Complementary directions for Truffle and liballocs

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge
So you’ve implemented a Truffle language…

You probably care about

- interop
- interop-enabled tools

We can probably do

- your language ↔ another Truffle language

What about

- your language ↔ native code?
- your language ↔ some other VM?
Quick summary of liballocs

Baseline infrastructure should be Unix(-like) process

- *not* VM-level mechanisms
- embrace native code
- embrace *other* VMs

liballocs is a runtime (+ tools) for

- extending Unix processes with in(tro)spection
- via a whole-process meta-level protocol
- $\approx$ “typed allocations”
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,

        (struct commit *)obj))
        return -1;
    return 0;
}
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker, 
        (assert(_is_a(obj, "struct_commit")), 
        (struct commit *)obj))
        return -1;
    return 0;
}
Making Unix processes more introspectable

```c
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,
                       (assert(_is_a(obj, "struct_commit")),
                       (struct commit *)obj))
        return -1;
    return 0;
}
```

Entails a runtime that can

- track *allocations*
- with type info
- efficiently
- language-agnostically?
Making native code more introspectable, efficiently

- exploit debugging info
- some source-level analysis for C
- add efficient disjoint metadata
- implementation is roughly per allocator
- mostly link- and run-time intervention

It works!

- one application: checking stuff about C code...
- another: as primitive for interop!
var ffi = require("node-ffi");

var libm = new ffi.Library("libm", { "ceil": [ "double", [ "double" ] ] });
libm.ceil(1.5); // 2

// You can also access just functions in the current process
var current = new ffi.Library(null, { "atoi": [ "int32", [ "string" ] ] });
current.atoi("1234"); // 1234
process.lm.ceil(1.5) // 2
process.lm.atoi("1234"); // 1234

/* Widget XtInitialize(String shell_name, String app_class,
     XrmOptionDescRec* options, Cardinal num_options,
     int* argc, char** argv) */

process.lm.dlopen("/usr/local/lib/libXt.so.6", 257)
var toplvl = process.lm.XtInitialize ( 
    process.argv[0], "simple", null, 0,
    [process.argv.length], process.argv
);
Goal: also make language runtimes more *transparent*. Why?

- bi-directional interop
- be transparent to whole-process tools (*gdb*, *perf*, …)

Means *retrofitting* VMs onto *liballocs*

- + some extra tool support needed

Designed to make this easy…
liballocs core: a simple meta-level allocator protocol

```c
struct uniqtype;    /* reified type */
struct allocator;   /* reified allocator */
uniqtype * alloc_get_type (void *obj);    /* what type? */
allocator * alloc_get_allocator (void *obj);    /* heap/stack? etc */
void * alloc_get_site (void *obj);    /* where allocated? */
void * alloc_get_base (void *obj);    /* base address? */
void * alloc_get_limit (void *obj);    /* end address? */
Dl_info alloc_dladdr (void *obj);    /* dladdr−like */
```

An object model, but not as we know it:

- (ideally) implemented across whole process
- embrace *plural* (many heaps)
- embrace *diversity* (native, VMs, ...)
Reifying data types at run time

```c
struct ellipse {
    double maj, min;
    struct { double x, y; } ctr;
};
```

- use the linker to keep them unique
- → “exact type” test is a pointer comparison
- `is_a()` is a short search
Disjoint metadata example: `malloc` heap index

- Index by high-order bits of virtual address
- Entries are one byte, each covering 512B of heap
- Interior pointer lookups may require backward search
- Pointers encoded compactly as local offsets (6 bits)
- Instrumentation adds a trailer to each heap chunk

Entries are one byte, each covering 512B of heap.
Helping liballocs grok native code

LIBALLOCS_ALLOC_FNS="xmalloc(Z)p xmalloc(Z)p xrealloc(pZ)p"
LIBALLOCS_SUBALLOC_FNS="ggc_alloc(Z)p ggc_alloc_cleared(Z)p"

export LIBALLOCS_ALLOC_FNS
export LIBALLOCS_SUBALLOC_FNS

allocscc -o myprog ... # call host compiler, postprocess metadata
Hierarchical model of allocations

- `mmap()`, `sbrk()`
  - `libc malloc()`
  - `custom malloc()`
  - `custom heap (e.g. Hotspot GC)`
    - `obstack (+ malloc)`
    - `gslice`
      - `client code`

...
<table>
<thead>
<tr>
<th>bench</th>
<th>normal/s</th>
<th>liballocs/s</th>
<th>liballocs %</th>
<th>no-load</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>4.91</td>
<td>5.05</td>
<td>+2.9%</td>
<td>+1.6%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.985</td>
<td>1.85</td>
<td>+88%</td>
<td>– %</td>
</tr>
<tr>
<td>gobmk</td>
<td>14.2</td>
<td>14.6</td>
<td>+2.8%</td>
<td>+0.7%</td>
</tr>
<tr>
<td>h264ref</td>
<td>10.1</td>
<td>10.6</td>
<td>+5.0%</td>
<td>+5.0%</td>
</tr>
<tr>
<td>hmmer</td>
<td>2.09</td>
<td>2.27</td>
<td>+8.6%</td>
<td>+6.7%</td>
</tr>
<tr>
<td>lbm</td>
<td>2.10</td>
<td>2.12</td>
<td>+0.9%</td>
<td>(−0.5%)</td>
</tr>
<tr>
<td>mcf</td>
<td>2.36</td>
<td>2.35</td>
<td>(−0.4%)</td>
<td>(−1.7%)</td>
</tr>
<tr>
<td>milc</td>
<td>8.54</td>
<td>8.29</td>
<td>(−3.0%)</td>
<td>+0.4%</td>
</tr>
<tr>
<td>perlbench</td>
<td>3.57</td>
<td>4.39</td>
<td>+23%</td>
<td>+1.6%</td>
</tr>
<tr>
<td>sjeng</td>
<td>3.22</td>
<td>3.24</td>
<td>+0.6%</td>
<td>(−0.7%)</td>
</tr>
<tr>
<td>sphinx3</td>
<td>1.54</td>
<td>1.66</td>
<td>+7.7%</td>
<td>(−1.3%)</td>
</tr>
</tbody>
</table>
Why Truffle + liballocs?

Lots of languages!

- more languages $\rightarrow$ more fragmentation
- need interop and cross-language tooling

Heresy: one VM can’t quite rule them all

- inevitably, native code (asm, Fortran, C++, …)
- inevitably, other VMs

$\rightarrow$ want a deeper basis for tools & interop

- Truffle ecosystem offers $>1$ good basis for exploring
TruffleC versus a liballocs approach to natives

- no need to wait for Truffle impl of all languages
- shared metamodel right down to native level

... but: no interprocedural optimisation

- conceivable, perhaps Dynamo-style
- natives’ type information available at run time
Not just about natives

Want to make Truffle languages transparent to liballocs

- implement the metaprotocol!
- also requires unwind support

Interested to learn

- what allocators/GCs are Truffle languages using?
- what metadata are Truffle languages keeping?
- synergy with Substrate ↔ Truffle langs

Likely benefits

- native interop, incl. embeddability into C/C++ programs
- help with native tools (gdb, perf etc.)
Pushing whole-process queries down into generated code

**JS property access via inline cache, currently:**

```
cmp [ebx,<class offset>],<cached class>; test
jne <inline cache miss>; miss? bail
mov eax,[ebx, <cached x offset>] ; hit; do load
```

**Same but “allocator-guarded” + slow/general path:**

```
xor ebx,<allocator mask>; get allocator
cmp ebx,<cached allocator prefix>; test
jne <allocator miss>; miss? bail
cmp [ebx,<class offset>],<cached class>; test class
jne <cached cache miss>; miss? bail
mov eax,[ebx, <cached x offset>] ; hit! do load
```

**Slow path goes via liballocs metaprotocol**
liballocs is a whole-process introspection infrastructure

- cross-language shared metamodel
- per-allocator API implementation
- good support for real/complex native code
- intended to be easy to retrofit VMs onto
- can help native interop now
- can help cross-VM/lang interop with some work!

Code is here: https://github.com/stephenrkell/

- look out for paper at Onward! later this year

Please ask questions!