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
- ≈ “typed allocations”

Making Unix processes more introspectable

```
if  (obj->type == OBJ_COMMIT) {
  if  (process_commit(walker,

           (struct commit *)obj))
    return -1;
  return 0;
}
```

Making Unix processes more introspectable

```
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

```
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!

Interop: what we *don't* want

```
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
```

No more FFIs…

```
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
);
```

Not only native interop

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

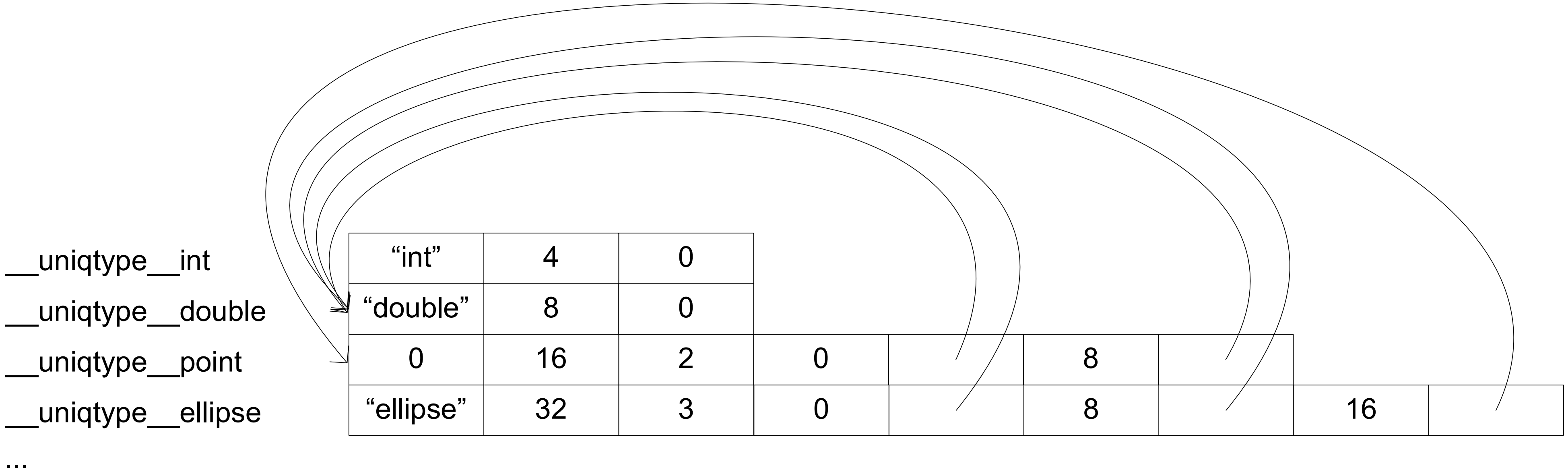
```
struct uniqtype;                                           /* reified type */
struct allocator ;                                         /* reified allocator */
uniqtype *    alloc_get_type        (void *obj);   /* what type? */
allocator  *  alloc_get_allocator (void *obj);   /* heap/stack? etc */
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 *plurality* (many heaps)
- embrace *diversity* (native, VMs, . . .)

Reifing data types at run time

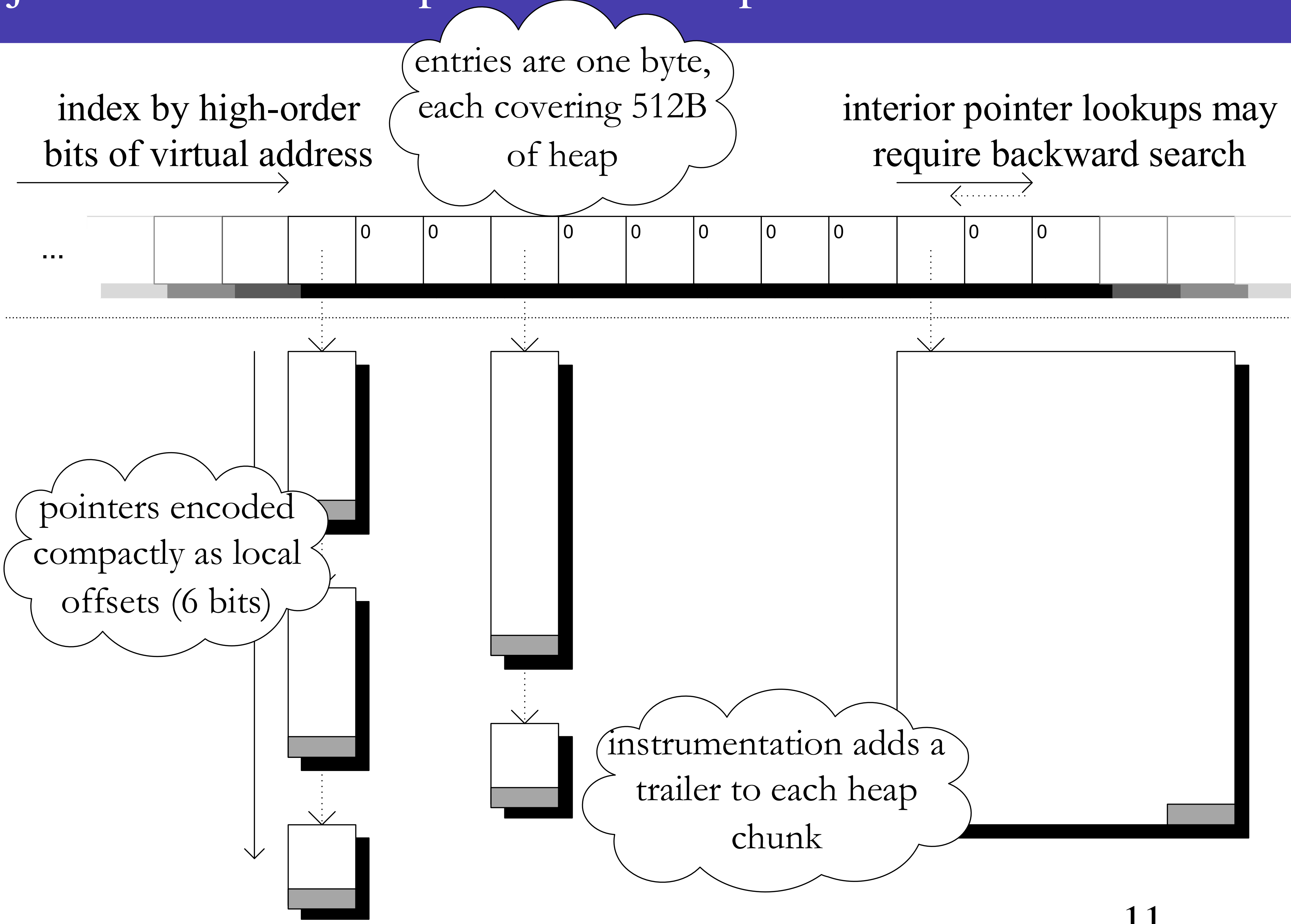
```
struct ellipse {
    double maj, min;
    struct { double x, y; } ctr ;
};
```



__uniqtype__int	"int"	4	0						
__uniqtype__double	"double"	8	0						
__uniqtype__point	0	16	2	0		8			
__uniqtype__ellipse	"ellipse"	32	3	0		8		16	
...									

- 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

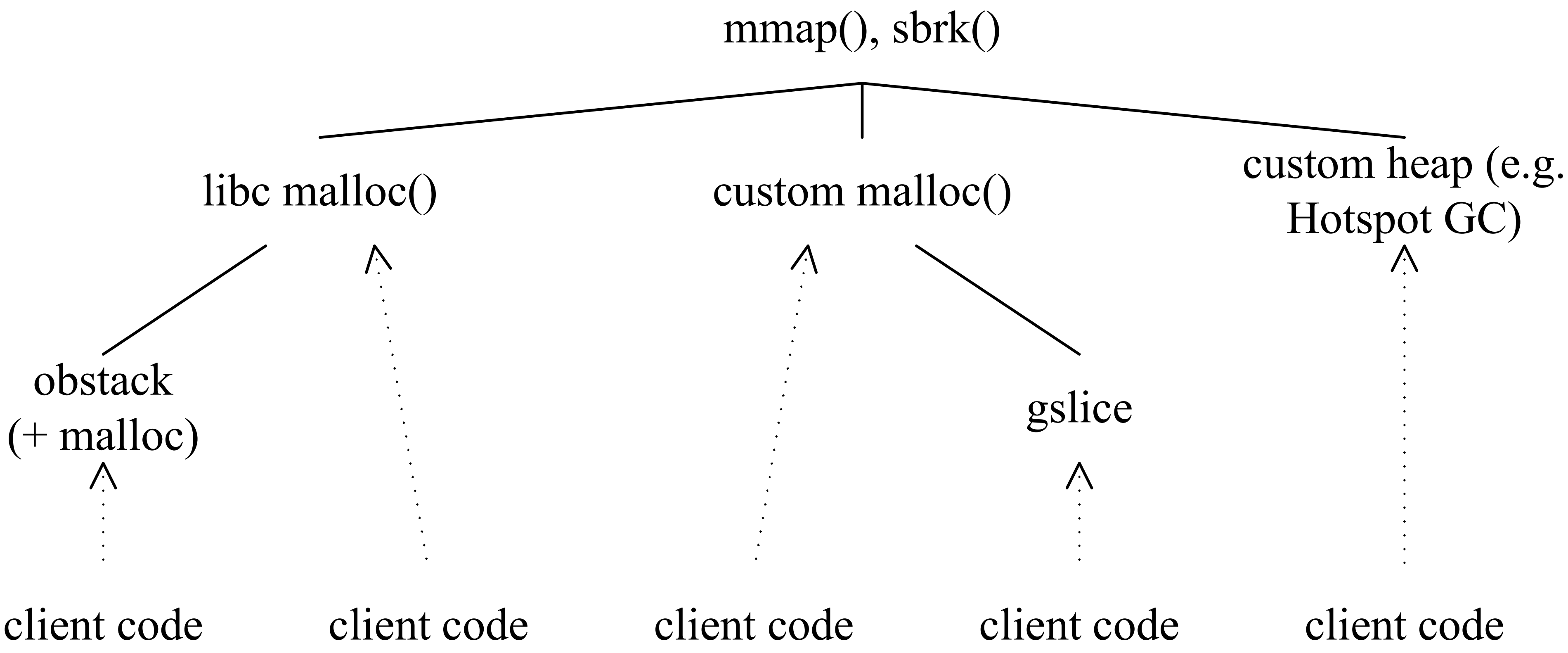
pointers encoded compactly as local offsets (6 bits)

instrumentation adds a trailer to each heap chunk

Helping liballocs grok native code

```
LIBALLOCS_ALLOC_FNS="xcalloc(zZ)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

client code

client code

client code

client code

liballocs vs C-language SPEC CPU2006 benchmarks

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

Why Truffle + liballocs?

Lots of languages!

- more languages → 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

→ 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

Conclusions

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!