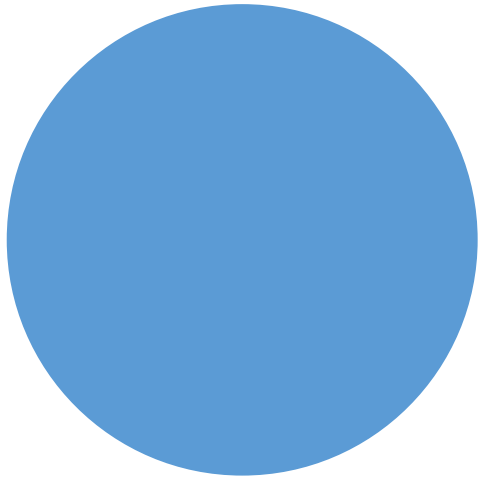
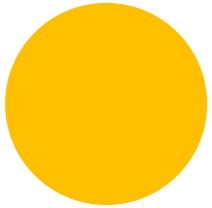



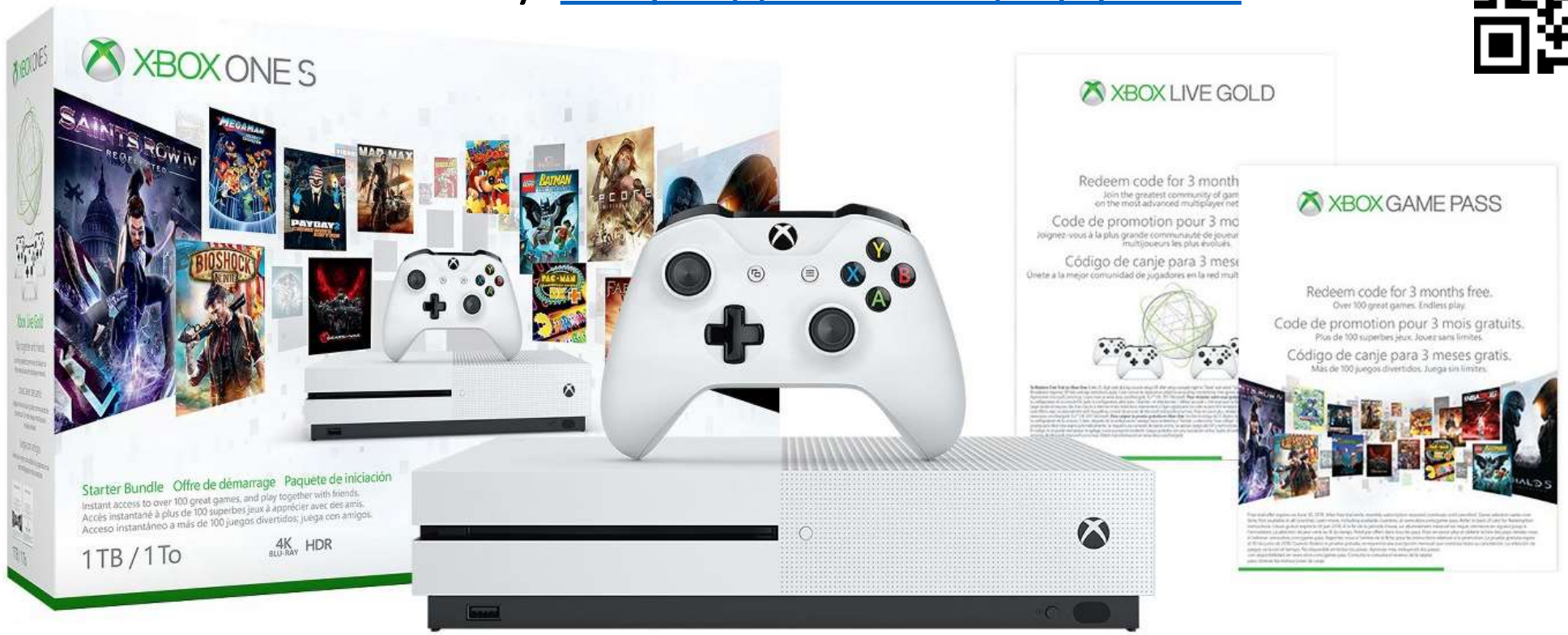
# Nano-coroutines

Curing your memory latency blues.

Negative overhead abstraction. Again?

Microsoft Visual C++ Team
Gor Nishanov

# Take our survey https://aka.ms/cppcon




# You can win an Xbox One S - Starter Bundle

9/27 10:30 – 12:00 // Breckenridge Hall (1st Floor)

**Thoughts on a More Powerful and Simpler C++ (5 of N), *Herb Sutter***

Reminder: A coroutine is a generalization of a function

# Coroutines

60 years ago

- Introduced in 1958 by Melvin Conway
- Donald Knuth, 1968: “generalization of subroutine”

| | subroutines | coroutines |
|---|---|---|
| call | Allocate frame, pass parameters | Allocate frame, pass parameters |
| return | Free frame, return result | Free frame, return eventual result |
| suspend | x | yes |
| resume | x | yes |

## 8.4 Function definitions [dcl.fct.def]

### 8.4.4 Coroutines [dcl.fct.def.coroutine]

Add this subclause to 8.4.

A function is a *coroutine* if it contains a *coroutine-return-statement* (6.6.3.1), an *await-expression* (5.3.8), a *yield-expression* (5.20), or a range-based `for` (6.5.4) with `co_await`.

```
generator<char> hello() {
   for (char ch: "Hello, world\n")
      co_yield ch;
}

int main() {
   for (char ch : hello())
      cout << ch;
}
```

```
task<void> sleepy() {
   cout << “Going to sleep…\n";
   co_await sleep_for(1ms);
   cout << “Woke up\n";
   co_return 42;
}

int main() {
   cout << sleepy.get();
}
```

# Coroutines Design Principles

- **Scalable** (to **b**illions of concurrent coroutines)
- **Efficient** (resume and suspend operations comparable in cost to a function call overhead)
- Seamless interaction with existing facilities **with no overhead**
- **Open** ended coroutine machinery allowing library designers to develop coroutine libraries exposing various high-level semantics, such as generators, goroutines, tasks and more.
- Usable in environments where **exception** are forbidden or not available

C++ source #1

```
template <typename T>
generator<T> seq() {
  for (T i = {};; ++i)
    co_yield i;
}

template <typename T>
generator<T> take_until(generator<T>& g, T limit) {
  for (auto&& v: g)
    if (v < limit) co_yield v;
    else break;
}

template <typename T>
generator<T> multiply(generator<T>& g, T factor) {
  for (auto&& v: g)
    co_yield v * factor;
}

template <typename T>
generator<T> add(generator<T>& g, T adder) {
  for (auto&& v: g)
    co_yield v + adder;
}

int main() {
  auto s = seq<int>();
  auto t = take_until(s, 10);
  auto m = multiply(t, 2);
  auto a = add(m, 110);
  return std::accumulate(a.begin(), a.end(), 0);
}
```

x86-64 clang 5.0.0 (Editor #1, Compiler #1)

x86-64 clang 5.0.0

-std=c++14 -O2 -stdlib=libc++ -fcoroutines-ts

11010 .LX0: .text // \s+ Intel Demangle

```
main:                                   # @main
        mov     eax, 1190
        ret
```

https://godbolt.org/g/26viuZ

clang version 5.0.0 (tags/RELEASE_500/final 312636)- cached

```
struct session {
  session(net::io_context &ioc, net::ip::tcp::socket s, size_t block_size)
      : io_context_(ioc), socket_(std::move(s)), block_size_(block_size),
        buf_(block_size), read_data_length_(0)
  {}

  void start() {
    std::error_code set_option_err;
    net::ip::tcp::no_delay no_delay(true);
    socket_.set_option(no_delay, set_option_err);
    if (!set_option_err) {
      socket_.async_read_some( net::buffer(buf_.data(), block_size_),
        make_custom_alloc_handler( allocator_,
          [this](auto ec, auto n) { handle_read(ec, n); }));
      return;
    }

    net::post(io_context_, [this] { destroy(this); });
  }

  void handle_read(const std::error_code &err, size_t length) {
    if (!err) {
      read_data_length_ = length;
      async_write(socket_, net::buffer(buf_.data(), read_data_length_),
        make_custom_alloc_handler( allocator_,
          [this](auto ec, auto) { handle_write(ec); }));
      return;
    }

    net::post(io_context_, [this] { destroy(this); });
  }
```

```
  void handle_write(const std::error_code &err) {
    if (!err) {
      socket_.async_read_some(net::buffer(buf_.data(), block_size_)
        make_custom_alloc_handler( allocator_,
          [this](auto ec, auto n) { handle_read(ec, n); }));
      return;
    }

    net::post(io_context_, [this] { destroy(this); });
  }

  static void destroy(session *s) { delete s; }

private:
  net::io_context &io_context_;
  net::ip::tcp::socket socket_;
  size_t block_size_;
  std::vector<char> buf_;
  size_t read_data_length_;
  handler_allocator allocator_;
};
```

```
struct server {
  server(net::io_context &ioc, const net::ip::tcp::endpoint &endpoint,
         size_t block_size)
      : io_context_(ioc), acceptor_(ioc, endpoint), block_size_(block_size)
  {
    acceptor_.listen();

    start_accept();
  }

  void start_accept()
  {
    acceptor_.async_accept(
        [this](auto ec, auto s) { handle_accept(ec, std::move(s)); });
  }

  void handle_accept(std::error_code err, net::ip::tcp::socket s)
  {
    if (!err) {
      session *new_session = new session(io_context_, std::move(s), block_size_);
      new_session->start();
    }
    start_accept();
  }

private:
  net::io_context &io_context_;
  net::ip::tcp::acceptor acceptor_;
  size_t block_size_;
};
```

# Coroutine based server

```
task<void> session(tcp::socket s, size_t block_size)
{
  s.set_option(tcp::no_delay(true));
  std::vector<char> buf(block_size);

  for(;;) {
    size_t n = co_await async_read_some(s, buffer(buf.data(), block_size));
    n = co_await async_write(s, buffer(buf.data(), n));
  }
}
```

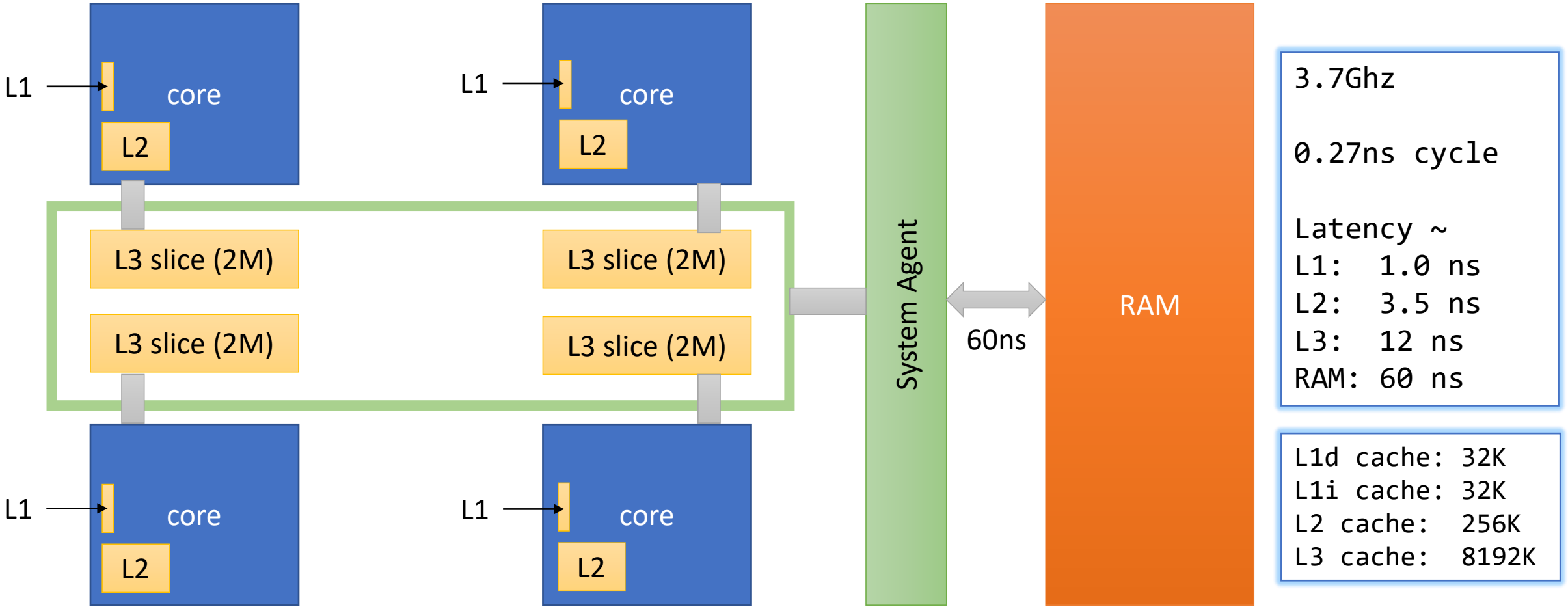
```
task<void> server(io_context &io, tcp::endpoint const &endpoint,
                  size_t block_size)
{
  tcp::acceptor acceptor(io, endpoint);
  acceptor.listen();

  for (;;)
    spawn(io, session(co_await async_accept(acceptor), block_size));
}
```

# Nano-coroutines?

# Coroutines and databases

- Interleaving with Coroutines: A Practical Approach for Robust Index Joins (2017)
  - https://infoscience.epfl.ch/record/231318/files/p230-psaropoulos.pdf
- Exploiting Coroutines to Attack the “Killer Nanoseconds” (2018)
  - http://www.vldb.org/pvldb/vol11/p1702-jonathan.pdf

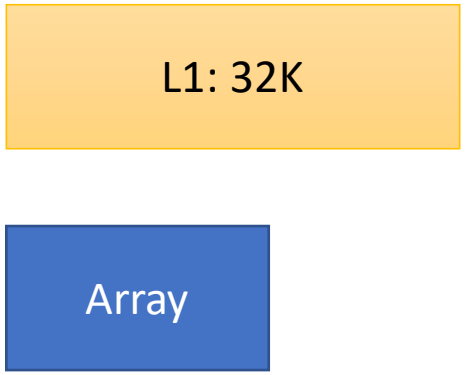
# Skylake Xeon(R) CPU E3-1505M v5



# Binary Search

```
template <typename Iterator, typename Value>
bool binary_search(Iterator first, Iterator last, Value val) {
  auto len = last - first;
  while (len > 0) {
    auto half = len / 2;
    auto middle = first + half;
    auto middle_key = *middle;
    if (middle_key < val) {
      first = middle + 1;
      len = len - half - 1;
    } else
      len = half;
    if (middle_key == val)
      return true;
  }
  return false;
}
```

# Array comfortably fits L1

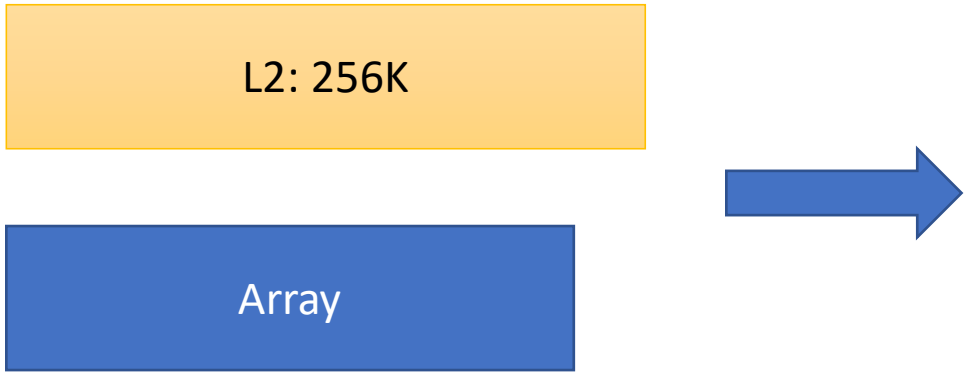



4 ns per lookup/log2(array-size)
0.1% L1 cache misses

```
3.7Ghz

0.27ns cycle

Latency ~
L1:   1.0 ns
L2:   3.5 ns
L3:   12 ns
RAM: 60 ns
```

```
L1d cache: 32K
L1i cache: 32K
L2 cache:  256K
L3 cache:  8192K
```

# Array comfortably fits L2

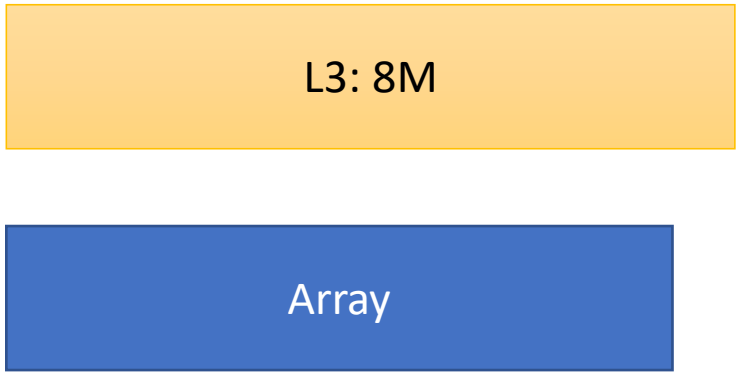



4.7 ns per lookup/log2(array-size)
**47%** L1 cache misses

```
3.7Ghz

0.27ns cycle

Latency ~
L1:   1.0 ns
L2:   3.5 ns
L3:   12 ns
RAM: 60 ns
```

```
L1d cache: 32K
L1i cache: 32K
L2 cache:  256K
L3 cache:  8192K
```

# Array comfortably fits L3

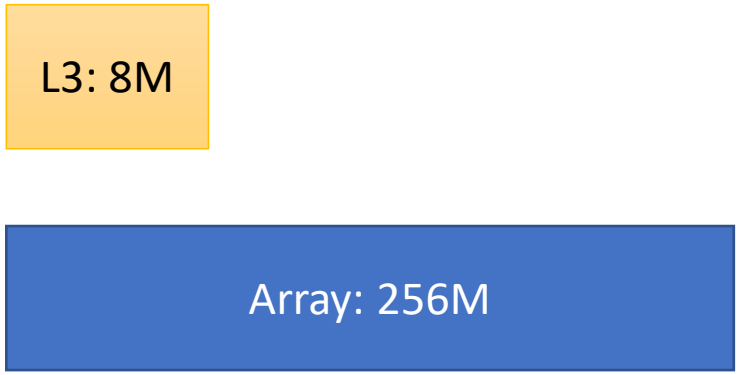



8.3 ns per lookup/log2(array-size)
**274%** L1 cache misses

```
3.7Ghz

0.27ns cycle

Latency ~
L1:   1.0 ns
L2:   3.5 ns
L3:   12 ns
RAM: 60 ns
```

```
L1d cache: 32K
L1i cache: 32K
L2 cache:  256K
L3 cache:  8192K
```

# Array is way bigger than L3




26 ns per lookup/log2(array-size)
**60% L3** cache misses

0.21 instructions per cycle

```
3.7Ghz

0.27ns cycle

Latency ~
L1:   1.0 ns
L2:   3.5 ns
L3:   12 ns
RAM: 60 ns
```

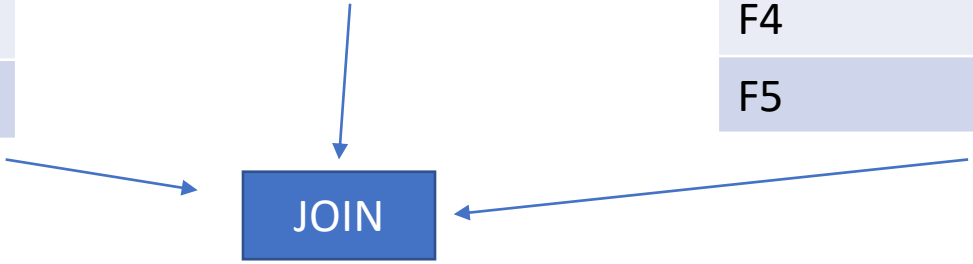
```
L1d cache:  32K
L1i cache:  32K
L2 cache:   256K
L3 cache:   8192K
```

# Database Join

| CustomerId | Date | Food |
|---|---|---|
| C1 | 2018/06/01 | F1 |
| C1 | 2018/07/11 | F2 |
| C2 | 2018/08/03 | F3 |
| C2 | 2018/09/01 | F4 |
| C3 | 2018/09/13 | F5 |

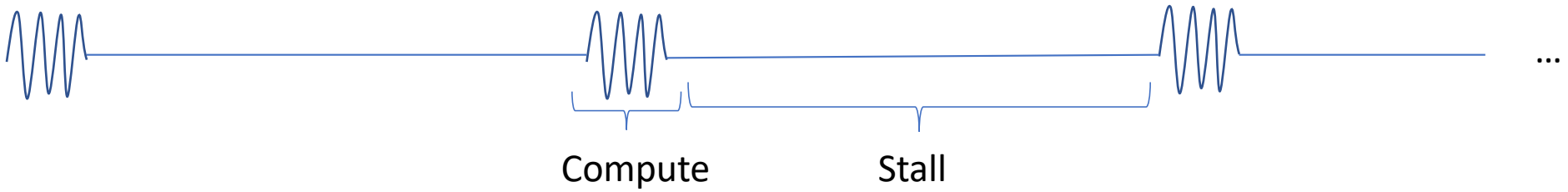
| CustomerId | Name |
|---|---|
| C1 | Reis |
| C2 | Stepanov |
| C3 | Smith |

| Food Code | Name |
|---|---|
| F1 | Fois Gras |
| F2 | Hamburger |
| F3 | Pirozhki |
| F4 | BBQ Alligator Ribs |
| F5 | Porridge |




| Name | Date | Food |
|---|---|---|
| Reis | 2018/06/01 | Fois Gras |
| Reis | 2018/07/11 | Hamburger |
| Stepanov | 2018/08/03 | Pirozhki |
| Stepanov | 2018/09/01 | BBQ Alligator Ribs |
| Smith | 2018/09/13 | Porridge |

# Very sad core

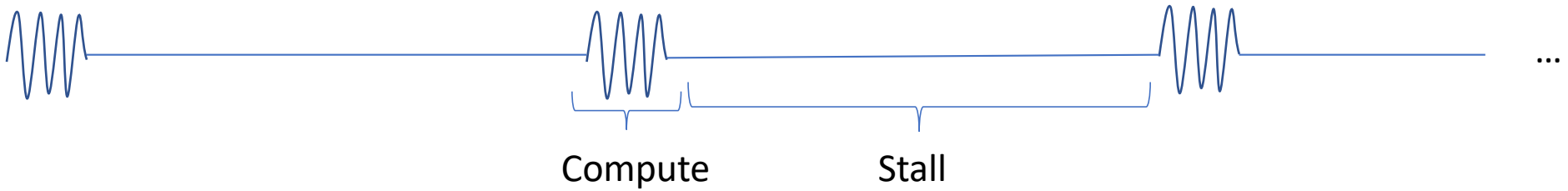


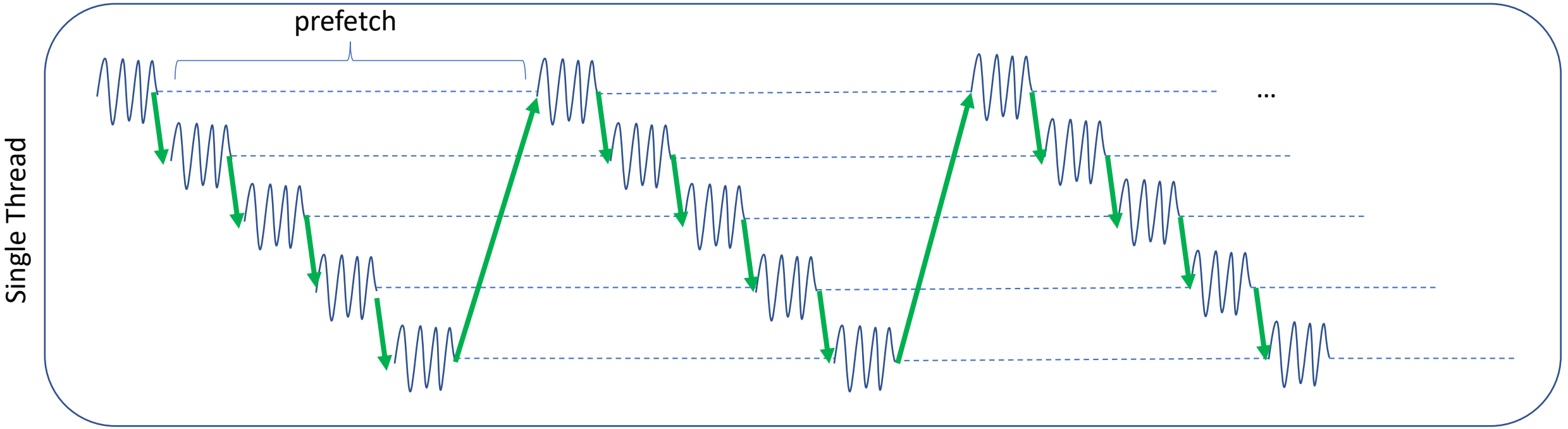
# Prefetching

```
#include <xmmintrin.h>


#define _MM_HINT_T0     1
#define _MM_HINT_T1     2
#define _MM_HINT_T2     3
#define _MM_HINT_NTA    0

void _mm_prefetch (
        char const *addr,
        int hint);
```

T0: prefetch data into all levels of the cache hierarchy
T1: prefetch data into level 2 cache or higher
T2: prefetch data into level 3 cache or higher
NTA: prefetch data to the closest cache to CPU minimizing cache pollution

# Instruction stream interleaving



# Binary Search

```
bool binary_search(int const* first, int const* last, int val) {
  auto len = last - first;
  while (len > 0) {
    auto half = len / 2;
    auto middle = first + half;
    auto middle_key = *middle;
    if (middle_key < val) {
      first = middle + 1;
      len = len - half - 1;
    } else
      len = half;
    if (middle_key == val)
      return true;
  }
  return false;
}
```

# Binary Search – state extraction

```
  auto len = last - first;
  while (len > 0) {
    auto half = len / 2;
    auto middle = first + half;
    auto middle_key = *middle;
    if (middle_key < val) {
      first = middle + 1;
      len = len - half - 1;
    } else
      len = half;
    if (middle_key == val)
      return true;
  }
  return false;
}
```

```
struct frame {
  int const* first;
  int const* last;
  int const* middle;
  size_t len;
  size_t half;
  int val;
  int state = EMPTY;

  void init(int const* first,
            int const* last, int val);

  bool run();

};
```

# Binary Search – init

```
  auto len = last - first;
  while (len > 0) {
    auto half = len / 2;
    auto middle = first + half;
    auto middle_key = *middle;
    if (middle_key < val) {
      first = middle + 1;
      len = len - half - 1;
    } else
      len = half;
    if (middle_key == val)
      return true;
  }
  return false;
}
```

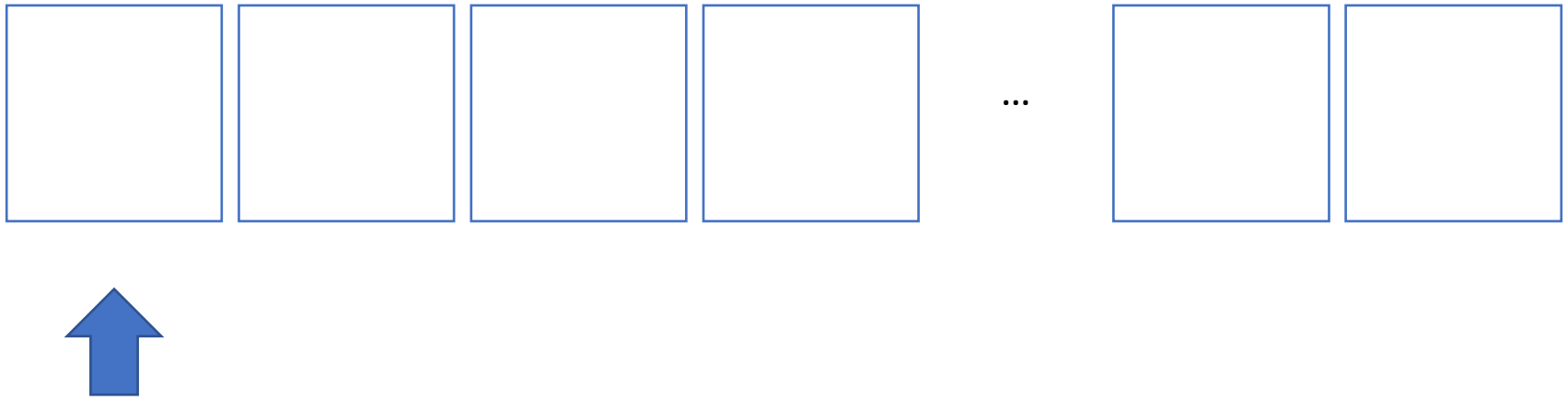
```
void frame::init(int const* first,
                 int const* last, int val)
{
  this->val = val;
  this->first = first;
  this->last = last;
  this->len = last - first;

  if (len == 0) {
    state = NOT_FOUND;
    return;
  }

  half = len / 2;
  middle = first + half;
  state = KEEP_GOING;
  prefetch(*middle);
}
```

# Binary Search – run

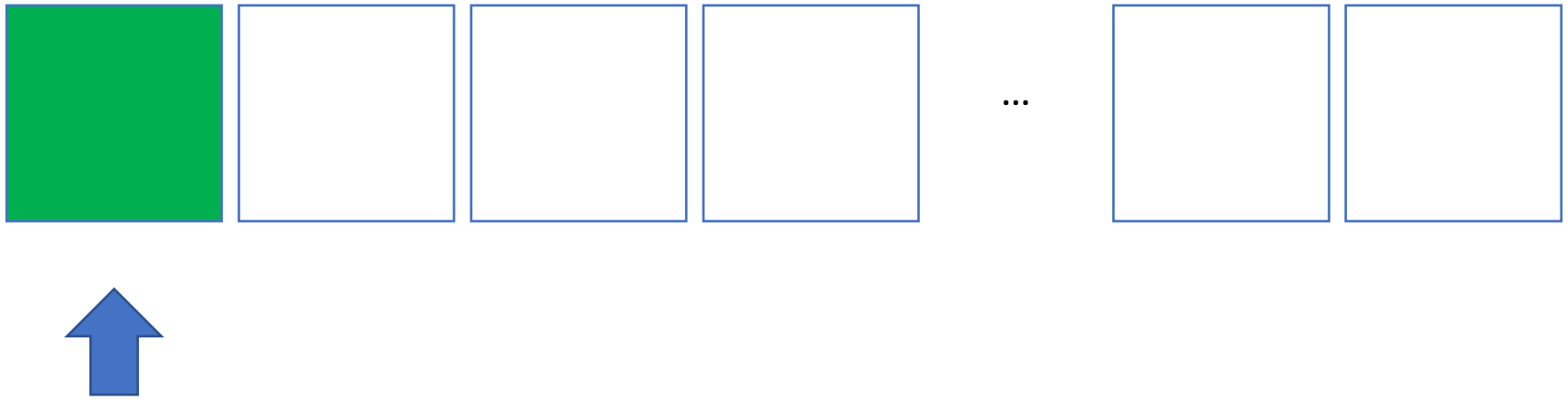
```
  auto len = last - first;
  while (len > 0) {
    auto half = len / 2;
    auto middle = first + half;
    auto middle_key = *middle;
    if (middle_key < val) {
      first = middle + 1;
      len = len - half - 1;
    } else
      len = half;
    if (middle_key == val)
      return true;
  }
  return false;
}
```

```
bool frame::run() {
  auto middle_key = *middle;
  if (middle_key < val) {
    first = middle + 1;
    len = len - half - 1;
  } else
    len = half;
  if (middle_key == val) {
    state = FOUND;
    return true;
  }
  if (len > 0) {
    half = len / 2;
    middle = first + half;
    prefetch(*middle);
    return false;
  }
  state = NOT_FOUND;
  return true;
}
```

# Run through

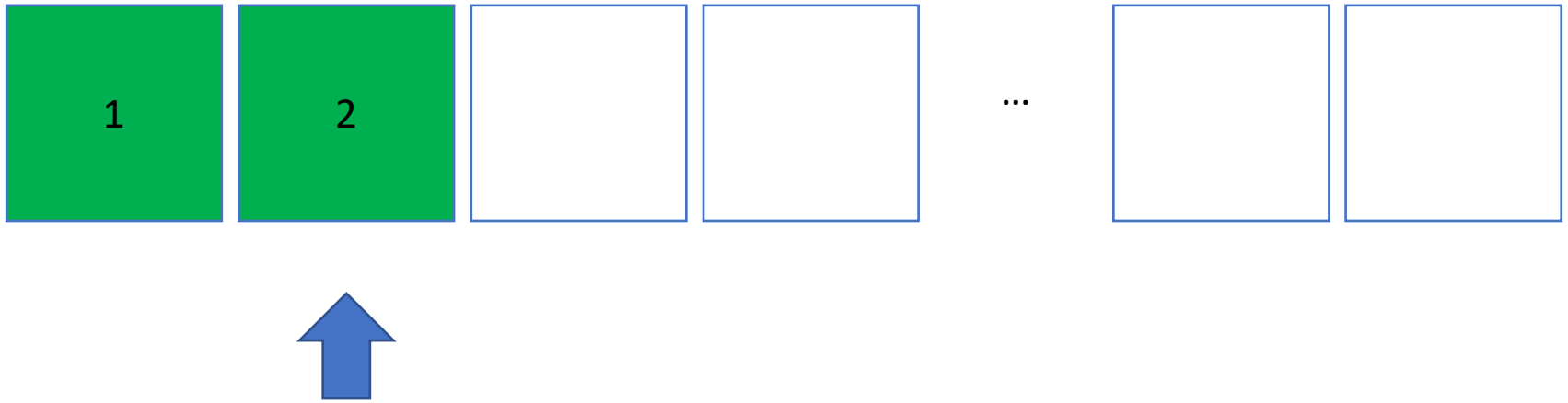


# Run through



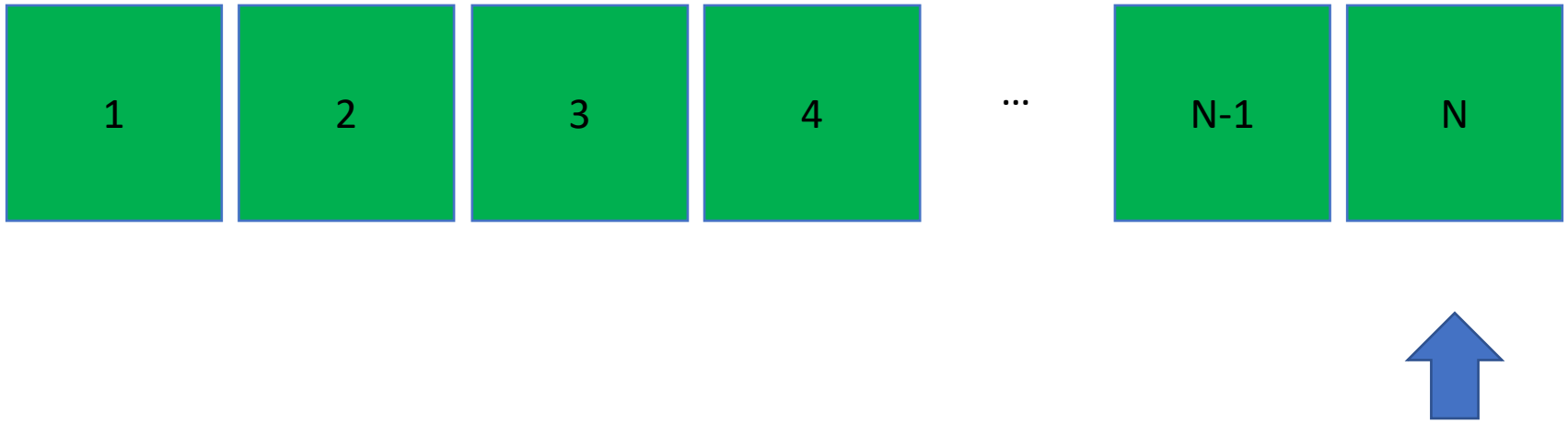

Run Init:
Start prefetch for frame 1

# Run through



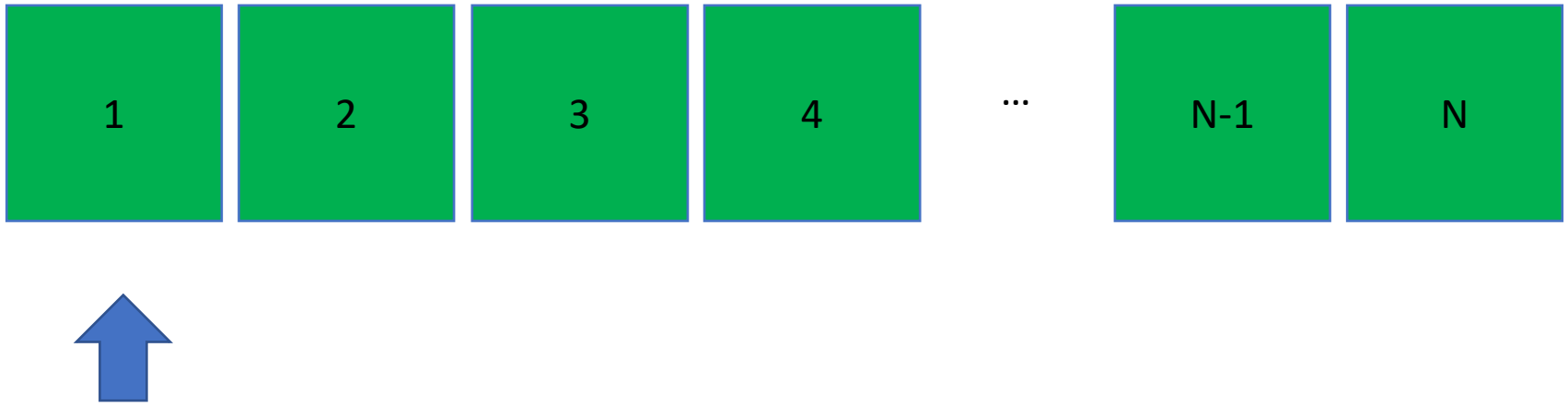

Run Init:
Start prefetch for frame 2

# Run through



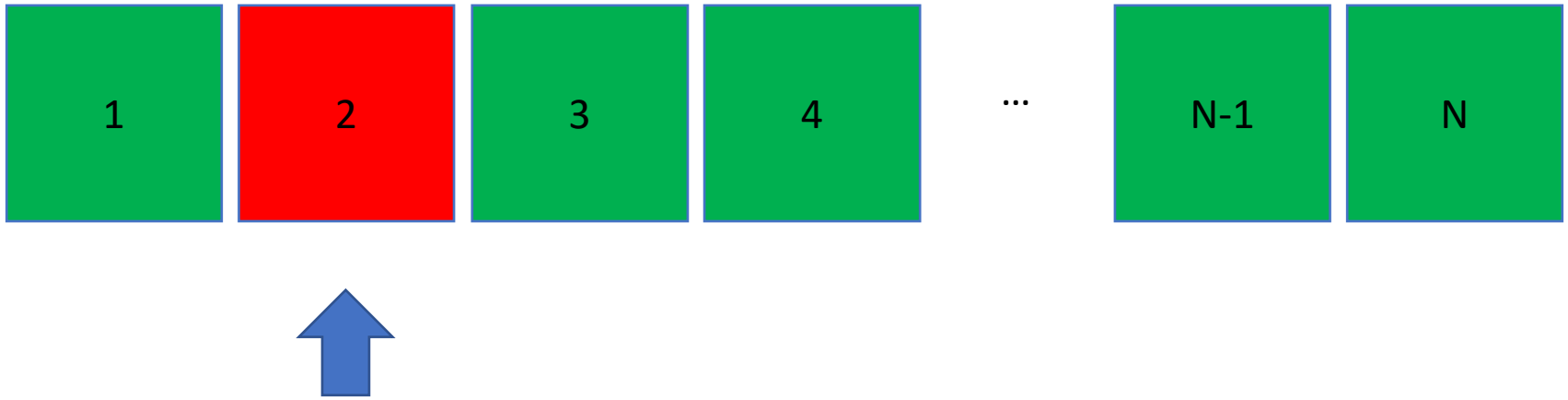

Run Init:
Start prefetch for frame N

# Run through



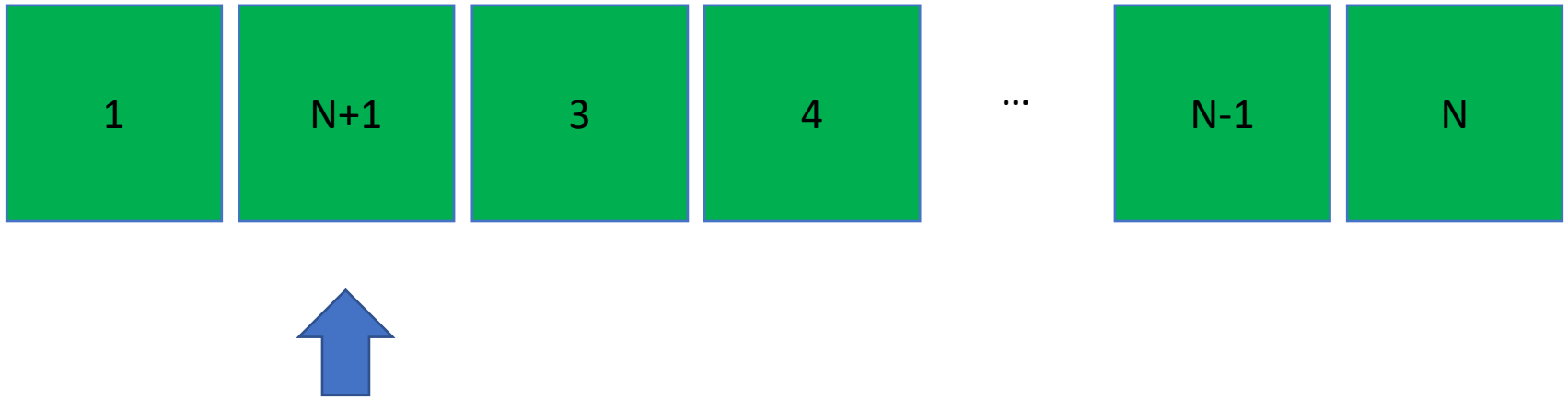

Run until we hit next
prefetch from frame 1

# Run through




Run stumbled on a value

# Run through




Run Init for value + 1:
Start prefetch

# Binary Search – orchestrator

```
std::vector<frame> f(part);
size_t N = part - 1;
size_t i = N;
long result = 0;

for (auto key: lookups) {
  auto* fr = &f[i];
  if (fr->state != KEEP_RUNNING) {
    fr->init(v.begin(), v.end(), key);
    if (i == 0)  i = N;  else --i;
  } else {
    for (;;) {
       … keep running until there is room …
    }
}

… deal with strugglers …
```

```
for (;;) {
  if (fr->run()) {
     // completed
     if (fr->state == FOUND) …;
     fr->init(v.begin(), v.end(), key);

     if (i == 0) i = N; else --i;
     break;
  }
  if (i == 0) i = N; else --i;
  fr = &f[i];
}
```

```
bool moreWork = false;
do {
  moreWork = false;
  for (auto& fr : f)
    if (fr.state == KEEP_RUNNING) {
      moreWork = true;
      if (fr.run() && fr.state == FOUND)
        …;
    }
} while (moreWork);
```

# Array is way bigger than L3

L3: 8M

Array: 256M

26 ns per lookup/log2(array-size)
**60% L3** cache misses

0.21 instructions per cycle

```
3.7Ghz

0.27ns cycle

Latency ~
L1:   1.0 ns
L2:   3.5 ns
L3:   12 ns
RAM: 60 ns
```

Not bad!

Interleaving with 16 streams

**10.0** ns per lookup/log2(array-size)
**3.5% L3** cache misses

0.83 instructions per cycle

```
L1d cache:  32K
L1i cache:  32K
L2 cache:   256K
L3 cache:   8192K
```

# Nice! But that is a lot of code!

```
struct frame {
  int const* first;
  int const* last;
  int const* middle;
  size_t len;
  size_t half;
  int val;
  int state = EMPTY;

  void init(int const* first,
            int const* last, int val);

  void run();

};

void frame::init(int const* first,
                 int const* last, int val)
{
  this->val = val;
  this->first = first;
  this->last = last;
  this->len = last - first;

  if (len == 0) {
    state = NOT_FOUND;
    return;
  }

  half = len / 2;
  middle = first + half;
  state = KEEP_GOING;
  prefetch(*middle);
}
```

```
bool frame::run() {
  auto middle_key = *middle;
  if (middle_key < val) {
    first = middle; ++first;
    len = len - half - 1;
  } else
    len = half;
  if (middle_key == val) {
    state = FOUND;
    return true;
  }
  if (len > 0) {
    half = len / 2;
    middle = first + half;
    prefetch(*middle);
    return false;
  }
  state = NOT_FOUND;
  return true;
}
```

```
std::vector<frame> f(part);
size_t N = part - 1;
size_t i = N;
long result = 0;

for (auto key: lookups) {
  auto* fr = &f[i];
  if (fr->state != KEEP_RUNNING) {
    fr->init(v.begin(), v.end(), key);
    if (i == 0)  i = N;  else --i;
  } else {
    for (;;) {
       … keep running until there is room …
    }
}

… deal with strugglers …
```

```
for (;;) {
  if (fr->run()) {
     // completed
     if (fr->state == FOUND) …;
     fr->init(v.begin(), v.end(), key);

     if (i == 0) i = N; else --i;
     break;
  }
  if (i == 0) i = N; else --i;
  fr = &f[i];
}
```

```
bool moreWork = false;
do {
  moreWork = false;
  for (auto& fr : f)
    if (fr.state == KEEP_RUNNING) {
      moreWork = true;
      if (fr.run() && fr.state == FOUND)
        …;
    }
} while (moreWork);
```


Can we do better?

# Coroutine based server

```
task<void> session(tcp::socket s, size_t block_size)
{
  s.set_option(tcp::no_delay(true));
  std::vector<char> buf(block_size);

  for(;;) {
    size_t n = co_await async_read_some(s, buffer(buf.data(), block_size));
    n = co_await async_write(s, buffer(buf.data(), n));
  }
}
```

```
task<void> server(io_context &io, tcp::endpoint const &endpoint,
                  size_t block_size)
{
  tcp::acceptor acceptor(io, endpoint);
  acceptor.listen();

  for (;;)
    spawn(io, session(co_await async_accept(acceptor), block_size));
}
```

# Binary Search

```
template <typename Iterator, typename Value>
bool binary_search(Iterator first, Iterator last, Value val) {
  auto len = last - first;
  while (len > 0) {
    auto half = len / 2;
    auto middle = first + half;
    auto middle_key = *middle;
    if (middle_key < val) {
      first = middle + 1;
      len = len - half - 1;
    } else
      len = half;
    if (middle_key == val)
      return true;
  }
  return false;
}
```

# Binary Search

```
template <typename Iterator, typename Value, typename Found, typename NotFound>
auto binary_search(Iterator first, Iterator last, Value val,
                   Found on_found, NotFound on_not_found) -> void {
  auto len = last - first;
  while (len > 0) {
    auto half = len / 2;
    auto middle = first + half;
    auto middle_key = *middle;
    if (middle_key < val) {
      first = middle + 1;
      len = len - half - 1;
    } else
      len = half;
    if (middle_key == val)
      return on_found(val, middle);
  }
  return on_not_found(val);
}
```

# Binary Search

```
template <typename Iterator, typename Value, typename Found, typename NotFound>
auto binary_search(Iterator first, Iterator last, Value val,
                   Found on_found, NotFound on_not_found) -> task<void> {
  auto len = last - first;
  while (len > 0) {
    auto half = len / 2;
    auto middle = first + half;
    auto middle_key = co_await prefetch(*middle);
    if (middle_key < val) {
      first = middle + 1;
      len = len - half - 1;
    } else
      len = half;
    if (middle_key == val)
      co_return on_found(val, middle);
  }
  co_return on_not_found(val);
}
```

# Coroutine based server

```
task<void> session(tcp::socket s, size_t block_size)
{
  s.set_option(tcp::no_delay(true));
  std::vector<char> buf(block_size);

  for(;;) {
    size_t n = co_await async_read_some(s, buffer(buf.data(), block_size));
    n = co_await async_write(s, buffer(buf.data(), n));
  }
}
```

```
task<void> server(io_context &io, tcp::endpoint const &endpoint,
                  size_t block_size)
{
  tcp::acceptor acceptor(io, endpoint);
  acceptor.listen();

  for (;;)
    spawn(io, session(co_await async_accept(acceptor), block_size));
}
```

# Orchestrator

```
void MultiLookup(std::vector<int> const& v,
                 std::vector<int> const& lookups, int concurrency)
{
  size_t found_count = 0;
  size_t not_found_count = 0;

  throttler t(concurrency);

  for (auto key: lookups)
    t.spawn(BinarySearch(v.begin(), v.end(), key,
                         [&](auto) { ++found_count; },
                         [&](auto) { ++not_found_count; }));

  t.join();
}
```

# Suspend while prefetching (1/3)

```
template <typename T>
auto prefetch(T &value) {
   return prefetch_Awaitable<T>{value};
}
```

```
co_await prefetch(*middle);
```

# `co_await` <expr>

Expands into an expression equivalent of

```
{
  auto && tmp = <expr>;
  if (! tmp.await_ready()) {
     tmp.await_suspend(<coroutine-handle>);
     
  }
  return  tmp.await_resume();
}
```

# Suspend while prefetching (2/3)

```
template <typename T> struct prefetch_Awaitable {
   T& value;

   prefetch_Awaitable(T& value) : value(value) {}

   bool await_ready() { return false; }

   auto await_suspend(coroutine_handle<> h) {
     _mm_prefetch(static_cast<char const *>(std::addressof(value)),
                  _MM_HINT_NTA);
     scheduler.push_back(h);

   }

   T& await_resume() { return value; }
};
```

# Suspend while prefetching (3/3)

```
template <typename T> struct prefetch_Awaitable {
   T& value;

   prefetch_Awaitable(T& value) : value(value) {}

   bool await_ready() { return false; }

   auto await_suspend(coroutine_handle<> h) {
     _mm_prefetch(static_cast<char const *>(std::addressof(value)),
                  _MM_HINT_NTA);
     scheduler.push_back(h);
     return scheduler.pop_front();
   }

   T& await_resume() { return value; }
};
```

# Just a queue

```
template <size_t N> struct scheduler_queue {
  size_t head = 0;
  size_t tail = 0;
  coroutine_handle<> arr[N];

  void push_back(coro_handle h) {
    arr[head] = h;
    head = (head + 1) % N;
  }
  auto pop_front() {
    auto result = arr[tail];
    tail = (tail + 1) % N;
    return result;
  }
  auto try_pop_front() { return head != tail ? pop_front() : coroutine_handle<>{}; }
  void run() { while (auto h = try_pop_front()) h.resume(); }
};
```

# Orchestrator

```
void MultiLookup(std::vector<int> const& v,
                 std::vector<int> const& lookups, int concurrency)
{
  size_t found_count = 0;
  size_t not_found_count = 0;

  throttler t(concurrency);

  for (auto key: lookups)
    t.spawn(BinarySearch(v.begin(), v.end(), key,
                         [&](auto) { ++found_count; },
                         [&](auto) { ++not_found_count; }));

  t.join();
}
```

# Binary Search

```
template <typename Iterator, typename Value, typename Found, typename NotFound>
auto binary_search(Iterator first, Iterator last, Value val,
                   Found on_found, NotFound on_not_found) -> task<void> {
  auto len = last - first;
  while (len > 0) {
    auto half = len / 2;
    auto middle = first + half;
    auto middle_key = co_await prefetch(*middle);
    if (middle_key < val) {
      first = middle + 1;
      len = len - half - 1;
    } else
      len = half;
    if (middle_key == val)
      co_return on_found(val, middle);
  }
  co_return on_not_found(val);
}
```

# Binary Search

```
template <typename Iterator, typename Value, typename Found, typename NotFound>
auto binary_search(Iterator first, Iterator last, Value val,
                   Found on_found, NotFound on_not_found) -> root_task {
  auto len = last - first;
  while (len > 0) {
    auto half = len / 2;
    auto middle = first + half;
    auto middle_key = co_await prefetch(*middle);
    if (middle_key < val) {
      first = middle + 1;
      len = len - half - 1;
    } else
      len = half;
    if (middle_key == val)
      co_return on_found(val, middle);
  }
  co_return on_not_found(val);
}
```

# Throttler

```
struct throttler {
  size_t limit;
  explicit throttler(size_t limit) : limit(limit) {}

  void spawn(root_task t) {
    if (limit == 0)
       scheduler.pop_front().resume();

    auto h = t.set_owner(this); // tell the task we own it
    scheduler.push_back(h);
    --limit;
  }
  void on_task_done() { ++limit; } // called when root_task completes
  void join() { scheduler.run(); }
  ~throttler() { join(); }
};
```

# Tweaks to root_task

```
struct root_task {
   struct promise_type {
      throttler *owner = nullptr;
      suspend_never final_suspend() { owner->on_task_done(); return {}; }
      …
   };

  auto set_owner(throttler *owner) {
    auto result = h;
    h.promise().owner = owner;
    h = nullptr;
    return result;
  }

  ~root_task() { if (h) h.destroy(); }
private:
  coroutine_handle<promise_type> h;
};
```

# Using recycling allocator

```
struct root_task {
   struct promise_type {
      throttler *owner = nullptr;
      suspend_never final_suspend() { owner->on_task_done(); return {}; }
      …
      void *operator new(size_t sz) { return recycle.alloc(sz); }
      void operator delete(void *p, size_t sz) { recycle.free(p, sz); }
   };
   …
};
```

# Array is bigger than L3

L3: 8M

Array: 256M

**Naive**

**26 ns** per lookup/log2(array-size)
60% L3 cache misses

0.21 instructions per cycle

**Hand-crafted state machine**
Interleaving with 16 streams

**10 ns** per lookup/log2(array-size)
3.5% L3 cache misses

0.83 instructions per cycle

**Coroutine based**
Interleaving with 20 streams

**7.56 ns** per lookup/log2(array-size)
2.56% L3 cache misses

1.46 instructions per cycle

```
3.7Ghz

0.27ns cycle

Latency ~
L1:   1.0 ns
L2:   3.5 ns
L3:   12 ns
RAM:  60 ns
```

```
L1d cache:  32K
L1i cache:  32K
L2 cache:   256K
L3 cache:   8192K
```

# Negative overhead abstraction, again???

# Nano-coroutines

- Beats straightforward hand-crafted state machine
- Looks similar to naïve code
- Allows to use prefetching and instruction stream interleaving techniques for more complicated algorithms that is possible by hand

# Coroutines – current status

- Proposal is working through C++ standardization committee (C++20?)
- Implementation in
  - VS 2015+,
  - Clang 5.1+
- GCC Implementation is in progress
- No longer naked
  - task<T> type – in standardization pipeline
  - generator<T> type – proposal coming soon
  - sync_wait, when_all – proposals coming soon

# Other sessions

**Monday, September 24th**

- 14:00 – 15:00
  **How to Write Well-Behaved Value Wrappers**
  - by Simon Brand
- 15:15 – 16:15
  **How C++ Debuggers Work**
  - by Simon Brand

**Tuesday, September 25th**

- 14:00 – 15:00
  **What Could Possibly Go Wrong?: A Tale of Expectations and Exceptions**
  - by Simon Brand and Phil Nash
- 15:15 – 15:45
  **Overloading: The Bane of All Higher-Order Functions**
  - by Simon Brand

**Wednesday, September 26th**

- 12:30 – 13:30
  **C++ Community Building Birds of a Feather**
  - with Stephan T. Lavavej and others
- 14:00 – 15:00
  **Latest and Greatest in the Visual Studio Family for C++ Developers 2018**
  - by Marian Luparu and Steve Carroll
- 15:15 – 15:45
  **Don't Package Your Libraries, Write Packagable Libraries!**
  - by Robert Schumacher

**Wednesday, September 26th**

- 15:15 – 15:45
  **What's new in Visual Studio Code for C++ Development**
  - by Rong Lu
- 15:50 – 16:20
  **Value Semantics: Fast, Safe, and Correct by Default**
  - by Nicole Mazzuca
- 16:45 – 17:45
  **Memory Latency Troubles You? Nano-coroutines to the Rescue! (Using Coroutines TS, of Course)**
  - by Gor Nishanov
- 18:45 – 20:00
  **Cross-Platform C++ Development is Challenging – let Tools Help!**
  - by Marc Goodner and Will Buik

**Thursday, September 27th**

- 9:00 – 10:00
  **Inside Visual C++'s Parallel Algorithms**
  - by Billy O'Neal
- 15:15 – 15:45
  **ConcurrencyCheck – Static Analyzer for Concurrency Issues in Modern C++**
  - by Anna Gringauze
- 16:45 – 17:45
  **Class Template Argument Deduction for Everyone**
  - by Stephan T. Lavavej

# Questions?

# Backup

# Orchestrator (no root_task)

```
void MultiLookup(std::vector<int> const& v,
                 std::vector<int> const& lookups, int concurrency)
{
  …
  for (auto key: lookups)
    t.spawn([&] { BinarySearch(v.begin(), v.end(), key,
                          [&](auto) { ++found_count; },
                          [&](auto) { ++not_found_count; });} );
  …
}
```

```
template <typename F>
void spawn(F f) {
  spawn([](F f)->root_task { co_await f(); }(f) );
}
```