appliscale

The Yin and Yang of Mutability

Péter Gömöri

CodeBEAM SF - San Francisco (6 March 2020)

We help companies scale

appliscale

Experts in:

- BEAM applications
- DevOps: CICD pipelines, Cloud infrastructure
- Adtech/Martech custom development
- Digital transformation

verizon media

STRÖER | labs

NOVOMATIC

Yin and Yang

Atomic counters ⟷ Persistent terms

very mutable ⟷ very immutable

Philosophy and history

Genesis

In the beginning there was Pure Erlang...

- All data was immutable
- No data was shared

...and it was good

Advantages

- Much easier concurrency
- Easier GC (no forward pointers)
- Simpler to understand and reason about
  (with the right mindset :))

Price:

- Lots of mem alloc and copying

Compromise for performance

But human was greedy and wanted more...

- Write intensive ⇒ mutability
- Read intensive ⇒ share

Cracks in the wall

- Off-heap binaries: shared, can be appended in place
  (if only 1 reference)
- ETS: table abstraction, no process overhead
  (still copy to/from, still could be a GenServer)
- ETS counters: "atomic" update of values (no get+put)

Nicely wrapped abstractions besides a consistent language

ETS counters, still not enough

- My experience: folsom 5% cpu
- Constant overhead: table indirection, temp allocs for new value, etc
- Concurrency overhead: lock contention
  even if write_concurrecy=true
- Irina Guberman CBSF 2018, Andy Till’s oneup lib

Atomics, Counters

Atomics

- 64 bit ints, signed/unsigned, overflow/underflow
- Mutable in place
- Shared (not owned by 1 proc, similar to off-heap bins)
- All ops atomic: eg.: `add_get`, `exchange`, `compare_exchange`
- Array of independent values

```
atomics:new(Arity, [])
```

Atomics internals

- No locks, no memory barriers (native CPU instr’s)
- Exposes already existing ERTS internal API (OTP 18, R15B01)
- Used by eg:
  - reference type
  - time handling
  - internal metrics (sched util, io bytes)

Counters

- 1 step higher abstraction
- smaller API: **`add`**, **`sub`**, **`get`**, **`put`**
- backend: atomics or write_concurrency

Write-concurrency counters

Further optimised for efficient concurrent writes

- 1 atomics per scheduler + 1 for base value
  (each in separate cache lines)
- Price: read inconsistency
- Only writes atomic

Atomics/counters reference

- `atomics:new` ⇒ magic ref
- How all procs get counter ref?
- Unnamed - need registry

Persistent terms

Persistent terms

- Globally shared key-value store
  - Concurrent, constant time reads (no locks, no copy)
  - Slow and globally expensive writes
- Simple API: put/get/erase
- Store unchanging: unnamed references, config structures, flags, feature switches

Literals

- Code constants

```
f(A) ->
  Lit = [11, <<"str">>],
  {A, Lit}.
```

```
{test_heap,3,1},
{put_tuple,2,{x,1}},
{put,{x,0}},
{put,{literal,[11,<<"bin">>]}},
{move,{x,1},{x,0}},
return]},
```

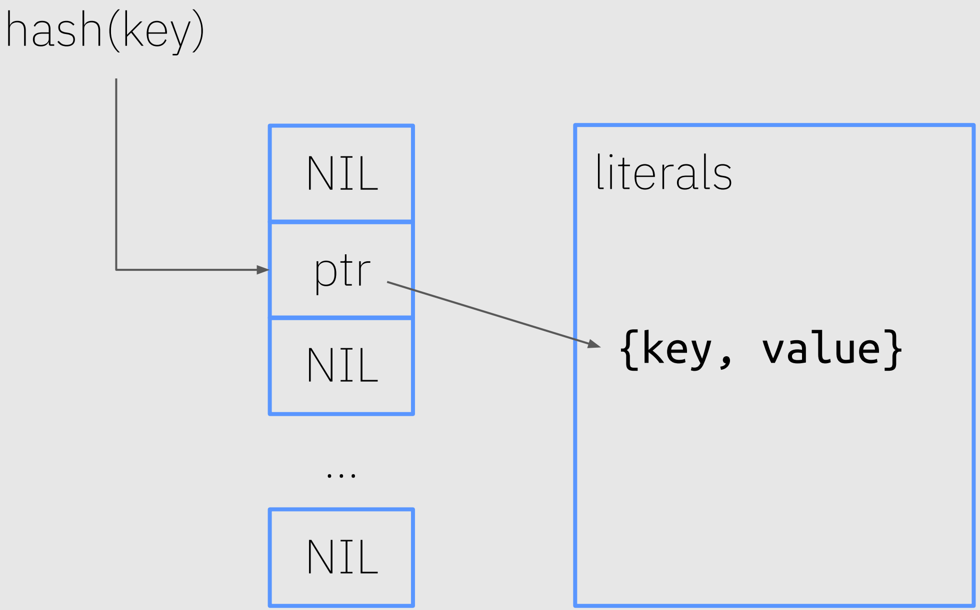
Literals

- Code load time: per module constant pool
- No-copy usage
- Module unload:
  - free constant pool
  - requires some housekeeping...

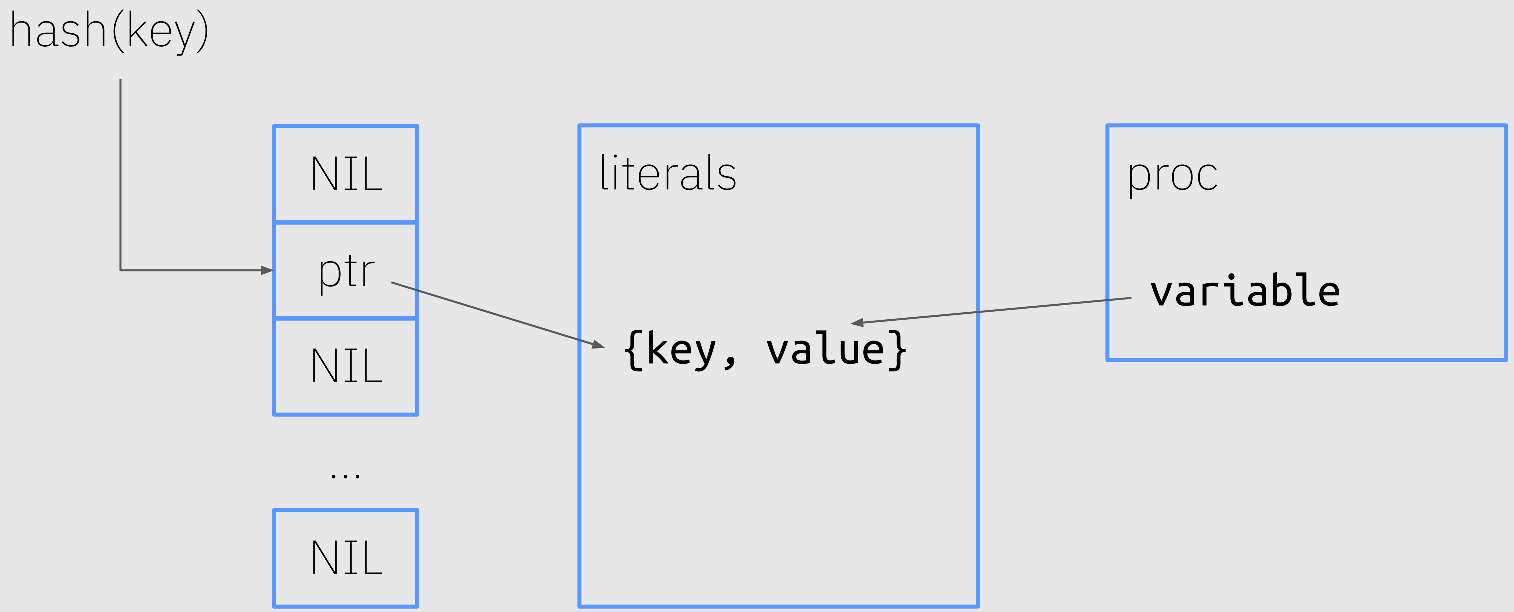
Persistent term vs code generation

- mochiglobal/FastGlobal (abusing literal pool)
- eg.: Elixir regexp sigil, compiled pattern (real, plain binary)
- Runtime terms: PIDs, ports, Refs (not literals per se)
  eg.: ETS TIDs (fake literals)
- official solution: PT supports all terms

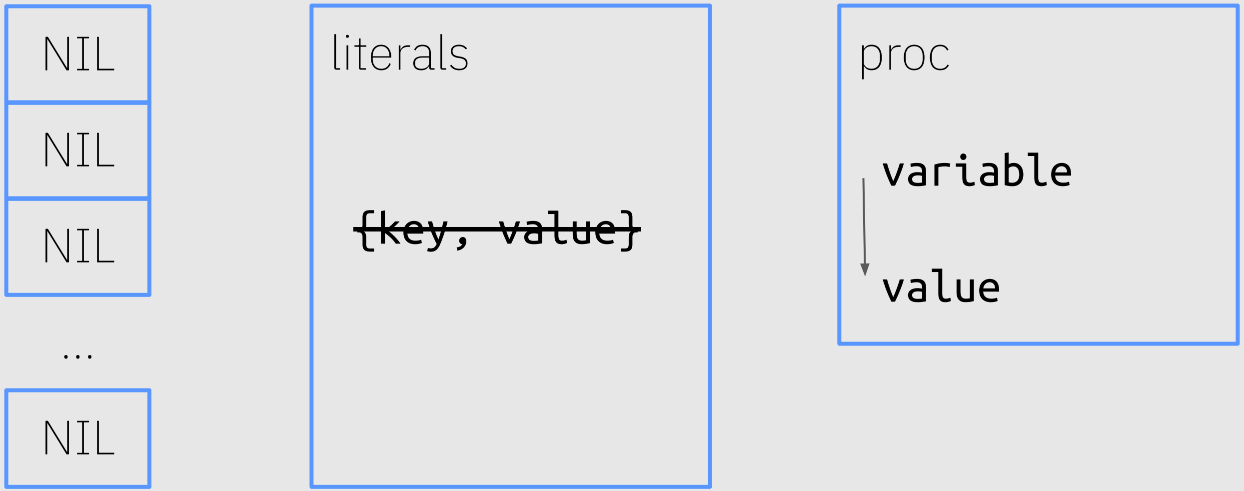
Persistent term - put



Persistent term - get



Persistent term - erase



Persistent term - update

- Any insert/update/delete will copy table
  (similar to updating tuple)
  Proportional to size of table
- Copy inserted term from heap
  (sharing-preserving copying)

Persistent term - update

Update/delete

(similar to unloading a module)

- Some procs still use old value
- Will scan all procs
- Will copy “complex” term to proc heap and trigger GC
  (can hit max heap size and kill proc)

PT update - Optimisations

- Immediates (terms that fit in 1 word)
  No GC: replace pointer with value on proc heap
  (eg.: OTP logger default log level)
- From OTP 23 no table copy, if no need to grow/shrink

PT update - Limitations

- Only 1 process can update at a time
  (no write_concurrency)
- Queue of processes waiting to update the hash table
- Only 1 key update at a time
  (use maps/tuples for multiple values)

Danger of PT update

- It's truly global (libraries have to share)
  - Use namespaced keys
  - An update by a lib affects other libs/apps/all procs
- One lib has no control over
  - The total size of the hash table
  - The total number of processes in the system

⇒ no idea about update impact

Apples vs oranges

Atomic counters | Persistent terms

concurrent writes | single process to update (the whole table)

Apples vs oranges

Atomic counters	Persistent terms
read inconsistency (write_concurrency)	super-cheap, no copy reads

Apples vs oranges

Atomic counters	Persistent terms
ets counters	proc dict
off-heap binaries	ets table

Summary

- Long awaited by the community
- No more hacks and NIFs
- More confidence in usage
- Just be careful

References

- Rickard Green, Patryk Nyblom: "Taking a Virtual Machine towards Many-Core" (EUC 2012)
- Irina Guberman: "High Performance Metrics Through Mutable Counters" (CodeBEAM SF 2018)
- Lukas Larsson: "OTP 22 Highlights", " Clever use of persistent_term" (Erlang/OTP Blog)

Thank you!

Benchmarks

Impact of PT update

Erasing one by one 10_000 entries takes

(processes have no references to PT)

- ~5s with ~50 processes in the system
- ~6.5s with ~100 procs
- ~9.8s with ~1000 procs
- ~18s with ~10_000 procs

Impact of PT update

20 procs bumping counter for 5 seconds

Counter reference updated in PT N times

- 0: 266M bumps
- 1: 264M bumps (99%)
- 5: 263M bumps (98%)
- 10: 256M bumps (96%)

Concurrent counters

Initialization

- `:ets.new(__MODULE__, [:public, :named_table, write_concurrency: true])`
- `:persistent_term.put({__MODULE__, ctr}, :counters.new(1, [:write_concurrency]))`

Concurrent counters

Increment

- ```
  :ets.update_counter(__MODULE__,
    {ctr, :erlang.system_info(:scheduler_id)}, 1)
  ```
- ```
  :counters.add(
    :persistent_term.get({__MODULE__, ctr}), 1, 1)
  ```

Concurrent counters

Counter bumps in 5 seconds - 20 processes (8 cores)

- ets: 104M bumps
- pt+counters: 251M bumps (240%)

Atomics

Counter bumps in 5 seconds - 1 process

- `:ets.new(__MODULE__, [:private, :set]).`
- `:counters.new(1, [:atomics]).`

- 81M bumps `:ets.update_counter(tid, :counter, 1).`
- 201M bumps (247%) `:counters.add(ref, 1, 1).`

Histogram with atomics

Histogram bumps in 5 seconds - 1 process

- `:ets.update_counter(h.table, h.metric,`
  `[{@total_cnt_idx, n},`
  `{index + @total_cnt_idx + 1, n}])`
- `:atomics.add(h.ref, @total_cnt_idx, n)`
  `:atomics.add(h.ref, index + @total_cnt_idx + 1, n)`

Histogram with atomics

Histogram bumps in 5 seconds - 1 process

- ets-based: 21M bumps
- atomics-based: 30M bumps (142%)