generic programming, now! *Proceedings of the 2006 International Conference on Datatype-generic Programming*. SSDGP'06, Springer-Verlag: Berlin, Heidelberg, 2007; 150–208. URL http://dl.acm.org/citation.cfm?id=1782894.1782897.
21. Neamtiu I, Hicks M, Stoyle G, Oriol M. Practical dynamic software updating for C. *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, ACM: New York, NY, USA, 2006; 72–83, doi:10.1145/1133981.1133991. URL http://doi.acm.org/10.1145/1133981.1133991.
22. Makris K, Bazzi RA. Immediate multi-threaded dynamic software updates using stack reconstruction. *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, USENIX Association: Berkeley, CA, USA, 2009; 31–31. URL http://dl.acm.org/citation.cfm?id=1855807.1855838.
23. Baumann A, Heiser G, Appavoo J, Da Silva D, Krieger O, Wisniewski RW, Kerr J. Providing dynamic update in an operating system. *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, USENIX Association: Berkeley, CA, USA, 2005; 32–32. URL http://dl.acm.org/citation.cfm?id=1247360.1247392.
24. Ferrari A, Chapin SJ, Grimshaw A. Heterogeneous process state capture and recovery through process introspection. *Cluster Computing* Apr 2000; **3**(2):63–73, doi:10.1023/A:1019067801346. URL http://dx.doi.org/10.1023/A:1019067801346.
25. massif. http://valgrind.org/info/tools.html.
26. Freyther H. memprof. http://www.secretlabs.de/projects/memprof/.