What Do You Mean by “Cache Friendly”?

Björn Fahller

```c
typedef uint32_t (*timer_cb)(void*);
struct timer {
    uint32_t deadline;
    timer_cb callback;
    void* userp;
    struct timer* next;
    struct timer* prev;
};

static timer timeouts = { 0, NULL, NULL, &timeouts, &timeouts };

timer* schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
    timer* iter = timeouts.prev;
    while (iter != &timeouts && is_after(iter->deadline, deadline))
        iter = iter->prev;
    add_behind(iter, deadline, cb, userp);
}
```

```c
typedef uint32_t (*timer_cb)(void*);
struct timer {
    uint32_t deadline;
    timer_cb callback;
    void* userp;
    struct timer* next;
    struct timer* prev;
};

static timer timeouts = { 0, NULL, NULL, &timeouts, &timeouts };

timer* schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
    timer* iter = timeouts.prev;
    while (iter != &timeouts && is_after(iter->deadline, deadline))
        iter = iter->prev;
    add_behind(iter, deadline, cb, userp);
}
```

```
typedef uint32_t (*timer_cb)(void*);
struct timer {
    uint32_t deadline;
    timer_cb callback;
    void* userp;
    struct timer* next;
    struct timer* prev;
};

static timer timeouts = { 0, NULL, NULL, &timeouts, &timeouts };

timer* schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
    timer* iter = timeouts.prev;
    while (iter != &timeouts && is_after(iter->deadline, deadline))
        iter = iter->prev;
    add_behind(iter, deadline, cb, userp);
}
```

```c
typedef uint32_t (*timer_cb)(void*);
struct timer {
    uint32_t deadline;
    timer_cb callback;
    void* userp;
    struct timer* next;
    struct timer* prev;
};
void cancel_timer(timer* t) {
    t->next->prev = t->prev; t->prev->next = t->next; free(t);
}
timer* schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
    timer* iter = timeouts.prev;
    while (iter != &timeouts && is_after(iter->deadline, deadline))
        iter = iter->prev;
    add_behind(iter, deadline, cb, userp);
}
```

What Do You Mean by “Cache Friendly”?

Björn Fahller

Simplistic model of cache behaviour

Includes

Simplistic model of cache behaviour

Includes

- The cache is small

Simplistic model of cache behaviour

Includes

- The cache is small
- and consists of fixed size lines

Simplistic model of cache behaviour

Includes

- The cache is small
- and consists of fixed size lines
- and data access hit is very fast

Simplistic model of cache behaviour

Includes

- The cache is small
- and consists of fixed size lines
- and data access hit is very fast
- and data acess miss is very slow

Simplistic model of cache behaviour

Includes

- The cache is small
- and consists of fixed size lines
- and data access hit is very fast
- and data acess miss is very slow

Excludes

Simplistic model of cache behaviour

Includes

- The cache is small
- and consists of fixed size lines
- and data access hit is very fast
- and data acess miss is very slow

Excludes

- Multiple levels of caches

Simplistic model of cache behaviour

Includes

- The cache is small
- and consists of fixed size lines
- and data access hit is very fast
- and data acess miss is very slow

Excludes

- Multiple levels of caches
- Associativity

Simplistic model of cache behaviour

Includes

- The cache is small
- and consists of fixed size lines
- and data access hit is very fast
- and data acess miss is very slow

Excludes

- Multiple levels of caches
- Associativity
- Threading

Simplistic model of cache behaviour

Includes

- The cache is small
- and consists of fixed size lines
- and data access hit is very fast
- and data acess miss is very slow

Excludes

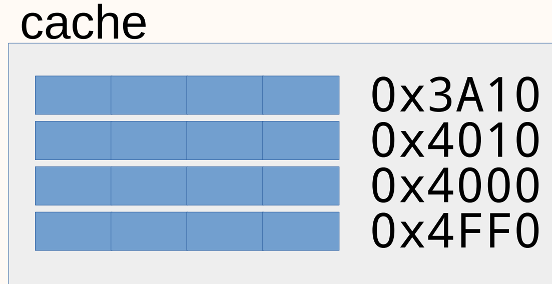
- Multiple levels of caches
- Associativity
- Threading

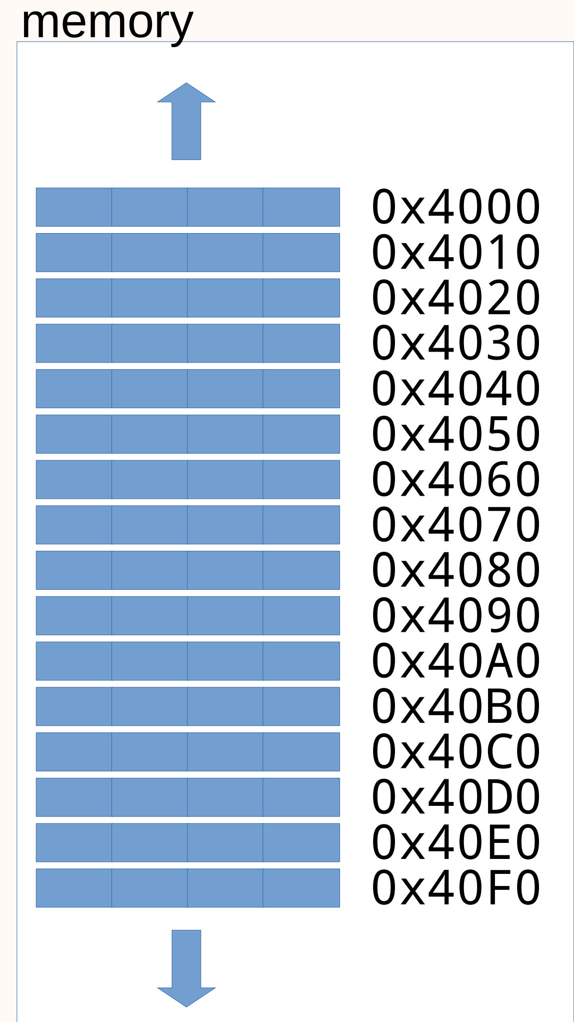
All models are wrong, but some are useful

Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

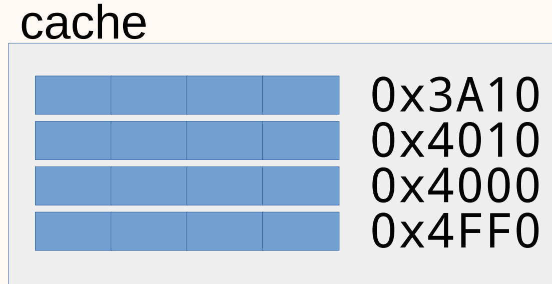


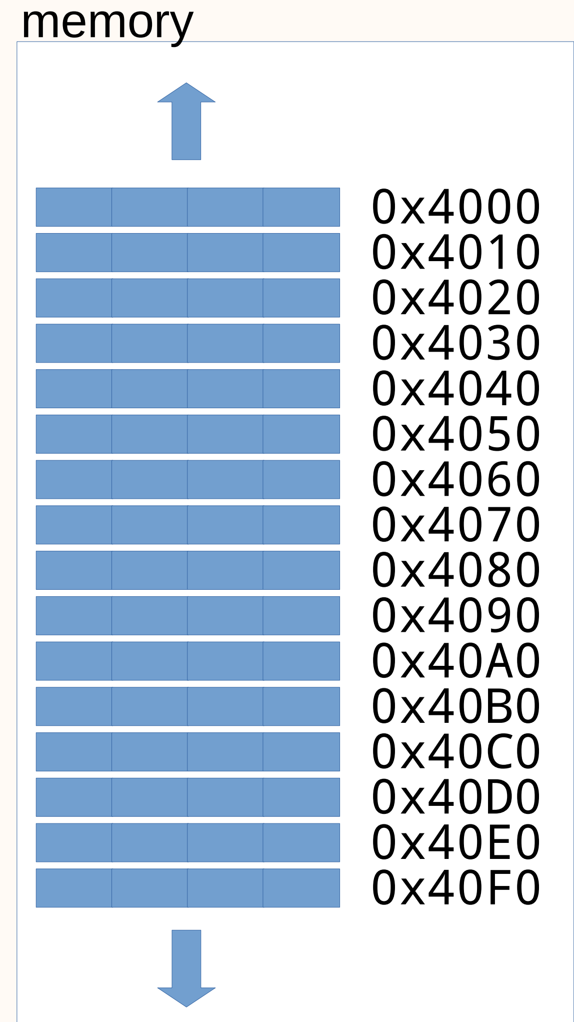


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

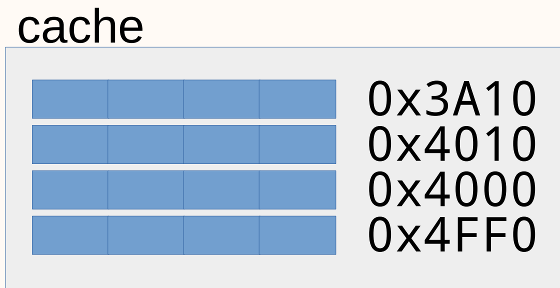


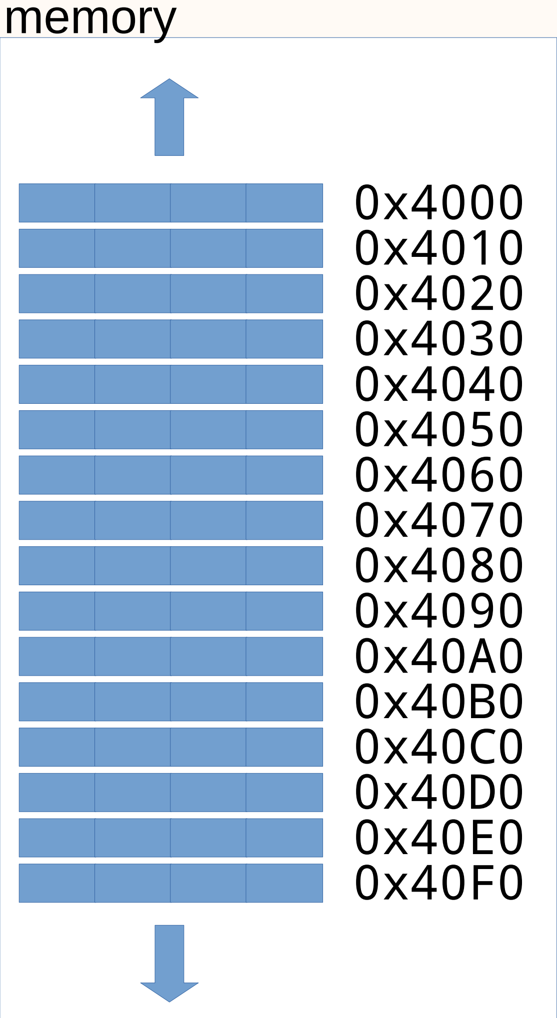


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

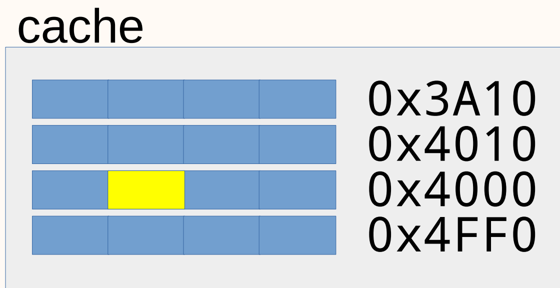


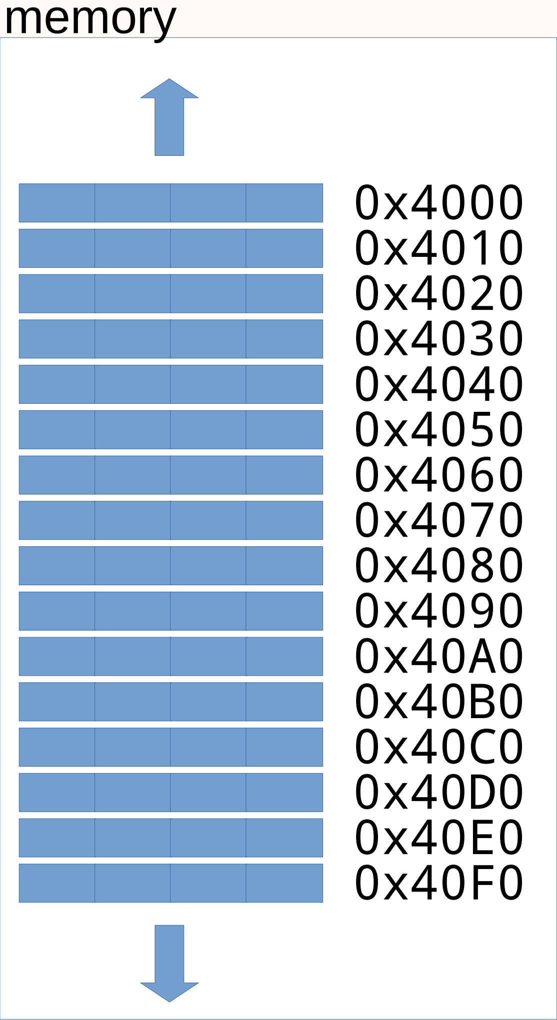


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

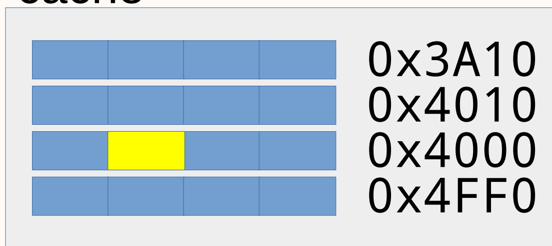


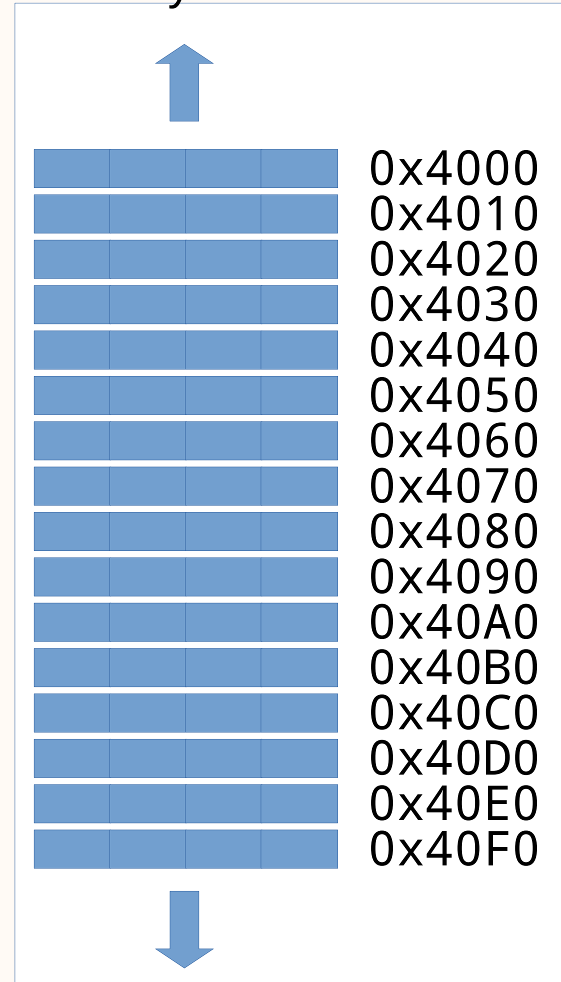


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

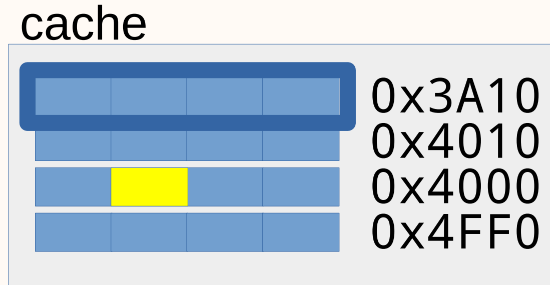


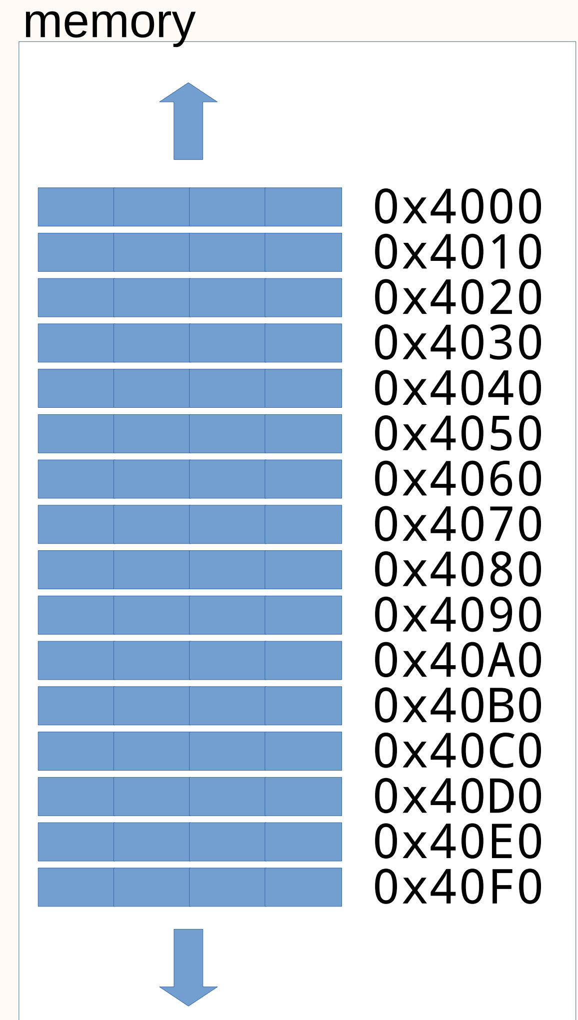


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

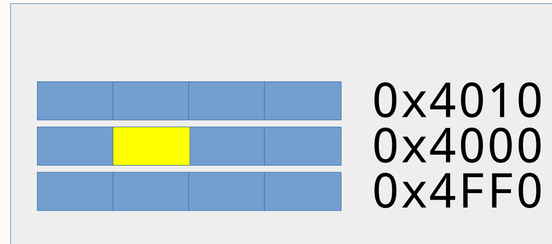


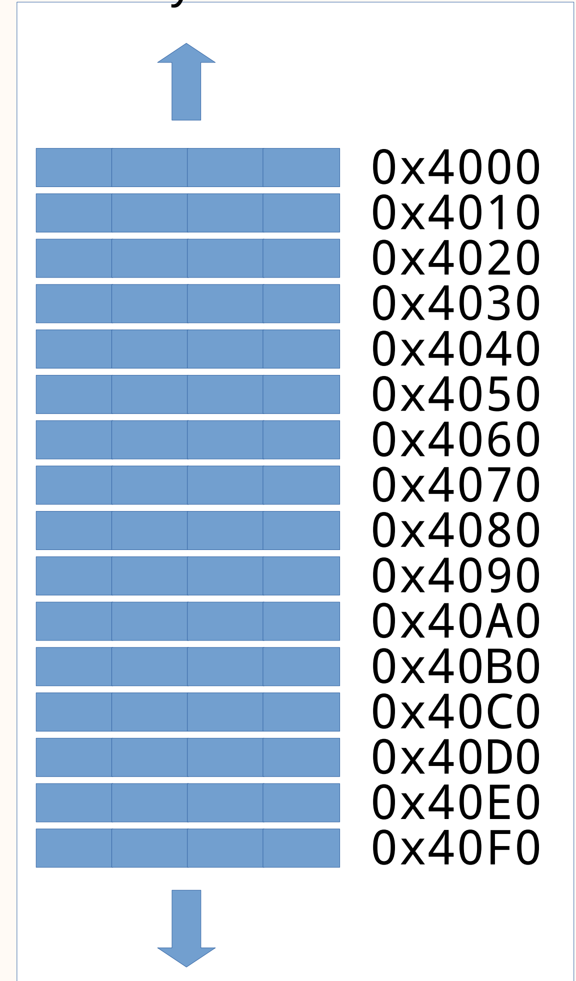


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

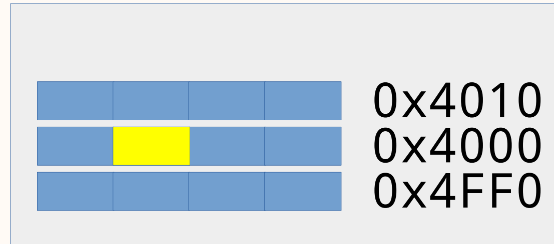


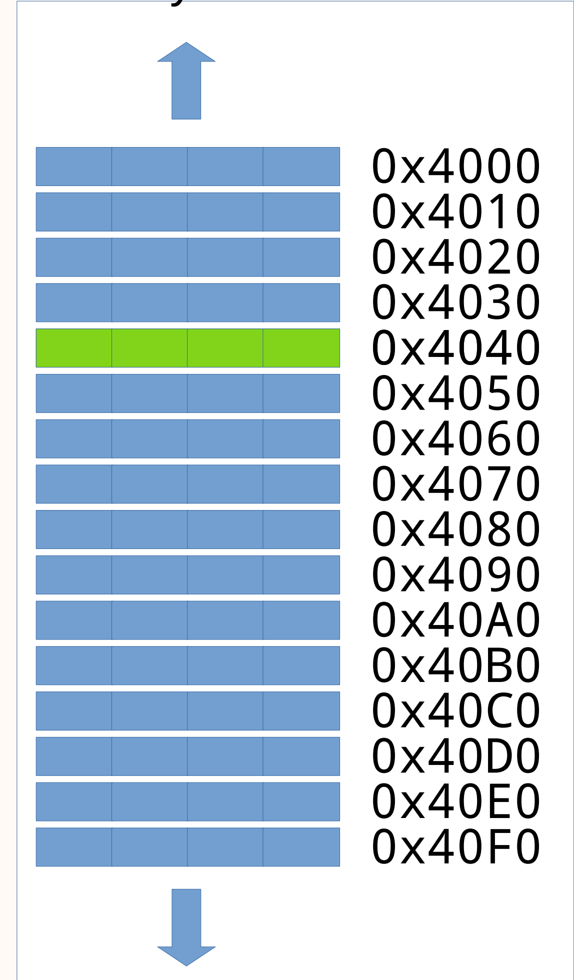


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

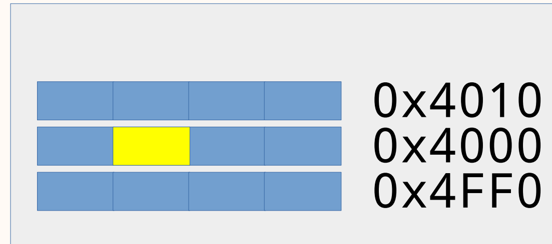


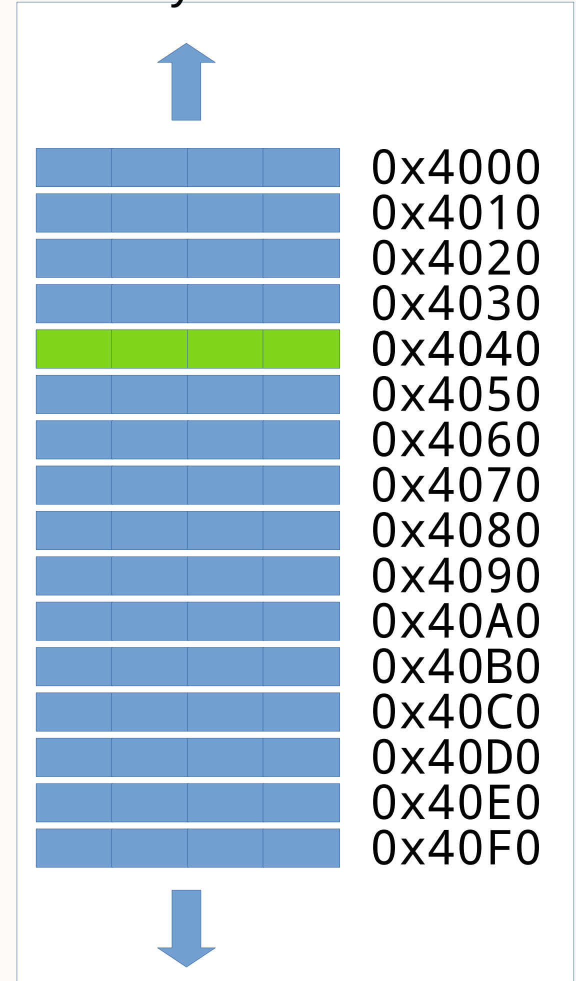


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

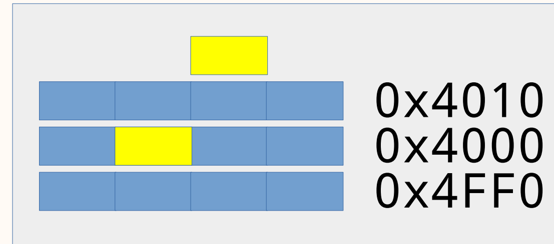


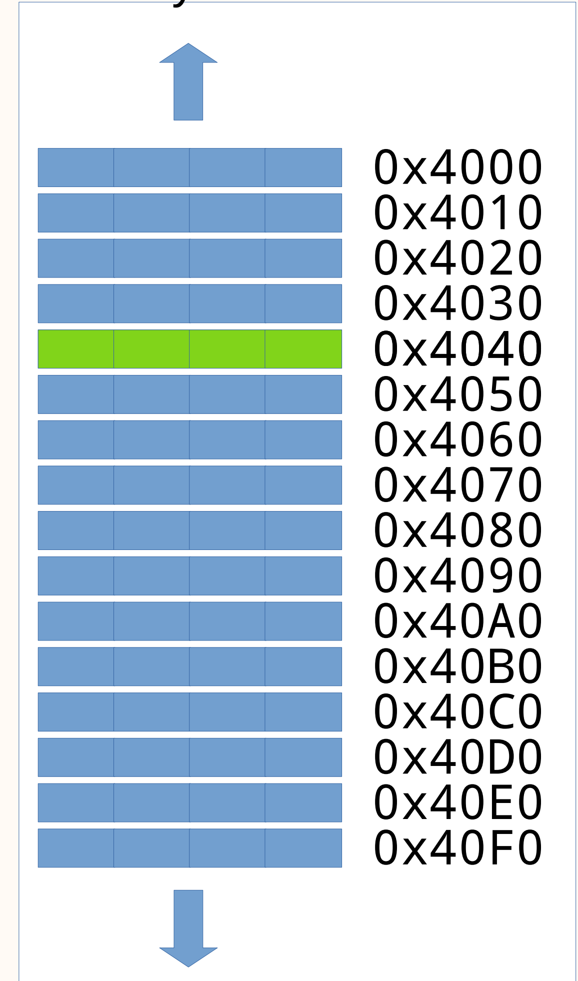


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

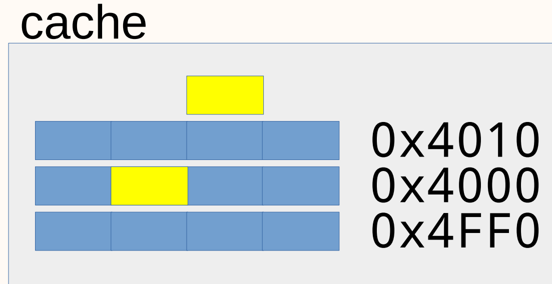


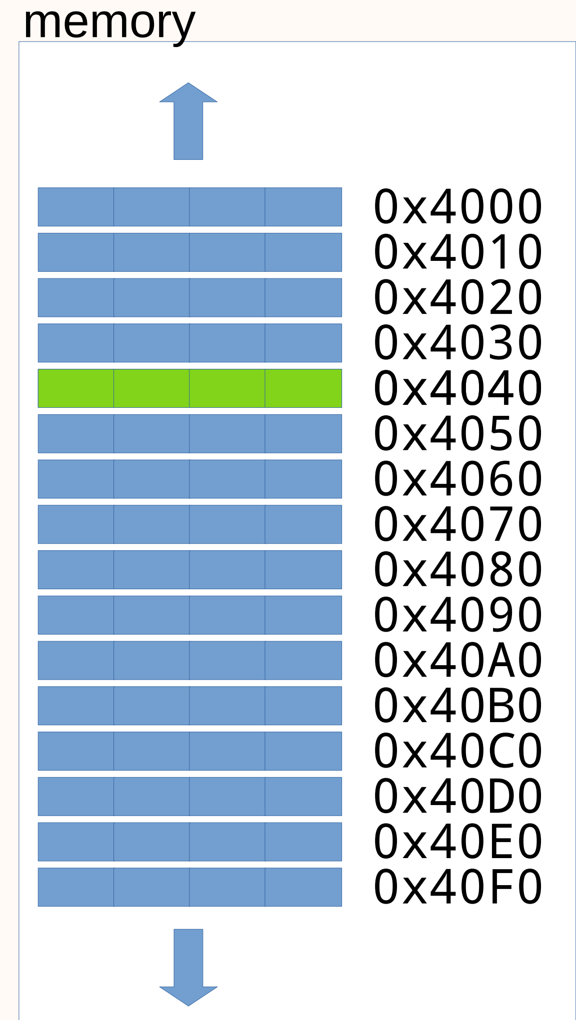


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

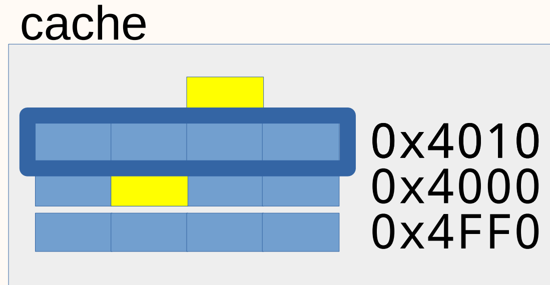


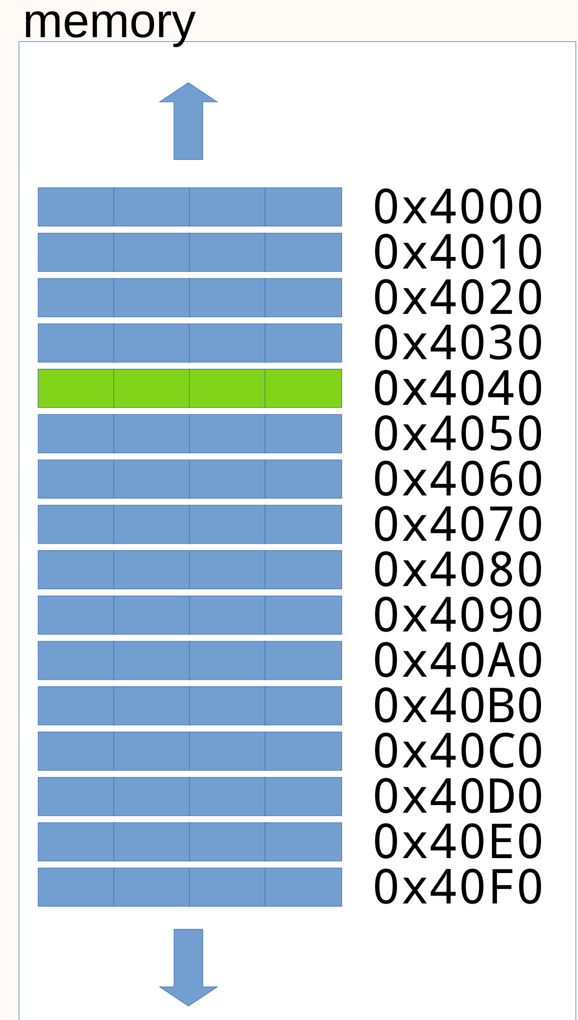


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

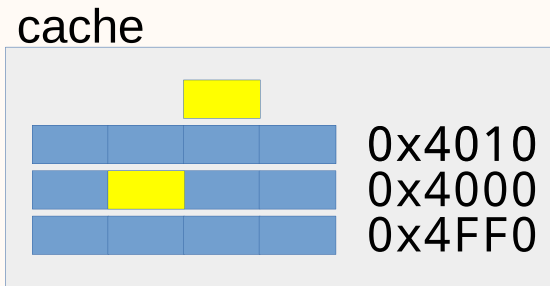


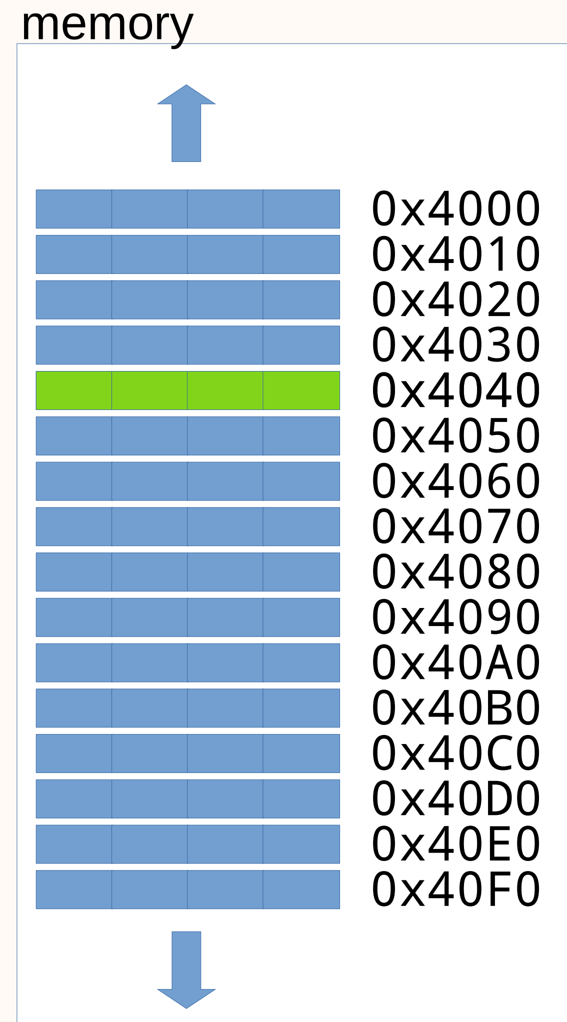


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

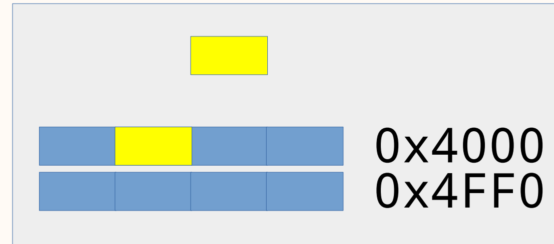


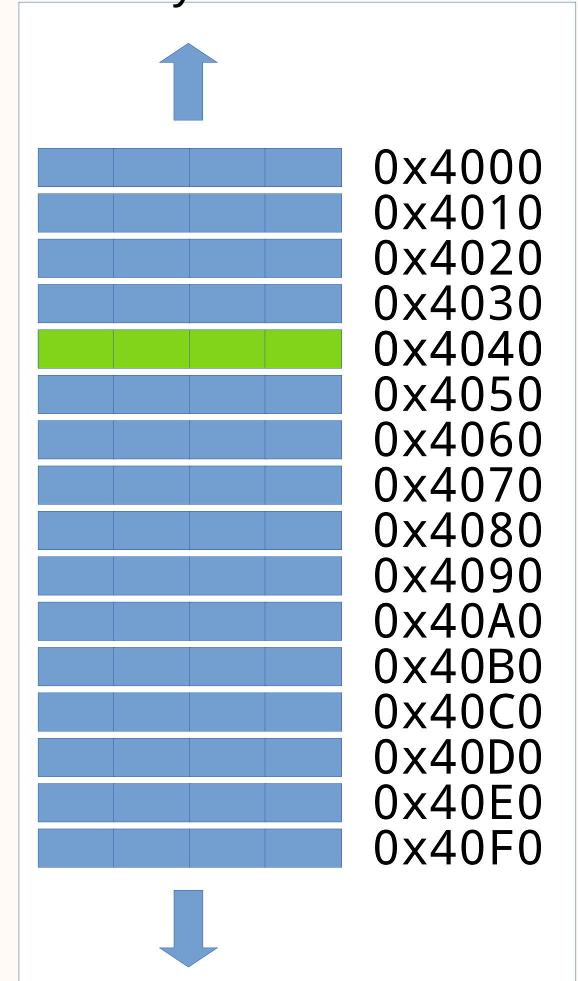


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

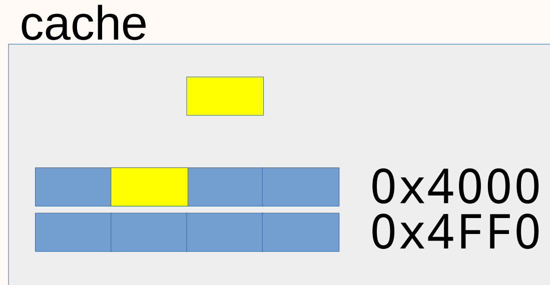


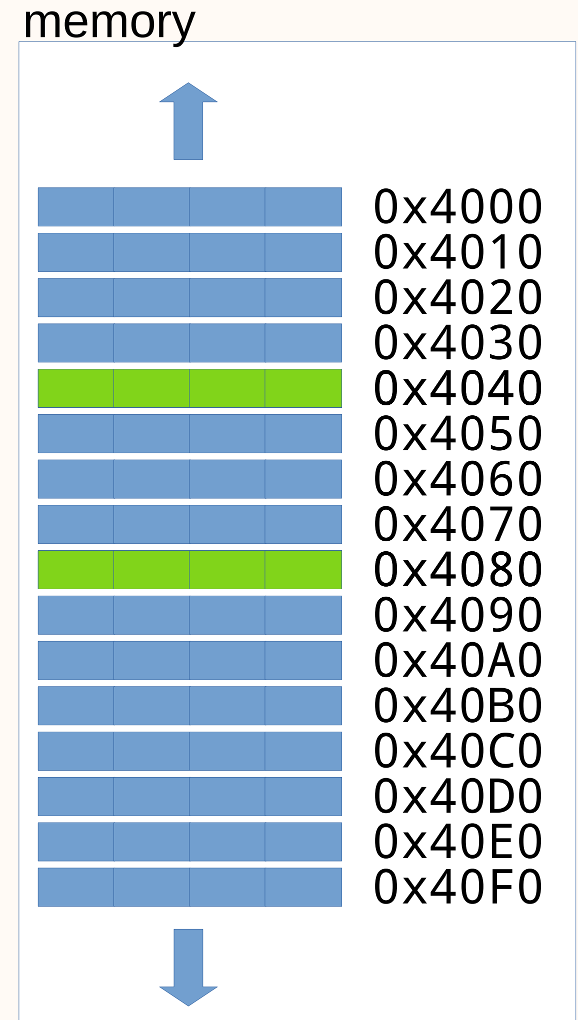


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

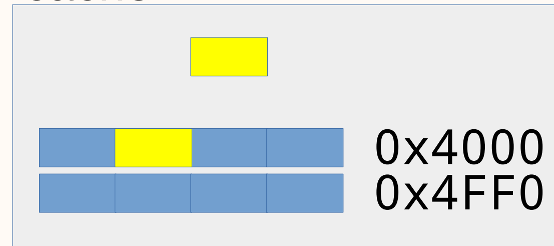


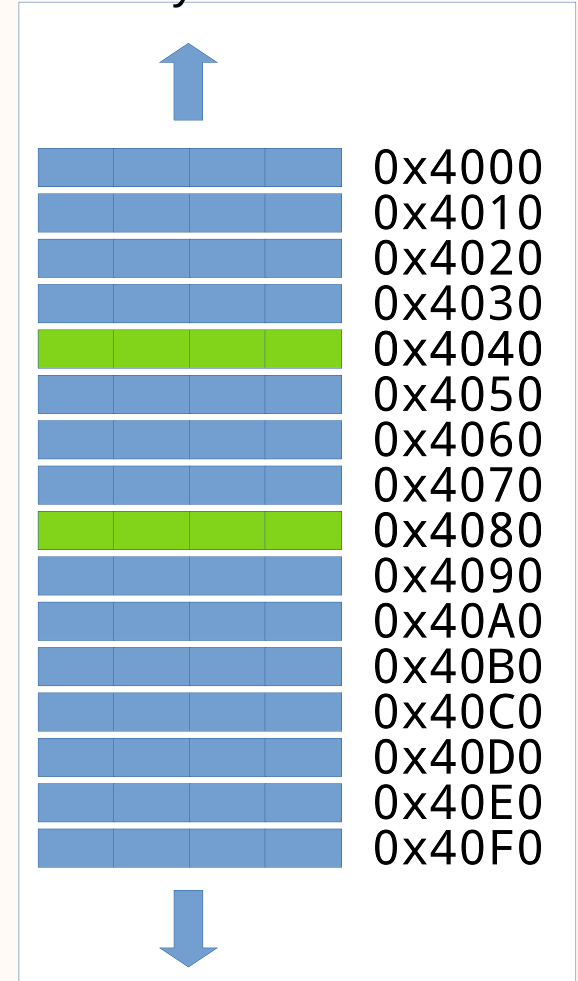


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

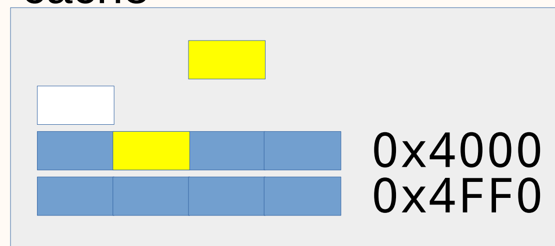


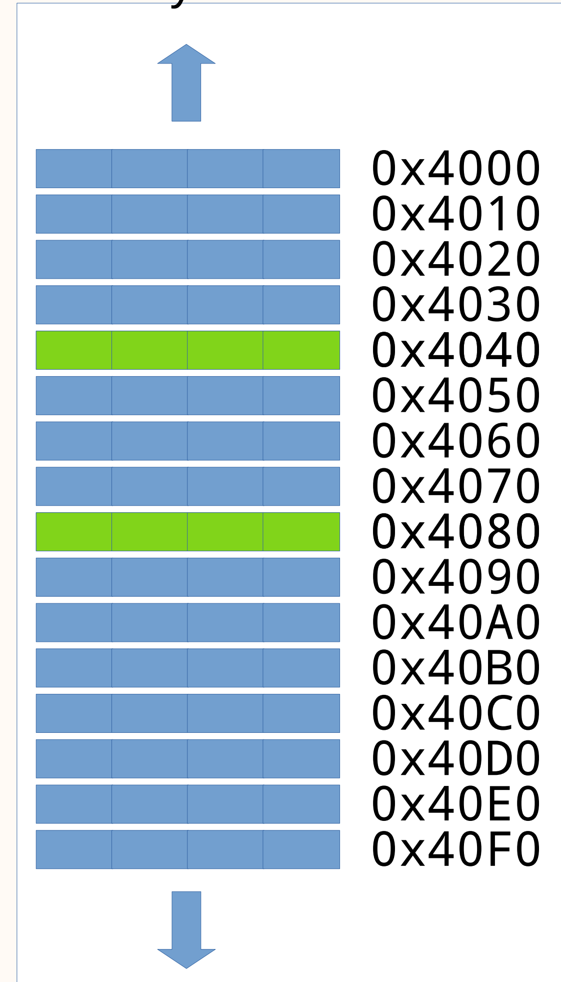


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

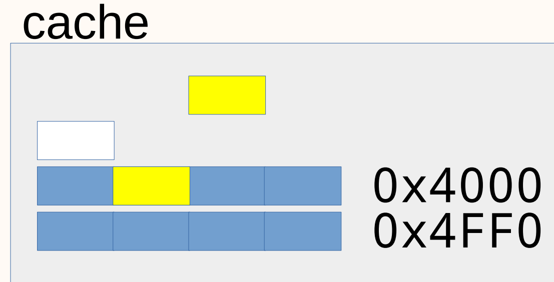


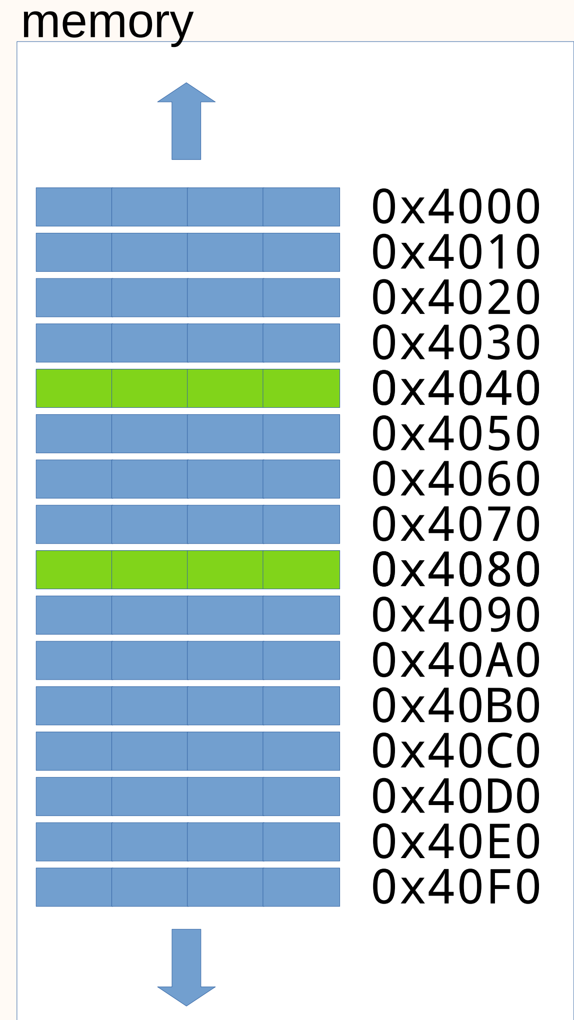


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

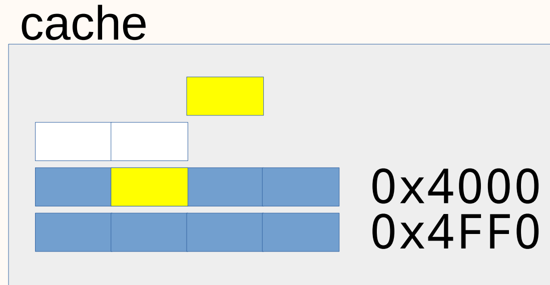


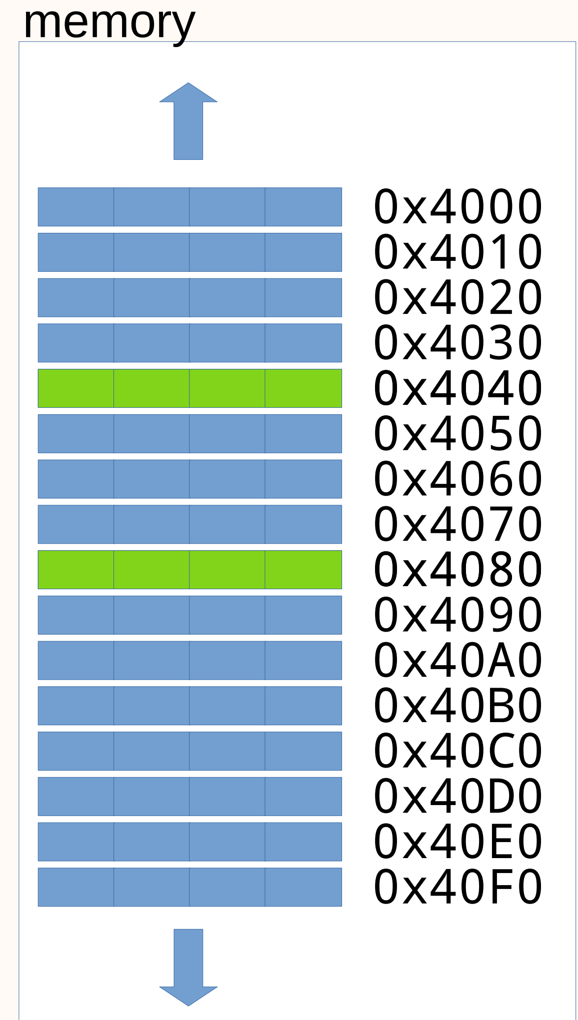


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

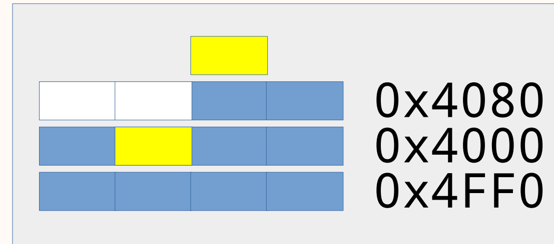


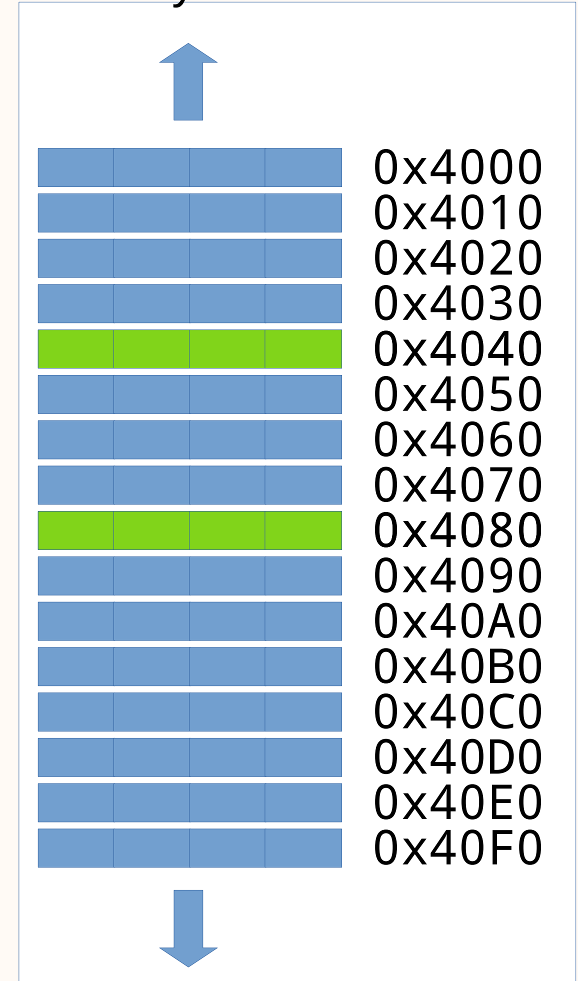


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```

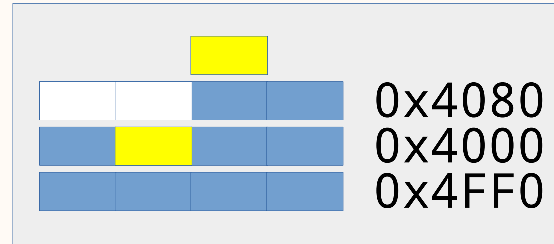


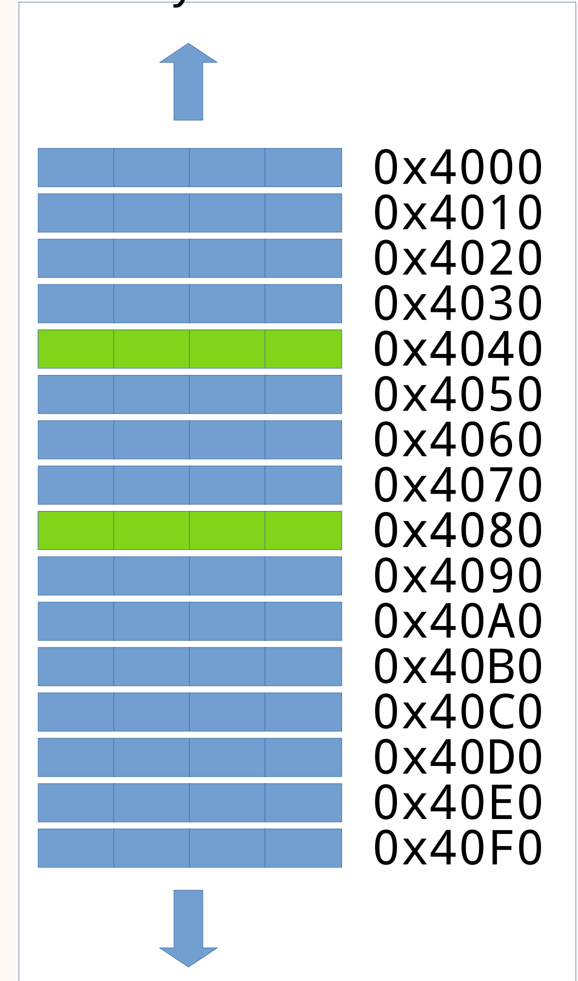


Simplistic model of cache behaviour

```
const int* hot = 0x4001;
const int* cold = 0x4042;
int* also_cold = 0x4080;


int a = *hot;
int c = *cold;
*also_cold = a;
also_cold[1] = c;
```





Analysis of implementation

```
int main() {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<uint32_t> dist;

  for (int k = 0; k < 10; ++k) {
    timer* prev = nullptr;
    for (int i = 0; i < 20'000; ++i) {
      timer* t = schedule_timer(
        dist(gen),
        [](void*){return 0U;}, nullptr);
      if (i & 1) cancel_timer(prev);
      prev = t;
    }
    while (shoot_first())
      ;
  }
}
```

Analysis of implementation

```
int main() {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<uint32_t> dist;

  for (int k = 0; k < 10; ++k) {
    timer* prev = nullptr;
    for (int i = 0; i < 20'000; ++i) {
      timer* t = schedule_timer(
        dist(gen),
        [](void*){return 0U;}, nullptr);
      if (i & 1) cancel_timer(prev);
      prev = t;
    }
    while (shoot_first())
      ;
  }
}
```

Analysis of implementation

```
int main() {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<uint32_t> dist;

  for (int k = 0; k < 10; ++k) {
    timer* prev = nullptr;
    for (int i = 0; i < 20'000; ++i) {
      timer* t = schedule_timer(
        dist(gen),
        [](void*){return 0U;}, nullptr);
      if (i & 1) cancel_timer(prev);
      prev = t;
    }
    while (shoot_first())
      ;
  }
}
```

Analysis of implementation

```
int main() {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<uint32_t> dist;

  for (int k = 0; k < 10; ++k) {
    timer* prev = nullptr;
    for (int i = 0; i < 20'000; ++i) {
      timer* t = schedule_timer(
        dist(gen),
        [](void*){return 0U;}, nullptr);
      if (i & 1) cancel_timer(prev);
      prev = t;
    }
    while (shoot_first())
      ;
  }
}
```

Analysis of implementation

```
int main() {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<uint32_t> dist;

  for (int k = 0; k < 10; ++k) {
    timer* prev = nullptr;
    for (int i = 0; i < 20'000; ++i) {
      timer* t = schedule_timer(
        dist(gen),
        [](void*){return 0U;}, nullptr);
      if (i & 1) cancel_timer(prev);
      prev = t;
    }
    while (shoot_first())
      ;
  }
}
```

```
bool shoot_first() {
  if (timeouts.next == &timeouts)
    return false;

  timer* t = timeouts.next;
  t->callback(t->userp);
  cancel_timer(t);
  return true;
}
```

Analysis of implementation

```
valgrind --tool=callgrind --cache-sim=yes --dump-instr=yes --branch-sim=yes
```

Analysis of implementation

```
valgrind --tool=callgrind --cache-sim=yes --dump-instr=yes --branch-sim=yes
```

Essentially a profiler that collects info about call hierarchies, number of calls, and time spent. The CPU simulator is not cycle accurate, so see timing results as a broad picture.

Analysis of implementation

```
valgrind --tool=callgrind --cache-sim=yes --dump-instr=yes --branch-sim=yes
```

Essentially a profiler that collects info about call hierarchies, number of calls, and time spent.
The CPU simulator is not cycle accurate, so see timing results as a broad picture.

Simulates a CPU cache, flattened to 2 levels, L1 and LL.
It shows you where you get cache misses.
L1 is by default a model of your host CPU L1, but you can change size, line-size, and associativity.

Analysis of implementation

```
valgrind --tool=callgrind --cache-sim=yes --dump-instr=yes --branch-sim=yes
```

Essentially a profiler that collects info about call hierarchies, number of calls, and time spent. The CPU simulator is not cycle accurate, so see timing results as a broad picture.

Simulates a CPU cache, flattened to 2 levels, L1 and LL. It shows you where you get cache misses. L1 is by default a model of your host CPU L1, but you can change size, line-size, and associativity.

Collects statistics per instruction instead of per source line. Can help pinpointing bottlenecks.

Analysis of implementation

```
valgrind --tool=callgrind --cache-sim=yes --dump-instr=yes --branch-sim=yes
```

- `--tool=callgrind`: Essentially a profiler that collects info about call hierarchies, number of calls, and time spent. The CPU simulator is not cycle accurate, so see timing results as a broad picture.
- `--cache-sim=yes`: Simulates a CPU cache, flattened to 2 levels, L1 and LL. It shows you where you get cache misses. L1 is by default a model of your host CPU L1, but you can change size, line-size, and associativity.
- `--dump-instr=yes`: Collects statistics per instruction instead of per source line. Can help pinpointing bottlenecks.
- `--branch-sim=yes`: Simulates a branch predictor.

Analysis of implementation

Very slow!

```
valgrind --tool=callgrind --cache-sim=yes --dump-instr=yes --branch-sim=yes
```

`--tool=callgrind`: Essentially a profiler that collects info about call hierarchies, number of calls, and time spent. The CPU simulator is not cycle accurate, so see timing results as a broad picture.

`--cache-sim=yes`: Simulates a CPU cache, flattened to 2 levels, L1 and LL. It shows you where you get cache misses. L1 is by default a model of your host CPU L1, but you can change size, line-size, and associativity.

`--dump-instr=yes`: Collects statistics per instruction instead of per source line. Can help pinpointing bottlenecks.

`--branch-sim=yes`: Simulates a branch predictor.

Live demo

```c
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;

    timer_cb callback;
    void* userp;
    struct timer* next;
    struct timer* prev;
} timer;
```

```c
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;          // 4 bytes

    timer_cb callback;
    void* userp;
    struct timer* next;
    struct timer* prev;
} timer;
```

```c
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;             // 4 bytes
                                   // 4 bytes padding for alignment
    timer_cb callback;
    void* userp;
    struct timer* next;
    struct timer* prev;
} timer;
```

```c
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;              // 4 bytes
                                    // 4 bytes padding for alignment
    timer_cb callback;              // 8 bytes
    void* userp;
    struct timer* next;
    struct timer* prev;
} timer;
```

```c
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;              // 4 bytes
                                    // 4 bytes padding for alignment
    timer_cb callback;              // 8 bytes
    void* userp;                    // 8 bytes
    struct timer* next;
    struct timer* prev;
} timer;
```

```c
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;                  // 4 bytes
                                        // 4 bytes padding for alignment
    timer_cb callback;                  // 8 bytes
    void* userp;                        // 8 bytes
    struct timer* next;                 // 8 bytes
    struct timer* prev;
} timer;
```

```c
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;              // 4 bytes
                                    // 4 bytes padding for alignment
    timer_cb callback;              // 8 bytes
    void* userp;                    // 8 bytes
    struct timer* next;             // 8 bytes
    struct timer* prev;             // 8 bytes
} timer;
```

```c
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;               // 4 bytes
                                     // 4 bytes padding for alignment
    timer_cb callback;               // 8 bytes
    void* userp;                     // 8 bytes
    struct timer* next;              // 8 bytes
    struct timer* prev;              // 8 bytes
} timer;                             // sum = 40 bytes
```

```
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;              // 4 bytes
                                    // 4 bytes padding for alignment
    timer_cb callback;              // 8 bytes
    void* userp;                    // 8 bytes
    struct timer* next;             // 8 bytes
    struct timer* prev;             // 8 bytes
} timer;                            // sum = 40 bytes
```

66% of all L1d cache misses

```c
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;                // 4 bytes
                                      // 4 bytes padding for alignment
    timer_cb callback;                // 8 bytes
    void* userp;                      // 8 bytes
    struct timer* next;               // 8 bytes
    struct timer* prev;               // 8 bytes
} timer;                              // sum = 40 bytes
```

66% of all L1d cache misses

Rule of thumb:
Follow pointer => cache miss

```
typedef uint32_t (*timer_cb)(void*);
typedef struct timer {
    uint32_t deadline;                   // 4 bytes
                                         // 4 bytes padding for alignment
    timer_cb callback;                   // 8 bytes
    void* userp;                         // 8 bytes
    struct timer* next;                  // 8 bytes
    struct timer* prev;                  // 8 bytes
} timer;                                 // sum = 40 bytes
```

66% of all L1d cache misses

33% of all L1d cache misses

Rule of thumb:
Follow pointer => cache miss

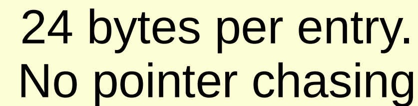
Chasing pointers is expensive.
Let’s get rid of the pointers.

```
typedef uint32_t (*timer_cb)(void*);
typedef uint32_t timer;
struct timer_data {
  uint32_t deadline;
  timer id;
  void* userp;
  timer_cb callback;
};

std::vector<timer_data> timeouts;
uint32_t next_id = 0;
```

```
typedef uint32_t (*timer_cb)(void*);
typedef uint32_t timer;
struct timer_data {
  uint32_t deadline;
  timer id;
  void* userp;
  timer_cb callback;
};

std::vector<timer_data> timeouts;
uint32_t next_id = 0;
```



```
typedef uint32_t (*timer_cb)(void*);
typedef uint32_t timer;
struct timer_data {
  uint32_t deadline;
  timer id;
  void* userp;
  timer_cb callback;
};

std::vector<timer_data> timeouts;
uint32_t next_id = 0;
```

24 bytes per entry.
No pointer chasing

Linear structure

```
typedef uint32_t (*timer_cb)(void*);
typedef uint32_t timer;
struct timer_data {
  uint32_t deadline;
  timer id;
  void* userp;
  timer_cb callback;
};
timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
  auto idx = timeouts.size();
  timeouts.push_back({});
  while (idx > 0 && is_after(timeouts[idx-1].deadline, deadline)) {
    timeouts[idx] = std::move(timeouts[idx-1]);
    --idx;
  }
  timeouts[idx] = timer_data{deadline, next_id++, userp, cb };
  return next_id;
}
```

```
typedef uint32_t (*timer_cb)(void*);
typedef uint32_t timer;
struct timer_data {
  uint32_t deadline;
  timer id;
  void* userp;
  timer_cb callback;
};
timer schedule_timer(uint32_t deadline, timer_                    )
{
  auto idx = timeouts.size();
  timeouts.push_back({});
  while (idx > 0 && is_after(timeouts[idx-1].deadline, deadline)) {
    timeouts[idx] = std::move(timeouts[idx-1]);
    --idx;
  }
  timeouts[idx] = timer_data{deadline, next_id++, userp, cb };
  return next_id;
}
```

Linear insertion sort

```
typedef uint32_t (*timer_cb)(void*);
typedef uint32_t timer;
struct timer_data {
  uint32_t deadline;
  timer id;
  void* userp;
  timer_cb callback;
};
timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
  auto idx = timeouts.size();
  timeouts.push_back({});
  while (idx > 0 && is_after(timeouts[idx-1].deadline, deadline)) {
    timeouts[idx] = std::move(timeouts[idx-1]);
    --idx;
  }
  timeouts[idx] = timer_data{deadline, next_id++, userp, cb };
  return next_id;
}
```

```
void cancel_timer(timer t)
{
  auto i = std::find_if(timeouts.begin(), timeouts.end(),
                        [t](const auto& e) { return e.id == t; });

  timeouts.erase(i);
}

timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
  auto idx = timeouts.size();
  timeouts.push_back({});
  while (idx > 0 && is_after(timeouts[idx-1].deadline, deadline)) {
    timeouts[idx] = std::move(timeouts[idx-1]);
    --idx;
  }
  timeouts[idx] = timer_data{deadline, next_id++, userp, cb };
  return next_id;
}
```

```
void cancel_timer(timer t)
{
  auto i = std::find_if(timeouts.begin(), timeouts.end(),
                        [t](const auto& e) { return e.id == t; });

  timeouts.erase(i);
}

timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
  auto idx = timeouts.size();
  timeouts.push_back({});
  while (idx > 0 && is_after(timeouts[idx-1].deadline, deadline)) {
    timeouts[idx] = std::move(timeouts[idx-1]);
    --idx;
  }
  timeouts[idx] = timer_data{deadline, next_id++, userp, cb };
  return next_id;
}
```

Linear search

Analysis of implementation

```
perf stat -e cycles,instructions,l1d-loads,l1d-load-misses
```

Analysis of implementation

```
perf stat -e cycles,instructions,l1d-loads,l1d-load-misses
```

Presents statistics from whole run of program, using counters from HW and linux kernel.

Analysis of implementation

```
perf stat -e cycles,instructions,l1d-loads,l1d-load-misses
```

Presents statistics from whole run of program, using counters from HW and linux kernel.

Number of cycles per instruction is a proxy for how much the CPU is working or waiting.

Analysis of implementation

```
perf stat -e cycles,instructions,l1d-loads,l1d-load-misses
```

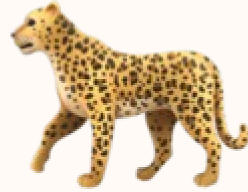
Presents statistics from whole run of program, using counters from HW and linux kernel.

Number of cycles per instruction is a proxy for how much the CPU is working or waiting.

Number of reads from L1d cache, and number of misses. Speculative execution can make these numbers confusing.

Analysis of implementation

Very fast!



```
perf stat -e cycles,instructions,l1d-loads,l1d-load-misses
```

Presents statistics from whole run of program, using counters from HW and linux kernel.

Number of cycles per instruction is a proxy for how much the CPU is working or waiting.

Number of reads from L1d cache, and number of misses. Speculative execution can make these numbers confusing.

Analysis of implementation

```
perf record -e cycles,instructions,l1d-loads,l1d-load-misses --call-graph=lbr
```

Analysis of implementation

```
perf record -e cycles,instructions,l1d-loads,l1d-load-misses --call-graph=lbr
```

Records where in your program the counters are gathered.

Analysis of implementation

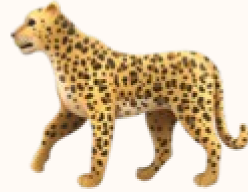
```
perf record -e cycles,instructions,l1d-loads,l1d-load-misses --call-graph=lbr
```

Records where in your program the counters are gathered.

Records call graph info, instead of just location. LBR requires no special compilation flags.

Very fast!

Analysis of implementation



```
perf record -e cycles,instructions,l1d-loads,l1d-load-misses --call-graph=lbr
```

Records where in your program the counters are gathered.

Records call graph info, instead of just location. LBR requires no special compilation flags.

Live demo

bjorn_fahller Björn Fahller Oct 6

What surprised you the most when you learned about CPU caches?

Miro Knejp
@mknejp

Replying to @bjorn_fahller

That doing more work can be faster than doing less.

12:42pm · 6 Oct 2019 · TweetDeck

Linear search is expensive.
Maybe try binary search?

```
typedef uint32_t (*timer_cb)(void*);
struct timer_data {
  uint32_t deadline;
  uint32_t id;
  void* userp;
  timer_cb callback;
};
struct timer {
  uint32_t deadline;
  uint32_t id;
};

std::vector<timer_data> timeouts;
uint32_t next_id = 0;
```

```
typedef uint32_t (*timer_cb)(void*);
struct timer_data {
  uint32_t deadline;
  uint32_t id;
  void* userp;
  timer_cb callback;
};
struct timer {
  uint32_t deadline;
  uint32_t id;
};
timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
 timer_data element{deadline, next_id, userp, cb};
 auto i = std::lower_bound(timeouts.begin(), timeouts.end(),
                           element, is_after);
 timeouts.insert(i, element);
 return {deadline, next_id++};
}
```

```
typedef uint32_t (*timer_cb)(void*);
struct timer_data {
  uint32_t deadline;
  uint32_t id;
  void* userp;
  timer_cb callback;
};
struct timer {
  uint32_t deadline;
  uint32_t id;
};
timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
 timer_data element{deadline, next_id, userp, cb};
 auto i = std::lower_bound(timeouts.begin(), timeouts.end(),
                           element, is_after);
 timeouts.insert(i, element);
 return {deadline, next_id++};
}
```

Binary search for insertion point

```
typedef uint32_t (*timer_cb)(void*);
struct timer_data {
  uint32_t deadline;
  uint32_t id;
  void* userp;
  timer_cb callback;
};
struct timer {
  uint32_t deadline;
  uint32_t id;
};
timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
 timer_data element{deadline, next_id, us
 auto i = std::lower_bound(timeouts.beg                        (),
                           element, is_after);
 timeouts.insert(i, element);
 return {deadline, next_id++};
}
```

Linear insertion

```cpp
void cancel_timer(timer t) {
 timer_data element{t.deadline, t.id, nullptr, nullptr};
 auto [lo, hi] = std::equal_range(timeouts.begin(), timeouts.end(),
                                  element, is_after);
 auto i = std::find_if(lo, hi,
                       [t](const auto& e) { return e.id == t.id; });

 if (i != hi) {
   timeouts.erase(i);
 }
}
timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
 timer_data element{deadline, next_id, us
 auto i = std::lower_bound(timeouts.beg                      (),
                           element, is_after);
 timeouts.insert(i, element);
 return {deadline, next_id++};
}
```

Linear insertion

```
void cancel_timer(timer t) {
 timer_data element{t.deadline, t.id, nullptr, nullptr};
 auto [lo, hi] = std::equal_range(timeouts.begin(), timeouts.end(),
                                  element, is_after);
 auto i = std::find_if(lo, hi,
                       [t](const auto& e) { return e.id == t.id; });

 if (i != hi) {
   timeouts.erase(i);
 }
}
timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
 timer_data element{deadline, next_id, us
 auto i = std::lower_bound(timeouts.begi                 (),
                           element, is_after);
 timeouts.insert(i, element);
 return {deadline, next_id++};
}
```

Binary search for timers with the same deadline

Linear insertion

```
void cancel_timer(timer t) {
 timer_data element{t.deadline, t.id, nullptr, nullptr};
 auto [lo, hi] = std::equal_range(timeouts.begin(), timeouts.end(),
                                  element, is_after);
 auto i = std::find_if(lo, hi,
                       [t](const auto& e) { return e.id == t.id; });

 if (i != hi) {
   timeouts.erase(i);
 }
}
timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
 timer_data element{deadline, next_id, us
 auto i = std::lower_bound(timeouts.beg                    (),
                           element, is_after);
 timeouts.insert(i, element);
 return {deadline, next_id++};
}
```

Linear search for matching id

Linear insertion

```
void cancel_timer(timer t) {
 timer_data element{t.deadline, t.id, nullptr, nullptr};
 auto [lo, hi] = std::equal_range(timeouts.begin(), timeouts.end(),
                                  element, is_after);
 auto i = std::find_if(lo, hi,
                       [t](const auto& e) { return e.id == t.id; });

 if (i != hi) {
   timeouts.erase(i);
 }
}
timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp)
{
 timer_data element{deadline, next_id, us
 auto i = std::lower_bound(timeouts.beg                (),
                           element, is_after);
 timeouts.insert(i, element);
 return {deadline, next_id++};
}
```

Linear removal

Linear insertion

Live demo

Searches not visible in profiling.
Number of reads reduced.
Number of cache misses high.
`memmove()` dominates.

Searches not visible in profiling.
Number of reads reduced.
Number of cache misses high.
`memmove()` dominates.

Failed branch predictions
can lead to cache entry eviction!

Searches not visible in profiling.
Number of reads reduced.
Number of cache misses high.
`memmove()` dominates.

Maybe try a `map<>`?

Failed branch predictions
can lead to cache entry eviction!

```
typedef uint32_t (*timer_cb)(void*);
struct timer_data {
  void* userp;
  timer_cb callback;
};

struct is_after {
  bool operator()(uint32_t lh, uint32_t rh) const {
    return lh < rh;
  }
};

using timer_map = std::multimap<uint32_t, timer_data, is_after>;
using timer = timer_map::iterator;

static timer_map timeouts;
```

```
typedef uint32_t (*timer_cb)(void*);
struct timer_data {
  void* userp;
  timer_cb callback;
};

struct is_after {
  bool operator()(uint32_t lh, uint32_t rh) const {
    return lh < rh;
  }
};

using timer_map = std::multimap<uint32_t, timer_data, is_after>;
using timer = timer_map::iterator;

static timer_map timeouts;
```

```
typedef uint32_t (*timer_cb)(void*);
struct timer_data {
  void* userp;
  timer_cb callback;
};



timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp) {
  return timeouts.insert(std::make_pair(deadline,
                                        timer_data{userp, cb}));
}

void cancel_timer(timer t) {
  timeouts.erase(t);
}
```

```
typedef uint32_t (*timer_cb)(void*);
struct timer_data {
  void* userp;
  timer_cb callback;
};



timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp) {
  return timeouts.insert(std::make_pair(deadline,
                                        timer_data{userp, cb}));
}

void cancel_timer(timer t) {
  timeouts.erase(t);
}
```

```
typedef uint32_t (*timer_cb)(void*);
struct timer_data {
  void* userp;
  timer_cb callback;
};



timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp) {
  return timeouts.insert(std::make_pair(deadline,
                                        timer_data{userp, cb}));
}

void cancel_timer(timer t) {
  timeouts.erase(t);
}
```

```
typedef uint32_t (*timer_cb)(void*);
struct timer_data {
  void* userp;
  timer_cb callback;
};



timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp) {
  return timeouts.insert(std::make_pair(deadline,
                                        timer_data{userp, cb}));
}

void cancel_timer(timer t) {
  timeouts.erase(t);
}
```

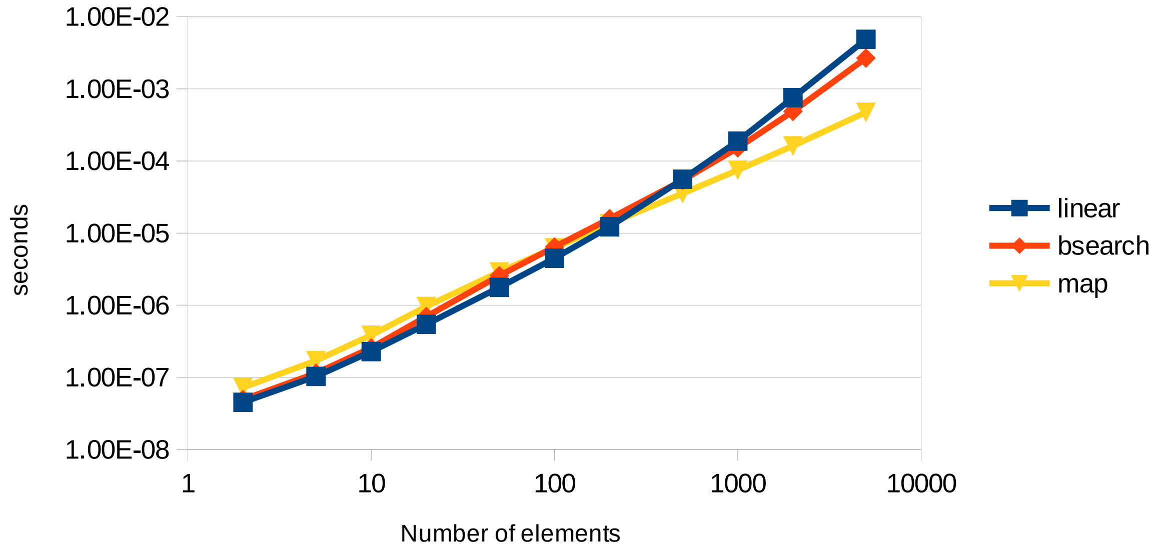
```
bool shoot_first() {
  if (timeouts.empty()) return false;
  auto i = timeouts.begin();
  i->second.callback(i->second.userp);
  timeouts.erase(i);
  return true;
}
```

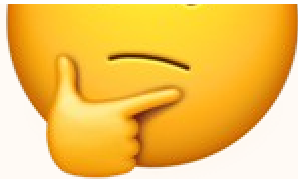
Live demo

Faster, but lots of cache misses when comparing keys and rebalancing the tree.

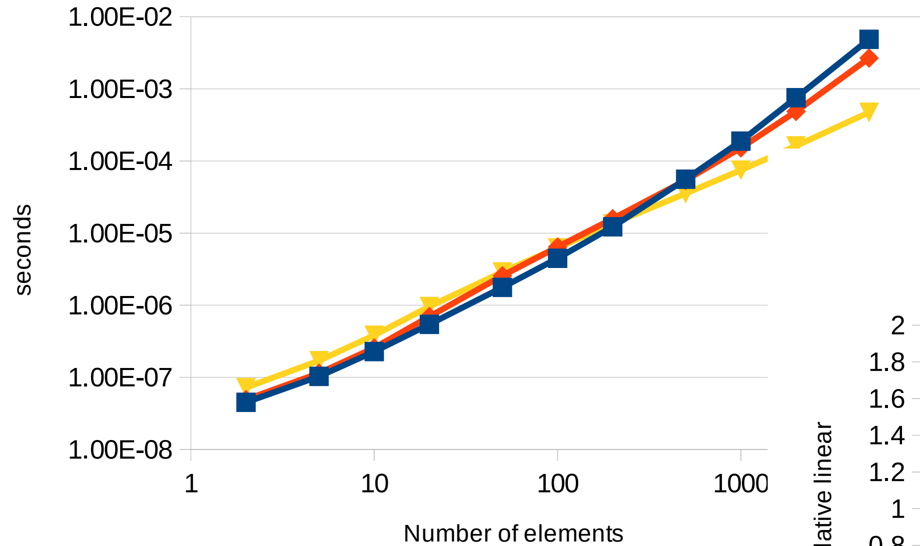
Faster, but lots of cache misses when comparing keys and rebalancing the tree.

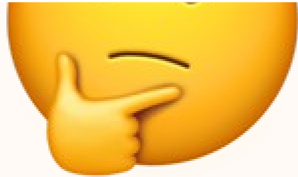
What did I say about chasing pointers?

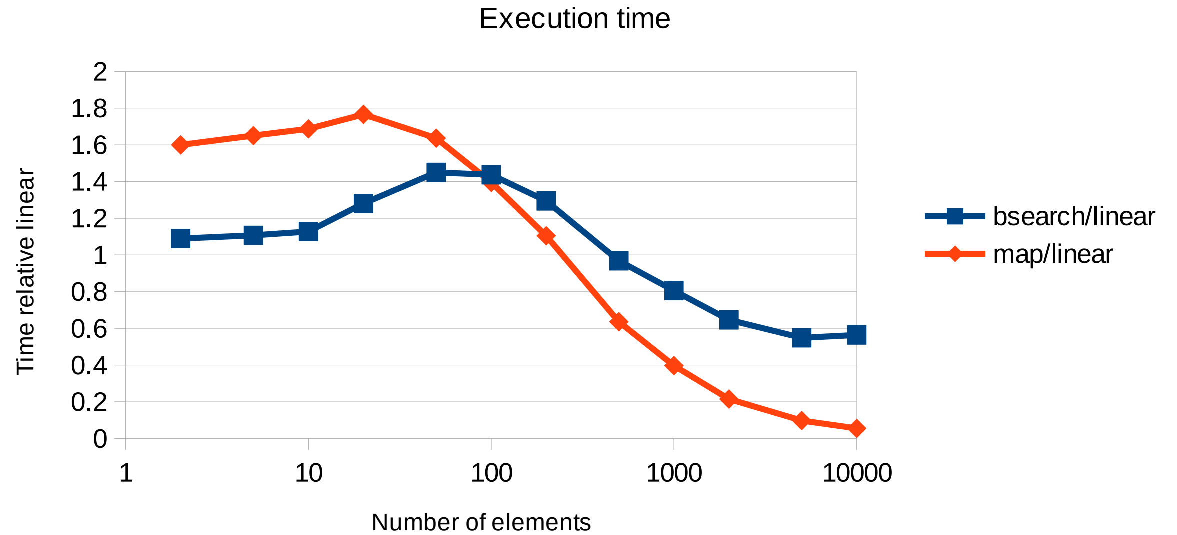




What did I say about chasing pointers?


Execution time
1.00E-02
1.00E-03
1.00E-04
1.00E-05
1.00E-06
1.00E-07
1.00E-08
seconds
1
10
100
1000
Number of elements




Performance relative to linear
Execution time
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Time relative linear
1
10
100
1000
10000
Number of elements
bsearch/linear
map/linear

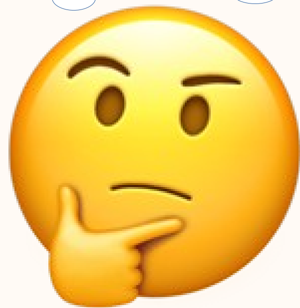
Faster, but lots of cache misses when comparing keys and rebalancing the tree.

What did I say about chasing pointers?

Faster, but lots of cache misses when comparing keys and rebalancing the tree.

Can we get log(n) lookup without chasing pointers?

What did I say about chasing pointers?



Enter the **HEAP**

Enter the **HEAP**

3

5 8

6 10 10 14

9 12 13 11 15

Enter the **HEAP**

- Perfectly balanced partially sorted tree

3

5 8

6 10 10 14

9 12 13 11 15

Enter the **HEAP**

- Perfectly balanced partially sorted tree
- Every node is sorted after or same as its parent

3

5 8

6 10 10 14

9 12 13 11 15

Enter the **HEAP**

- Perfectly balanced partially sorted tree
- Every node is sorted after or same as its parent
- No relation between siblings

3

5 8

6 10 10 14

9 12 13 11 15

Enter the **HEAP**

- Perfectly balanced partially sorted tree
- Every node is sorted after or same as its parent
- No relation between siblings
- At most one node with only one child, and that child is the last node

3

5 8

6 10 10 14

9 12 13 11 15

3

5 8

6 10 14

9 12 13 11 15

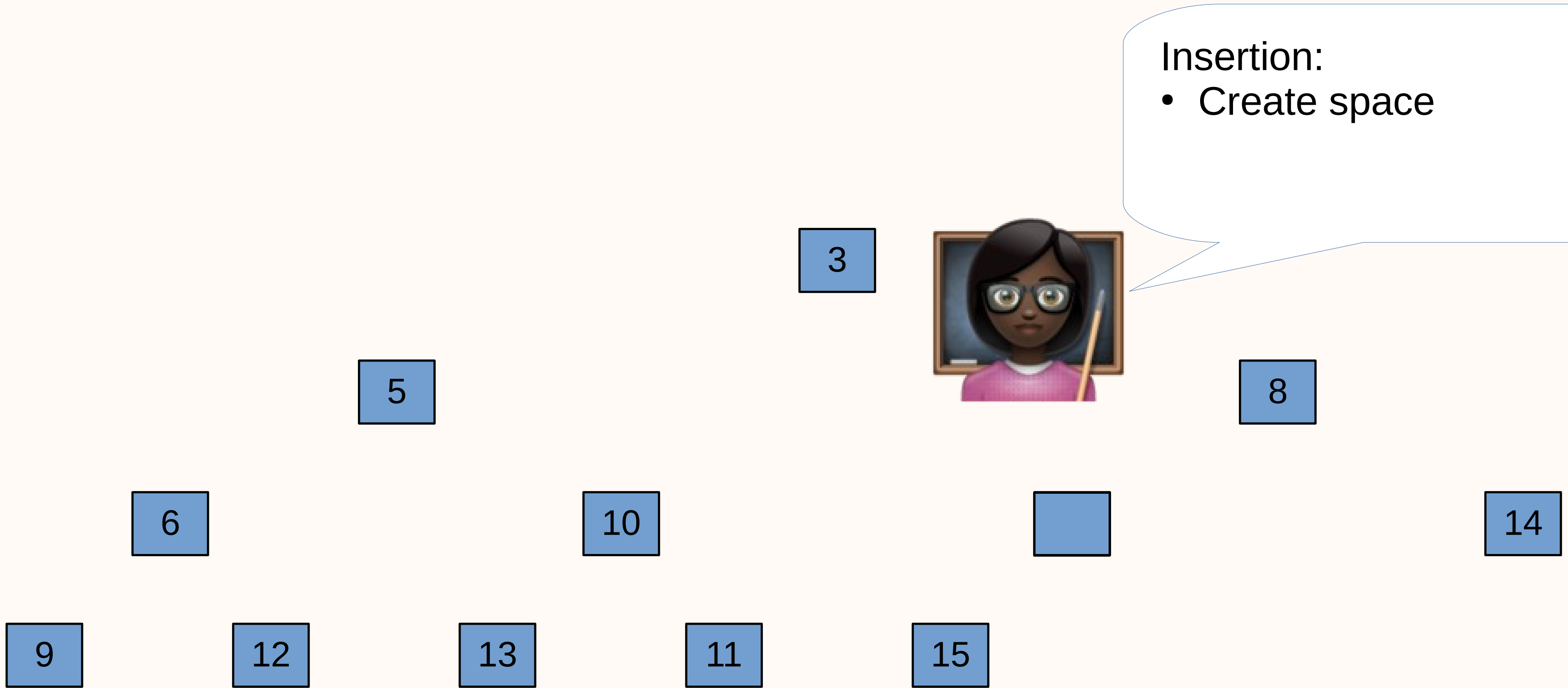
Insertion:

3

5 | 8

6 | 10 | | 14

9 | 12 | 13 | 11 | 15


Insertion:
• Create space
3
5
8
6
10
14
9
12
13
11
15

Insertion:
- Create space
- Trickle down greater nodes

3

5 8

6 10 14

9 12 13 11 15

Insertion:
- Create space
- Trickle down greater nodes
- Insert into space

3

5

8

6

10

14

9

12

13

11

15

Insertion:

- Create space
- Trickle down greater nodes
- Insert into space

3

5 8

6 10 14

9 12 13 11 15

7

Insertion:

- Create space
- Trickle down greater nodes
- Insert into space

3

5 8

6 10 10 14

9 12 13 11 15

7

Insertion:

- Create space
- Trickle down greater nodes
- Insert into space

3

5 8

6 10 14

9 12 13 11 15

7

Insertion:

- Create space
- Trickle down greater nodes
- Insert into space

3

5 8

6 10 14

9 12 13 11 15

7

3

5 7

6 10 8 14

9 12 13 11 15 10

Pop top:

3

5

7

6

10

8

14

9

12

13

11

15

10

Pop top:

- Remove top

3

5

7

6

10

8

14

9

12

13

11

15

10

Pop top:

- Remove top
- Trickle up lesser child

3

5 7

6 10 8 14

9 12 13 11 15 10

Pop top:

- Remove top
- Trickle up lesser child
- move-insert last into hole

3

5 7

6 10 8 14

9 12 13 11 15 10

Pop top:

- Remove top
- Trickle up lesser child
- move-insert last into hole

5 7

6 10 8 14

9 12 13 11 15 10

Pop top:
- Remove top
- Trickle up lesser child
- move-insert last into hole

		5				7		
	6		10		8		14	
9	12		13	11	15	10		

Pop top:

- Remove top
- Trickle up lesser child
- move-insert last into hole

			5								7				
	6				10				8				14		
9		12		13		11		15		10					

Pop top:
- Remove top
- Trickle up lesser child
- move-insert last into hole

| |

5 7

6 10 8 14

9 12 13 11 15 10

Pop top:

- Remove top
- Trickle up lesser child
- move-insert last into hole

5 7

6 10 8 14

9 12 13 11 15

Pop top:

- Remove top
- Trickle up lesser child
- move-insert last into hole

5 7

6 10 8 14

9 12 13 11 15 10

5
1

6 7
2 3

9 10 8 14
4 5 6 7

10 12 13 11 15
8 9 10 11 12

Addressing:
The index of a parent node is half (rounded down) of that of a child.

Value	Index
5	1
6	2
7	3
9	4
10	5
8	6
14	7
10	8
12	9
13	10
11	11
15	12

Addressing:
The index of a parent node is half (rounded down) of that of a child.

Index	1	2	3	4	5	6	7	8	9	10	11	12
Value	5	6	7	9	10	8	14	10	12	13	11	15

Addressing:
The index of a parent node is half (rounded down) of that of a child.

Value	Index
5	1
6	2
7	3
9	4
10	5
8	6
	7
10	8
12	9
13	10
11	11
15	12

Array indexes!
No pointer chasing!

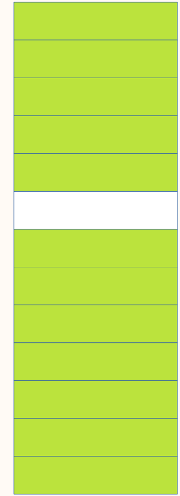
The heap is not searchable,
so how handle cancellation?

The heap is not searchable, so how handle cancellation?

```
struct timer_action {
  uint32_t (*callback)(void*);
  void* userp;
};
```

The heap is not searchable,
so how handle cancellation?

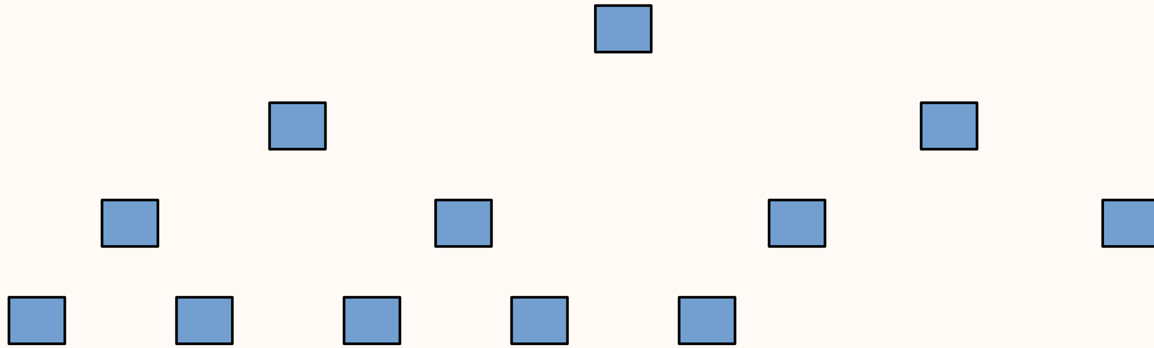
```
struct timer_action {
  uint32_t (*callback)(void*);
  void* userp;
};
```

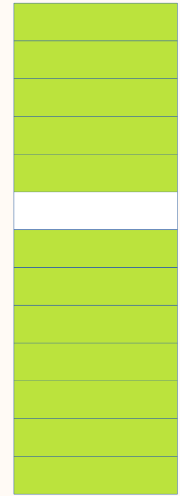


actions

The heap is not searchable, so how handle cancellation?

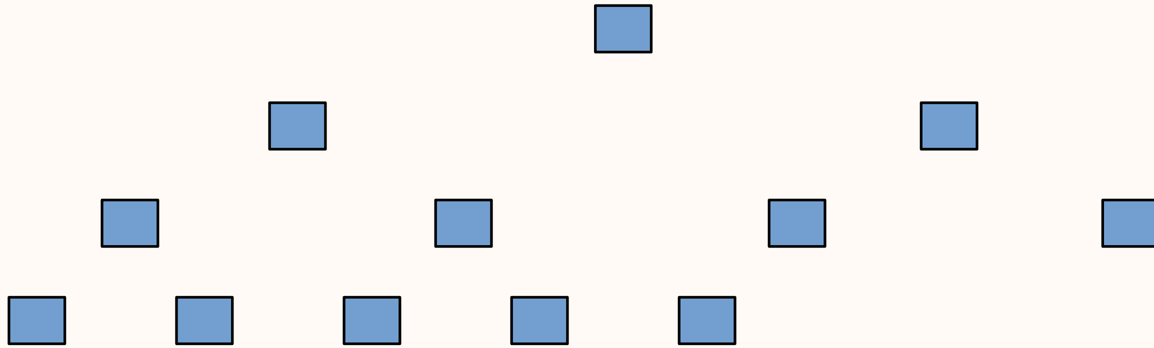
```
struct timer_action {
  uint32_t (*callback)(void*);
  void* userp;
};
```

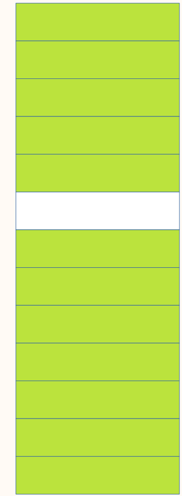




```
struct timer_action {
  uint32_t (*callback)(void*);
  void* userp;
};
```

The heap is not searchable, so how handle cancellation?

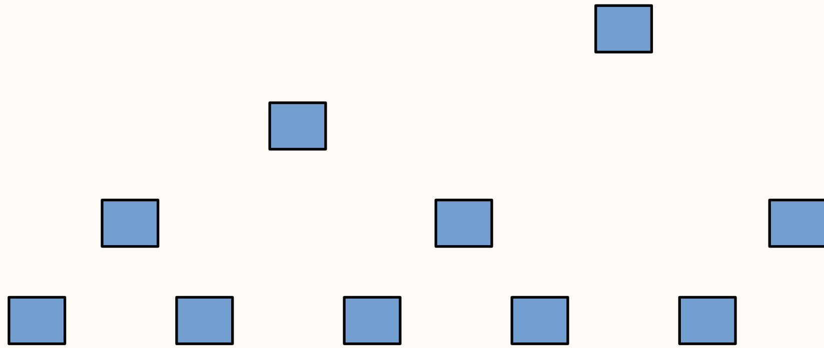


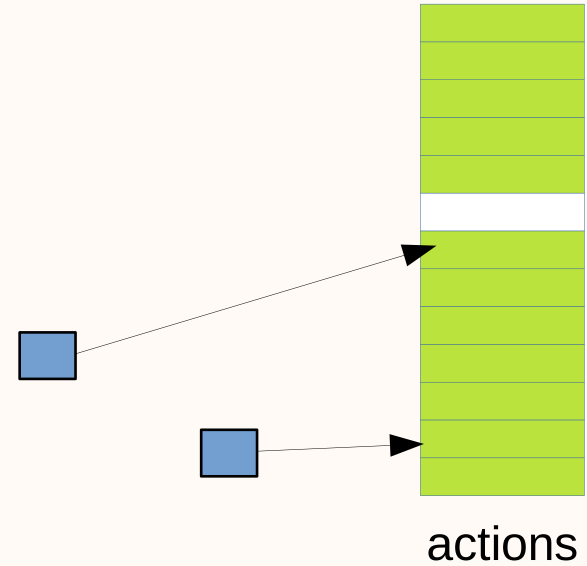



```
struct timeout {
  uint32_t deadline;
  uint32_t action_index;
};
```

The heap is not searchable,
so how handle cancellation?

```
struct timer_action {
  uint32_t (*callback)(void*);
  void* userp;
};
```

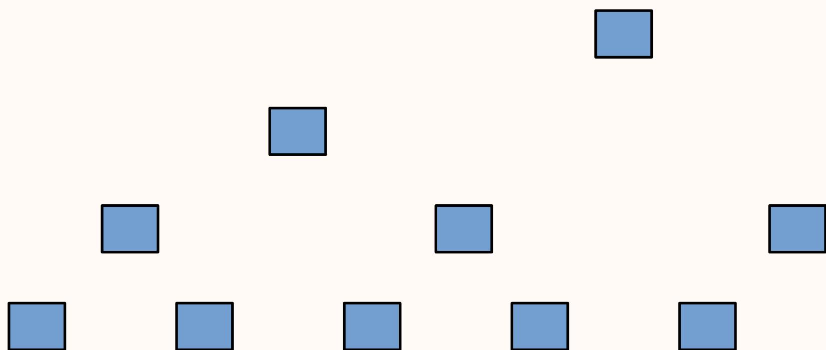


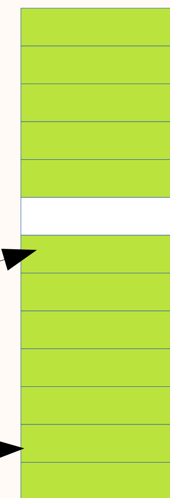



```
struct timeout {
  uint32_t deadline;
  uint32_t action_index;
};
```

```
struct timer_action {
  uint32_t (*callback)(void*);
  void* userp;
};
```

The heap is not searchable, so how handle cancellation?



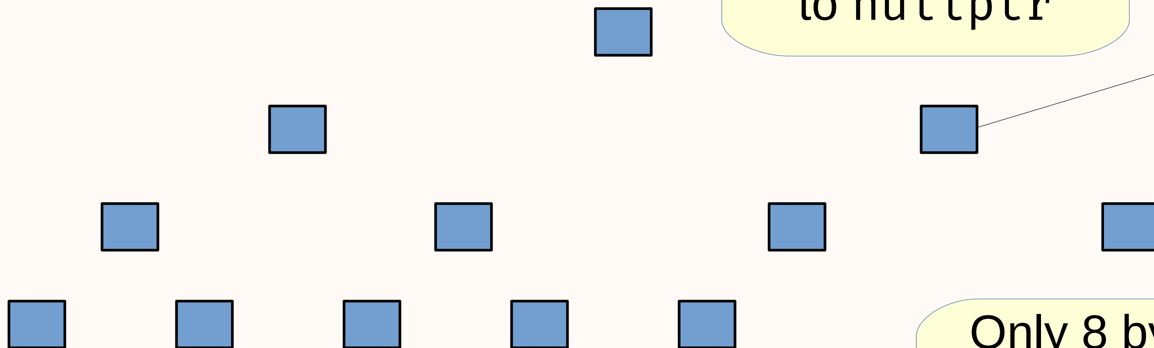


Only 8 bytes per element of working data in the heap.

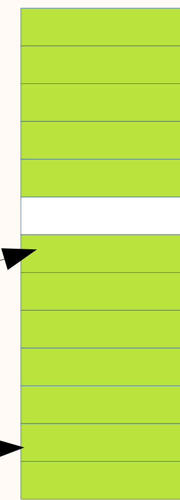
```
struct timeout {
  uint32_t deadline;
  uint32_t action_index;
};
```

```
struct timer_action {
  uint32_t (*callback)(void*);
  void* userp;
};
```

The heap is not searchable, so how handle cancellation?



Cancel by setting `callback` to `nullptr`



Only 8 bytes per element of working data in the heap.

```
struct timeout {
  uint32_t deadline;
  uint32_t action_index;
};
```

```cpp
struct timer_data {
  uint32_t deadline;
  uint32_t action_index;
};

struct is_after {
  bool operator()(const timer_data& lh, const timer_data& rh) const {
    return lh.deadline < rh.deadline;
  }
};

std::priority_queue<timer_data, std::vector<timer_data>, is_after>
timeouts;

timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp) {
  auto action_index = actions.push(cb, userp);
  timeouts.push(timer_data{deadline, action_index});
  return action_index;
}
```

```
struct timer_data {
  uint32_t deadline;
  uint32_t action_index;
};

struct is_after {
  bool operator()(const timer_data& lh, const timer_data& rh) const {
    return lh.deadline < rh.deadline;
  }
};

std::priority_queue<timer_data, std::vector<timer_data>, is_after>
timeouts;

timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp) {
  auto action_index = actions.push(cb, userp);
  timeouts.push(timer_data{deadline, action_index});
  return action_index;
}
```

Container adapter that implements a heap

```cpp
struct timer_data {
  uint32_t deadline;
  uint32_t action_index;
};

struct is_after {
  bool operator()(const timer_data& lh, const timer_data& rh) const {
    return lh.deadline < rh.deadline;
  }
};

std::priority_queue<timer_data, std::vector<timer_data>, is_after>
timeouts;

timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp) {
  auto action_index = actions.push(cb, userp);
  timeouts.push(timer_data{deadline, action_index});
  return action_index;
}
```

Container adapter that implements a heap

```
struct timer_data {
  uint32_t deadline;
  uint32_t action_index;
};

struct is_after {
  bool operator()(const timer_data& lh, const timer_data& rh) const {
    return lh.deadline < rh.deadline;
  }
};

std::priority_queue<timer_data, std::vector<timer_data>, is_after>
timeouts;

timer schedule_timer(uint32_t deadline, timer_cb cb, void* userp) {
  auto action_index = actions.push(cb, userp);
  timeouts.push(timer_data{deadline, action_index});
  return action_index;
}
```

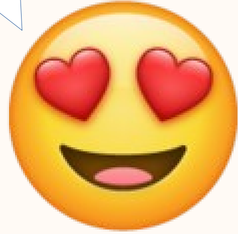
Container adapter that implements a heap

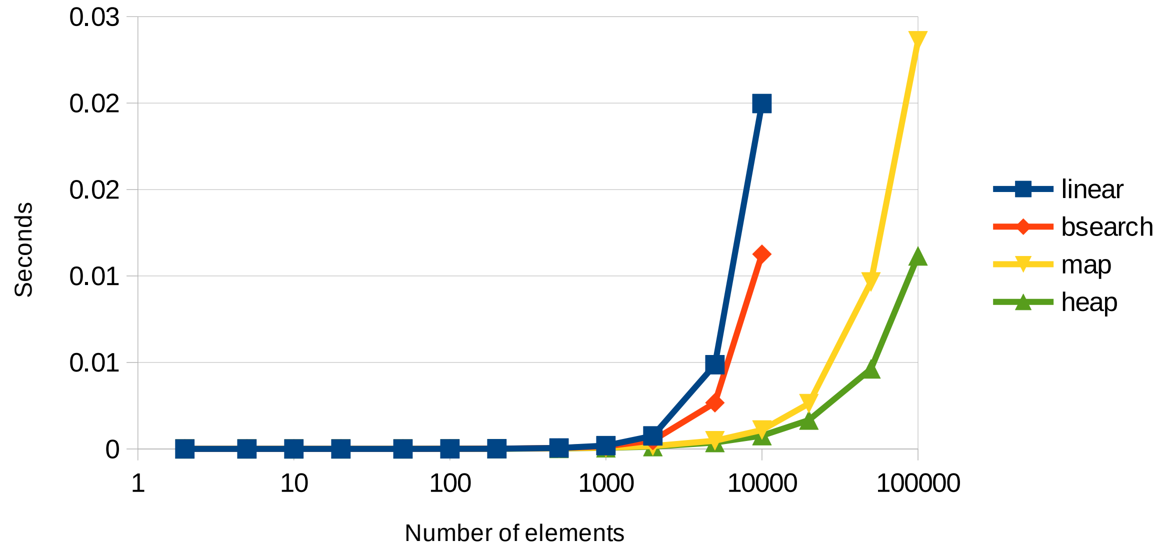
```cpp
bool shoot_first() {
  while (!timeouts.empty()) {
    auto& t = timeouts.top();
    auto& action = actions[t.action_index];
    if (action.callback) break;
    actions.remove(t.action_index);
    timeouts.pop();
  }
  if (timeouts.empty()) return false;
  auto& t = timeouts.top();
  auto& action = actions[t.action_index];
  action.callback(action.userp);
  actions.remove(t.action_index);
  timeouts.pop();
  return true;
}
```

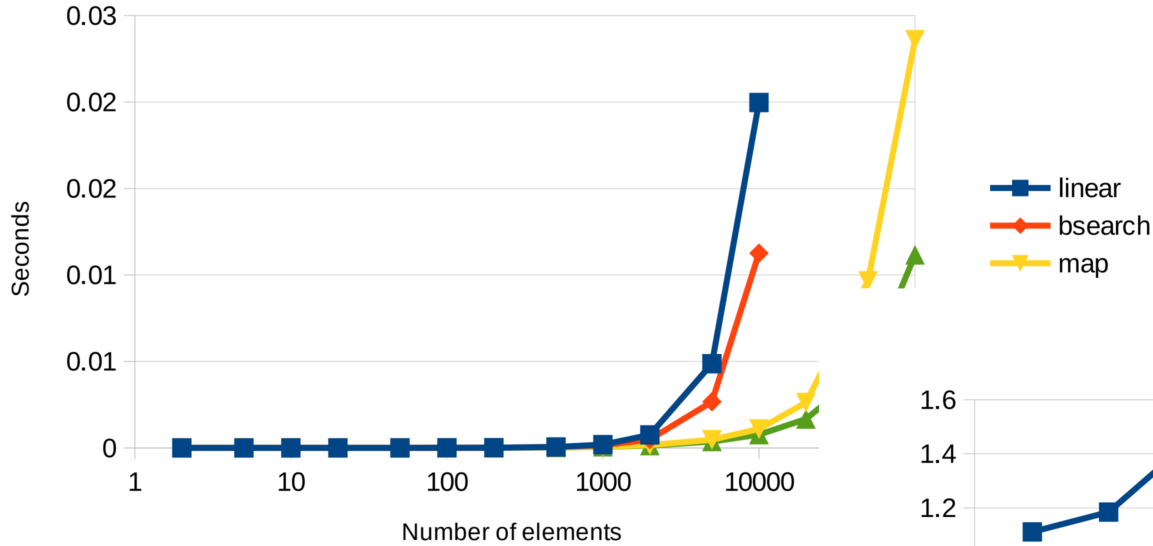
Pop-off any cancelled items

Live demo

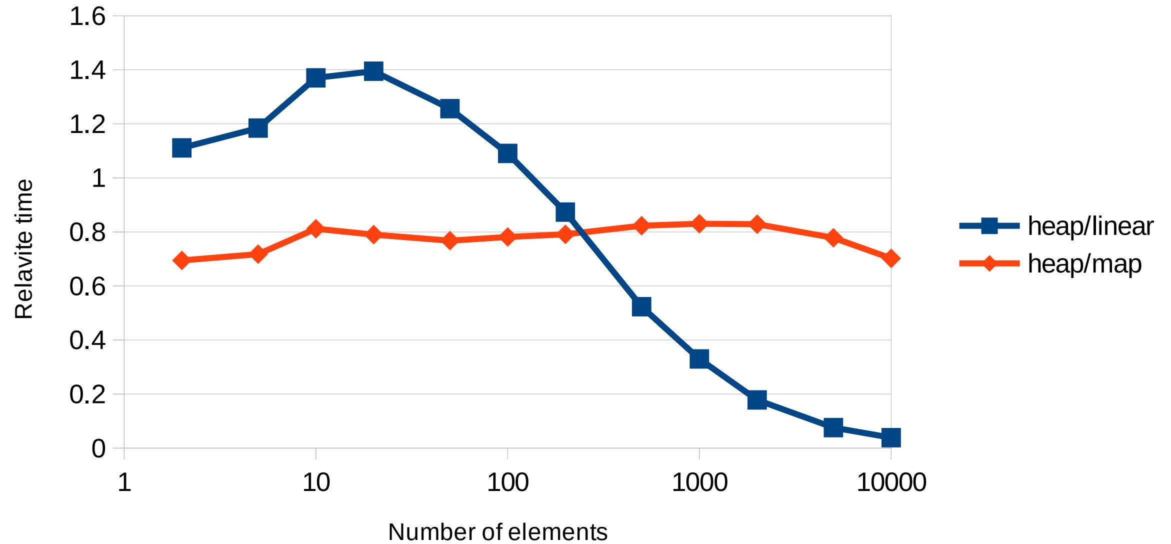
A lot fewer everything!
and nearly twice as fast too




Execution time
Seconds
0.03
0.02
0.02
0.01
0.01
0
1
10
100
1000
10000
100000
Number of elements
linear
bsearch
map
heap

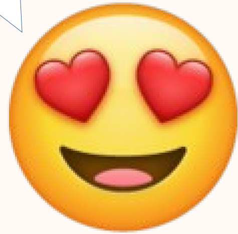


Execution time
0.03
0.02
0.02
0.01
0.01
0
Seconds
1
10
100
1000
10000
Number of elements
linear
bsearch
map

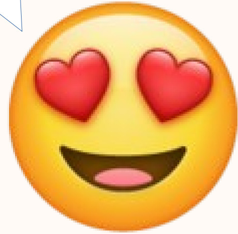


Relative execution time
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Relavite time
1
10
100
1000
10000
Number of elements
heap/linear
heap/map

A lot fewer everything!
and nearly twice as fast too

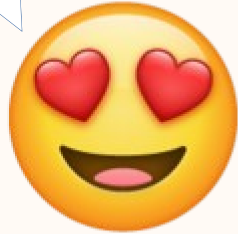


A lot fewer everything!
and nearly twice as fast too



But there are many cache misses
in the adjust-heap functions

A lot fewer everything!
and nearly twice as fast too

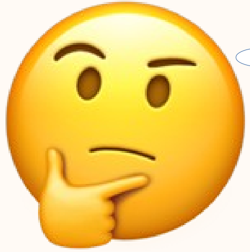


But there are many cache misses
in the adjust-heap functions

Can we do better?

How do the entries fit in cache lines?

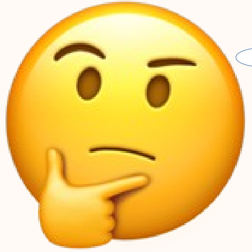
How do the entries fit in cache lines?



How do the entries fit in cache lines?

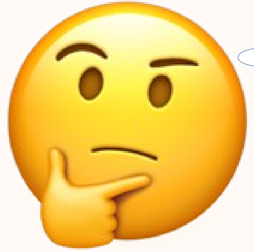
How do the entries fit in cache lines?

How do the entries fit in cache lines?

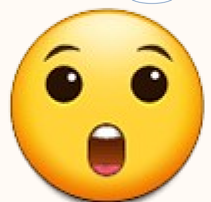


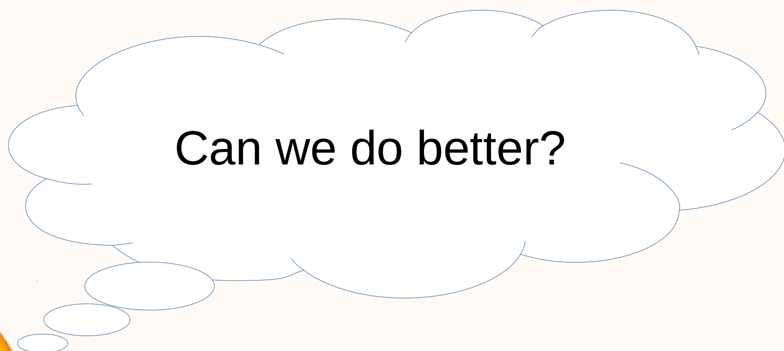
How do the entries fit in cache lines?

How do the entries fit in cache lines?

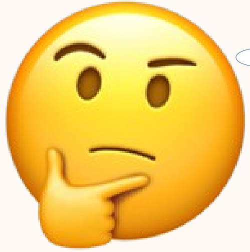


Every generation is
on a new cache line

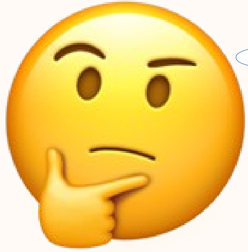



Can we do better?

Can we do better?



Can we do better?



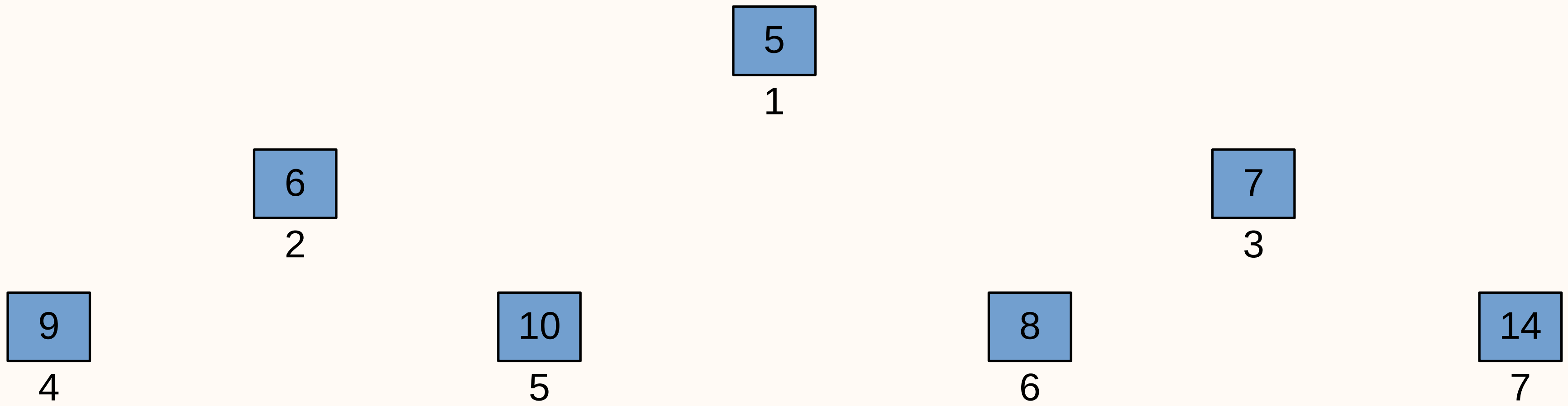
Can we do better?

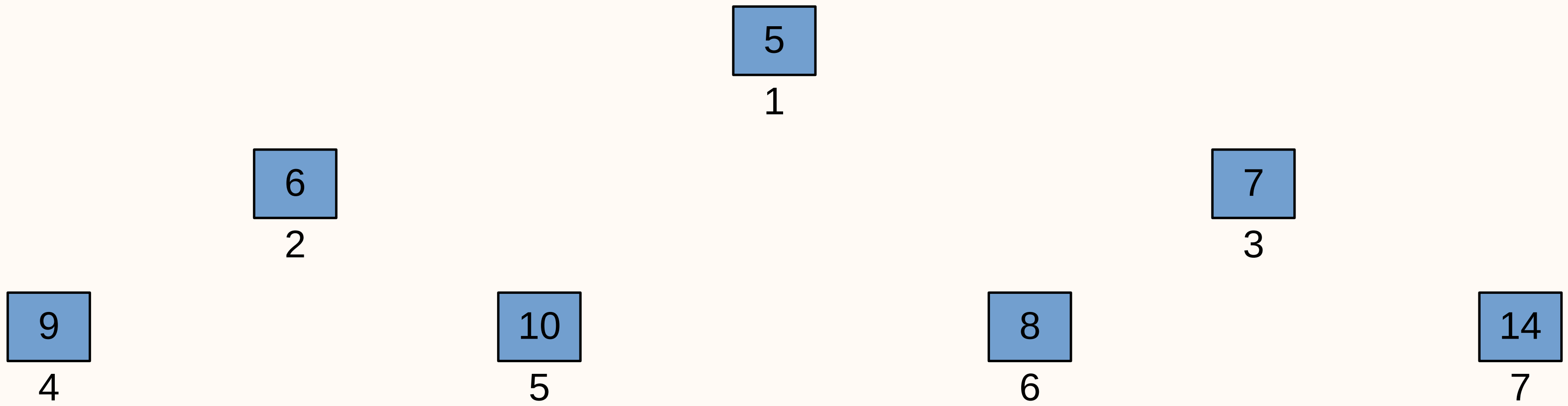
Can we do better?

Can we do better?

Can we do better?

Three generations per cache line!

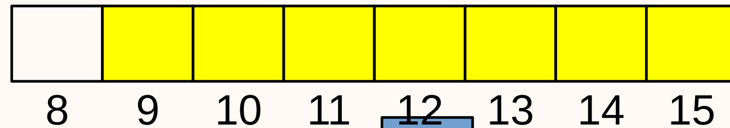

5
1
6
2
7
3
9
4
10
5
8
6
14
7


5
1
6
2
7
3
9
4
10
5
8
6
14
7

Index	0	1	2	3	4	5	6	7
Value		5	6	7	9	10	8	14

5
1

0



6
2

7
3

9
4

10
5

8
6

14
7

5

1

0

8 9 10 11 12 13 14 15

6

2

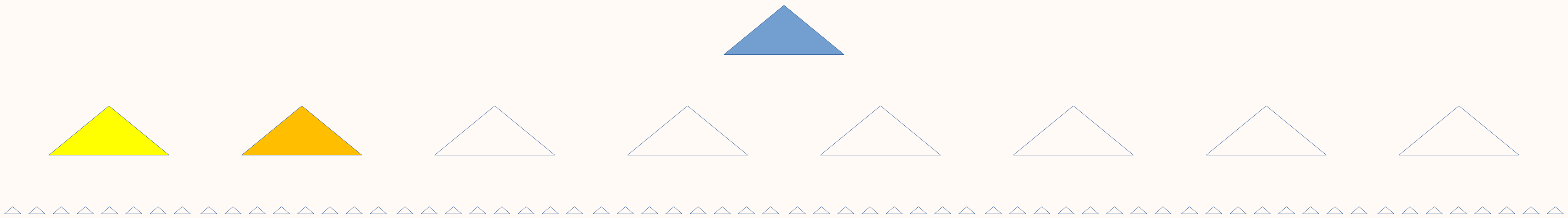
7

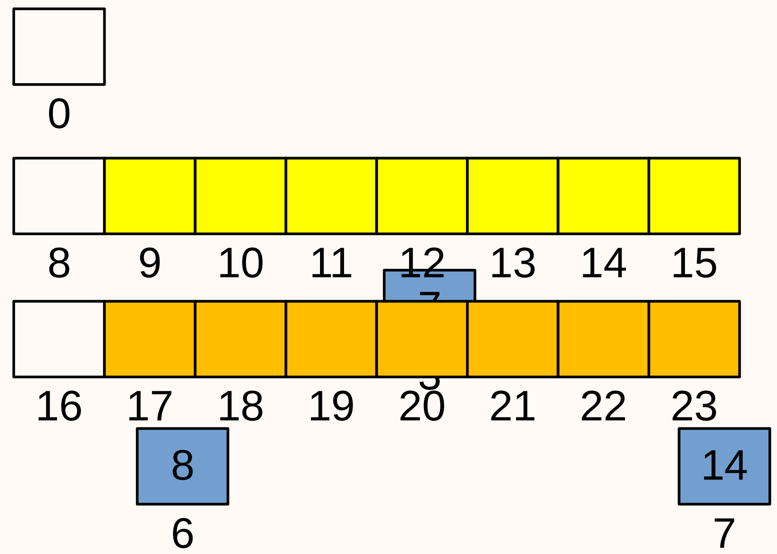
16 17 18 19 20 21 22 23

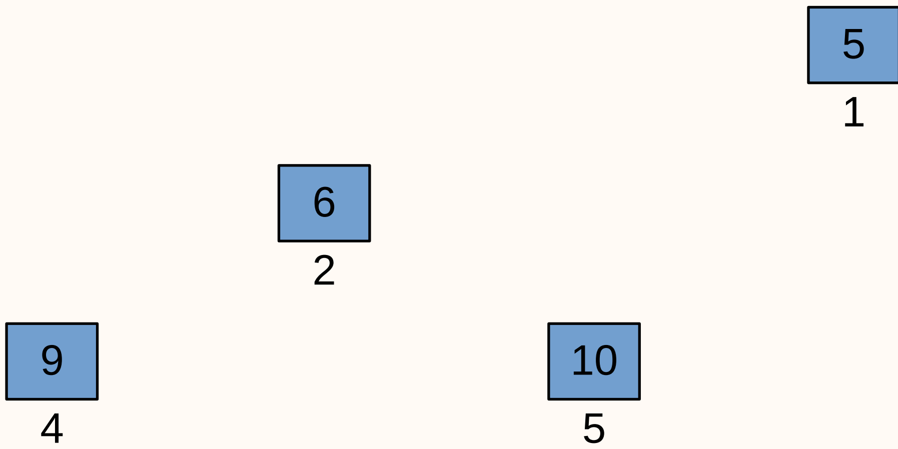
3

9 10 8 14

4 5 6 7

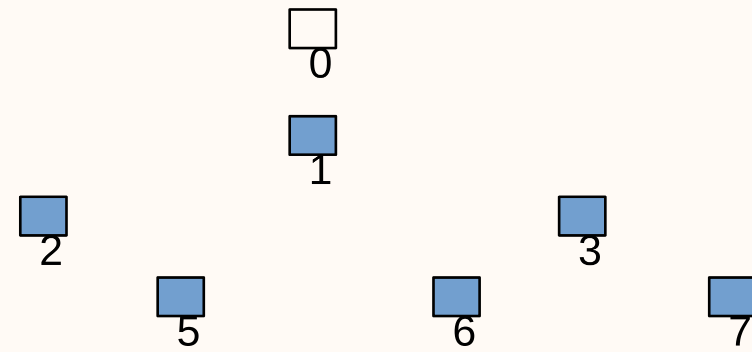



0
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
8
6
14
7


5
1
6
2
9
4
10
5

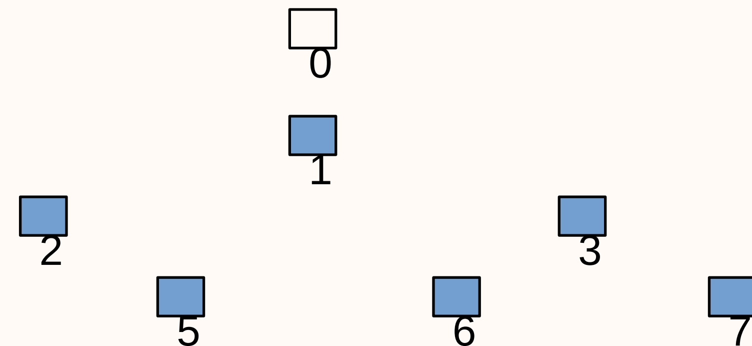
```
class timeout_store {
  static constexpr size_t block_size = 8;
  static constexpr size_t block_mask = block_size - 1U;

  static size_t block_offset(size_t idx) {
    return idx & block_mask;
  }
  static size_t block_base(size_t idx) {
    return idx & ~block_mask;
  }
  static bool is_block_root(size_t idx) {
    return block_offset(idx) == 1;
  }
  static bool is_block_leaf(size_t idx) {
    return (idx & (block_size >> 1)) != 0U;
  }
  ...
};
```



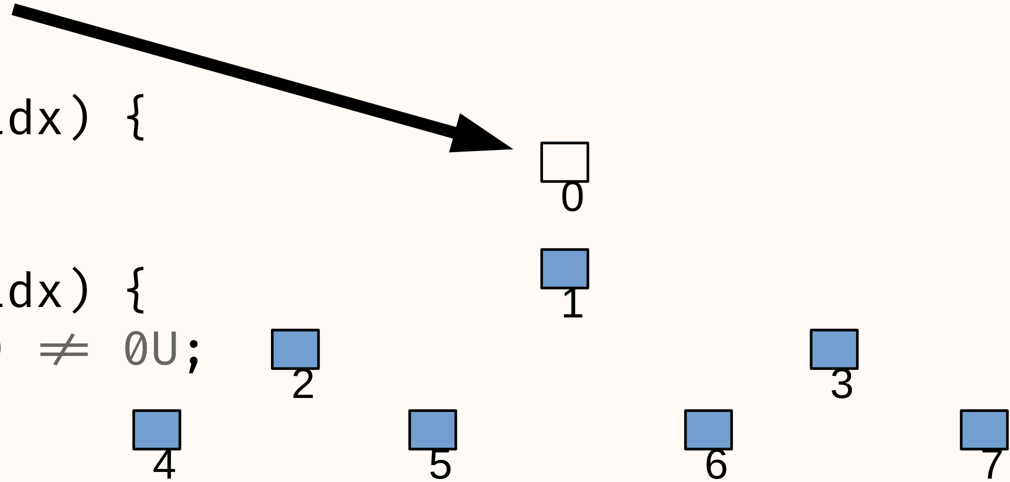
```
class timeout_store {
  static constexpr size_t block_size = 8;
  static constexpr size_t block_mask = block_size - 1U;

  static size_t block_offset(size_t idx) {
    return idx & block_mask;
  }
  static size_t block_base(size_t idx) {
    return idx & ~block_mask;
  }
  static bool is_block_root(size_t idx) {
    return block_offset(idx) == 1;
  }
  static bool is_block_leaf(size_t idx) {
    return (idx & (block_size >> 1)) != 0U;
  }
  ...
};
```



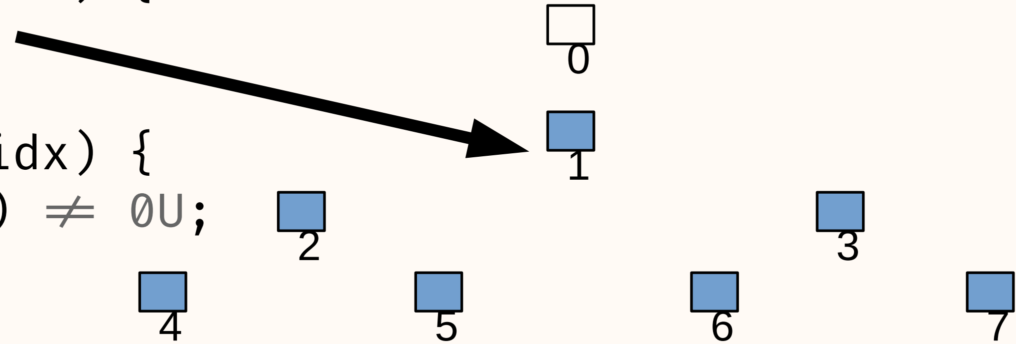
```
class timeout_store {
  static constexpr size_t block_size = 8;
  static constexpr size_t block_mask = block_size - 1U;

  static size_t block_offset(size_t idx) {
    return idx & block_mask;
  }
  static size_t block_base(size_t idx) {
    return idx & ~block_mask;
  }
  static bool is_block_root(size_t idx) {
    return block_offset(idx) == 1;
  }
  static bool is_block_leaf(size_t idx) {
    return (idx & (block_size >> 1)) != 0U;
  }
  ...
};
```



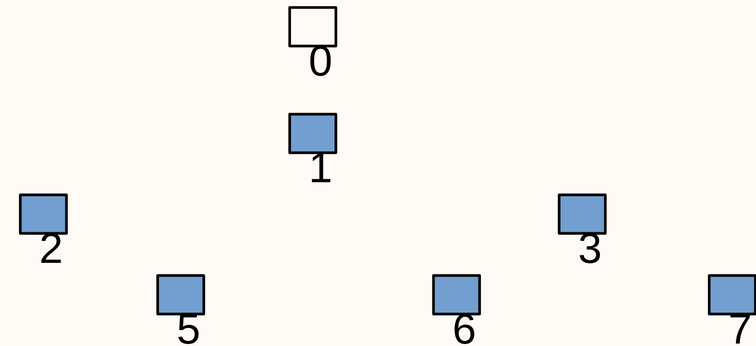
```
class timeout_store {
  static constexpr size_t block_size = 8;
  static constexpr size_t block_mask = block_size - 1U;

  static size_t block_offset(size_t idx) {
    return idx & block_mask;
  }
  static size_t block_base(size_t idx) {
    return idx & ~block_mask;
  }
  static bool is_block_root(size_t idx) {
    return block_offset(idx) == 1;
  }
  static bool is_block_leaf(size_t idx) {
    return (idx & (block_size >> 1)) != 0U;
  }
  ...
};
```



```
class timeout_store {
  static constexpr size_t block_size = 8;
  static constexpr size_t block_mask = block_size - 1U;

  static size_t block_offset(size_t idx) {
    return idx & block_mask;
  }
  static size_t block_base(size_t idx) {
    return idx & ~block_mask;
  }
  static bool is_block_root(size_t idx) {
    return block_offset(idx) == 1;
  }
  static bool is_block_leaf(size_t idx) {
    return (idx & (block_size >> 1)) != 0U;
  }
  ...
};
```



```
class timeout_store {
  static constexpr size_t block_size = 8;
  static constexpr size_t block_mask = block_size - 1U;

  static size_t block_offset(size_t idx);
  static size_t block_base(size_t idx);
  static bool is_block_root(size_t idx);
  static bool is_block_leaf(size_t idx);

  static size_t left_child_of(size_t idx) {
    if (!is_block_leaf(idx)) return idx + block_offset(idx);
    auto base = block_base(idx) + 1;
    return base * block_size + child_no(idx) * block_size * 2 + 1;
  }


   ...
};
```

```
class timeout_store {
  static constexpr size_t block_size = 8;
  static constexpr size_t block_mask = block_size - 1U;

  static size_t block_offset(size_t idx);
  static size_t block_base(size_t idx);
  static bool is_block_root(size_t idx);
  static bool is_block_leaf(size_t idx);

  static size_t left_child_of(size_t idx) {
    if (!is_block_leaf(idx)) return idx + block_offset(idx);
    auto base = block_base(idx) + 1;
    return base * block_size + child_no(idx) * block_size * 2 + 1;
  }


   ...
};
```

```
class timeout_store {
  static constexpr size_t block_size = 8;
  static constexpr size_t block_mask = block_size - 1U;

  static size_t block_offset(size_t idx);
  static size_t block_base(size_t idx);
  static bool is_block_root(size_t idx);
  static bool is_block_leaf(size_t idx);

  static size_t parent_of(size_t idx) {
    auto const node_root = block_base(idx);
    if (!is_block_root(idx)) return node_root + block_offset(idx) / 2;
    auto parent_base = block_base(node_root / block_size - 1);
    auto child = ((idx - block_size) / block_size - parent_base) / 2;
    return parent_base + block_size / 2 + child;
  }
  ...
};
```

```cpp
class timeout_store {
  static constexpr size_t block_size = 8;
  static constexpr size_t block_mask = block_size - 1U;

  static size_t block_offset(size_t idx);
  static size_t block_base(size_t idx);
  static bool is_block_root(size_t idx);
  static bool is_block_leaf(size_t idx);

  static size_t parent_of(size_t idx) {
    auto const node_root = block_base(idx);
    if (!is_block_root(idx)) return node_root + block_offset(idx) / 2;
    auto parent_base = block_base(node_root / block_size - 1);
    auto child = ((idx - block_size) / block_size - parent_base) / 2;
    return parent_base + block_size / 2 + child;
  }
  ...
};
```

```
class timeout_store {
  ...
  using allocator = align_allocator<64>::type<timer_data>;
  std::vector<timer_data, allocator> bheap_store;
};
```

```
class timeout_store {
  ...
  using allocator = align_allocator<64>::type<timer_data>;
  std::vector<timer_data, allocator> bheap_store;
};
```

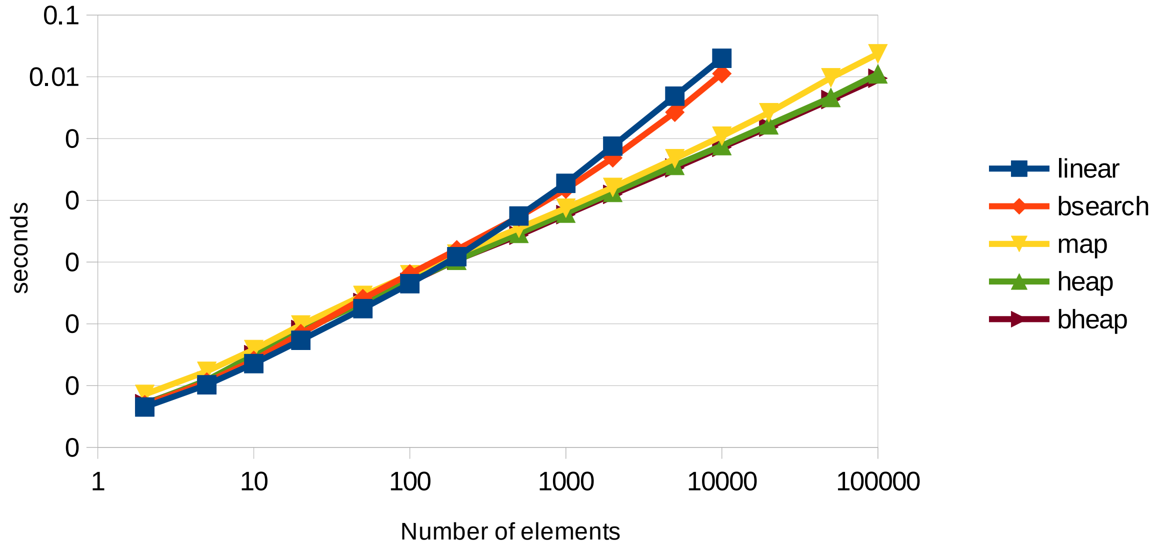
```
template <size_t N>
struct align_allocator {
  template <typename T>
  struct type {
    using value_type = T;
    static constexpr std::align_val_t alignment{N};
    T* allocate(size_t n) {
      return static_cast<T*>(operator new(n*sizeof(T), alignment));
    }
    void deallocate(T* p, size_t) {
      operator delete(p, alignment);
    }
  };
};
class timeout_store {
  ...
  using allocator = align_allocator<64>::type<timer_data>;
  std::vector<timer_data, allocator> bheap_store;
};
```

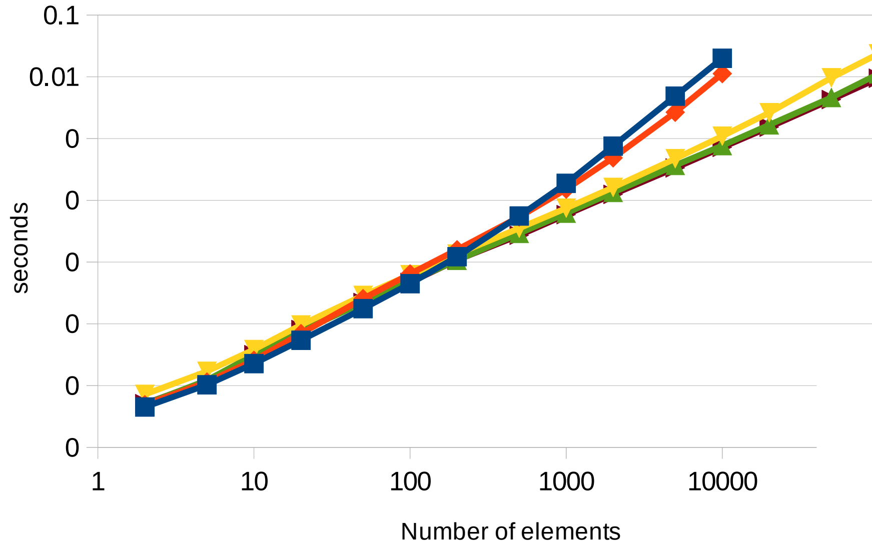
```
template <size_t N>
struct align_allocator {
  template <typename T>
  struct type {
    using value_type = T;
    static constexpr std::align_val_t alignment{N};
    T* allocate(size_t n) {
      return static_cast<T*>(operator new(n*sizeof(T), alignment));
    }
    void deallocate(T* p, size_t) {
      operator delete(p, alignment);
    }
  };
};
class timeout_store {
  ...
  using allocator = align_allocator<64>::type<timer_data>;
  std::vector<timer_data, allocator> bheap_store;
};
```

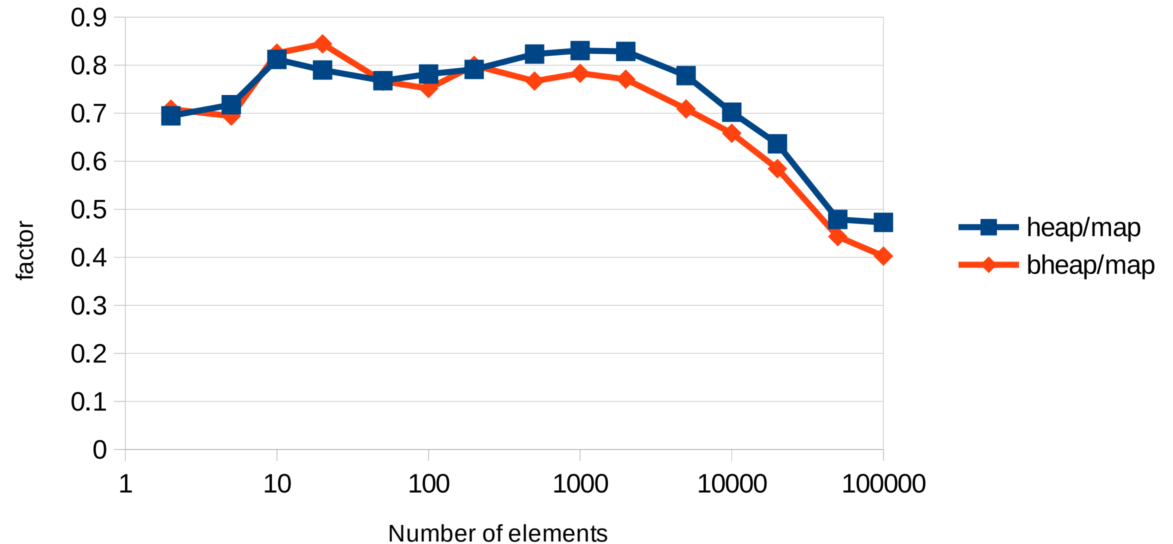
```
template <size_t N>
struct align_allocator {
  template <typename T>
  struct type {
    using value_type = T;
    static constexpr std::align_val_t alignment{N};
    T* allocate(size_t n) {
      return static_cast<T*>(operator new(n*sizeof(T), alignment));
    }
    void deallocate(T* p, size_t) {
      operator delete(p, alignment);
    }
  };
};
class timeout_store {
  ...
  using allocator = align_allocator<64>::type<timer_data>;
  std::vector<timer_data, allocator> bheap_store;
};
```

Aligned operator new and delete came with C++ 17

Live demo


Execution time
0.1
0.01
0
0
0
0
0
0
seconds
1
10
100
1000
10000
100000
Number of elements
linear
bsearch
map
heap
bheap


Execution time
0.1
0.01
0
0
0
0
0
0
seconds
1
10
100
1000
10000
Number of elements
linear
bsearch
map
heap
bheap


Execution time relative map
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
factor
1
10
100
1000
10000
100000
Number of elements
heap/map
bheap/map

Rules of thumb

Rules of thumb

- Following a pointer is a cache miss, unless you have information to the contrary

Rules of thumb

- Following a pointer is a cache miss, unless you have information to the contrary
- Smaller working data set is better

Rules of thumb

- Following a pointer is a cache miss, unless you have information to the contrary
- Smaller working data set is better
- Use as much of a cache entry as you can

Rules of thumb

- Following a pointer is a cache miss, unless you have information to the contrary
- Smaller working data set is better
- Use as much of a cache entry as you can
- Sequential memory accesses can be very fast due to prefetching

Rules of thumb

- Following a pointer is a cache miss, unless you have information to the contrary
- Smaller working data set is better
- Use as much of a cache entry as you can
- Sequential memory accesses can be very fast due to prefetching
- Fewer evicted cache lines means more data in hot cache for the rest of the program

Rules of thumb

- Following a pointer is a cache miss, unless you have information to the contrary
- Smaller working data set is better
- Use as much of a cache entry as you can
- Sequential memory accesses can be very fast due to prefetching
- Fewer evicted cache lines means more data in hot cache for the rest of the program
- Mispredicted branches can evict cache entries

Rules of thumb

- Following a pointer is a cache miss, unless you have information to the contrary
- Smaller working data set is better
- Use as much of a cache entry as you can
- Sequential memory accesses can be very fast due to prefetching
- Fewer evicted cache lines means more data in hot cache for the rest of the program
- Mispredicted branches can evict cache entries
- Measure measure measure

Resources

Ulrich Drepper - “What every programmer should know about memory”
http://www.akkadia.org/drepper/cpumemory.pdf

Milian Wolff - “Linux perf for Qt Developers”
https://www.youtube.com/watch?v=L4NClVxqdMw

Travis Downs - “Cache counters rant”
https://gist.github.com/travisdowns/90a588deaaa1b93559fe2b8510f2a739

Emery Berger - “Performance Matters”
https://www.youtube.com/watch?v=r-TLSBdHe1A

What Do You Mean by “Cache Friendly”?

Björn Fahller

bjorn@fahller.se

@bjorn_fahller

@rollbear

#include <C++>