Encoding for Objects Matters

Dave Mason
Toronto Metropolitan University

Toronto Metropolitan University

Memory is Slow

Memory is Slow

- at least relatively slow

Memory is Slow

- at least relatively slow

Storage Type	Slowed Time Scale	Real Time Scale
Single CPU Instruction (at 3 GHz)	1 second	0.3 nSec

-

Memory is Slow

- at least relatively slow

Storage Type	Slowed Time Scale	Real Time Scale
Single CPU Instruction (at 3 GHz)	1 second	0.3 nSec
Registers (storage for active instructions)	1 to 3 seconds	0.3 to 1 nSec

-

Memory is Slow

- at least relatively slow

Storage Type	Slowed Time Scale	Real Time Scale
Single CPU Instruction (at 3 GHz)	1 second	0.3 nSec
Registers (storage for active instructions)	1 to 3 seconds	0.3 to 1 nSec
Memory Caches	2 to 12 seconds	0.7 to 4 nSec

Memory is Slow

- at least relatively slow
-

Storage Type	Slowed Time Scale	Real Time Scale
Single CPU Instruction (at 3 GHz)	1 second	0.3 nSec
Registers (storage for active instructions)	1 to 3 seconds	0.3 to 1 nSec
Memory Caches	2 to 12 seconds	0.7 to 4 nSec
Main System Memory (RAM)	30 to 60 seconds	10 to 20 nSec

Memory is Slow

- at least relatively slow

Storage Type	Slowed Time Scale	Real Time Scale
Single CPU Instruction (at 3 GHz)	1 second	0.3 nSec
Registers (storage for active instructions)	1 to 3 seconds	0.3 to 1 nSec
Memory Caches	2 to 12 seconds	0.7 to 4 nSec
Main System Memory (RAM)	30 to 60 seconds	10 to 20 nSec
NVMe SSD	3 to 11 **days**	100 to 200 uSec

Memory is Slow

- at least relatively slow
Storage Type	Slowed Time Scale	Real Time Scale
Single CPU Instruction (at 3 GHz)	1 second	0.3 nSec
Registers (storage for active instructions)	1 to 3 seconds	0.3 to 1 nSec
Memory Caches	2 to 12 seconds	0.7 to 4 nSec
Main System Memory (RAM)	30 to 60 seconds	10 to 20 nSec
NVMe SSD	3 to 11 **days**	100 to 200 uSec
- modern machines get a lot of their preformance from deep pipelines

Memory is Slow

- at least relatively slow
Storage Type	Slowed Time Scale	Real Time Scale
Single CPU Instruction (at 3 GHz)	1 second	0.3 nSec
Registers (storage for active instructions)	1 to 3 seconds	0.3 to 1 nSec
Memory Caches	2 to 12 seconds	0.7 to 4 nSec
Main System Memory (RAM)	30 to 60 seconds	10 to 20 nSec
NVMe SSD	3 to 11 **days**	100 to 200 uSec
- modern machines get a lot of their preformance from deep pipelines
- memory references to executable addresses stall pipelines

Types

- untyped
  - assembler, BCPL, B
  - only machine instruction controls interpretation of bits

Types

- untyped
  - assembler, BCPL, B
  - only machine instruction controls interpretation of bits
- static typing
  - C, Rust, Zig, C++ Java
  - variables, functions, expressions have type
  - compile-time overload/operator polymorphism
  - allows operators like +, -, *, and others to behave differently based on the data types of their operands

Types

- untyped
  - assembler, BCPL, B
  - only machine instruction controls interpretation of bits
- static typing
  - C, Rust, Zig, C++ Java
  - variables, functions, expressions have type
  - compile-time overload/operator polymorphism
  - allows operators like +, -, *, and others to behave differently based on the data types of their operands
- dynamic typing
  - Smalltalk, Python, JavaScript
  - values, expressioins have type
  - run-time overload/operator polymorphism

Encoding Considerations for Objects

- many implementation, from byte interpreters to threaded execution to JIT'ed native code

Encoding Considerations for Objects

- many implementation, from byte interpreters to threaded execution to JIT'ed native code
- encoding choices will affect all of the implementations, to varying degrees

Encoding Considerations for Objects

- many implementation, from byte interpreters to threaded execution to JIT'ed native code
- encoding choices will affect all of the implementations, to varying degrees
- **Determining types**
  - operations must be parameterized at run time by the types of the values
  - the first step is to access the types of the values - may slow pipeline
  - types in memory can cause a pipeline stall until they can be loaded
  - may be shifting/masking operations required to access types

Encoding Considerations for Objects

- many implementation, from byte interpreters to threaded execution to JIT'ed native code
- encoding choices will affect all of the implementations, to varying degrees
- **Determining types**
  - operations must be parameterized at run time by the types of the values
  - the first step is to access the types of the values - may slow pipeline
  - types in memory can cause a pipeline stall until they can be loaded
  - may be shifting/masking operations required to access types
- **Accessing values**
  - In order to perform operations, the values must be made available to the CPU.
  - accessing values in memory is an order of magnitude slower than accessing them in registers
  - even if the values happen to be in cache they will be several times slower than in registers
  - may be shifting/masking operations required to access values

Encoding Considerations for Objects

- many implementation, from byte interpreters to threaded execution to JIT'ed native code
- encoding choices will affect all of the implementations, to varying degrees
- **Determining types**
  - operations must be parameterized at run time by the types of the values
  - the first step is to access the types of the values - may slow pipeline
  - types in memory can cause a pipeline stall until they can be loaded
  - may be shifting/masking operations required to access types
- **Accessing values**
  - In order to perform operations, the values must be made available to the CPU.
  - accessing values in memory is an order of magnitude slower than accessing them in registers
  - even if the values happen to be in cache they will be several times slower than in registers
  - may be shifting/masking operations required to access values
- **Supporting Memory Management**
  - dynamic languages invariably have automatic memory management, (reference counting or garbage collection)
  - objects need to be allocated, and removed when no longer needed
  - both actions have a significant cost, and the more objects allocated in memory, the greater the cost

Catalog of Encodings

- all mutable object need to be in memory
- many possible ways to encode the range of immutable values that arise in normal calculations
- we discuss 5 possible encodings that encompass the range from all values being in memory, to treating as many things as immadiate as possible
- we examine each from the perspective of the three considerations

Every Object in Memory

- This is the simplest model.
- Determining the value type requires a memory access, as does accessing the actual value. Interoperation with foreign functions is complicated.
- The result of every operation must be allocated to memory.
- This will create a huge amount of churn, although most of the values wil be very short-lived.
- Immutable values ~~nil~~, ~~true~~, ~~false~~, and ASCII Characters are allocated at fixed addresses (Special Objects array)

Every Object in Memory with SmallInteger Cache

- This is a simple optimization by statically defining a set of small integers (say -5..100).
- This means that if a result is in this pre-allocated set, then no allocation or collection is required.

Tag *SmallInteger*

- Early Lisp and Smalltalk implementers noticed that the vast majority of values that were created were for *SmallInteger* objects.
- Most such systems tag small integers with a 1 bit tag (either in the low bit leaving all other addresses natural by aligning all objects on at least a word boundary, or (less commonly) using the sign bit).
- This encoding was so important that the SPARC architecture had basic arithmetic instructions that checked that the low bit of the parameters to verify they were flagged as integers.
- A small drawback of this encoding is that 1 bit of precision is lost, but most such systems move automatically to big integers with unbounded precision on overflow and on modern, 64-bit systems that one bit of precision is fairly irrelevant.

Tag *SmallInteger*, *Character*, *Float*

- Miranda/Bera created the SPUR encoding
- cleverly extends the 1-bit tag to 3 bits and in addition to *SmallInteger* and general (memory) objects, supports encoding for Unicode characters and a subset of *Float*.
- Examining the low 3 bits of an object, a coding of 0 is a pointer to a memory object, 1 is a 61-bit *SmallInteger*, 2 is a Unicode *Character*, and 4 is a *Float*.
- Only the pointer tag is a natural value - all the others require some decoding before use.
- As with tagging reducing precision on integers, the Spur encoding limits the range of floating point values.

NaN Encoding

- NaN encoding utilizes the large number of code points in the IEEE-7544 floating point encoding that do not represent valid floating-point numbers, by using those bit patterns to represent values of other types, including pointers and *SmallInteger*.
- This encoding has been used by Spidermonkey and was the originally planned encoding for Zag Smalltalk
- This encoding supports, within 64 bits, all *Float* values naturally, as well as 51-bit *SmallInteger* values, pointers to memory objects, *nil*, booleans, symbols, characters, as well as several common *BlockClosure* values.

Tag Most Possible Values

- Zag Smalltalk now uses a modified-Spur encoding.
- It uses the bottom 3 bits of an object: 0 to naturally encode pointers to memory objects (and *nil*); 1 to encode 31 immediate immutable classes; 2-7 to encode a broader range of *Float* values than encoded by Spur.
- The immediate classes include 56-bit *SmallInteger* values, booleans, symbols, Unicode *Character*.
- The immediate classes also include 13 kinds of special block closures, including some non-local returns, that are common in existing code.

Experiment

- we are currently implementing all 6 of these encodings
- initial experiment with integer and float fibonacci
- will have inlined and non-inlined versions
- will exercise the integer and floating-point paths, and the version with no inlining will exercise the *BlockClosure* creation and non-local return
- unfortunately, don't yet have results

Limitations

- Zag design features other than the choice of encoding will influence some of the runtimes
- In particular, Zag has a three-tier memory system: stack, nursery, and global heap, and references are only allowed to go to the right, but none of the test runs should hit the global heap, and the nursery is a copying collector, so should be quite efficient.
- Zag does no inlining of special selectors, so the “no inlining” versions will create actual *BlockClosure* s.
- The non-local return block is particularly tricky, because it references the context from which it must return, non-immediate closures can still be stack allocated, but create some extra work and stack pressure
- This may lead to significant churn in the first 4 cases.
- The last 2 encodings create immediate values for this block, so do not force any context spilling, and may, in fact, create no nursery allocations at all.

Hypothesis

- **Every Object in Memory**
  - The result of every addition will allocate in the nursery, which will create a great deal of churn for the numbers

Hypothesis

- **Every Object in Memory**
  - The result of every addition will allocate in the nursery, which will create a great deal of churn for the numbers
- **Every Object in Memory with SmallInteger Cache**
  - The first 11 fibonacci numbers are in the cache we pre-allocated, and those are the most frequently generated values, so the integer versions of the benchmark should run significantly faster.
  - The floating-point version will have no speedup.

Hypothesis

- **Every Object in Memory**
  - The result of every addition will allocate in the nursery, which will create a great deal of churn for the numbers
- **Every Object in Memory with SmallInteger Cache**
  - The first 11 fibonacci numbers are in the cache we pre-allocated, and those are the most frequently generated values, so the integer versions of the benchmark should run significantly faster.
  - The floating-point version will have no speedup.
- **Tag** *SmallInteger*
  - Now all integers (for the test) have immediate values, so the integer tests should speed up significantly.
  - The floating-point version will have no speedup.

Hypothesis

- **Every Object in Memory**
  - The result of every addition will allocate in the nursery, which will create a great deal of churn for the numbers
- **Every Object in Memory with SmallInteger Cache**
  - The first 11 fibonacci numbers are in the cache we pre-allocated, and those are the most frequently generated values, so the integer versions of the benchmark should run significantly faster.
  - The floating-point version will have no speedup.
- **Tag** *SmallInteger*
  - Now all integers (for the test) have immediate values, so the integer tests should speed up significantly.
  - The floating-point version will have no speedup.
- **Tag** *SmallInteger* **,** *Character* **,** *Float*
  - Because SPUR encodes immediate values for all of the generated floating-point values, the *Float* version of this should run significantly faster.
  - The integer version should experience no speedup.

Hypothesis

- **Every Object in Memory**
  - The result of every addition will allocate in the nursery, which will create a great deal of churn for the numbers
- **Every Object in Memory with SmallInteger Cache**
  - The first 11 fibonacci numbers are in the cache we pre-allocated, and those are the most frequently generated values, so the integer versions of the benchmark should run significantly faster.
  - The floating-point version will have no speedup.
- **Tag** *SmallInteger*
  - Now all integers (for the test) have immediate values, so the integer tests should speed up significantly.
  - The floating-point version will have no speedup.
- **Tag** *SmallInteger* **,** *Character* **,** *Float*
  - Because SPUR encodes immediate values for all of the generated floating-point values, the *Float* version of this should run significantly faster.
  - The integer version should experience no speedup.
- **NaN Encoding**
  - NaN encoding has the floating-point already in the correct format, therefore this should have the fastest floating point version
  - integer versions will be somewhat slower (more manipulation is required to retrieve the integral value)
  - Because the blocks are immediate values, the “no-inlining” version should speed up noticeably

Hypothesis

- **Every Object in Memory**
  - The result of every addition will allocate in the nursery, which will create a great deal of churn for the numbers
- **Every Object in Memory with SmallInteger Cache**
  - The first 11 fibonacci numbers are in the cache we pre-allocated, and those are the most frequently generated values, so the integer versions of the benchmark should run significantly faster.
  - The floating-point version will have no speedup.
- **Tag** *SmallInteger*
  - Now all integers (for the test) have immediate values, so the integer tests should speed up significantly.
  - The floating-point version will have no speedup.
- **Tag** *SmallInteger* **,** *Character* **,** *Float*
  - Because SPUR encodes immediate values for all of the generated floating-point values, the *Float* version of this should run significantly faster.
  - The integer version should experience no speedup.
- **NaN Encoding**
  - NaN encoding has the floating-point already in the correct format, therefore this should have the fastest floating point version
  - integer versions will be somewhat slower (more manipulation is required to retrieve the integral value)
  - Because the blocks are immediate values, the “no-inlining” version should speed up noticeably
- **Tag Most Possible Values**
  - Zag native format may be very slightly slower than Spur for inlined integer
  - should also be a bit faster than Spur for floating point, because the decoding is a couple of instructions faster
  - Again, the blocks are immediate values, the “no-inlining” version should be comparable to the NaN-encoding

Conclusions

- there is a large design space for object encoding
- can be a memory-first or an immediate-first design philosophy
- lots of potential implications
- **Future work** is required to qualify these implications

Questions?

dmason@torontomu.ca

https://github.com/Zag-Research/Zag-Smalltalk