Tachyon's GC

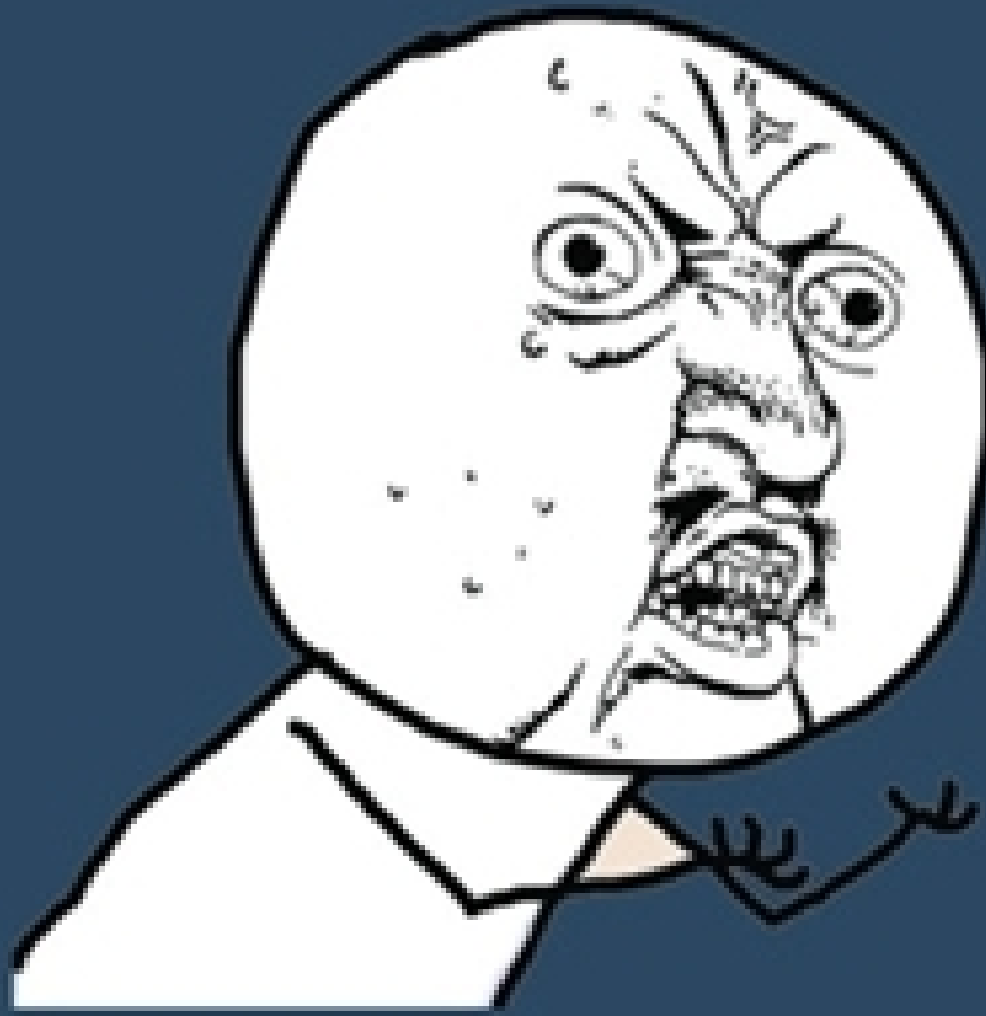
Maxime Chevalier-Boisvert

Jan. 11, 2012

Tachyon needs a GC

- JS without GC is only a toy implementation
- Previously, allocator bumps pointer to allocate
  - If we run out of heap space, failure happens
  - Allocate big heap, hope we have enough heap space
  - Long-running benchmarks not possible
- To bootstrap, Tachyon needed ~50GB of allocations
- Tachyon should be able to bootstrap on a "normal" desktop
  - ~4GBs RAM
- Basic GC needs
  - Collect dead objects when the heap is full
  - Speed is not extremely critical

TACHYON



Y U NO BOOTSTRAP IN 32 BIT?


COLLECT
ALL THE THINGS!
memegenerator.net

KISS: simplifying our work

- Stop-the-world, semi-space, compacting GC
  - Cheney's algorithm
- Fixed-size, contiguous heap
  - Pre-allocate one big block of memory with malloc
- The C world cannot hold references to Tachyon objects
  - By design, this should never be needed
- Some objects are not currently collected
  - Context lives outside GC'd heap
  - No special handling of string table (weak references)
  - Machine code is not collected
- Are these issues?
  - Not difficult to improve
  - Not a problem for our current applications

Tachyon compiles its own GC

- Runtime code already written in extended JS
- Already have bulk of code to allocate/touch Tachyon objects
  - Auto-generated based on object layout info
- Could have written the GC in C
  - Trying to avoid this by design
  - Don't want redundant code in C
- GC in extended JS ties well into current design
  - GC can use getters/settters to access memory object
  - Can auto-generate layout visit functions for GC
- Result: fairly simple implementation
  - runtime/gc.js -> 961 lines
  - gc code generation code -> ~200 lines
  - Implementation + debugging time: ~2 weeks


COOL
HIGH
OFF
LOW

Debugging

- Mark's wise words of warning: GCs are notoriously bug-prone and difficult to debug
  - Mean bugs that will make you cry yourself to sleep
- Testing strategy
  - Write representative unit tests
    - Graph updating, mini VM, corner cases
  - Make sure to use standard library
  - Make sure to use all weird calling conventions
  - Shrink heap, force many collections (30+)
- Debugging tools
  - Many many assertions, self-checking
    - e.g.: can't allocate in heap during GC
  - iir.trace_print("string"), printInt(intVal)
  - GCC's mprobe

Implementation details

- Allocations go through heapAlloc(size)
  - Checks if space is available, if not, calls gcCollect()
- GC traverses roots, copies references objects
  - Context object -> gc_visit_ctx(), auto-generated
  - Dynamic stack traversal code, top-to-bottom
- References come in 3 kinds:
  - boxed values (tag bits + payload)
  - reference values (heap object pointer, no tag bits)
  - raw pointers (pointer to data outside of heap)
- Meta-info encoded in machine code blocks
  - Stack frame format after return address
  - Location of references, pointers inside MCB at end

Stack traversal (1/2)

- Has multiple potential applications
  - GC
  - Exceptions
  - Debugger, stack-frame introspection
  - On-stack replacement
- Need to know
  - Location and type of references in a stack frame
  - Offset to the next stack frame down
  - Know return address, base pointer of topmost frame
    - iir.get_ra(), iir.get_bp()
- For now, simple approach
  - call fptr, jmp POST_INFO, <... info bits ...>, POST_INFO:
  - Attempt to benefit from hardware ret branch prediction
  - Ultimately : external table hashing on ra probably best
    - Less instruction cache interference

Stack traversal (2/2)

- Backend knows call conv., encodes stack frame info
  - Info written after call instruction
  - Info is about the caller function's frame
- Stack frame info:
  - Magic number: 1337 (16-bit)
  - Call align/pad space (16-bit)
    - May be used to align callee stack frame
  - Num stack slots (16-bit)
  - Return address slot index (16-bit)
  - * [kind: other:0, rptr:1, ref:2, box:3 (2 bits)]
  - Function name (ASCII string, for debugging)
- But JS is more complicated than your ex:
  - call_apply -> pushes caller sp under args
  - arguments -> special call to createArgTable

Performance

- Performance not critical, but still interesting
- To collect a basic heap:
  - All primitive functions
  - All stdlib objects
  - All strings
  - Simple test program
- Collection time ~5ms
- Time includes
  - All assertions
  - Some amount of console output
- Performance fairly good, acceptable for our purposes
- Surprisingly unslow?
  - GC coded with typed pointers, integers
  - JavaScript only in syntax