Zero-Cost Abstractions

~ or ~

Why fixing the inliner for hash_value was actually important

C++?

Loop-heavy?

Fortran?

Servers?

Small?

Haskell?

What code matters?

Scientific?

En/Decoding?

Big projects?

Parallel?

Small projects?

Kernels?

C++?

Loop-heavy?

Fortran?

Servers?

Small?

Haskell?

What code matters?

Scientific?

En/Decoding?

Big projects?

Parallel?

Small projects?

Kernels?

C++?

Loop-heavy?

Fortran?

Servers?

Small?

Haskell?

Big, C++ Codebases!

Scientific?

En/Decoding?

Big projects?

Parallel?

Small projects?

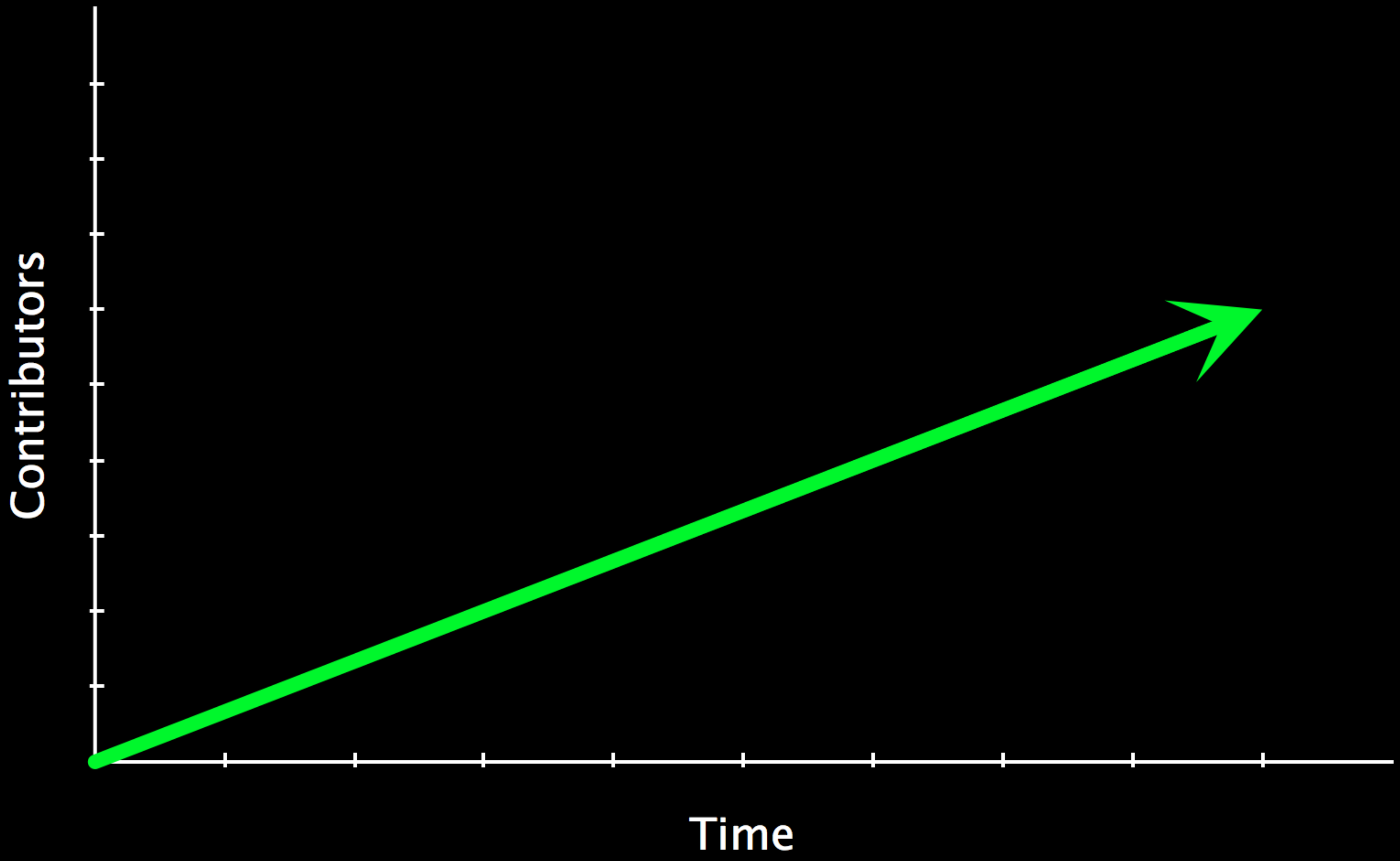
Kernels?

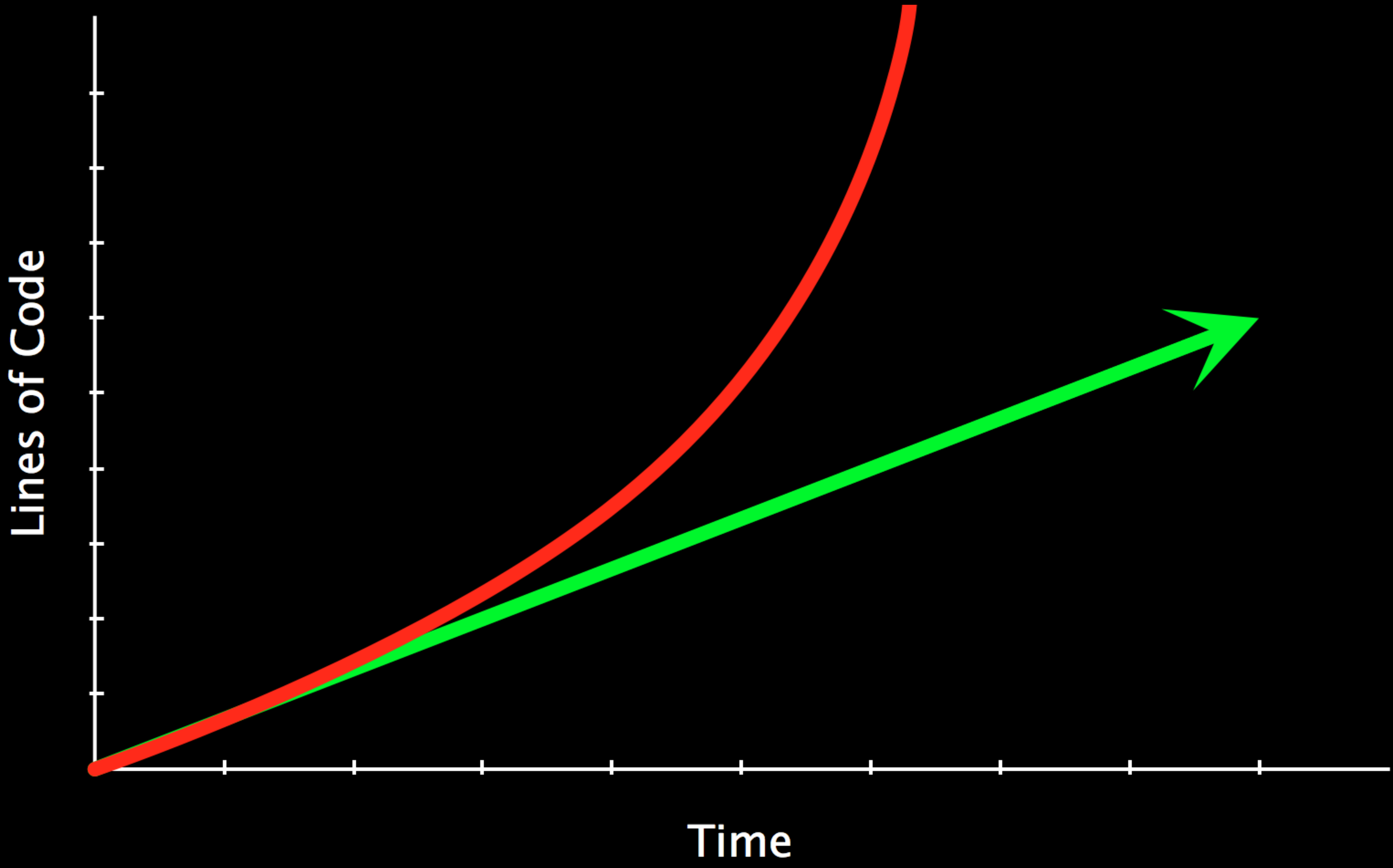
How does C++ scale up?

Abstractions

Abstractions

have cost!


Contributors
Time


Lines of Code
Time

C++ strives for zero-cost abstractions

Zero-cost abstractions need optimizer help

- Inlining
- Devirtualization
- Iterative combining
- SRoA
- Loop idioms
- Scalar idioms
- Hardware patterns
- Global-opt

A case study: hash_value

First off, what is Hashing.h?

```
namespace llvm {

class hash_code;

/// \brief Compute a hash_code for any integer value.
template <typename T>
typename enable_if<is_integral_or_enum<T>, hash_code>::type
  hash_value(T value);

/// \brief Compute a hash_code for a pointer's address.
template <typename T> hash_code hash_value(const T *ptr);

/// \brief Compute a hash_code for a pair of objects.
template <typename T, typename U>
hash_code hash_value(const std::pair<T, U> &arg);

/// \brief Compute a hash_code for a standard string.
template <typename T>
hash_code hash_value(const std::basic_string<T> &arg);

}
```

```
namespace llvm {

/// \brief Combine values into a single hash_code.
template <typename ...Ts>
hash_code hash_combine(const Ts &...args);

/// \brief Compute a hash_code for a sequence of values.
template <typename InputIteratorT>
hash_code hash_combine_range(InputIteratorT first,
                             InputIteratorT last);

}
```

```
using namespace llvm;

hash_code hash_combine_range(char* start, char* stop) {
  uint64_t h = 0;
  while (start != stop)
    h = 33*h + *start++;
  return hash_code(h);
}
```

We need some abstractions…

```
struct hash_state {
  uint64_t h0, h1, h2, h3, h4, h5, h6, seed;
  static hash_state create(char* s, uint64_t seed);
  void mix(char* s);
  uint64_t finalize(size_t length);
};

hash_code hash_combine_range(char* start, char* stop) {
  size_t seed = ...;
  size_t length = std::distance(start, stop);
  if (length <= 64)
    return hash_short(start, length, seed);
  char *aligned_stop = start + (length & ~63);
  hash_state state = state.create(start, seed);
  start += 64;
  while (start != aligned_stop) {
    state.mix(start);
    start += 64;
  }
  if (length & 63) state.mix(stop - 64);
  return state.finalize(length);
}
```

```c
uint64_t hash_short(const char *s, size_t length,
                    uint64_t seed) {
  if (length >= 4 && length <= 8)
    return hash_4to8_bytes(s, length, seed);
  if (length > 8 && length <= 16)
    return hash_9to16_bytes(s, length, seed);
  if (length > 16 && length <= 32)
    return hash_17to32_bytes(s, length, seed);
  if (length > 32)
    return hash_33to64_bytes(s, length, seed);
  if (length != 0)
    return hash_1to3_bytes(s, length, seed);

  return k2 ^ seed;
}
```

```
uint32_t fetch32(const char *p) {
  uint32_t result;
  memcpy(&result, p, sizeof(result));
  return result;
}
uint64_t hash_16_bytes(uint64_t low, uint64_t high) {
  // Murmur-inspired hashing.
  const uint64_t kMul = 0x9ddfea08eb382d69ULL;
  uint64_t a = (low ^ high) * kMul;
  a ^= (a >> 47);
  uint64_t b = (high ^ a) * kMul;
  b ^= (b >> 47);
  b *= kMul;
  return b;
}
uint64_t hash_4to8_bytes(const char *s, size_t len,
                        uint64_t seed) {
  uint64_t a = fetch32(s);
  return hash_16_bytes(len + (a << 3),
                       seed ^ fetch32(s + len - 4));
}
```

Imagine hashing 8 bytes…

Disaster #1


INLINE ALL THE FUNCTIONS
imgflip.com

```
uint32_t fetch32(const char *p) {
  uint32_t result;
  memcpy(&result, p, sizeof(result));
  return result;
}
uint64_t hash_16_bytes(uint64_t low, uint64_t high) {
  // Murmur-inspired hashing.
  const uint64_t kMul = 0x9ddfea08eb382d69ULL;
  uint64_t a = (low ^ high) * kMul;
  a ^= (a >> 47);
  uint64_t b = (high ^ a) * kMul;
  b ^= (b >> 47);
  b *= kMul;
  return b;
}
uint64_t hash_4to8_bytes(const char *s, size_t len,
                        uint64_t seed) {
  uint64_t a = fetch32(s);
  return hash_16_bytes(len + (a << 3),
                       seed ^ fetch32(s + len - 4));
}
```

```
uint64_t hash_short(const char *s, size_t length,
                   uint64_t seed) {
  if (length >= 4 && length <= 8)
    return hash_4to8_bytes(s, length, seed);
  if (length > 8 && length <= 16)
    return hash_9to16_bytes(s, length, seed);
  if (length > 16 && length <= 32)
    return hash_17to32_bytes(s, length, seed);
  if (length > 32)
    return hash_33to64_bytes(s, length, seed);
  if (length != 0)
    return hash_1to3_bytes(s, length, seed);

  return k2 ^ seed;
}
```

```
struct hash_state {
  uint64_t h0, h1, h2, h3, h4, h5, h6, seed;
  static hash_state create(char* s, uint64_t seed);
  void mix(char* s);
  uint64_t finalize(size_t length);
};

hash_code hash_combine_range(char* start, char* stop) {
  size_t seed = ...;
  size_t length = std::distance(start, stop);
  if (length <= 64)
    return hash_short(start, length, seed);
  char *aligned_stop = start + (length & ~63);
  hash_state state = state.create(start, seed);
  start += 64;
  while (start != aligned_stop) {
    state.mix(start);
    start += 64;
  }
  if (length & 63) state.mix(stop - 64);
  return state.finalize(length);
}
```

Bottom-up inliner?
Top-down inliner?
Yes.

Disaster #2

When inlining, context is *everything*

```c
uint64_t hash_short(const char *s, size_t length,
                    uint64_t seed) {
  if (length >= 4 && length <= 8)
    return hash_4to8_bytes(s, length, seed);
  if (length > 8 && length <= 16)
    return hash_9to16_bytes(s, length, seed);
  if (length > 16 && length <= 32)
    return hash_17to32_bytes(s, length, seed);
  if (length > 32)
    return hash_33to64_bytes(s, length, seed);
  if (length != 0)
    return hash_1to3_bytes(s, length, seed);

  return k2 ^ seed;
}
```

Inline cost rewrite...

- Context sensitive cost per *call site*
- Constant value call args primary context
- Walk the callee instructions, estimate live code cost and propagate context
- Perfect dead-code cost pruning
- Uses InstructionSimplify to collapse simple constant-forming constructs during propagation.

```c
uint64_t hash_short(const char *s, size_t length,
                    uint64_t seed) {
  if (length >= 4 && length <= 8)
    return hash_4to8_bytes(s, length, seed);
  if (length > 8 && length <= 16)
    return hash_9to16_bytes(s, length, seed);
  if (length > 16 && length <= 32)
    return hash_17to32_bytes(s, length, seed);
  if (length > 32)
    return hash_33to64_bytes(s, length, seed);
  if (length != 0)
    return hash_1to3_bytes(s, length, seed);

  return k2 ^ seed;
}
```

Disaster #3

I'll come in again....

```
uint64_t hash_short(const char *s, size_t length,
                    uint64_t seed) {
  if (length >= 4 && length <= 8)
    return hash_4to8_bytes(s, length, seed);
  if (length > 8 && length <= 16)
    return hash_9to16_bytes(s, length, seed);
  if (length > 16 && length <= 32)
    return hash_17to32_bytes(s, length, seed);
  if (length > 32)
    return hash_33to64_bytes(s, length, seed);
  if (length != 0)
    return hash_1to3_bytes(s, length, seed);

  return k2 ^ seed;
}
```

```
struct hash_state {
  uint64_t h0, h1, h2, h3, h4, h5, h6, seed;
  static hash_state create(char* s, uint64_t seed);
  void mix(char* s);
  uint64_t finalize(size_t length);
};

hash_code hash_combine_range(char* start, char* stop) {
  size_t seed = ...;
  size_t length = std::distance(start, stop);
  if (length <= 64)
    return hash_short(start, length, seed);
  char *aligned_stop = start + (length & ~63);
  hash_state state = state.create(start, seed);
  start += 64;
  while (start != aligned_stop) {
    state.mix(start);
    start += 64;
  }
  if (length & 63) state.mix(stop - 64);
  return state.finalize(length);
}
```

So we're done, right?

Not even close.

- Partial SRoA
- Struct + methods vs. parameters

```
struct hash_combine_recursive_helper {
  char buffer[64], *buffer_ptr, *buffer_end;
  size_t length;
  hash_state state;
  const size_t seed;

  template <typename T> char *combine_data(T data);
  template <typename T, typename ...Ts>
  hash_code combine(const T &arg, const Ts &...args) {
    combine_data(arg);

    // Recurse to the next argument.
    return combine(args...);
  }
}
template <typename ...Ts>
hash_code hash_combine(const Ts &...args) {
  // Recursively hash each argument using a helper class.
  hash_combine_recursive_helper helper;
  return helper.combine(args...);
}
```

So we're done, right?

Not even close.

- Partial SRoA
- Struct + methods vs. parameters
- LTO-aware top-down / bottom-up inline strategy balancing
- Function splitting or "outlining"
- Function merging
- ???
- PROFIT!

Questions?

chandlerc@gmail.com