THIS SLIDE
INTENTIONALLY
LEFT BLANK.

WRITING A TOP-10 CHESS ENGINE IN RUST

CSMR KRAR

MOVE EMISSION

```
pub struct MoveSelector {
  pub stage : Stage,
  // ...
}

impl MoveSelector {
  pub fn next(&mut self, /* context */) -> Option<Move> {
    if self.stage == Stage::A {
      // ...
      if /* ... */ { return Some(mv); }
      self.stage = Stage::B;
    }
    if self.stage == Stage::B {
      // ...
    }
    // ...
  }
}
```

CONTROL FLOW

This is nice because it allows fall-through: once we've exhausted the moves from a stage, we can immediately move on to the next stage. However, we're not guaranteed to jump to the right stage upon entry.

(*In practice, it probably doesn't matter—the compiler might emit either a test-and-jump sequence or a jump table based on the number of locations independently of whether we write an if-then sequence or a match construct.*)

CONTROL FLOW

But suppose we did want to write it this way:

```
pub fn next(&mut self, /* context */) -> Option<Move> {
  loop {
    match self.stage {
      Stage::A => {
        if /* ... */ { return Some(mv); }
        self.stage = Stage::B;
        continue;
      }
      Stage::B => { /* ... */ }
      // ...
    }
  }
}
```

Now this no longer indicates fall-through!

If you wanted to express that in the source, it used to be the case you'd have to write something like

```
pub fn example(stage : usize, mut x : i32) -> i32 {
  'e: loop {
    'd: loop {
      'c: loop {
        'b: loop {
          match stage {
            0 => {} //   'a
            1 => { break 'b; }
            2 => { break 'c; }
            3 => { break 'd; }
            _ => { break 'e; }
          }
          x *= 2; break;  // case A
        }
        x *= 3; break;    // case B
      }
      x *= 5; break;      // case C
    }
    x *= 7; break;        // case D
  }
  x *= 11; return x;      // case E
}
```

CONTROL FLOW

In version 1.55, this compiled to

```
example:
        cmp     rdi, 3
        ja      .LBB0_3
        lea     rax, [rip + .LJTI0_0]
        movsxd  rcx, dword ptr [rax + 4*rdi]
        add     rcx, rax
        jmp     rcx
.LBB0_4:
        add     sil, sil
.LBB0_5:
        movzx   eax, sil
        lea     esi, [rax + 2*rax]
.LBB0_6:
        movzx   eax, sil
        lea     esi, [rax + 4*rax]
.LBB0_2:
        movzx   eax, sil
        lea     esi, [8*rax]
        sub     esi, eax
.LBB0_3:
        movzx   eax, sil
        lea     ecx, [rax + 4*rax]
        lea     eax, [rax + 2*rcx]
        ret
.LJTI0_0:
        .long   .LBB0_4-.LJTI0_0
        .long   .LBB0_5-.LJTI0_0
        .long   .LBB0_6-.LJTI0_0
        .long   .LBB0_2-.LJTI0_0
```

CONTROL FLOW

When there are four stages rather than five, the compiler emits a series of test-and-jump instructions, but when the number of cases is sufficiently large, the compiler instead emits a jump table (as just seen).

The source of example is, of course, *disgusting*. But we have labelled blocks now! so we can instead write... the same thing, but without loop and break.

PROSAIC OBSERVATION

This is why we like compiler optimizations.

DREAMING OF TOMORROW

In the future, you could instead do a recursive call to `next` at the end of each stage and use the `become` keyword to avoid creating a nested stack frame. (Thank you, Waffle *et alia!* See issue **112788**.)

POETIC OBSERVATION

> Rust is getting better all the time.

SHARED CACHE

Or the "transposition table", in traditional parlance.
(The size of the transposition table is also known as the "hash size", despite the fact that the data structure is not a hash map but a cache.)

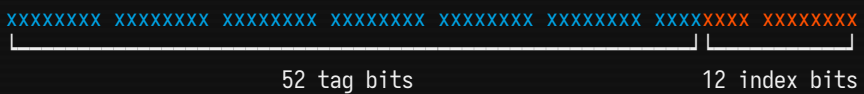
Commonly 3-way set associative. The cache is the only means by which threads communicate during search.

```
#[repr(C)]
struct CacheEntry {
  tag   : u16,
  move  : Option<Move>,
  eval  : i16,
  score : i16,
  depth : u8,
  meta  : PackedMetadata,
}
```

```
#[repr(C, align(32))]
struct CacheSet {
  entries : [CacheEntry; 3],
  padding : [u8; 2],  // to avoid UB
}
```

INDEXING THE CACHE

In processor caches (where the keys are addresses), the cache is often indexed by partitioning the key into tag bits and index bits.*

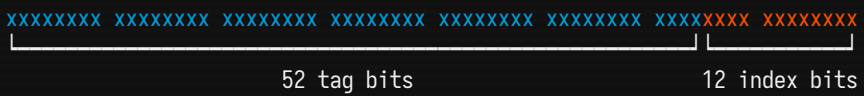


There are $2^{num\text{-}idx\text{-}bits}$ sets in the cache
and the size of each set is $\lceil entry\text{-}size \times ways \rceil$.

*And offset bits, but let's ignore that for now.

INDEXING THE CACHE

In processor caches (where the keys are addresses), the cache is often indexed by partitioning the key into tag bits and index bits.*



There are $2^{num\text{-}idx\text{-}bits}$ sets in the cache
and the size of each set is $[entry\text{-}size \times ways]$.

QUERY

What if the number of sets in our cache isn't a power of two?

*And offset bits, but let's ignore that for now.

INDEXING THE CACHE

If keys are 64-bit numbers, then $key \,/\, 2^{64}$ is a fraction between zero and one. We can multiply the number of sets by this fraction to get an index between zero and *num-sets*.

To do this with integer arithmetic, we reärrange:

$$num\text{-}sets \times (key \,/\, 2^{64}) = (num\text{-}sets \times key) \,/\, 2^{64}$$

INDEXING THE CACHE

```
fn mulhi(a : u64, b : u64) -> u64 {
  ((a as u128 * b as u128) >> 64) as u64  // wrapping_mul unnecessary because the
}                                         // compiler can elide the overflow check

let index = mulhi(num_sets, key); // derived from the upper bits
let tag   = key as u16;           // derived from the lower 16 bits
```

We use the least-significant bits of the key for the tag because they don't figure into the index.

PROOF

If we imagine s and k as two halves, $s = (s^H \ll 32) + s^L$ and $k = (k^H \ll 32) + k^L$, their product is

$$(s^H \times k^H) \ll 64 + (s^H \times k^L + s^L \times k^H) \ll 32 + (s^L + k^L).$$

The sum $s^L + k^L$ is only 33 bits wide, so after taking the upper 64 bits, this disappears entirely (except for propagated carries). In practice, s is about 2^{23} to 2^{33}, so s^H is zero, and then $s^H \times k^L$ is zero — and so k^L does not affect the result.

MEMORY MODEL

Consider two threads accessing the same memory, where at least one of the accesses is a write.

There is a race condition if the accesses are concurrent (unsynchronized).

There is a data race unless
- the accesses are synchronized or
- the accesses are all atomic.

A race condition is a data race if the accesses are not atomic.

[ACHTUNG] *The behavior of a program containing a data race is undefined.*

MEMORY MODEL

Synchronization is established by release–acquire* pairs, by combinations of atomic accesses and memory fences, thread spawning, and thread joining (or by constructions that use these, like mutexes).

The operations of an individual thread are sequenced by program order, so non-atomic accesses through a raw pointer are transitively synchronized (and have defined behavior) if the accesses are properly guarded by a lock.

* Or stronger.

ACCESSING THE CACHE

We've two options:

0 Guard access with striped locks.

The compiler can optimize our lookup into a single 256-bit load if that's better, but the performance penalty from obtaining a lock is unacceptable (or at least unnecessary, however slight it may be — hence unacceptable).

1 Use atomics.

Four 64-bit loads, which appears to be as fast as anything else in practice. This is more or less the only place it makes sense to use Relaxed ordering.

ACCESSING THE CACHE ATOMICALLY

```
pub fn lookup(&self, key : u64) -> CacheEntry
{
  let index = mulhi(self.num_sets, key) as usize;
  let b0 = self.sets[index].block[0].load(Relaxed);
  let b1 = self.sets[index].block[1].load(Relaxed);
  let b2 = self.sets[index].block[2].load(Relaxed);
  let b3 = self.sets[index].block[3].load(Relaxed);
  let cache_set = unsafe {
    transmute::<[u64; 4], CacheSet>([b0, b1, b2, b3])
  };
  let tag = key as u16;
  for entry in cache_set.entries {
    if entry.tag == tag { return entry; }
  }
  return CacheEntry::ZERO;
}
```

ACCESSING THE CACHE WRONGLY BUT WISHFULLY

```
pub fn lookup(cache : *const [CacheSet], num_sets : u64, key : u64) -> CacheEntry
{
  let index = mulhi(num_sets, key) as usize;
  let ptr = cache.add(index as usize).cast();
  let bytes = unsafe { std::arch::x86_64::_mm256_load_epi64(ptr) };
  let cache_set = unsafe { transmute::<_, CacheSet>(bytes) };
  let tag = key as u16;
  for entry in cache_set.entries {
    if entry.tag == tag { return entry; }
  }
  return CacheEntry::ZERO;
}
```

The behavior of this is undefined because access is not synchronized.

There's no reason to do this (and many reasons not to).

THE DUALITY OF MEM

FEARLESS CONCURRENCY

Rust is a great systems language because of its powerful safe primitives (like Arc, Condvar, Mutex, OnceLock, and so on).

FEARFUL CONCURRENCY

Rust is *also* a great systems language because it takes you seriously as a user and lets you create your own primitives*!* (After all, the standard library is largely implemented in ordinary Rust.)

[ACHTUNG]

When you use unsafe, the compiler will not check that all invariants are upheld; you are responsible for doing so.

DU HAST VERSPROCHEN

REMEMBER YOUR PROMISE

INJUNCTION

READ MARA'S BOOK.

TORN READS

The widest atomic type is AtomicU64 (eight bytes), but an entry is ten bytes and a set is thirty(-two) bytes, so neither can be accessed atomically.

Reading an entry requires multiple loads, and in theory, if another thread is issuing stores at the same time, part of the data we read might be from the ongoing write and part of the data might be from a previous write.

NOTE

Even if torn reads weren't possible, since we store too few tag bits to recover the key, we have to handle entries that appear valid but are in fact invalid. (There's also the very small chance of key collisions.)

TORN READS

We don’t have to mitigate this, amazingly enough.

- It’s safe because all bit patterns are valid.
- The performance* impact is small (but enough that it’s worth countering).

Why might this be so? We already expect the engine to be mistaken in its evaluätion of positions, and reading an incorrect entry is simply another misevaluätion, which merely degrades the quality of analysis but does not threaten algorithmic correctness.

We can reject the vast majority of incorrect lookups by checking the legality of the stored move. *This guards against torn reads, tag aliasing, and key collisions, but imperfectly — a move might be legal in two colliding positions.*

* Game-playing strength measured in Elo rating points.

ATOMIC PER BYTE

RFC **3301** proposes an “atomic memcpy” that permits tearing.

It might also provide the means to read uninitialized memory, allowing more performant implementation of data structures like sparse sets.

RUST THROUGH THE AGES

We both started using Rust in January 2021. What's changed since then?

- 1.51 const generics · {integer}::unsigned_abs
- 1.58 identifiers in format strings
- 1.59 inline assembly · destructuring assignments
- 1.60 {integer}::abs_diff
- 1.65 let-else statements · break in labelled blocks
- 1.79 inline const expressions
- 1.82 raw pointer syntax
- 1.87 {integer}::is_multiple_of · {integer}::midpoint
- 1.88 let chains
- 1.92 Box::new_zeroed
- 1.93 [T]::as_array

△

{ 终 }

THIS SLIDE
INTENTIONALLY
LEFT BLANK.