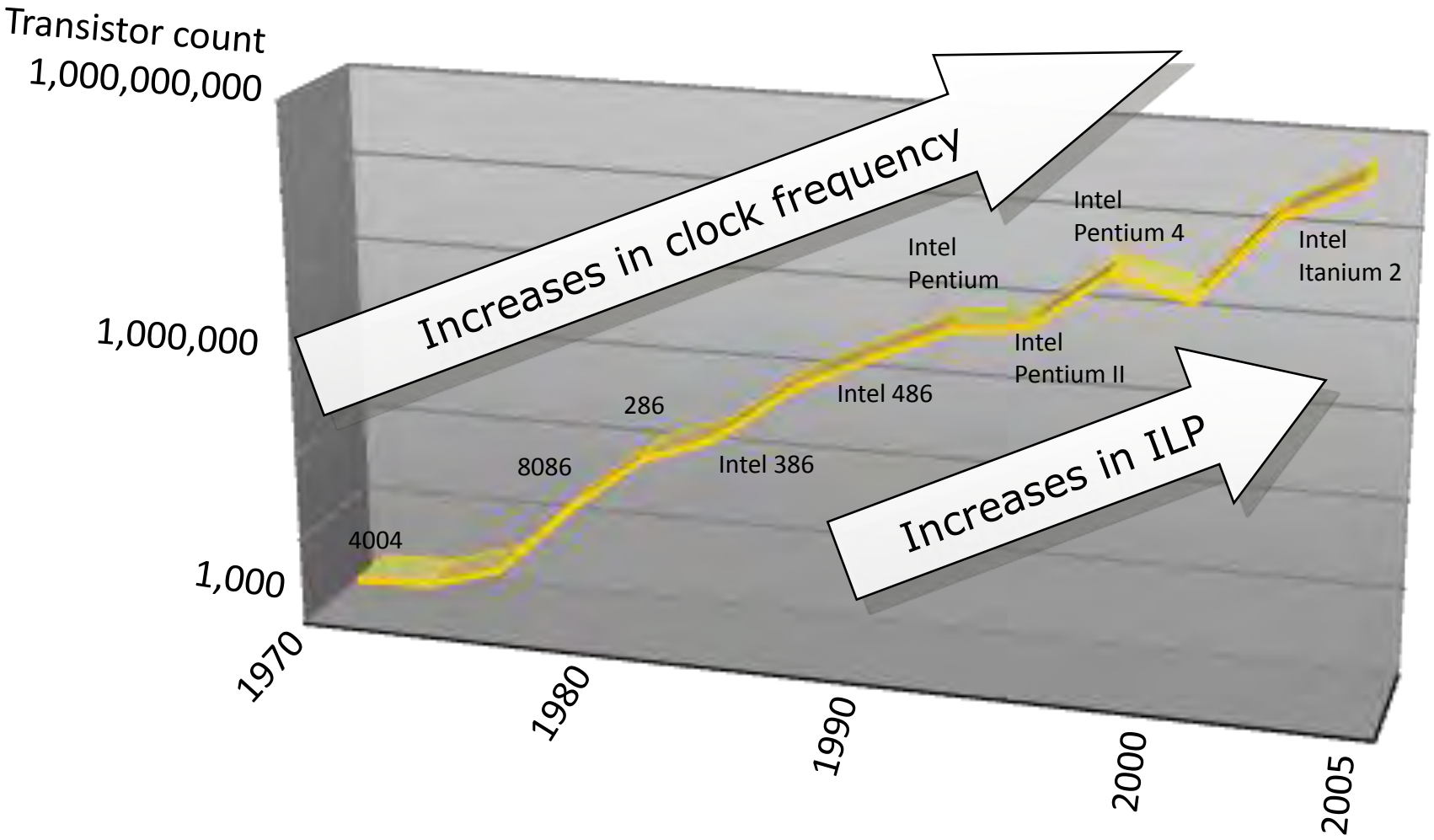

# Multi-core programming with automatic parallelisation

Tim Harris and Satnam Singh
Microsoft Research Cambridge

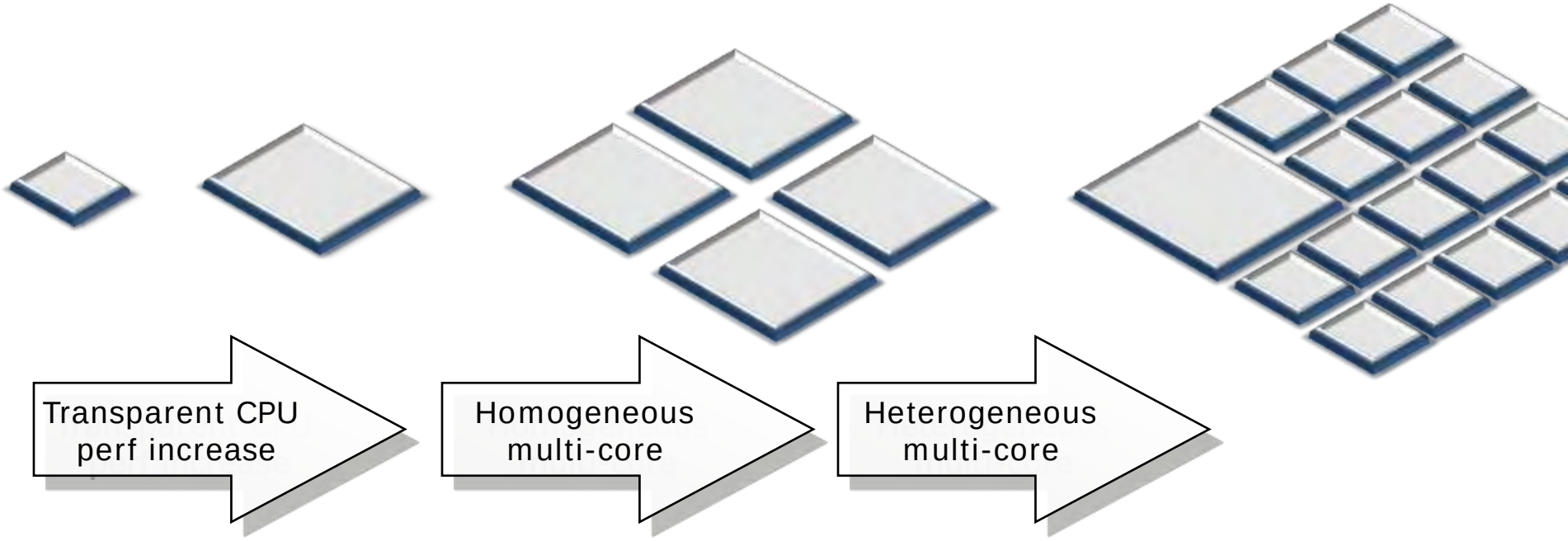
# Moore's law; the free lunch



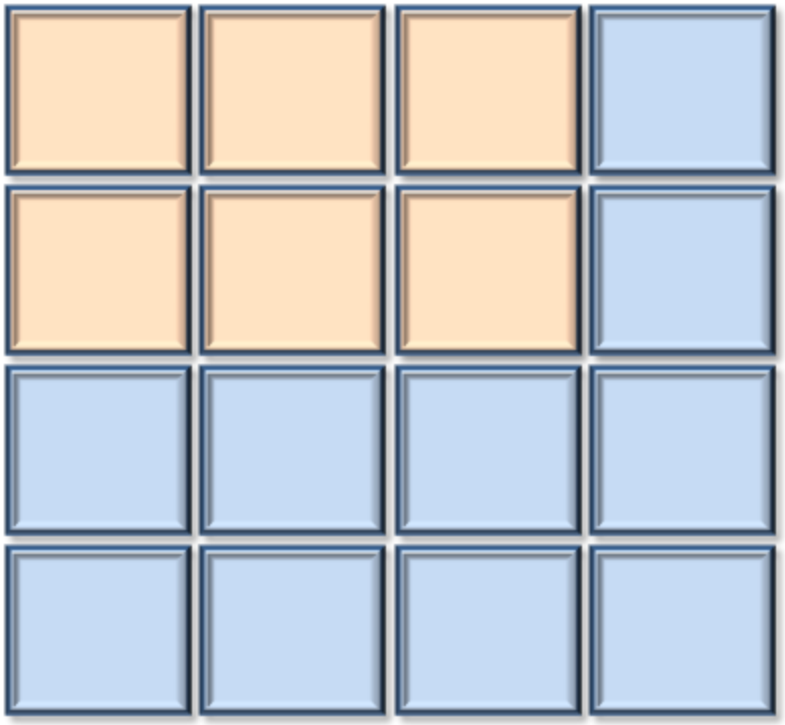

Source: table from http://www.intel.com/technology/mooreslaw/index.htm

Microsoft Research

# The free lunch is over

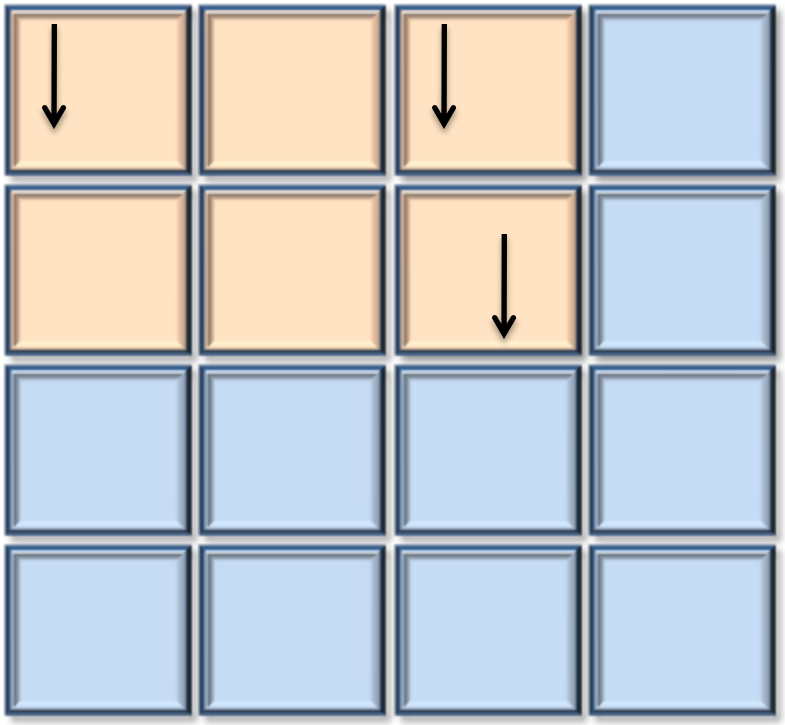


# Using multi-cores: space scheduling



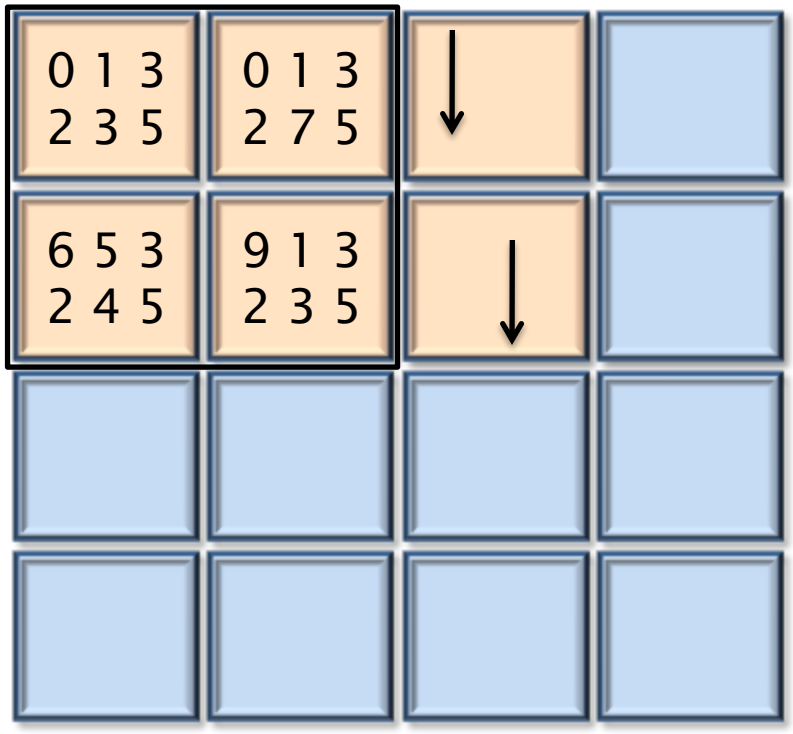
- Partition resources between concurrent apps or VMs
- Machine consolidation
- Preserve responsiveness
- VMs for stronger isolation

# Using multi-cores: control parallelism



- Find control-parallelism within an app
- Multiple threads in a shared address space
- Handle different clients
- GC, indexing, etc concurrent with application work

# Using multi-cores: data parallelism




- Exploit data parallelism in large computations
- OpenMP
- Nested data-parallelism

# What about existing client code?

- Centred around interactions with a single user…
  - No easy parallelism through multiple clients
- Not written with parallelism in mind…
  - No explicit threading for performance
  - No calls onto parallel libraries

# Implicit parallelism

```
for (int i = 0; i < 100; i++) {
   MyFunction(i);
}
```

Run each iteration in parallel?

```
F1(F2(x), F3(x))
```

Evaluate F2(x) and F3(x) in parallel?

# The key problems

- Getting the granularity right

```
for (int i = 0; i < 16; i++) {
   x[i]++;
}
```

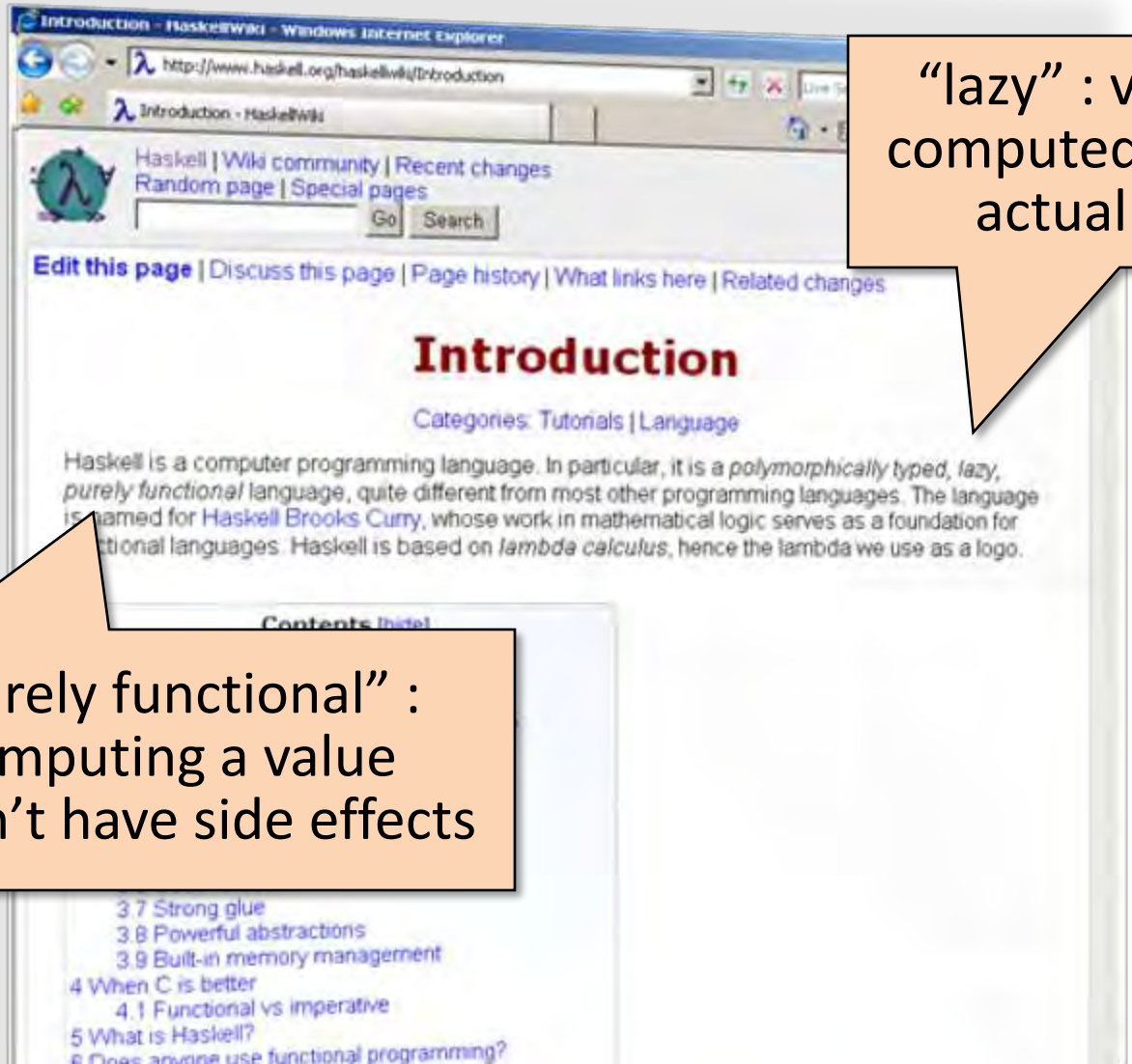
- Keeping it transparent

```
for (int i = 0; i < 16; i++) {
   total += x[i];
}
```


# Overview

- Haskell
- Limit study: is there anything out there?
- Making it more real: what does the granularity look like?
- Actual implementation: can we get a free lunch after all?

# Haskell

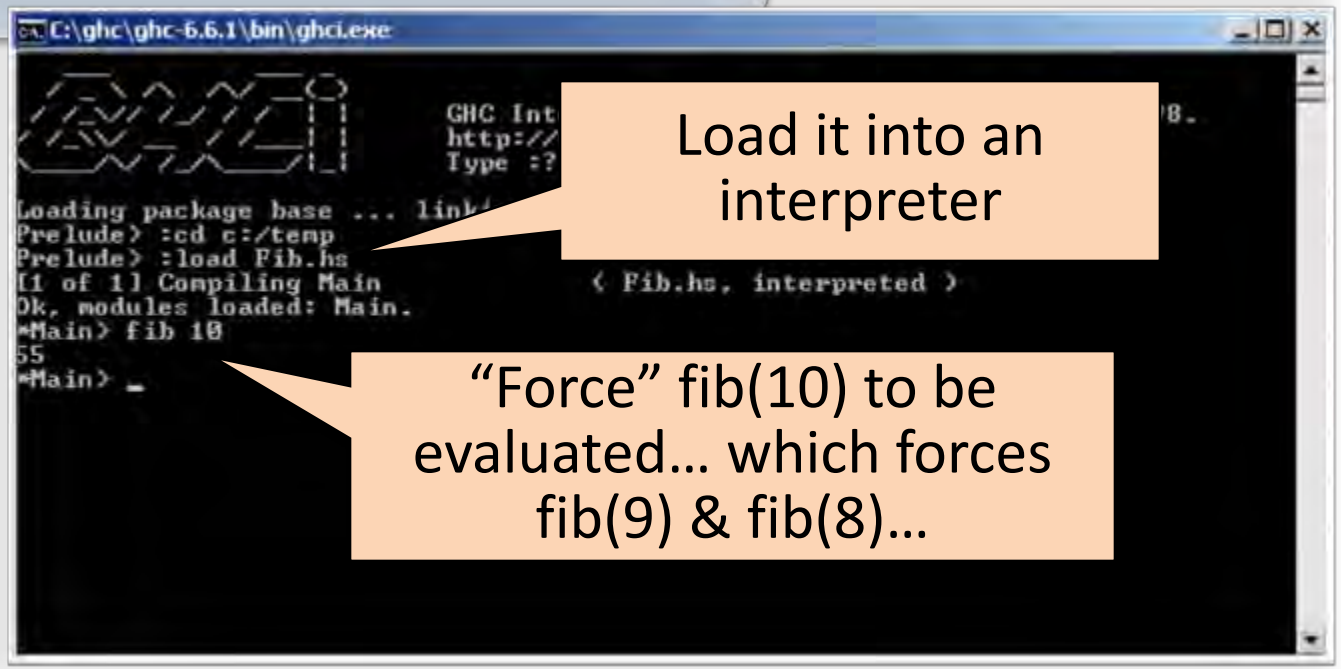



"lazy" : values aren't computed until they're actually needed

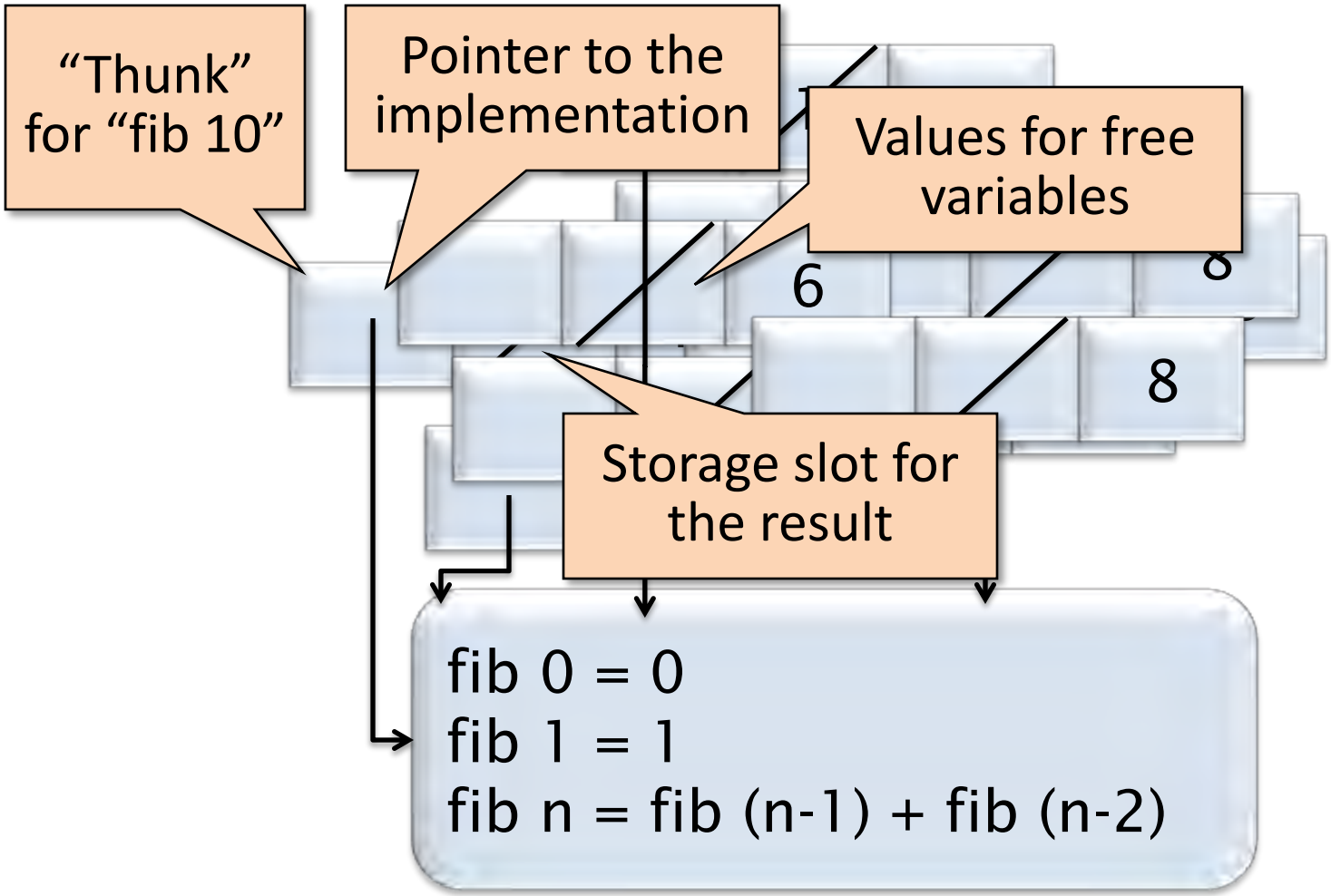
"purely functional" : computing a value doesn't have side effects

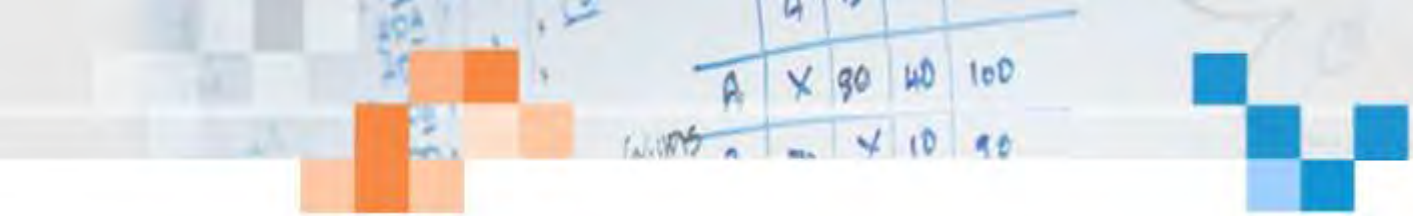
# A toy example: Fibonacci

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```



# Running the toy example



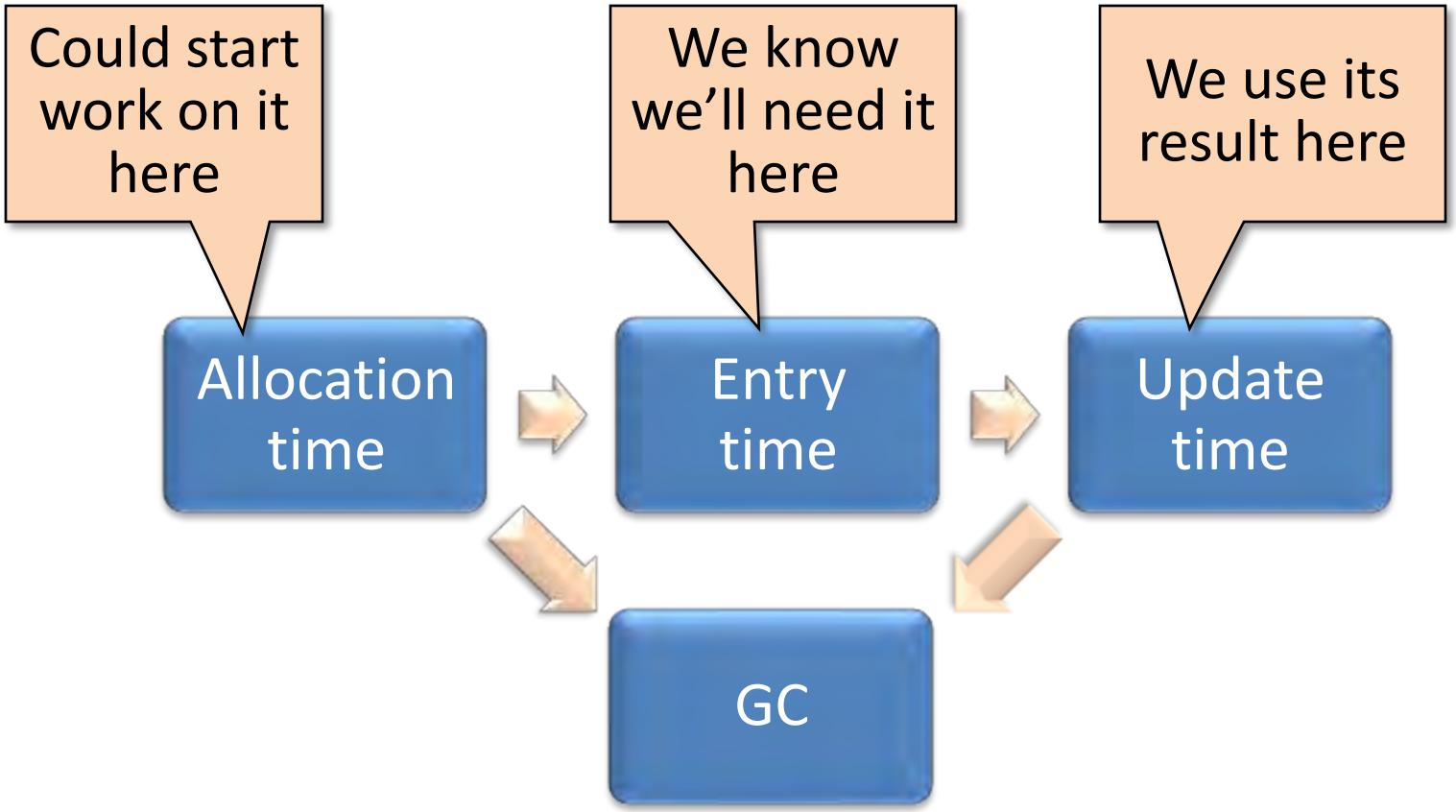


Microsoft Research

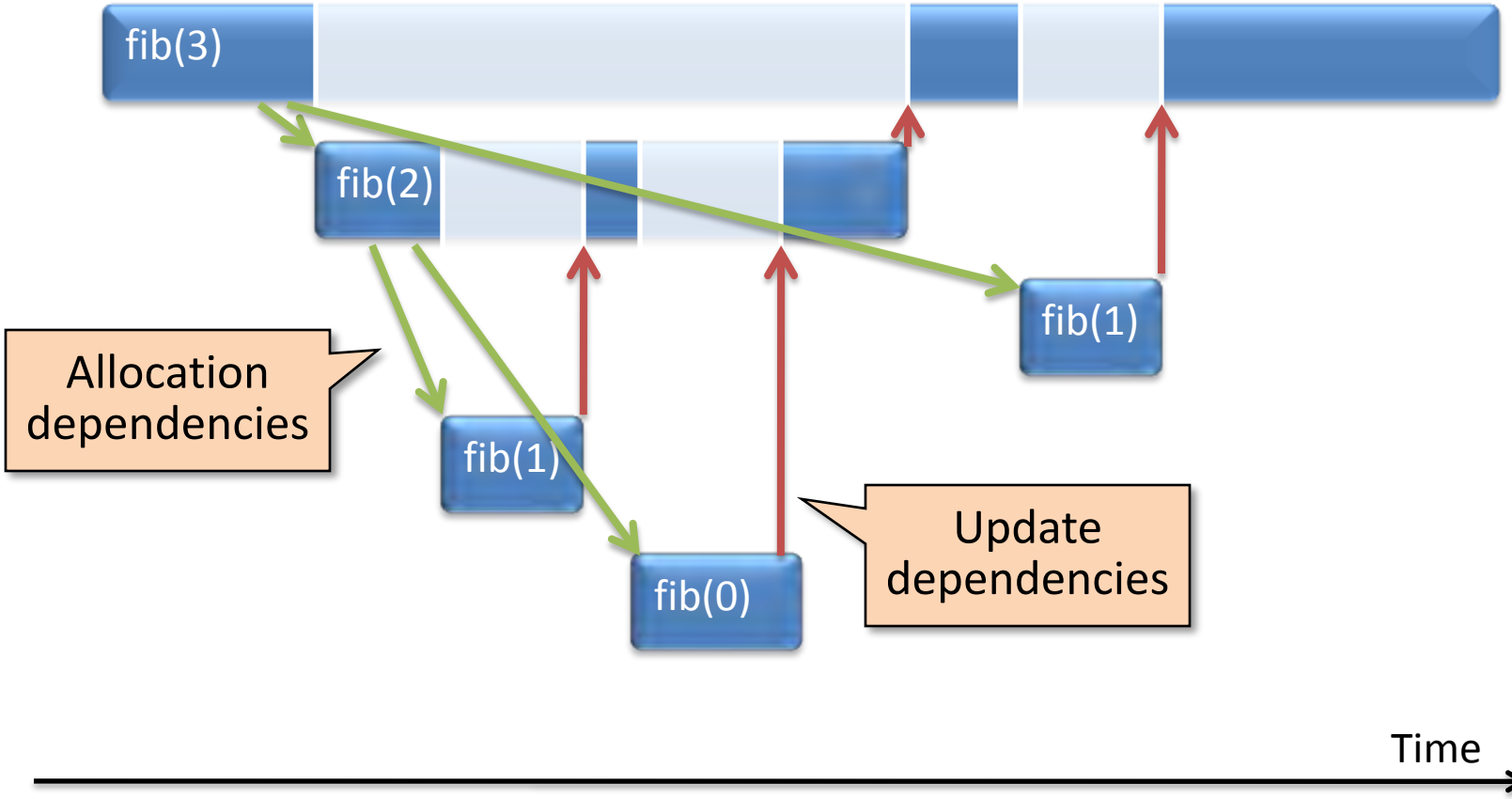
# Overview

- Haskell
- Limit study: is there anything out there?
- Making it more real: what does the granularity look like?
- Actual implementation: can we get a free lunch after all?

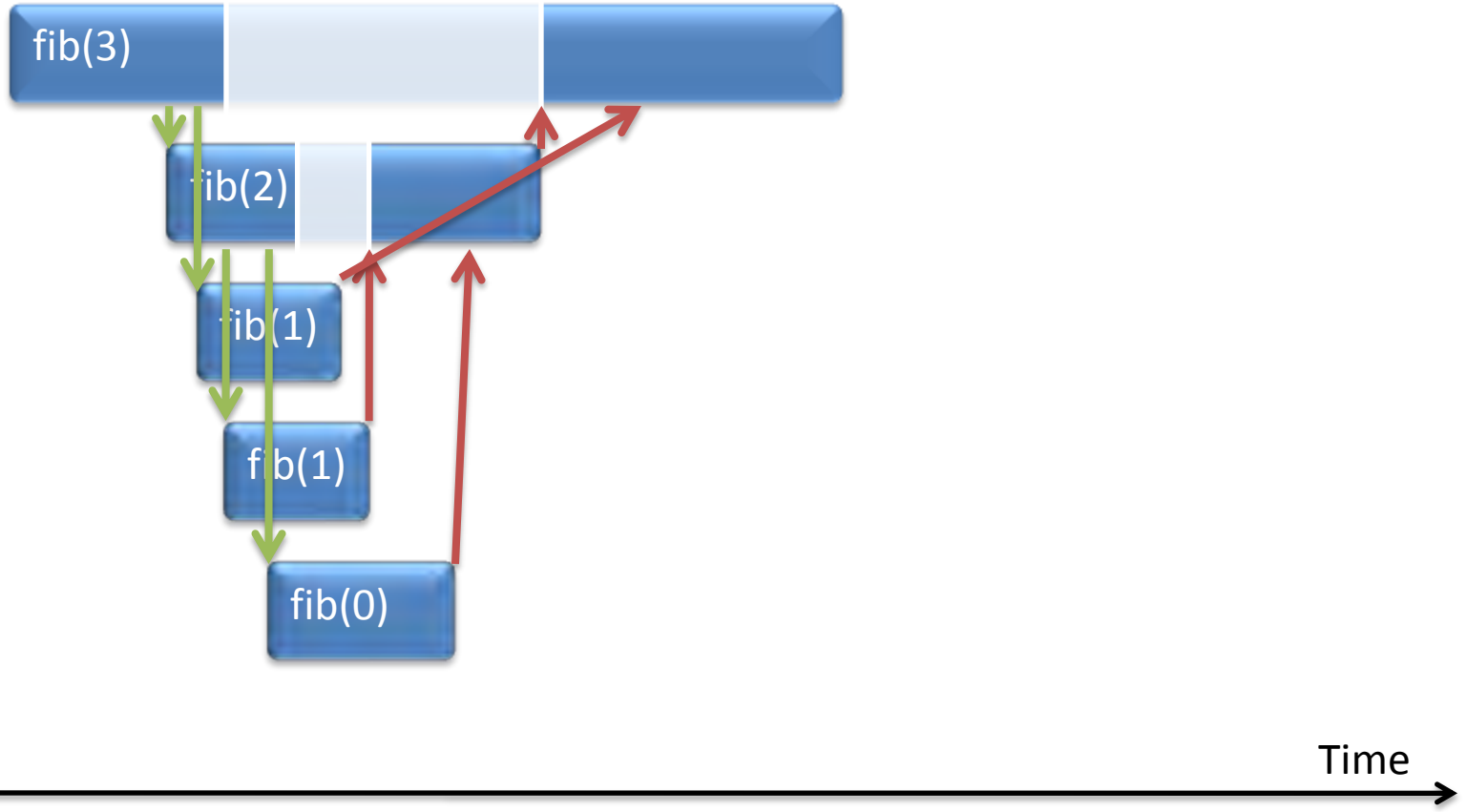
# Thunk lifecycle



# Thunk dependencies

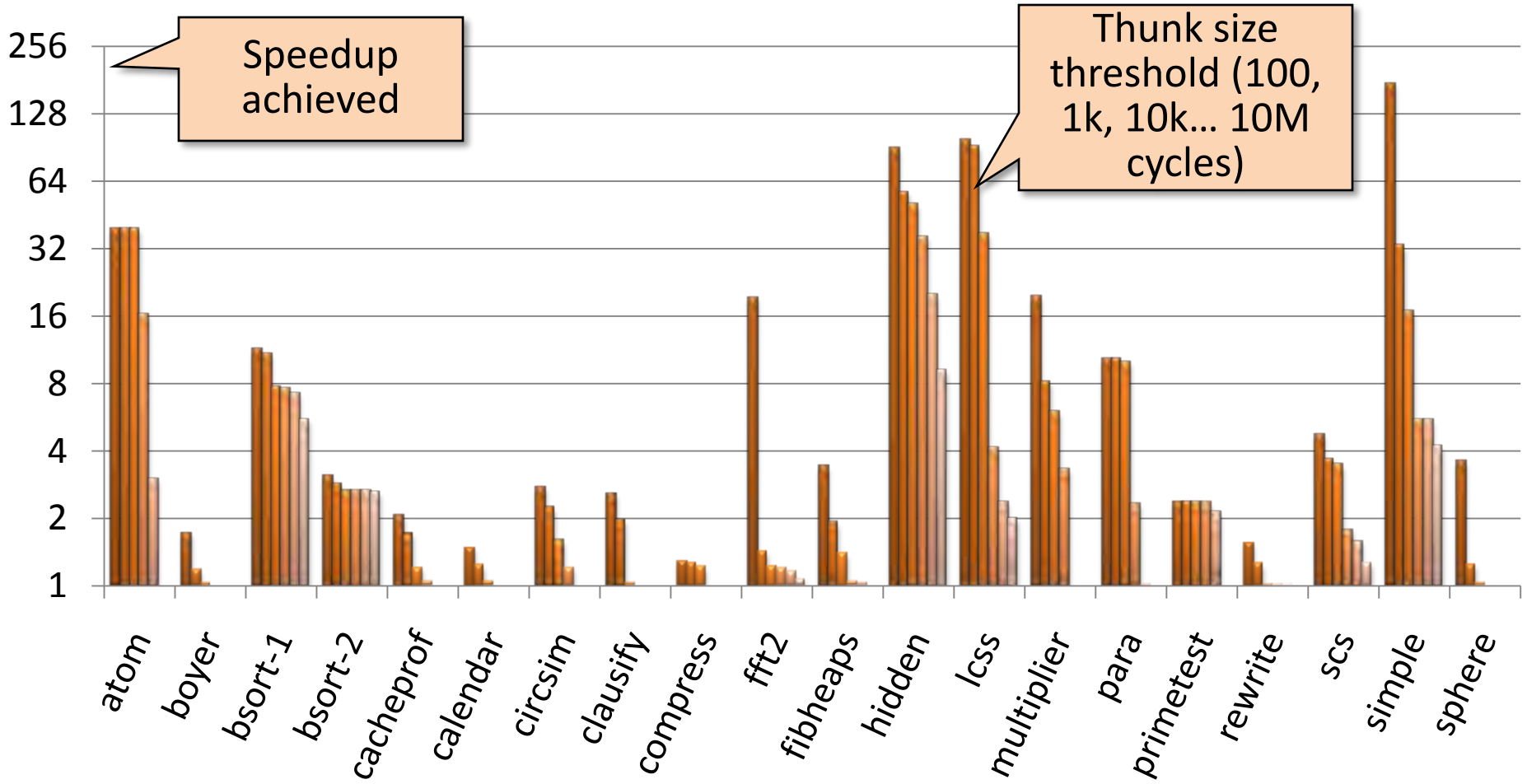


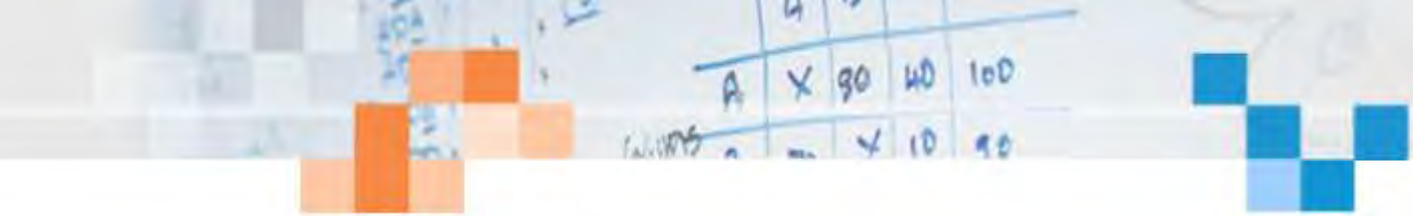
# Potential parallelism



Microsoft Research

# Limit study





Microsoft Research

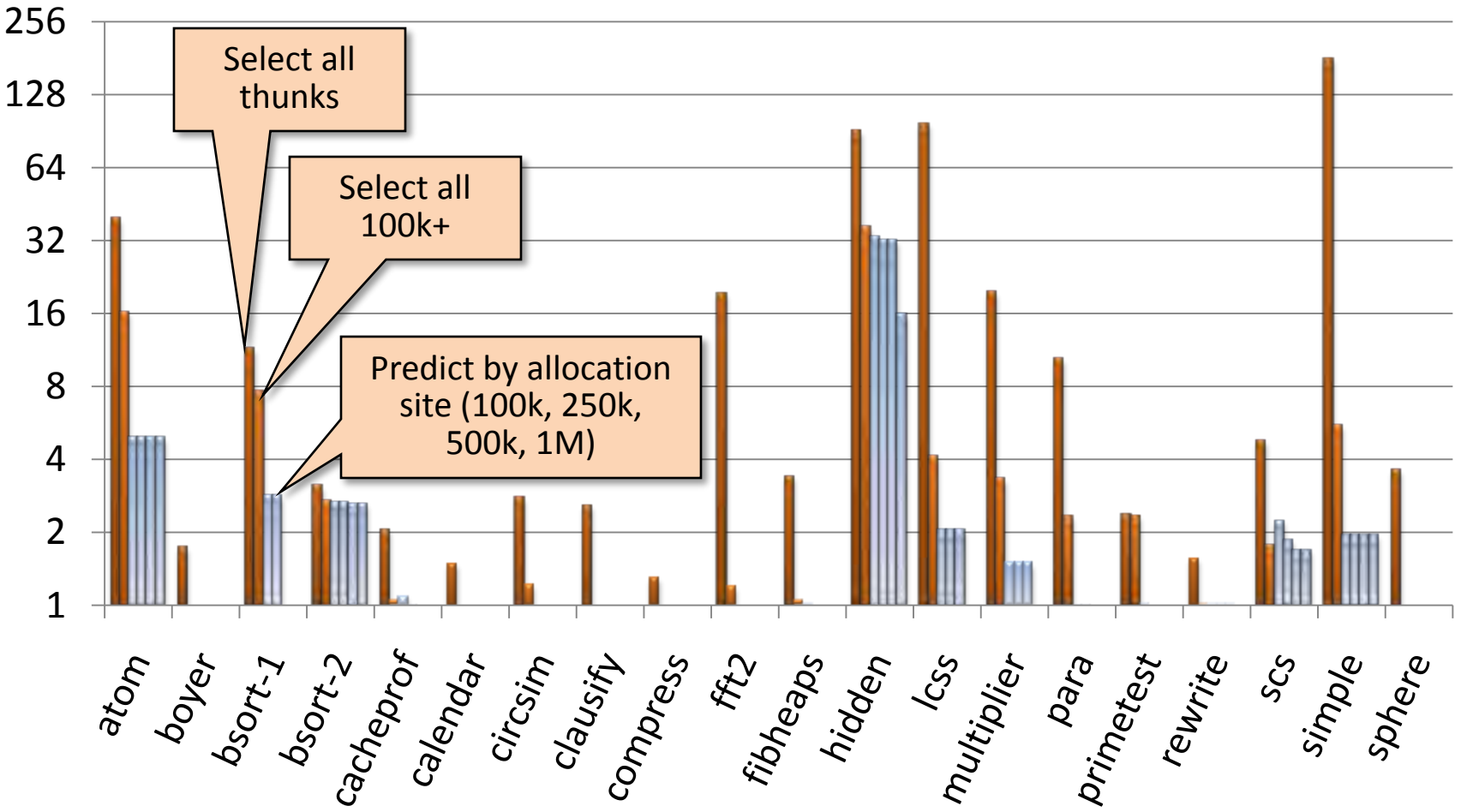
# Overview

- Haskell
- Limit study: is there anything out there?
- Making it more real: what does the granularity look like?
- Actual implementation: can we get a free lunch after all?

# Which thunks to select?

- Select based on thunk allocation site
- Binary spark / no-spark decision
- Threshold for needed/not-needed (3/4)
- Consider total work for allocation site

# Simulation, select by alloc site




# Overview

- Haskell
- Limit study: is there anything out there?
- Making it more real: what does the granularity look like?
- Actual implementation: can we get a free lunch after all?

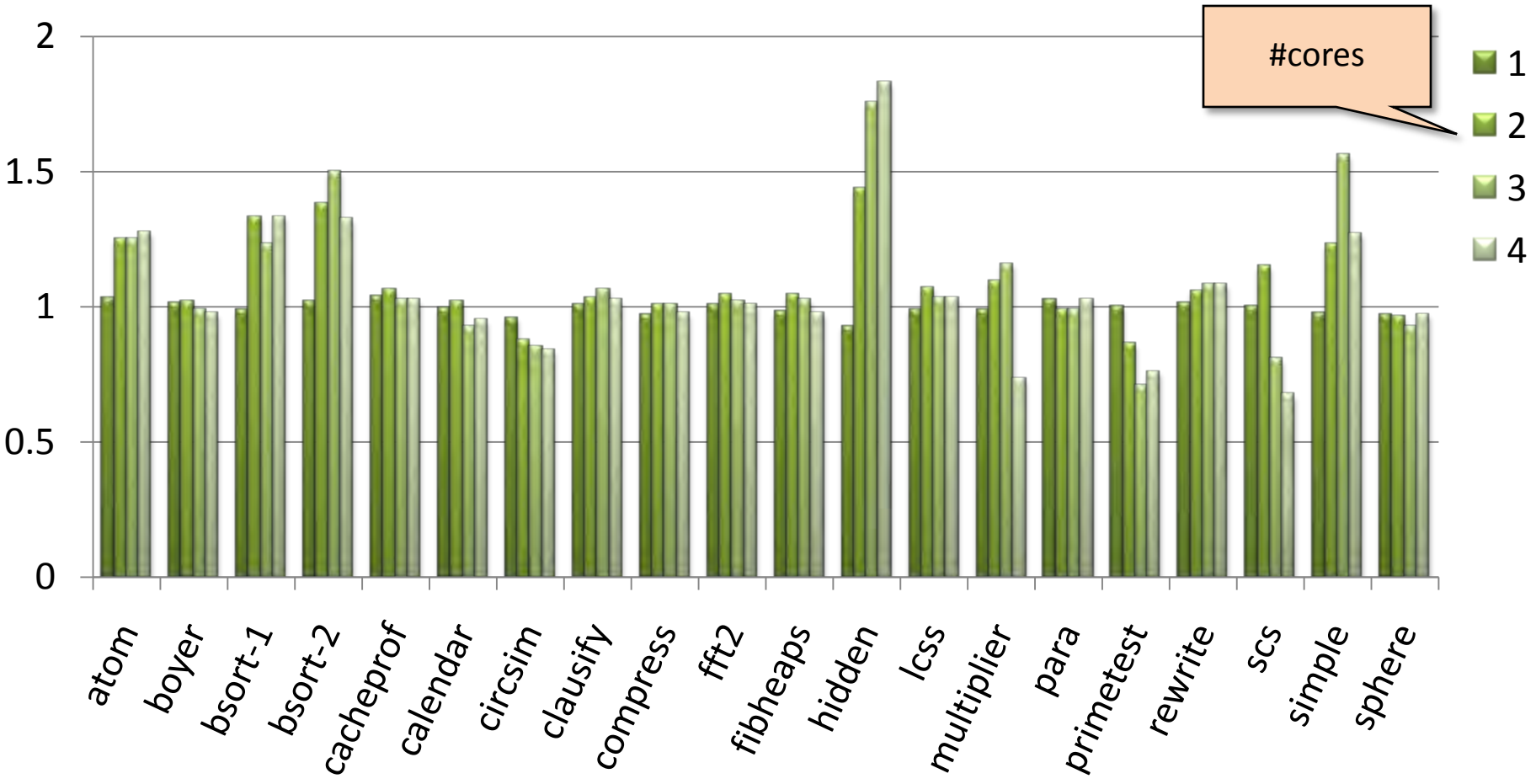
# Avoiding duplicate work

- By default GHC *does not* prevent concurrent evaluation of the same thunk
- But here the penalty is bad
  - We're selecting big thunks
  - Independence of thunk evaluation is compromised
- We add a level of locking to avoid duplication

# Avoiding speculation of unsafe I/O

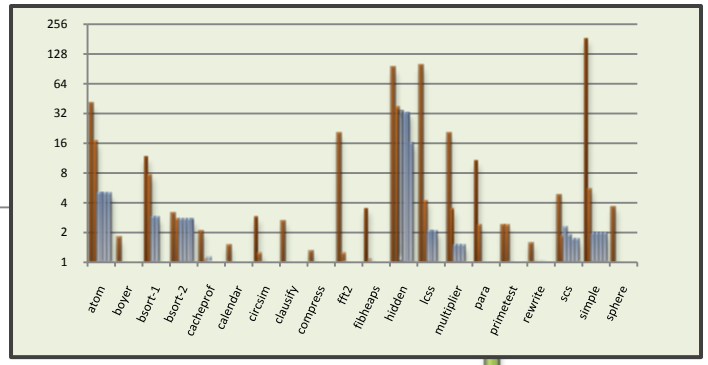
- Dynamically detect attempts to perform unsafe I/O while speculating
- Upon detection suspend speculation of current thunk
- Defined using new I/O action which has no side effect in application thread but suspends under speculation

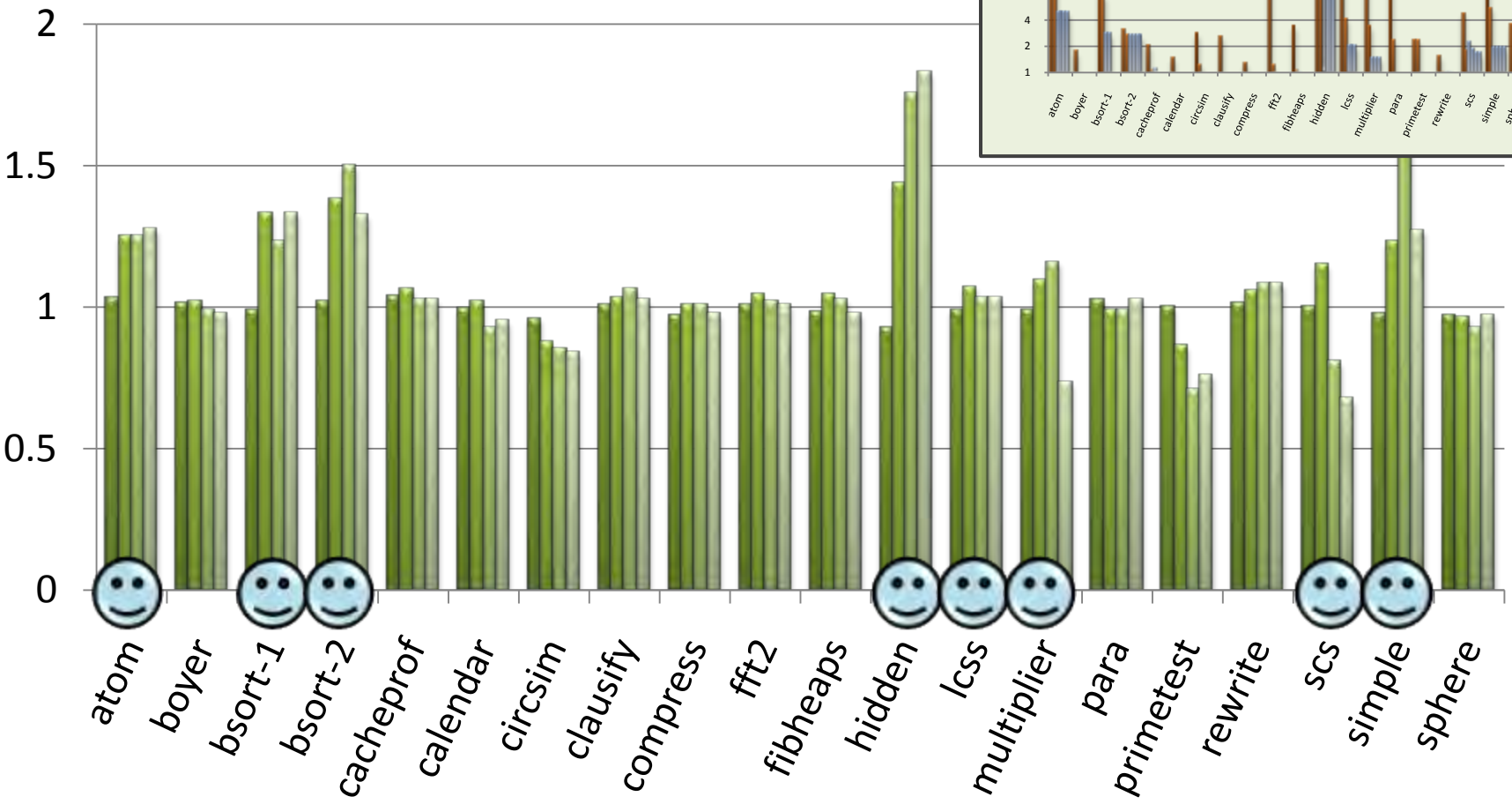
# Measured performance



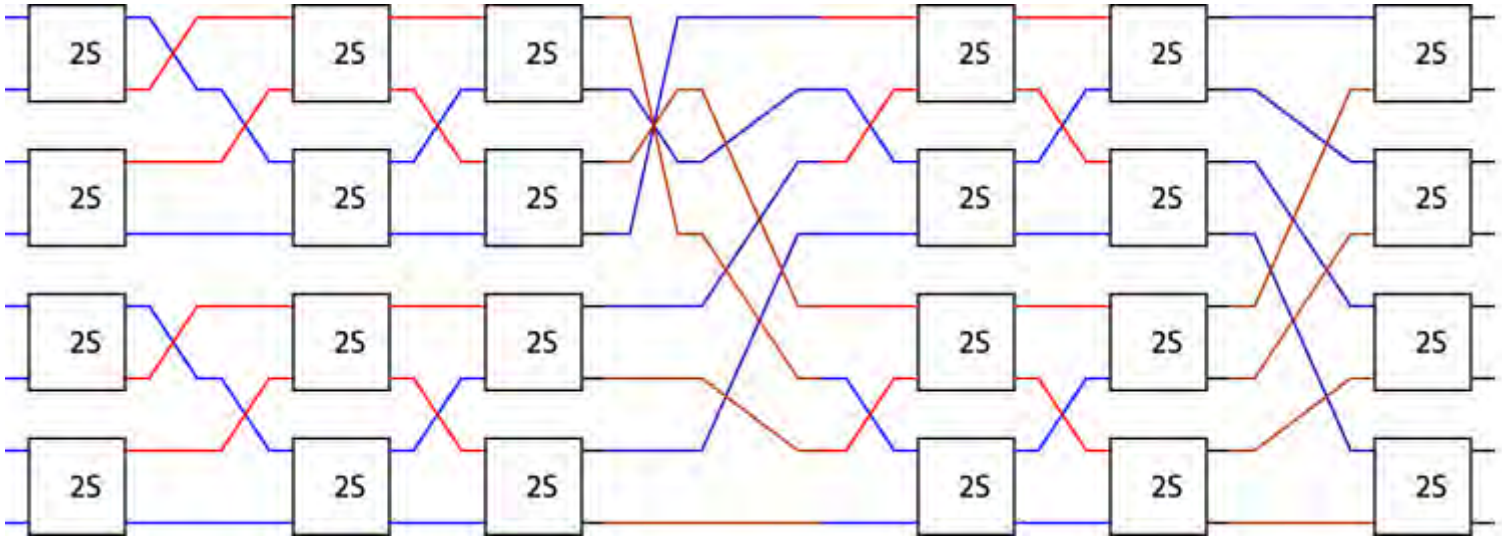
Microsoft Research

# Avoiding slow-downs

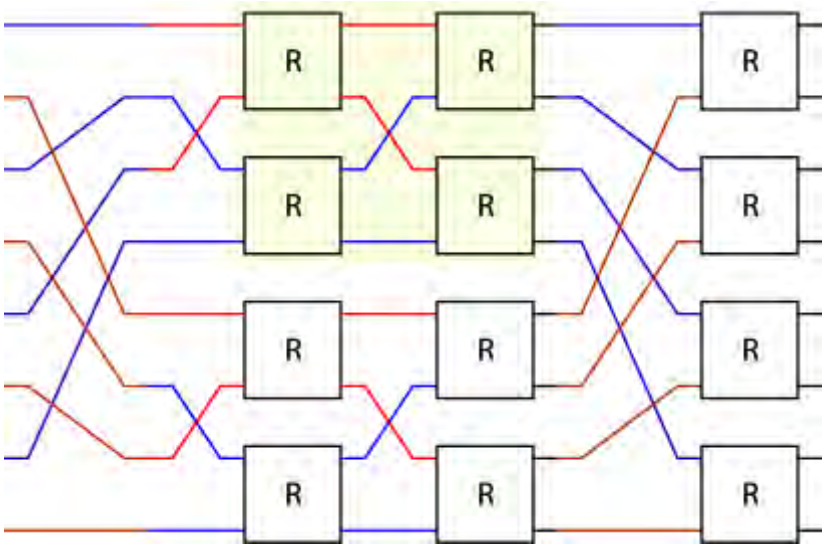


# Batcher's Parallel Sorter



# Parallelism in the sorter




```
two :: ([a] -> [b]) -> [a] -> [b]
two r = halve >-> toBoth r r >-> unhalve
```

# Related work

- Parallel function programming [Hammond 94].
- Speculative evaluation in Multilisp [Osborne 90].
- `par` and `seq` [Roe 91].
- Parallel strategies [Trinder et. al. 98].
- Combining non-strictness with eager evaluation: Id90 and pH [Nikhil and Arvind 01].
- Fine grain concurrency: Monsson [Traub 91].
- Optimistic evaluation [Ennals and Peyton-Jones 03].
- GranSim [Loidl 98].

# Future work

- Can we use a single program execution that avoids the need for a profile-collection phase?
- Impact of adjusting optimization level
- Use of feedback information to guide manual parallelization

# Conclusions

- For selected programs an automatic speedup of 10-80% can be achieved
- Simulator can identify programs that are not amenable to automatic parallelization
- Not a silver bullet but does provide an interesting approach for parallelizing single application across 2, 3 or 4 cores