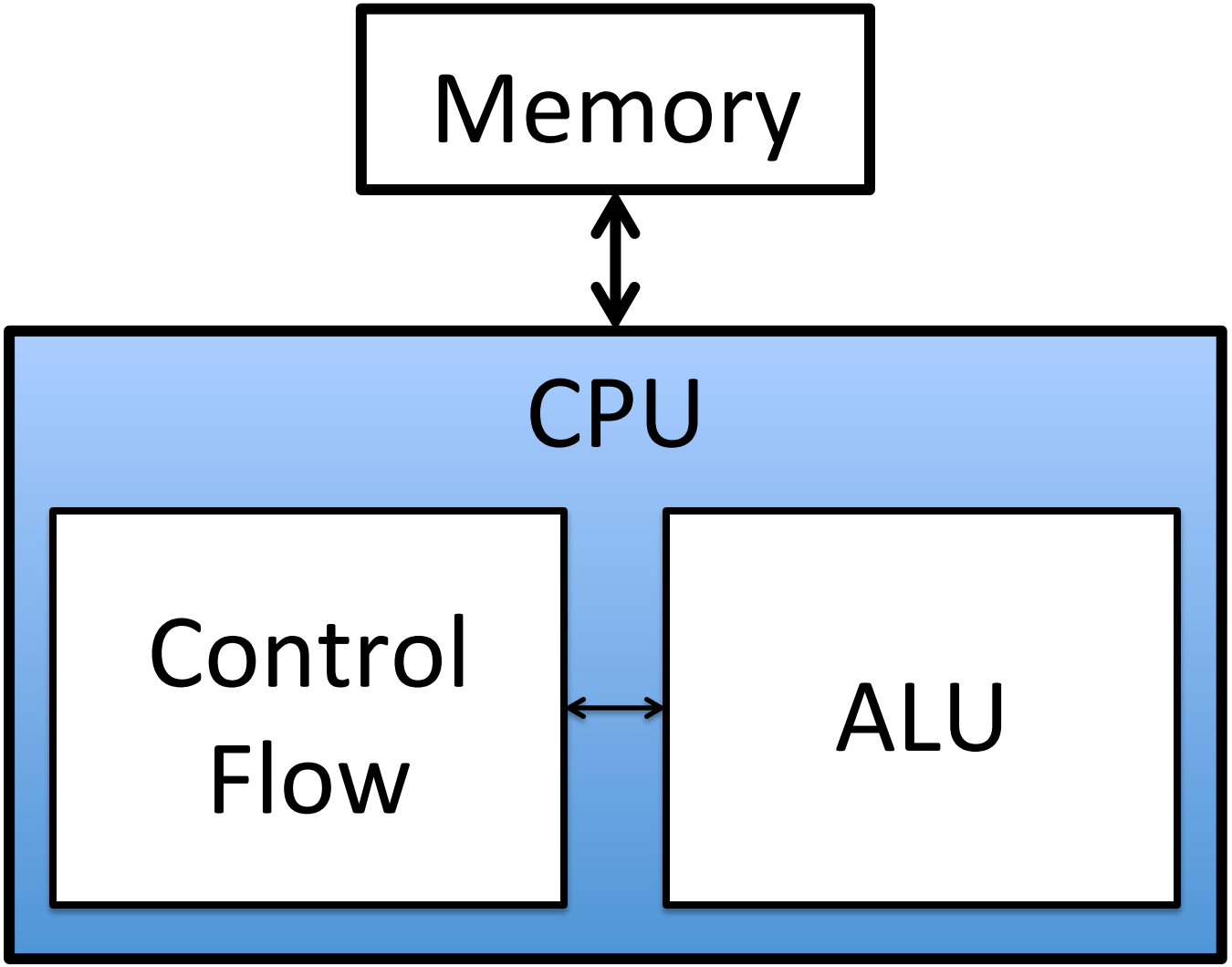
# Memory, What Is It All About?

Amaury SECHET
@deadalnix

# Von Neumann Architecture



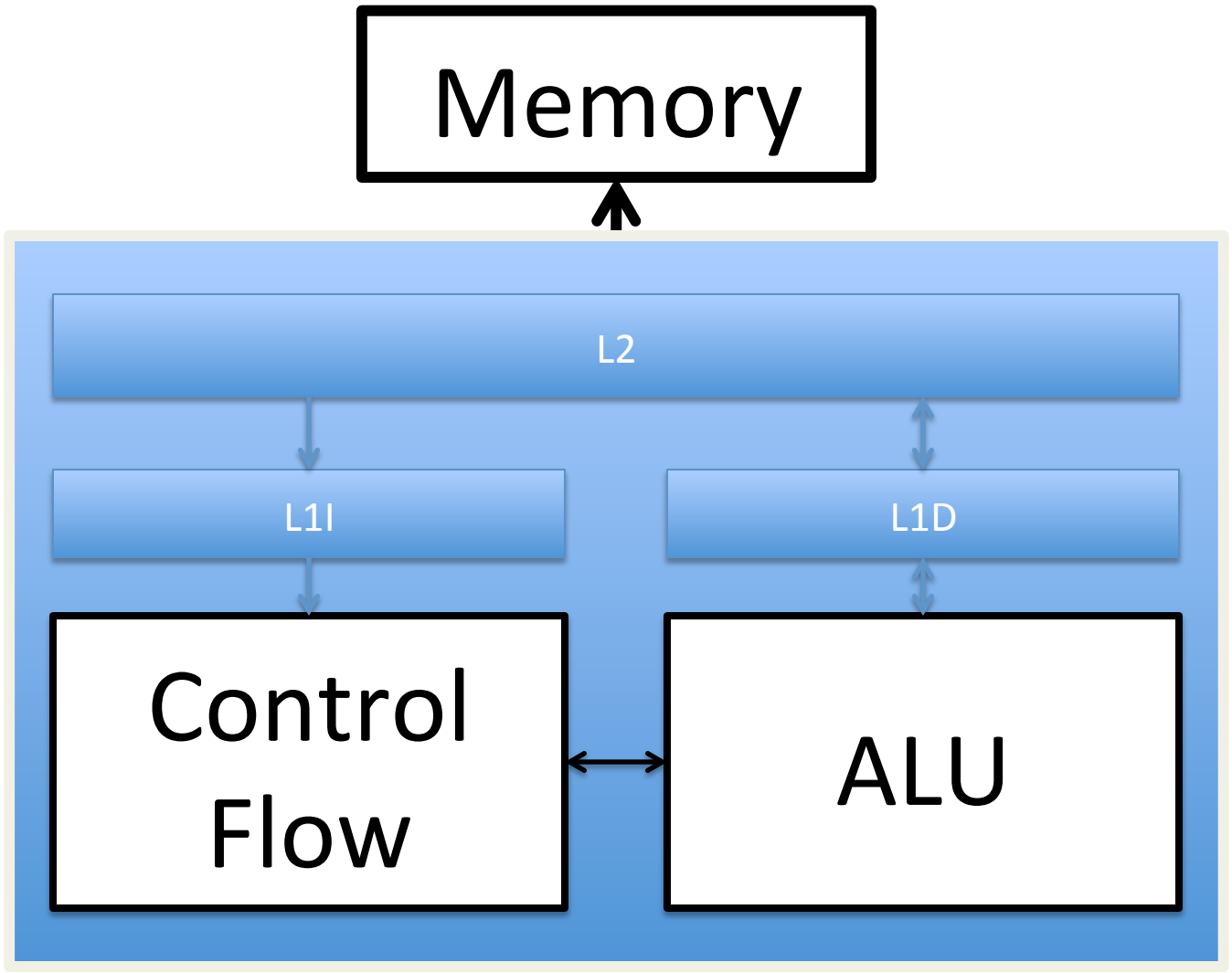
# Memory is slow

- About 300 cycles to hit memory
- Bandwidth still increasing
- Latency only marginally increasing

# Memory is slow - Caching

- Add faster memory on CPU.
- Various size and speed
  - Signal needs time to travel
  - L1: 3-4 cycles, 32kb
    - Instruction
    - Data
  - L2: 8-14 cycles, 256kb
  - L3: tens of cycles, few Mb, often shared
  - Cache line: 64 bytes

# Memory is slow - Caching



# Memory is slow – Prefetching

- CPU observe memory access pattern and try to predict what is going to be accessed next

- Trade memory bandwidth for hope
- Usually worthwhile

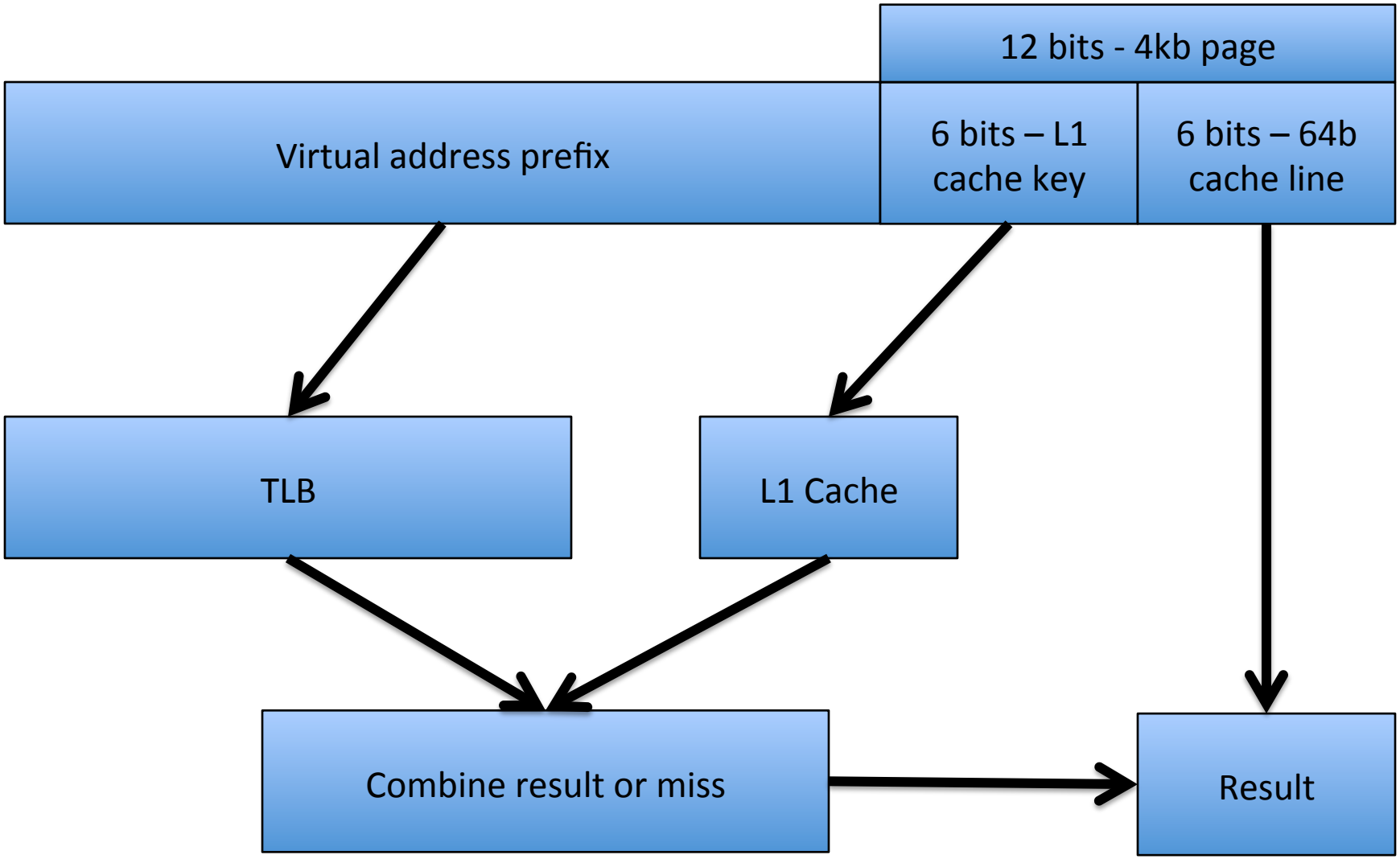
# Memory is slow – Good Practices

- Put what you can on the stack, it's hot
- Pack you data !
  - std.bitmanip is your friend.
- Avoid indirections
  - Sometime duplication is preferable
  - This include indirect branches
  - NB: The optimizer don't like them either
- Use linear or pseudo linear access pattern
- Size your data structures
  - LZ4 use a 16kb dictionary
  - ZSTD uses a 16kb finite state automaton

# MMU

- Map virtual addresses to physical addresses
  - Only the OS knows about physical addresses
- Check access right
- Done by pages, usually 4kb
- Translation must be done before cache lookup
  - Kind of
- Data about pages are cached in the TLB
  - Usually 16 entries in the first level

# MMU


12 bits - 4kb page
Virtual address prefix
6 bits – L1 cache key
6 bits – 64b cache line
TLB
L1 Cache
Combine result or miss
Result

# Welcome multicore

- It is everywhere
  - In servers
  - In desktops
  - In laptops
- Even in mobile devices

# Welcome multicore

- Improvement of single core CPU give smaller and smaller returns
- Technology still allow for more transistors in a chip
- Multi core is the next logical step

# Welcome multicore

- Unlike previous CPU improvement
  - Visible to the programmer
  - Require to adapt programs
  - A lot of non obvious subtleties
  - Programmer sanity is in danger

# Welcome multicore

"… a folk definition of insanity is to do the same thing over and over again and to expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs."

Edward A. Lee

# Usual suspects

# Usual suspects

- Language semantic
- Compiler
- CPU

# Language semantic

- Most languages defined before multicores
- Do not multithread at all
- Do it in undefined manner
- Hard to retrofit the right semantic
  - C++11
  - Java 1.5 (JSR-133)

# Language semantic

- Can't change semantic of existing code
- Or make it way slower
- Add new well defined options
  - Can't be made safe by default
  - Rely on programmer
  - Mistakes are undefined behavior
- Newer languages like D MUST have a solution

# Compiler

- The compiler do its best to ensure maximum performances
- Single threaded semantic is preserved
- Optimizations can become visible on multicore
- Optimizer must be essentially disabled for multicore correctness
  - Only a subset of the code will suffer if language semantic makes it explicit which are which
  - Otherwise everything is slow or unsafe

# CPU

- Each core write in its cache
- Data is committed to memory asynchronously
- Other core see write out of order…
  - when reading from memory
  - when reading from cache
- Or even can erase each other's writes

# CPU – MOESI Cache coherency

- Cache coherency protocol
  - Each core exchanges cache state with others
  - Via a dedicated bus
- MOESI is a cache coherency protocol
- It is simple…
  - To not say simplistic
- Actual CPUs use more complex mechanisms
- But the basics remains

# MOESI – Cache line states

| State | Read/Write | Read only |
|---|---|---|
| Dirty | Modified | Owned |
| Clean | Exclusive | Shared |
| None | Invalid | |

Red text indicate ownership

# MOESI - Reading

- MOES : read from cache
- I : acquire cache line
  - O : Get a copy as S
  - M : Get a copy as S, other core goes to O
  - E : Get a copy as S, other core goes to S
  - S or I : Get a copy from memory

# MOESI - Writing

- M : write to cache
- E : write to cache, go to M
- OS : acquire cache line in write mode
  - Invalidate other entries
  - Write and go to M
- I : acquire cache data in write mode
  - Similar to reading process
  - Invalidate other entries
  - Write and go to M

# MOESI - Conclusion

- 2 nice scenarios
  - Only one core use a cache line
    - Fast read and write
  - No write to the cache line
    - Several core can share the line
    - Line can be fetched from other cores
- Avoid sharing and writing
  - Core will have to pass the line back and forth
  - Slow and use a lot of bandwidth

# MOESI - Conclusion

- Keep data thread local
- Prefers sharing immutable data
- Avoid sharing mutable data
- Avoid contention on a cache line
  - Add padding around frequently written variables
  - LMAX disruptor PaddedLong

# Sequential Consistency

- Memory operations appear in a consistent order for all cores
- It turns out that developers like it
  - Non linear time is confusing
  - If you think you can handle it you are likely wrong
  - Try to watch Primer and reconsider
- Some extra optimizations get into the way
  - Memory barrier required

# CPU – Store buffer

- Store can stall
  - Cache line must be acquired
  - Or other core's cache line invalidated
- We don't want to stall execution
- Store are put in a buffer
  - And CPU continue execution

# CPU – Store buffer

- Store are placed in a buffer
- Wait until the cache line is acquired
- Store is done when the cache line is ready
- Load look in store buffer and cache
- Execution continue in the meantime
- Store can be done out of order
- Store buffer is invisible to other cores

# CPU – Invalidate Queue

- Invalidates are acknowledged immediately
- But processed later
- CPU can read outdated data from cache
- CPU must check its queue to send invalidate or copies of a cache line
  - Must issue an acquire-invalidate instead

# CPU – Memory barrier

- Required to ensure sequential consistency
- 4 kind
  - LoadLoad
  - StoreStore
  - LoadStore
  - StoreLoad

# CPU – StoreLoad

- Flush Store buffer
- Flush invalidate queue
- The most expensive barrier
  - No mainstream architecture ensures it

# CPU – On existing CPU

| Barrier | X86 | ARM |
|---|---|---|
| LoadLoad | no-op | dmb |
| StoreStore | no-op | dmb-str |
| LoadStore | no-op | dmb |
| StoreLoad | Locked instruction<br>mfence | dmb |

# We are doomed /o\

- Years of optimizations are firing back
- Existing languages provide terrible semantic
  - Memory shared implicitly
  - Correctness is not ensured for concurrent access
  - Developer must ensure correctness manually

# We are doomed /o\

- Standard testing practices do not work
- Algorithms must be proven correct
- Failures are non deterministic and hard to reproduce
- That means hard to debug
- And impossible to bisect

D

# D - Memory Model

- Everything is thread local by default
  - Global aren't really global
  - Data can't be shared between threads

# D - Thread Local

```
Widget w;  // Thread local

void fun() {
     // Widget is thread local
     w = new Widget();
}
```

# D - Thread Local

- Thread local is the default
- The compiler can optimize maximally
- No need for synchronization
- By default, each thread lives in its own universe
  - Information sharing is always explicit
  - Makes life easier for the developer
  - Makes life easier for the compiler

# D - Memory Model

- Everything is thread local by default
  - Global aren't really global
  - Data can't be shared between threads
- Objects can be declared immutable
  - Immutable can only reference immutable
  - Immutable can be shared between threads

# D - Immutable

```
struct S {
    int* i;
}

void foo(int* i) {
    auto a = S(i);                  // OK
    auto b = immutable(S)(i);       // Fail
}

void bar(immutable int* i) {
    auto a = S(i);                  // Fail
    auto b = immutable(S)(i);       // OK
}
```

# D - Immutable

- Everything you can reach through an immutable object is immutable
  - Immutable is transitive
- No synchronization needed
- Compiler can optimize based on immutability
  - Not done right now

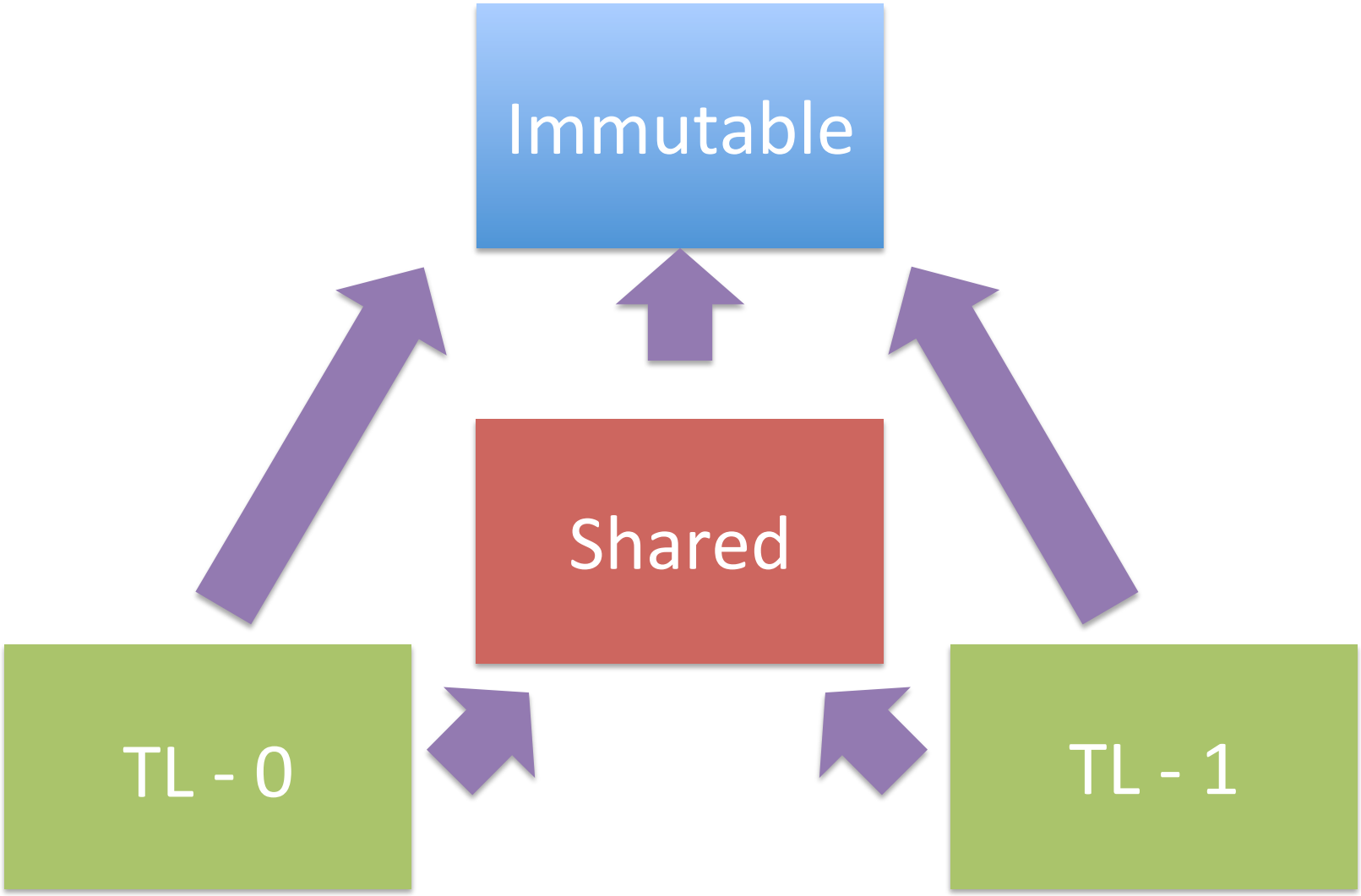
# D - Memory Model

- Everything is thread local by default
  - Global aren't really global
  - Data can't be shared between threads
- Objects can be declared immutable
  - Immutable is transitive
  - Immutable can be shared between threads
- Or shared…
  - Also called "I'm asking for trouble"

# D - Shared

- Transitive as immutable
- Everything you can reach via shared is either shared or immutable
- Compiler ensure sequential consistency
  - Except DMD
  - Your lock free code is probably wrong anyway
- You generally want to avoid it

# D – Memory Model



# Current D's GC

- Stop the world
- Mark and sweep
- Lock on all allocations/deallocations
- Find memory pool in log(N) while scanning
- Not much
  - Not precise
  - Not concurrent
  - Not generational
  - Not parallel
- Improving, but the API between compiler and runtime ultimately very limiting

# A Good Alocator for D

- Fast at malloc/free
- Fast at GC
- Interior pointer detection
- Low fragmentation
  - Internal (requested size vs allocated size)
  - External (No “holes”)
  - Reduce pressure on cache and MMU
- Comply with system programming
- Maintain metadata for arrays

# SDC – work in progress

- Segregate the heap
  - One heap for thread local for each thread
  - One heap for shared and immutable
- TL heap is stop the world
  - The world in only one thread
- Immutable/Shared is concurrent
  - Write barrier ?
  - Not as bad as it seems

# SDC – Thread local GC

- No need to lock or be thread safe
- Segregate the heap
- Thread that do not generate garbage are not impacted
- Mostly done
  - Array metadata
  - GC scan, but do not collect
- Inspired from jemalloc
  - Modified to find interior pointer quickly
  - Remove TL cache and locking

# SDC – Shared heap

- Use thread local cache to reduce contention
  - tcmalloc and jemalloc
  - At the cost of fragmentation
  - Tuning and tweaking required
- Concurrent collection problem :
  - Reference is read from memory
  - Reference is then overwritten in memory
  - Local copy exist, but GC don’t see it

# SDC - Concurrent collection

- No weird GC pauses
- Free concurrency !
- Even more floating garbage
  - Not a problem on desktop/server
  - Not suitable for embedded
- Require write barrier
  - Tricky to do AOT

# SDC - Write barrier

- Always insert them
  - Make all stores expensive
  - VM language can swap implementation
  - @system code able to bypass barrier (scary)
- Use MMU
  - Trap everything, not only pointer
  - Can be enabled/disabled at will
  - VERY expensive (few thousands cycles)

# SDC - Concurrent collection

- New allocations :
  - Considered live
  - No need to scan, reference in there come from
    - Other new allocations (live)
    - Preexisting allocation (will be scanned)
- Protect unscaned memory
  - Mark old value before updating
  - Use MMU to do so
- Used in ML language with great success

# SDC - Concurrent collection

- Write in memory are mainly
  - Immutable initialization
  - Shared write
- New allocs aren't scanned, no protection needed
- Shared is assumed to be a minor part of the heap
- MMU can work

# Implicit sharing

- Immutable delegate can have mutable context
  - Need fully qualified delegates: DIP30
  - Will not handle
- Exceptions
  - Only when the thread terminate
- Pure function
  - Result can be promoted to immutable
- std.allocator needs to handle this

# Heap merging

- Merge heap on thread join
- Pure functions
  - Create new heap before calling
  - Set it as TL heap
  - Merge it into the TL heap on return
- Perform GC on merge
- Performance ?
- Proposal: make it a language construct (isolated/ owned)
  - Can pass reference while keeping segregation
  - Allow compiler to optimize further
  - Safely enable many multithreading use cases

# SDC – Optimizations ?

- Immutable naturally generational
  - Can this be leveraged ?
- When a reference does not escape, add free
- Promote pair alloc/free on stack
  - Is size unknown, add a check
- Do it by adding passes into the optimizer
  - It can benefit from inlining

# SDC – This should not allocate !

```
int[] getArray() {
    return [1, 2, 3];
}

void main() {
    foreach(i; getArray()) {
        import std.stdio;
        writeln(i);
    }
}
```

# SDC – This should not allocate !

```
class Foo {
    void doSomething() { ... }
}

class Builder {
    Foo build() { return new Foo(); }
}

void main() {
    auto b = new Builder();
    auto f = b.build();
    f.doSomething();
}
```