Heartbeat Scheduling

Provable Efficiency for Nested Parallelism

Umut Acar
Carnegie Mellon University and Inria

Arthur Charguéraud
Inria & University of Strasbourg, ICube

Adrien Guatto
Inria

Mike Rainey
Inria & Indiana University

Filip Sieczkowski
Inria

Motivation: make it easier to write high-level and efficient fork-join parallel code

Running example:

(using notation of the Cilk language extensions for C/C++)

Applies function f to iterates in the range [lo, hi)

```
void map(lo, hi, f)
  if lo <= hi
    return
  else if lo + 1 == hi
    f(lo)
    return
  int mid = (lo + hi) / 2
  spawn map(lo, mid, f)
        map(mid, hi, f)
  sync
```

Fork point enables calls to go in parallel

Join point blocks until both calls return

Fibers and their overheads

- We consider languages with support for fork join, on a multicore system.
- Every fork point potentially creates a *fiber*.
- Each fiber creation imposes a noticeable cost at runtime.
- The total cost can range from a few percent to a large enough to negate parallelism.

Can we design a runtime technique that ensures, for <u>any</u> fork-join program, bounded overheads on the <u>overall</u> cost of fiber creation?

fiber = representation of a fork point that can move between cores by load balancing

aka: task descriptor, lightweight thread, spark, etc.

Related work & contribution

Main approaches to taming fiber-creation overheads

Related work & contribution

Main approaches to taming fiber-creation overheads

Reduce the cost of each fiber creation

(useful, but not sufficient)

Related work & contribution

Main approaches to taming fiber-creation overheads

Reduce the cost of each fiber creation

(useful, but not sufficient)

Reduce the number of fibers created (i.e., *prune* excess parallelism)

Related work & contribution

Main approaches to taming fiber-creation overheads

Reduce the cost of each fiber creation

(useful, but not sufficient)

Reduce the number of fibers created (i.e., *prune* excess parallelism)

Granularity control: Prediction of running time to throttle fiber creation

(depends on predicting execution time, requires additional information, not always available)

Related work & contribution

Main approaches to taming fiber-creation overheads

- Reduce the cost of each fiber creation
  (useful, but not sufficient)
- Reduce the number of fibers created (i.e., *prune* excess parallelism)
  - Granularity control: Prediction of running time to throttle fiber creation
    (depends on predicting execution time, requires additional information, not always available)
  - Lazy Scheduling: Delay creating a fiber until it's needed to realize parallelism
    (no formal guarantees; known adversarial inputs)

Related work & contribution

Main approaches to taming fiber-creation overheads

Reduce the cost of each fiber creation
(useful, but not sufficient)

Reduce the number of fibers created
(i.e., *prune* excess parallelism)

Granularity control:
Prediction of running time
to throttle fiber creation

(depends on predicting execution time, requires additional information, not always available)

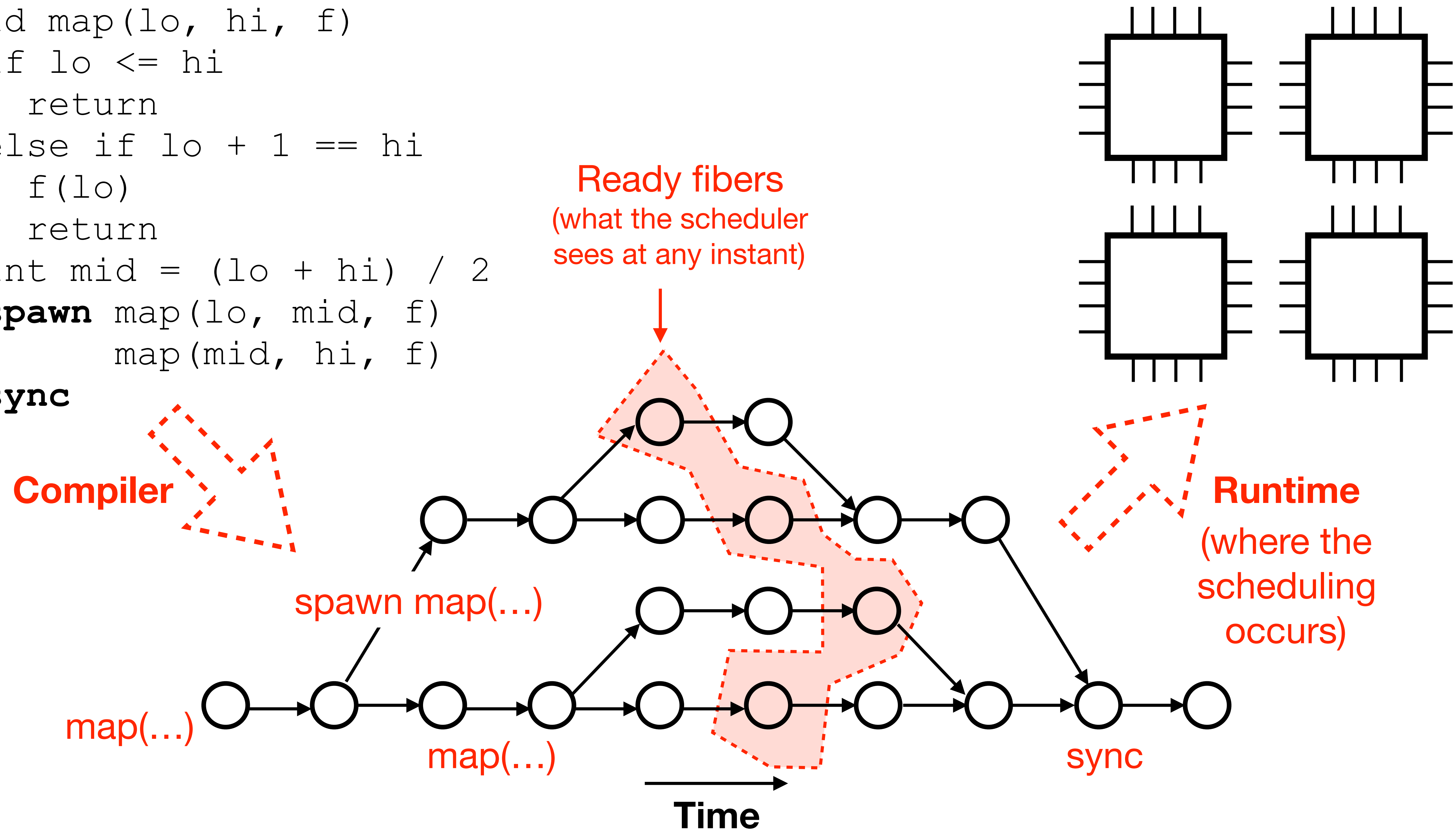
Lazy Scheduling:
Delay creating a fiber until it's
needed to realize parallelism

(no formal guarantees; known adversarial inputs)

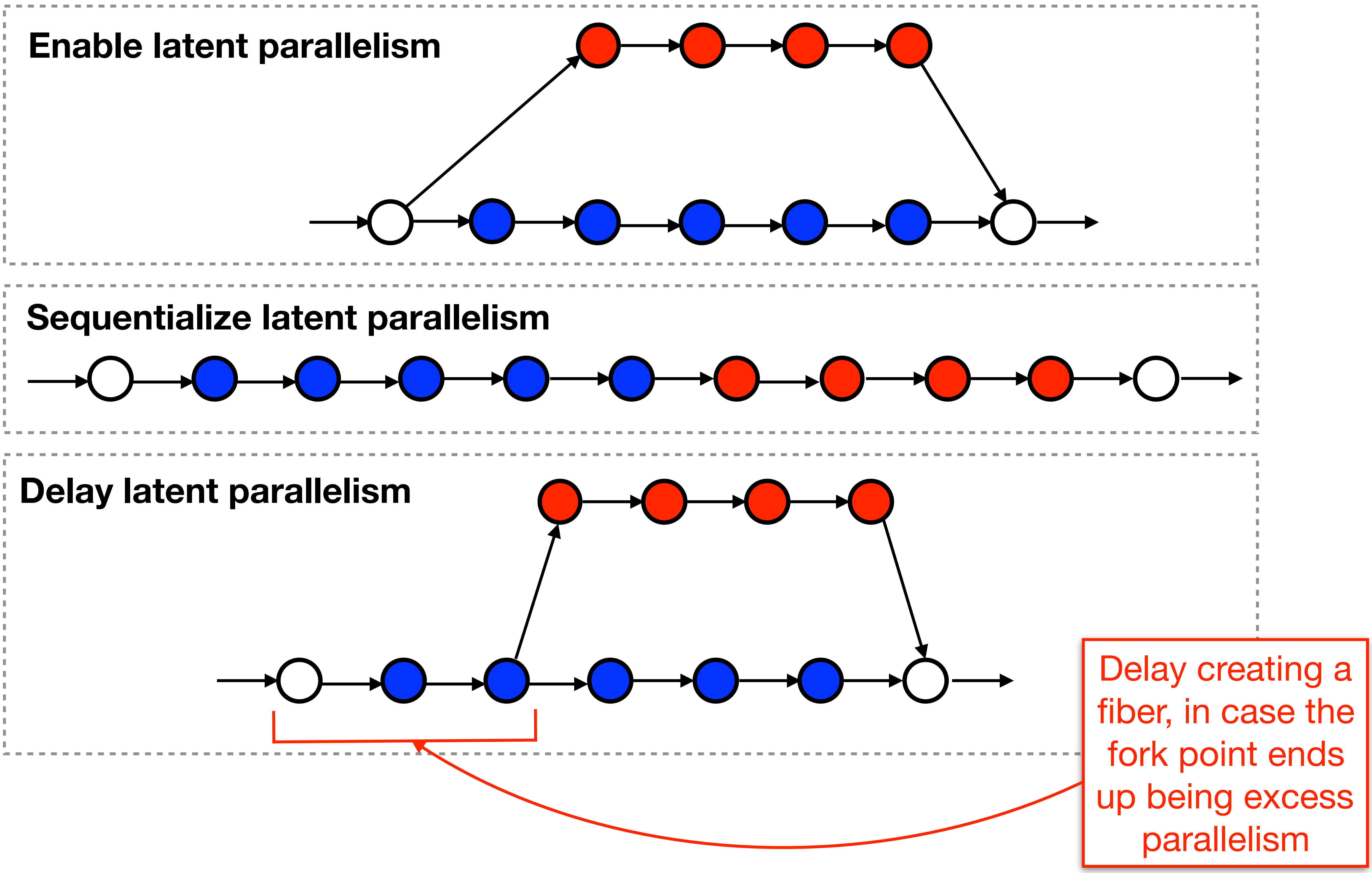
Heartbeat Scheduling: a runtime technique that, for any fork join program and any input, ensures **provably small overheads** and **good utilization**.

Scheduling fork-join programs

```
void map(lo, hi, f)
  if lo <= hi
    return
  else if lo + 1 == hi
    f(lo)
    return
  int mid = (lo + hi) / 2
  spawn map(lo, mid, f)
        map(mid, hi, f)
  sync
```



Decision to be made by the runtime for each fork point



The problem with manual granularity control

```
void map(lo, hi, f)
  if hi - lo < grain
    foreach i in [lo, hi)
      f(i)
    return
  int mid = (lo + hi) / 2
  spawn map(lo, mid, f)
        map(mid, hi, f)
  sync
```

Manual serializing for small calls

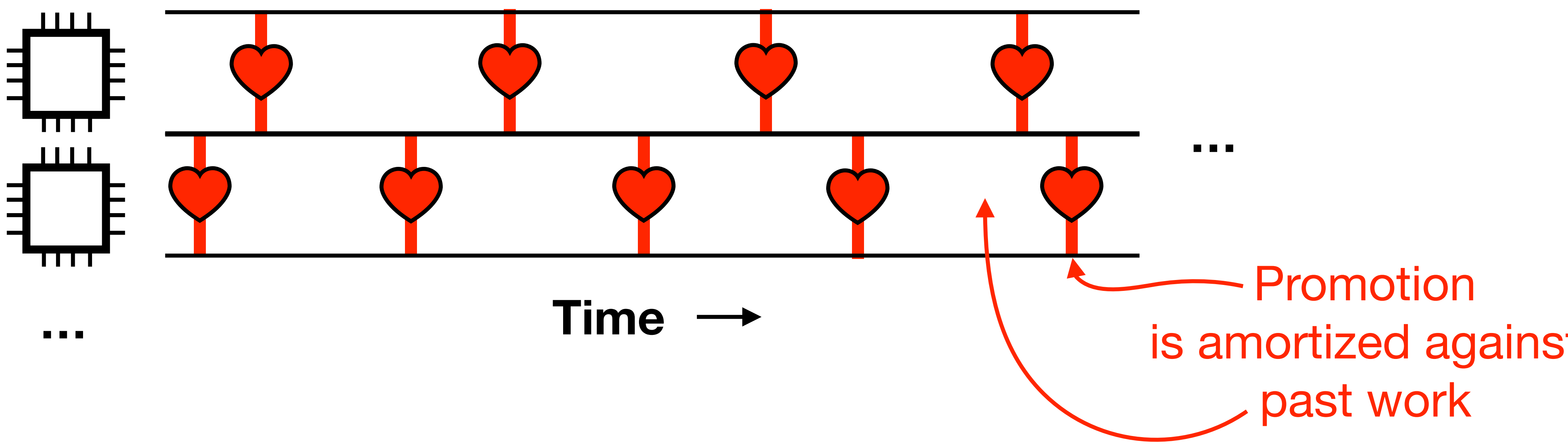
Manual granularity control degrades code quality and is not performance portable.

An acceptable setting of `grain` depends on:
(Tzannes et al 2014)

- The calling context
  - e.g., function `f` might perform little to a lot of work, might perform a call to map
- The execution environment
  - Vagaries of chip architecture
  - Number of cores
  - Operating system / software environment

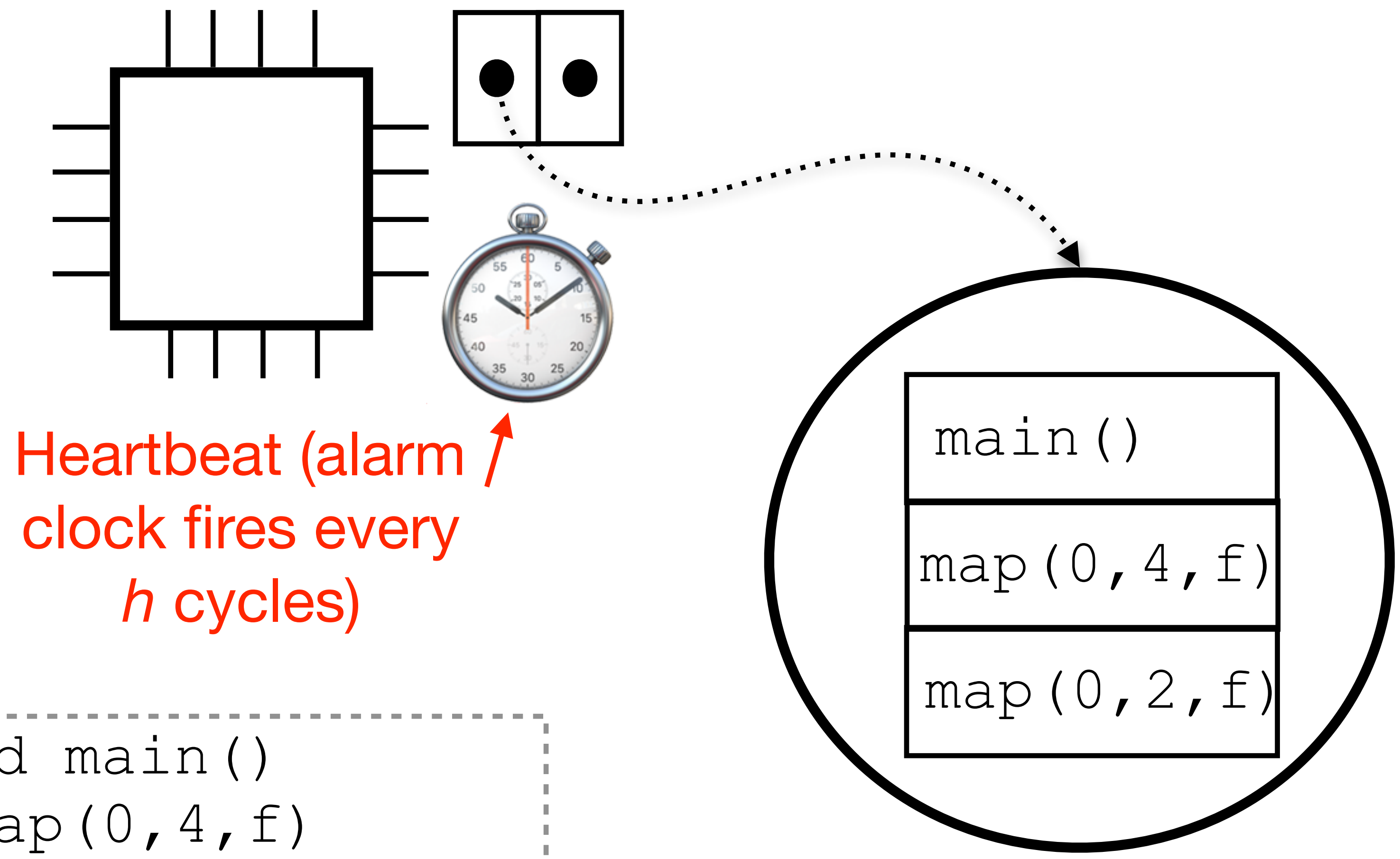
Heartbeat scheduling

Key idea: **amortize** fiber-creation overhead against past work



- At runtime, each core keeps track of how long it's been since the previous fiber creation.
- When it's been long enough, the core inspects the call stack of its current running fiber.
- If there's some latent parallel call in the call stack, the core promotes the parallel call into a new fiber.

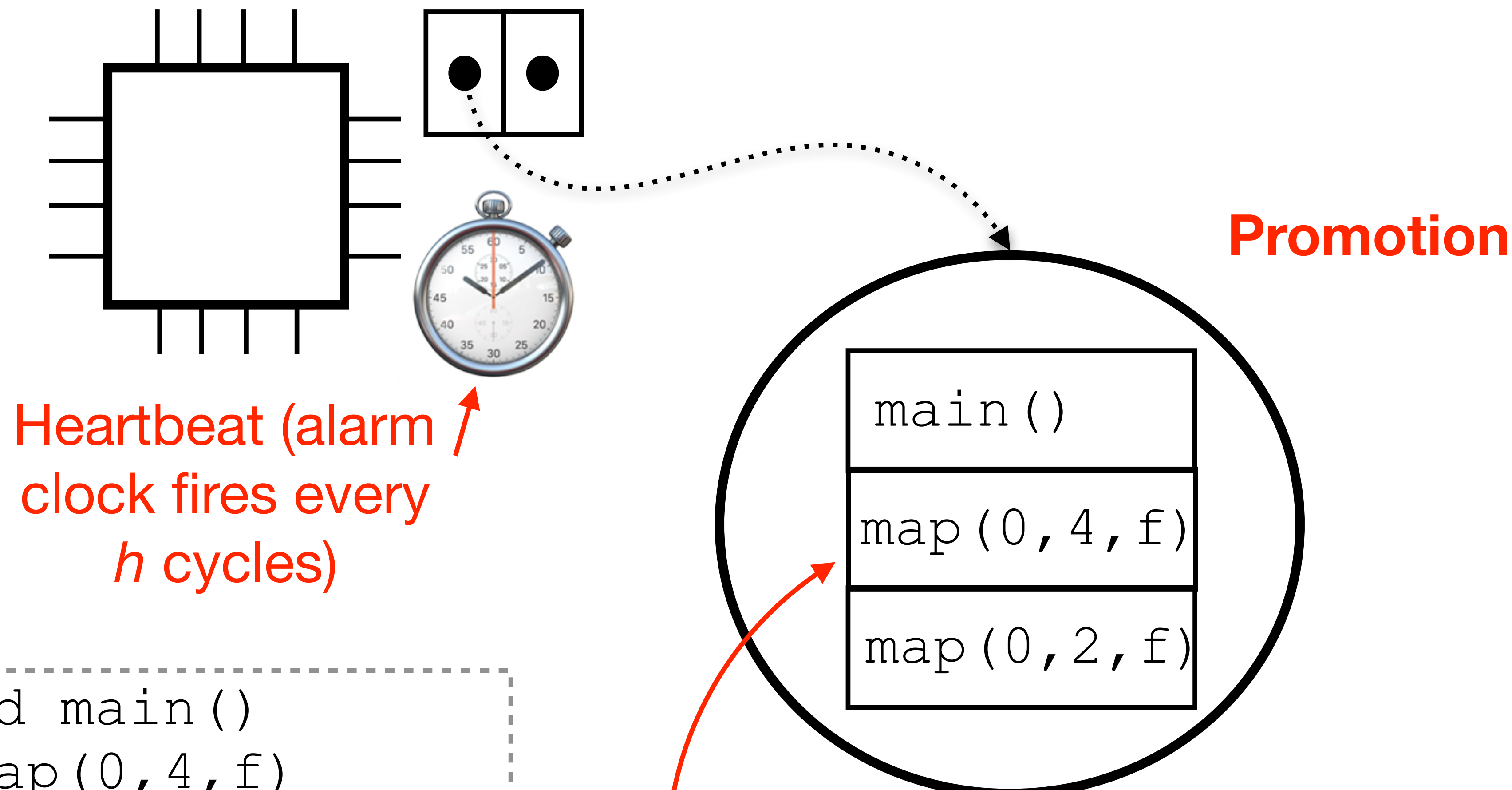
How heartbeat scheduling works




```
void main()
  map(0,4,f)
  return
```

```
void map(lo, hi, f)
  if lo <= hi
    return
  else if lo + 1 == hi
    f(lo)
    return
  int mid = (lo + hi) / 2
  spawn map(lo, mid, f)
        map(mid, hi, f)
  sync
```

How heartbeat scheduling works



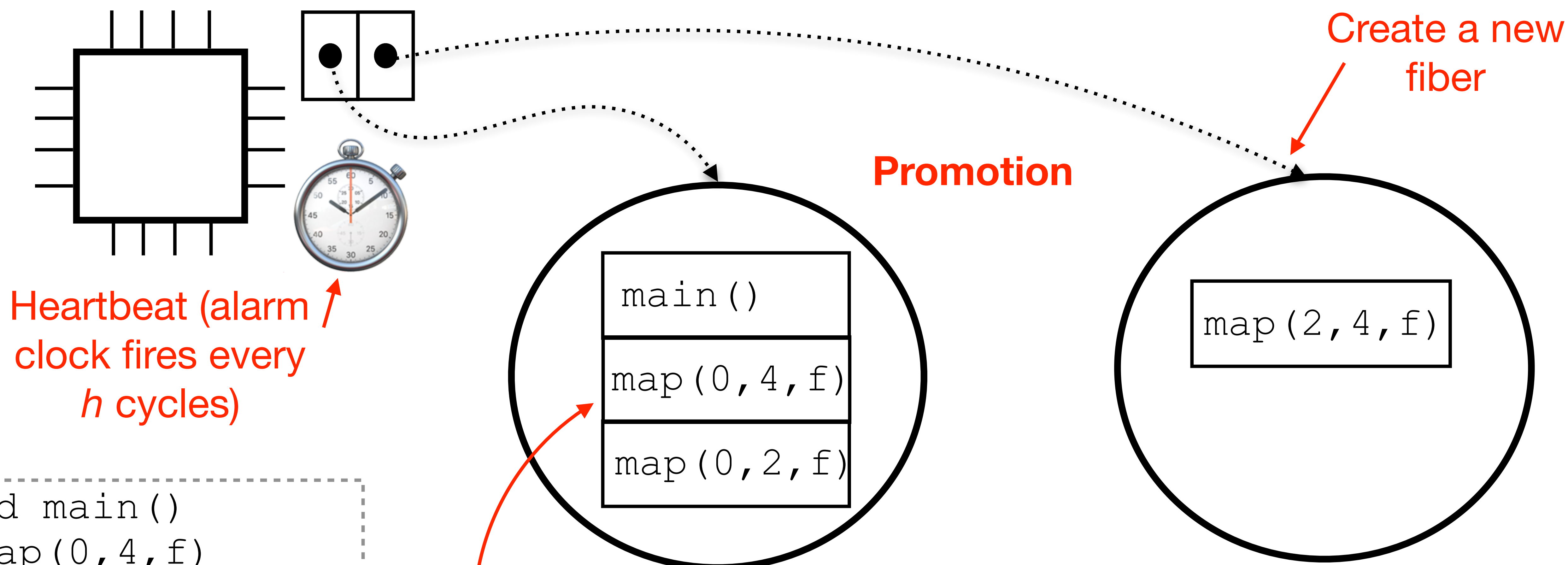

```
void main()
  map(0,4,f)
  return
```

```
void map(lo, hi, f)
  if lo <= hi
    return
  else if lo + 1 == hi
    f(lo)
    return
  int mid = (lo + hi) / 2
  spawn map(lo, mid, f)
        map(mid, hi, f)
  sync
```

Snapshot of the call stack, just after second recursive call to `map`.

The stack grows down.

How heartbeat scheduling works



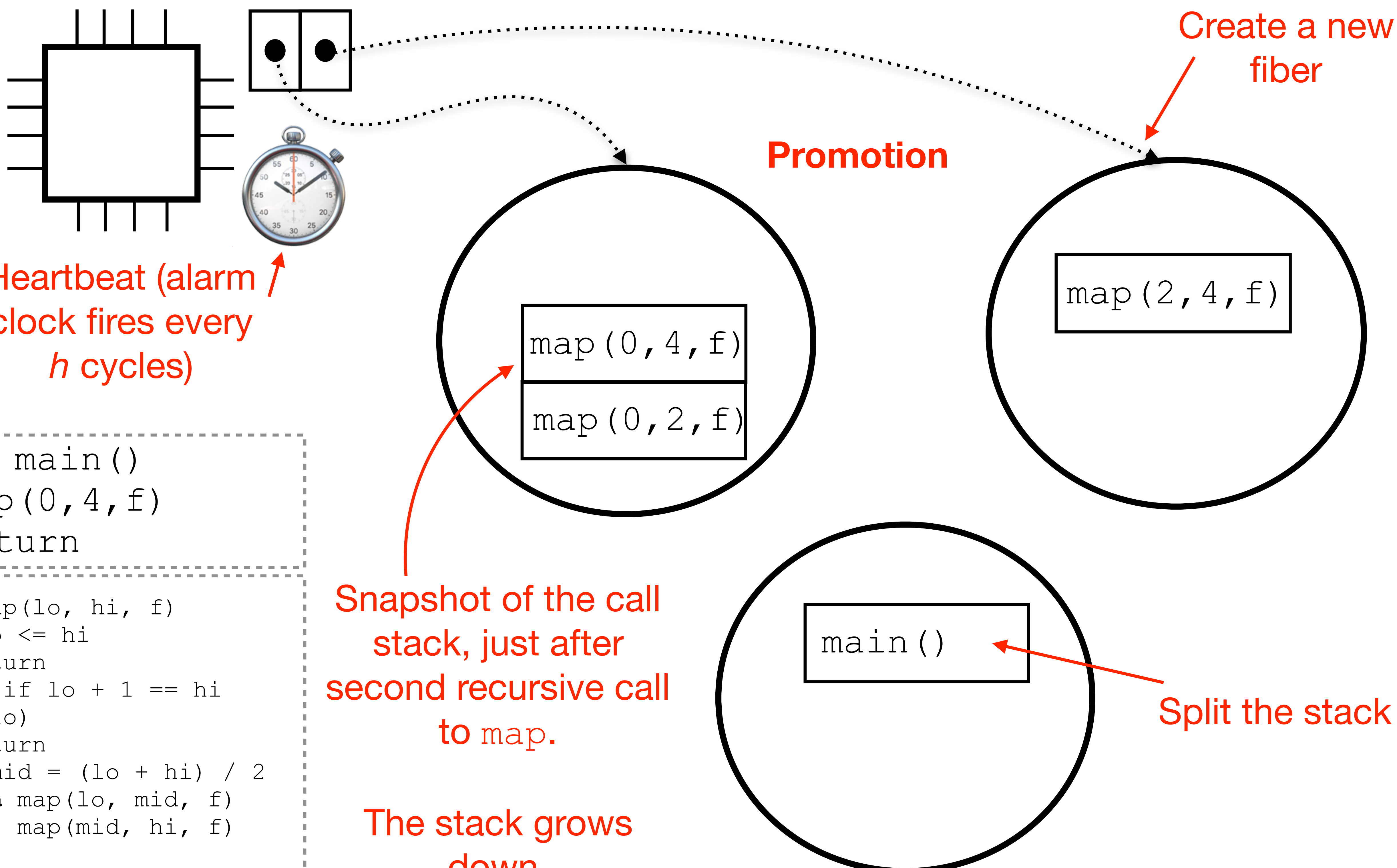

```
void main()
  map(0,4,f)
  return
```

```
void map(lo, hi, f)
  if lo <= hi
    return
  else if lo + 1 == hi
    f(lo)
    return
  int mid = (lo + hi) / 2
  spawn map(lo, mid, f)
        map(mid, hi, f)
  sync
```

Snapshot of the call stack, just after second recursive call to `map`.

The stack grows down.

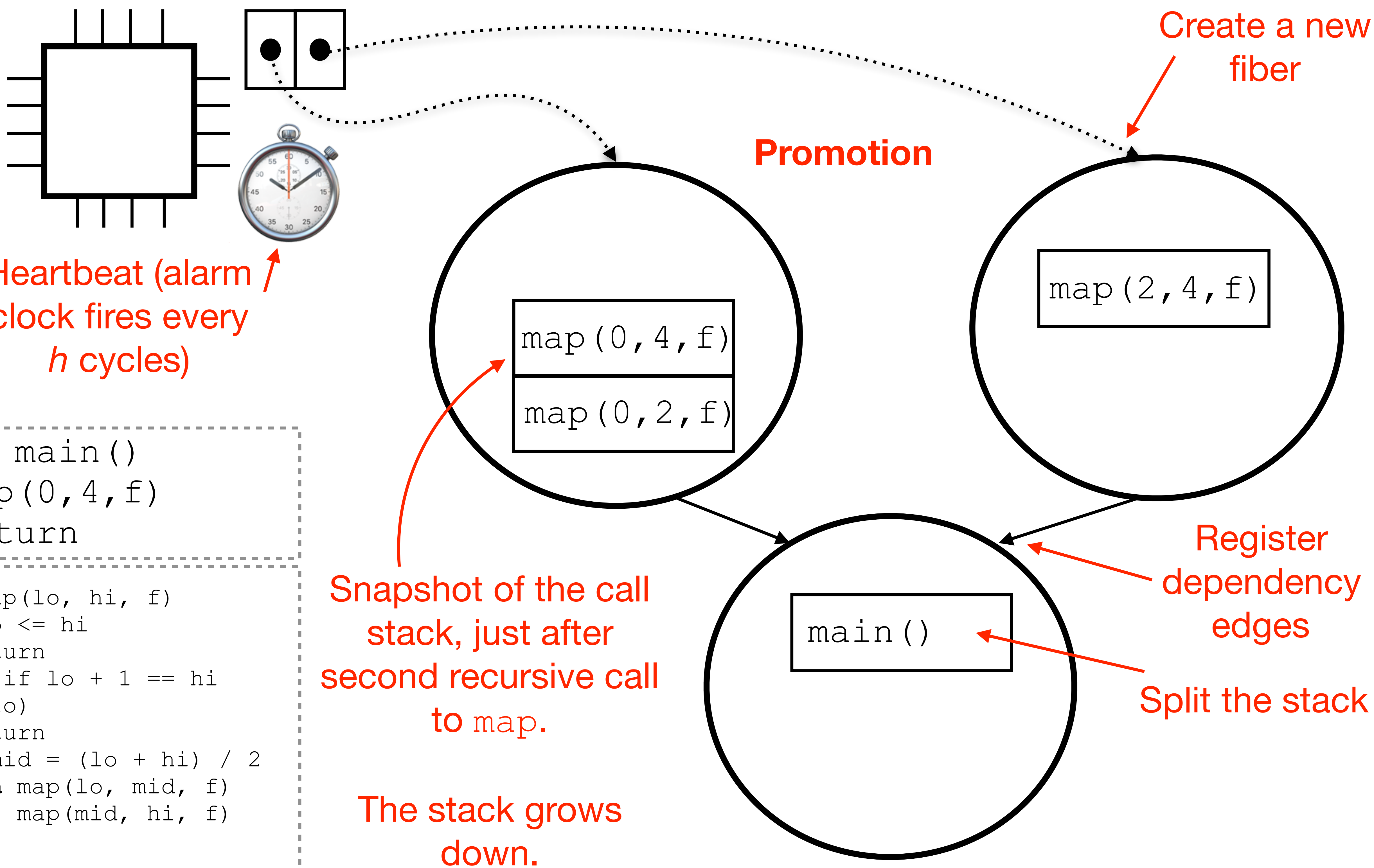
How heartbeat scheduling works




```
void main()
  map(0,4,f)
  return
```

```
void map(lo, hi, f)
  if lo <= hi
    return
  else if lo + 1 == hi
    f(lo)
    return
  int mid = (lo + hi) / 2
  spawn map(lo, mid, f)
        map(mid, hi, f)
  sync
```

How heartbeat scheduling works



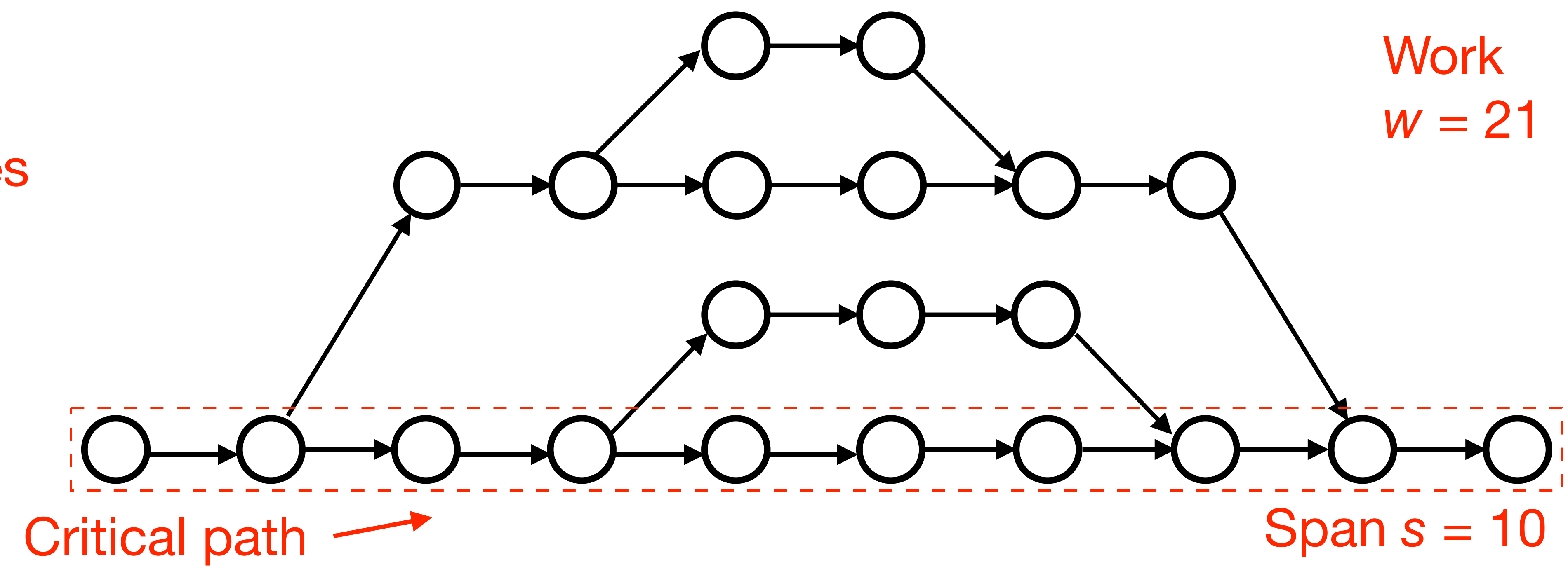

```
void main()
  map(0,4,f)
  return
```

```
void map(lo, hi, f)
  if lo <= hi
    return
  else if lo + 1 == hi
    f(lo)
    return
  int mid = (lo + hi) / 2
  spawn map(lo, mid, f)
        map(mid, hi, f)
  sync
```

Cost model and time bound

Work
w = total # of vertices

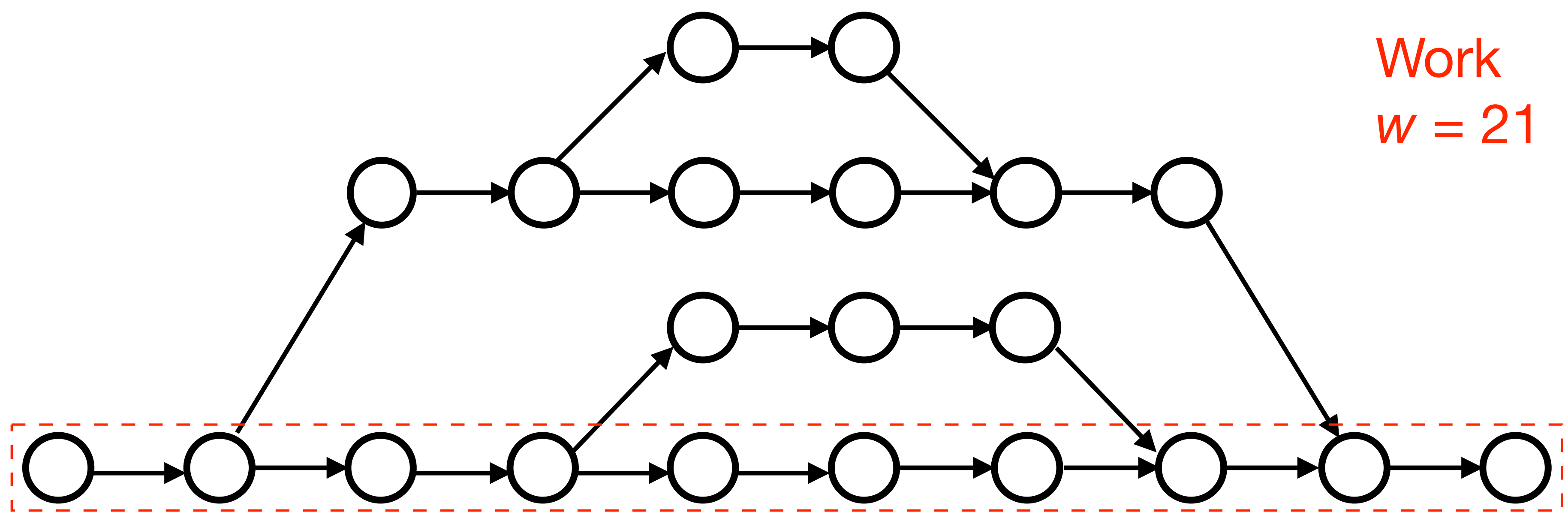
Span
s = length of critical path



Cost model and time bound

Work
w = total # of vertices

Span
s = length of critical path



Work
$w = 21$

Critical path

Span $s = 10$

Work-stealing bound:
(Blumofe & Leiserson)

For any fork-join program:

$$E[t_p] \le w/p + O(s)$$

Expected time to execute on p cores

The bound accounts for the cost of load balancing fibers, but assigns to each scheduling operation a unit cost.

Time bound for heartbeat scheduling

Definitions:

w Work (total # vertices)

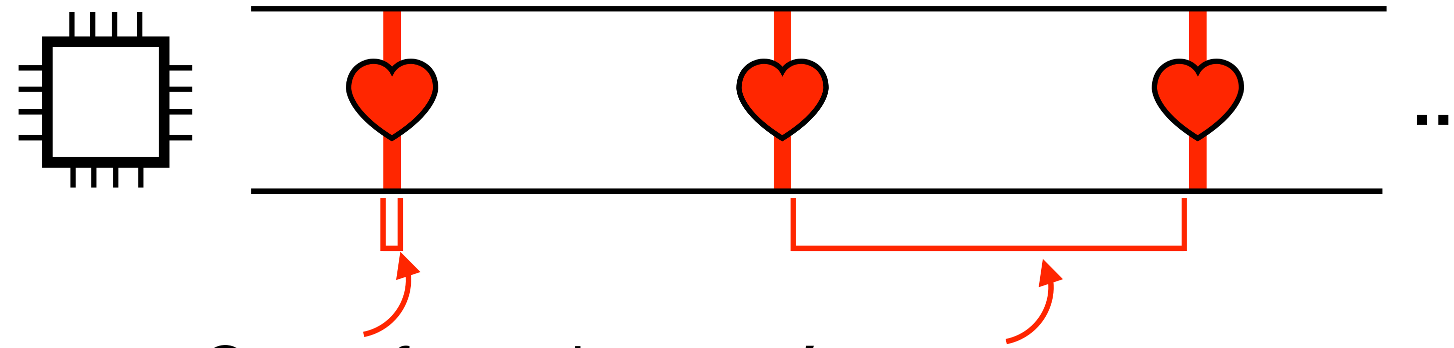
s Span (critical-path length)

t_p Running time of the program on *p* cores

Work stealing: $E[t_p] \leq w/p +$ $O(s)$

Time bound for heartbeat scheduling

Definitions:



w Work (total # vertices)

s Span (critical-path length)

t_p Running time of the program on p cores

τ Cost of creating a fiber

h Heartbeat duration

(Per fiber overhead = τ/h.)

Work stealing: $E[t_p] \leq w/p + \qquad O(s)$

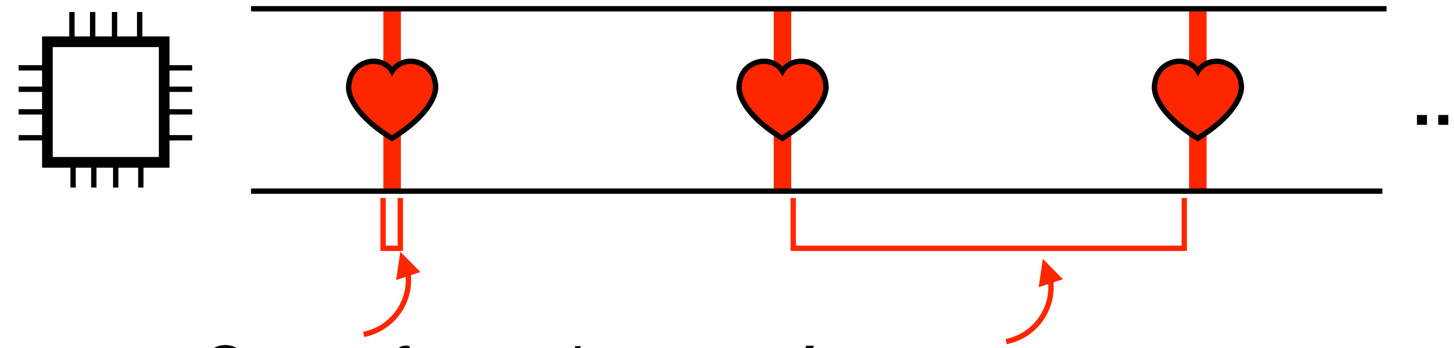
Time bound for heartbeat scheduling

Definitions:

w Work (total # vertices)

s Span (critical-path length)

t_p Running time of the program on p cores



τ Cost of creating a fiber

h Heartbeat duration

(Per fiber overhead = τ/h.)

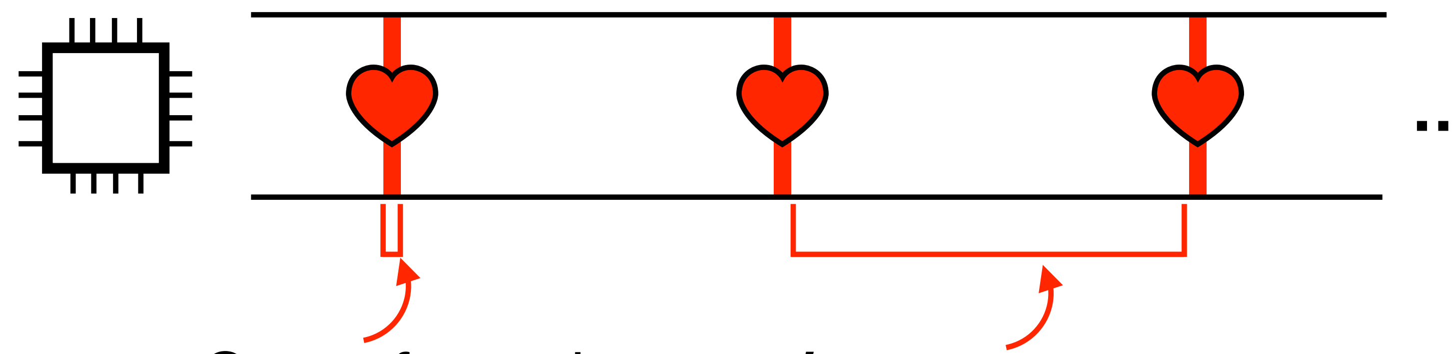
Work stealing: $E[t_p] \leq w/p +$ $O(s)$

$h = k\tau$

We can pick h to be a multiple k of τ.

Time bound for heartbeat scheduling

Definitions:



w Work (total # vertices)

s Span (critical-path length)

t_p Running time of the program on p cores

τ Cost of creating a fiber

h Heartbeat duration

(Per fiber overhead = τ/h.)

Work stealing: $E[t_p] \leq w/p + O(s)$

Work stealing with heartbeat, accounting for sched. overheads: $E[t_p] \leq w/p + (1/k * w/p) + O(k * s)$

1. Bounded increase in overheads (e.g., 5% of work, if $k = 20$)

2. Bounded increase in the span

$h = k\tau$

We can pick h to be a multiple k of τ.

Prototype implementation

Heartbeat mechanism

Need to wake up and try to promote ≈ 20-50μs.

The heartbeat can be realized by software polling or hardware interrupts.

Native support for parallel loops

Should avoid introducing a new stack frame for each parallel loop invocation.

Our solution: extend frame representation to expose *loop descriptor*.

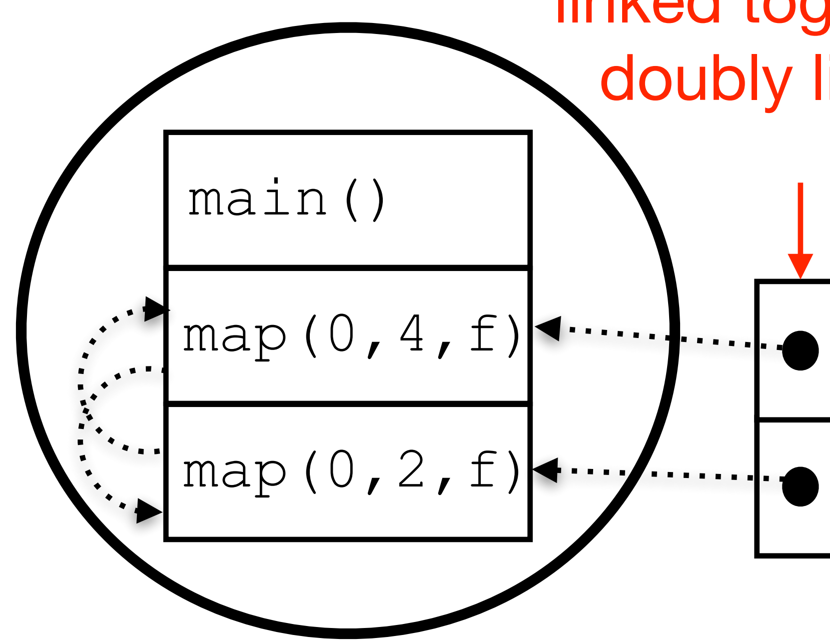
Cactus stack

(+ heartbeat acceleration structure)

For calling convention: we use the classic cactus-stack representation.

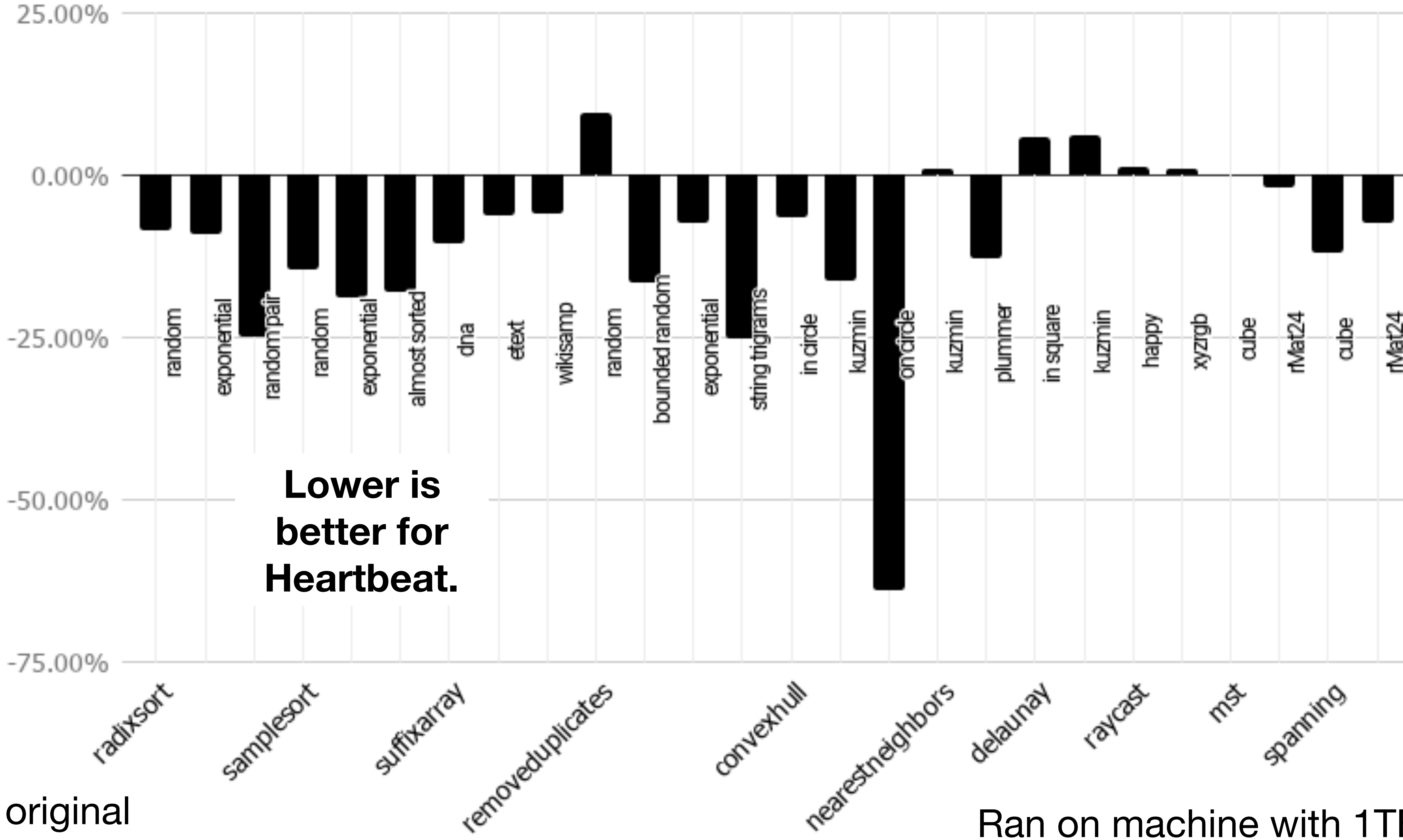
Bookkeeping needed because we need O(1) access to top-most promotable frame.

Promotable frames are linked together by a doubly linked list



Experimental results

% difference in execution time between Heartbeat and baseline



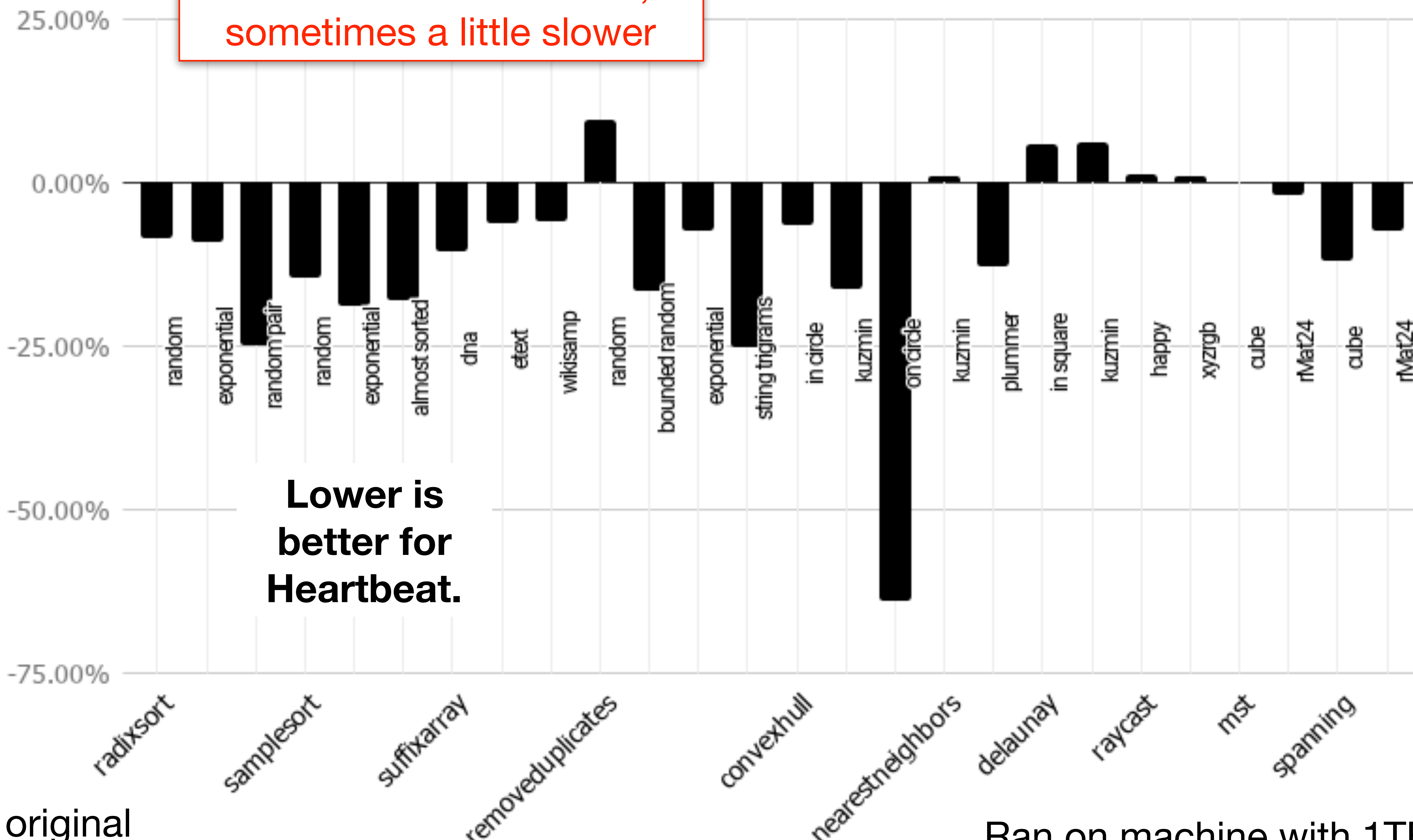
Baseline = original authors' Cilk code

Ran on machine with 1TB RAM, using all 40 cores

Experimental results

Heartbeat is almost always faster than the baseline, sometimes a little slower

% difference in execution time between Heartbeat and baseline



Lower is better for Heartbeat.

Baseline = original authors' Cilk code

Ran on machine with 1TB RAM, using all 40 cores

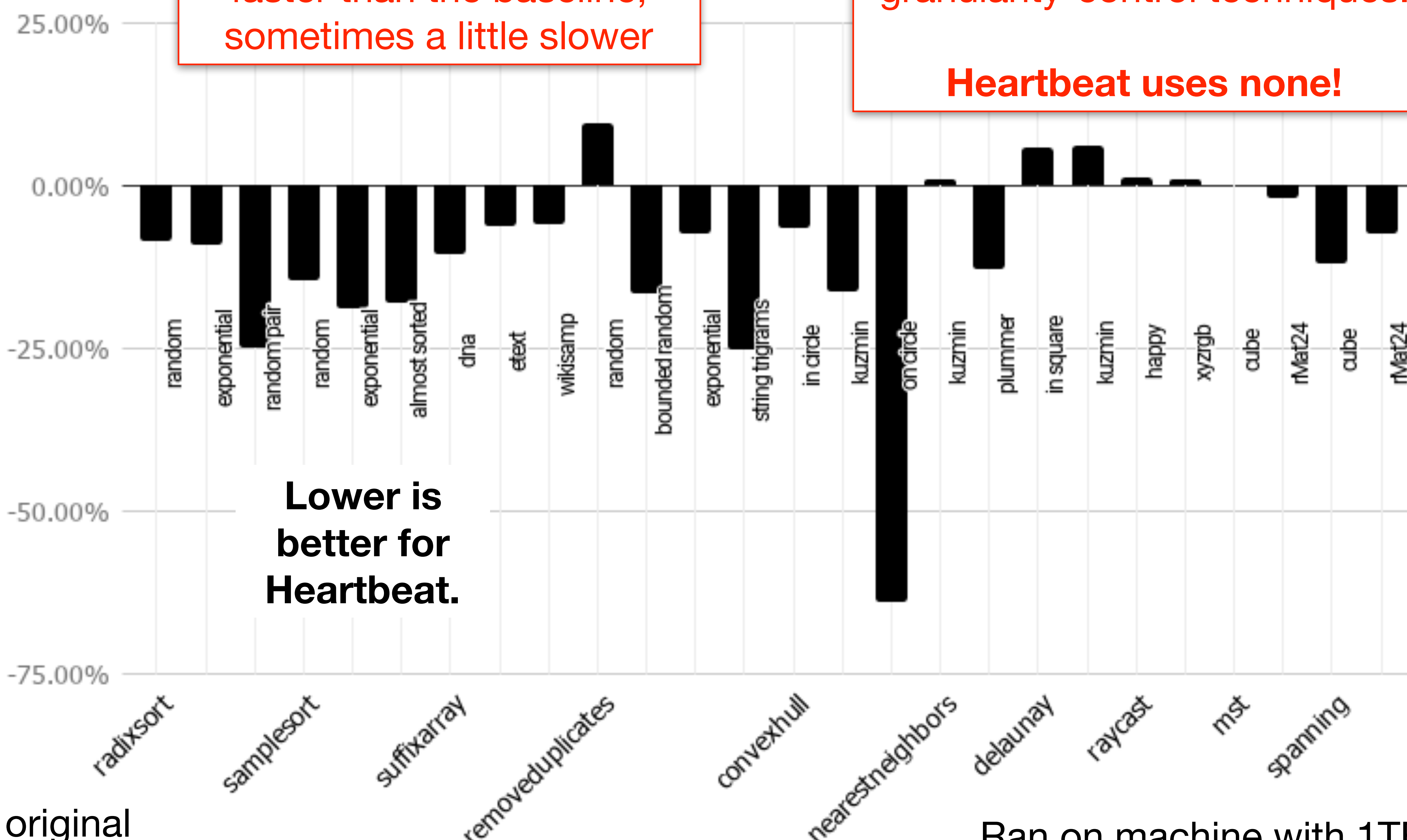
Experimental results

Heartbeat is almost always faster than the baseline, sometimes a little slower

Code simplification: the baseline codes use several manual granularity-control techniques.

Heartbeat uses none!

% difference in execution time between Heartbeat and baseline



Baseline = original authors' Cilk code

Ran on machine with 1TB RAM, using all 40 cores

Related work

Formal bounds for scheduling fork join

Brent ’74, Arora et al ’98, Blumofe & Leiserson ’99, Agarwal et al ’07, Acar et al ‘11

Heartbeat is the first to show analytical bounds on scheduling overheads for all fork join programs.

Lazy-scheduling methods

Mohr et al ’91, Feeley ’93, Goldstein et al ’96, Frigo et al ’98, Imam et al ’14, Tzannes et al ‘14

Heartbeat is the first in this class of approaches to have a state-of-the-art implementation and be backed by end-to-end bounds.

Prediction-based methods

Weening ’89, Pehoushek et al ’90, Lopez et al ’96, Duran et al ’08, Acar et al ’16, Iwasaki et al ’16, Shintaro et al ‘16

Heartbeat offers similar but stronger guarantees than Oracle-Guided Granularity Control, and delivers state-of-the-art in performance.

Conclusion

- Heartbeat scheduling supports really lightweight nested parallelism:
  - It simplifies code: no need for manual granularity control.
  - It is protected by formal bounds from adversary programs.
  - It can, on ten benchmarks, achieve comparable or better performance to Cilk, a carefully engineered implementation.
- Future work:
  - Optimized compiler implementation
  - Generalizing beyond fork join (e.g., futures)
- **Thanks for you attention!**