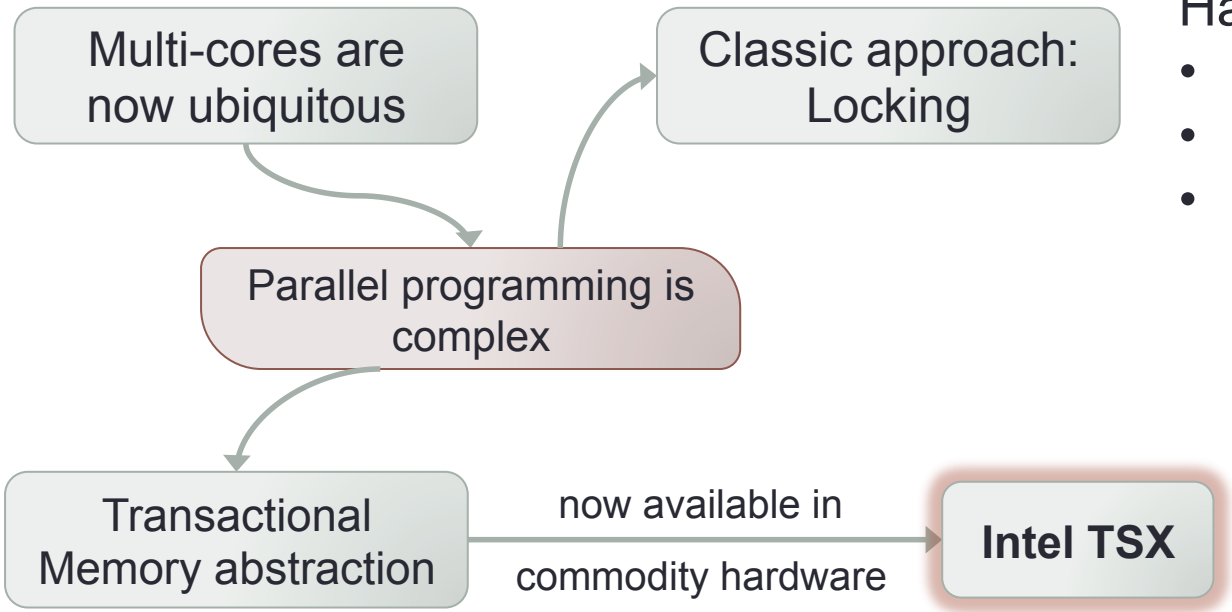
ICAC 2014

# SELF-TUNING INTEL TSX

Nuno Diegues and **Paolo Romano**


# Intel Transactional Synchronization Extensions




Hard to get right:

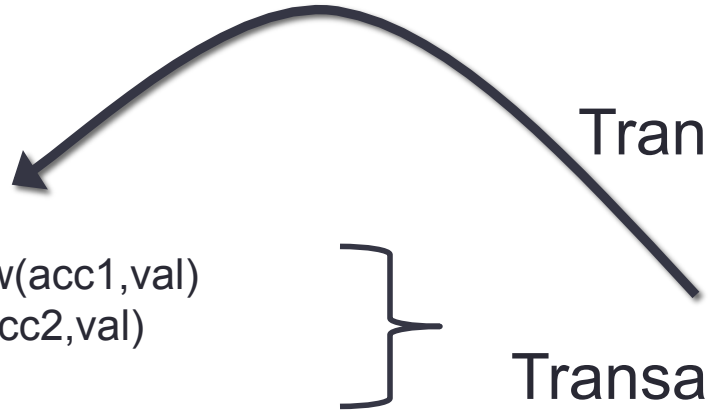
- fine-grained locks
- deadlocks
- correctness

```
atomic {
 withdraw(acc1,val);
 deposit(acc2,val);
}
```

x86 instruction set extension:
xbegin, xend, xabort…

Programmer identifies atomic blocks
Runtime implements synchronization

# In an ideal world…




Transactions may abort:

- because of contention on same memory locations

…and every transaction shall eventually succeed

# …in practice: Best-Effort Nature

No progress guarantees:

- A transaction may **always** abort

…due to a number of reasons:

- Forbidden instructions
- Capacity of caches (L1 for writes, L2 for reads)
- Faults and signals
- Contending transactions, aborting each other

TSX alone is not enough

# TSX with a fall-back: a single lock

```
start:
int status = xbegin
if (status = ok)          // !=ok upon restart
    if (isFree(lock))     // read global lock
            goto code     // fast-path
    else xabort                     // fall-back in use
if (shouldRetry())        // retry policy
    goto start
else
    acquire(lock)         // fall-back

code:
 application logic

if (inFastPath)                     // fast-path
            xend
else                                // fall-back
    release(lock)
```

Still simple enough.

Single lock not an issue if taken rarely.

But **when** should it be taken?

# Objective

Self-tune the management of the fall-back policy

- Focus on TSX and single-lock
- Performance-oriented
- Generalizable to others (IBM, Hybrid TMs…)

# Retry Policies

Define when and how the software fall-back should be used.

- How **many retries** before triggering the fall-back?
  - Ranges from never retrying to insisting many times
- How to cope with **capacity aborts**?
  - **Giveup** – exhaust all retries left
  - **Half** – drop half of the retries left
  - **Stubborn** – drop only one retry left
- How to implement the **fall-back** synchronization?
  - **Wait** – single lock should be free before retrying
  - **None** – retry immediately and hope the lock will be freed
  - **Aux** – serialize conflicting transactions on auxiliary lock

# Static tuning

***Heuristic***:

Try to tune the parameters according to best practices

- Empirical work in recent papers [SC13, HPCA14]
- Intel optimization manual

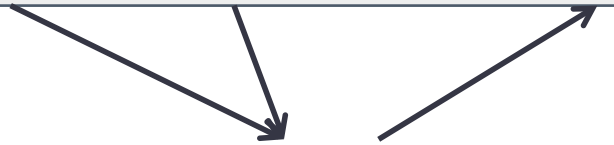
***GCC***:

Use the existing support in GCC out of the box

# Why Static Tuning is not enough

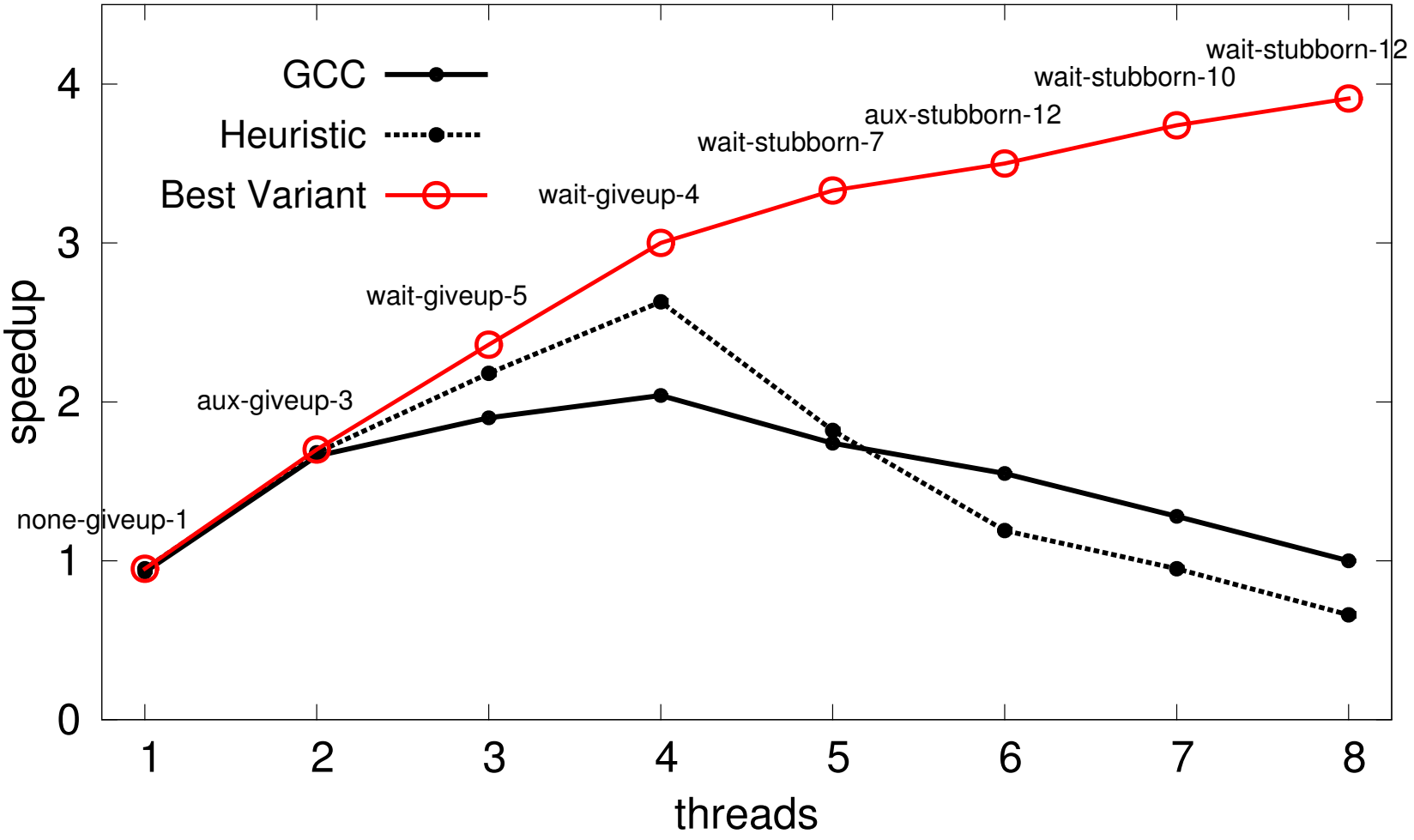
Speedup with 4 threads (vs 1 thread)

| Benchmark | *GCC* | *Heuristic* | Best Tuning | |
|---|---|---|---|---|
| genome | 1.54 | 3.14 | 3.36 | wait-giveup-4 |
| intruder | 2.03 | 1.81 | 3.02 | wait-giveup-4 |
| kmeans-h | 2.73 | 2.66 | 3.03 | none-stubborn-10 |
| rbt-l-w | 2.48 | 2.43 | 2.95 | aux-stubborn-3 |
| ssca2 | 1.71 | 1.69 | 1.78 | wait-giveup-6 |
| vacation-h | 2.12 | 1.61 | 2.51 | aux-half-5 |
| yada | 0.19 | 0.47 | 0.81 | wait-stubborn-15 |




Intel Haswell Xeon with 4 cores (8 hyperthreads)

# Why Static Tuning is not enough




Intruder from STAMP benchmarks

# The Need for Self-tuning

**No-one-size-fits-all** static tuning solution.

We exploit two **reinforcement learning** techniques in synergy:

- **Upper Confidence Bounds**: how to cope with capacity aborts?
- **Gradient Descent**: how many retries in hardware?

For the fall-back synchronization:

- Either **aux** or **wait** were similar
- When **none** was best, was by a marginal amount
- Reduce this dimension in the optimization problem

# How to handle capacity aborts?

Reduction to "**Bandit Problem**"

- 3-levers slot machine with unknown reward distributions


**Exploitation vs Exploration dilemma**

how often to test apparently unfavorable levers?

**Too little**: convergence to wrong solution ☹

**Too much**: many suboptimal choices ☹

# Upper Confidence Bounds (UCB)

Solution to **exploration** vs **exploitation** dilemma

- Online estimation of "uncertainty" of each strategy
  - upper confidence bound on expected reward
- Appealing theoretical guarantees:
  - logarithmic bound on optimization error
- Very lightweight and efficient:
  - **…practical**!

# Upper Confidence Bounds (UCB)

- Basic reward function for each strategy *i* :

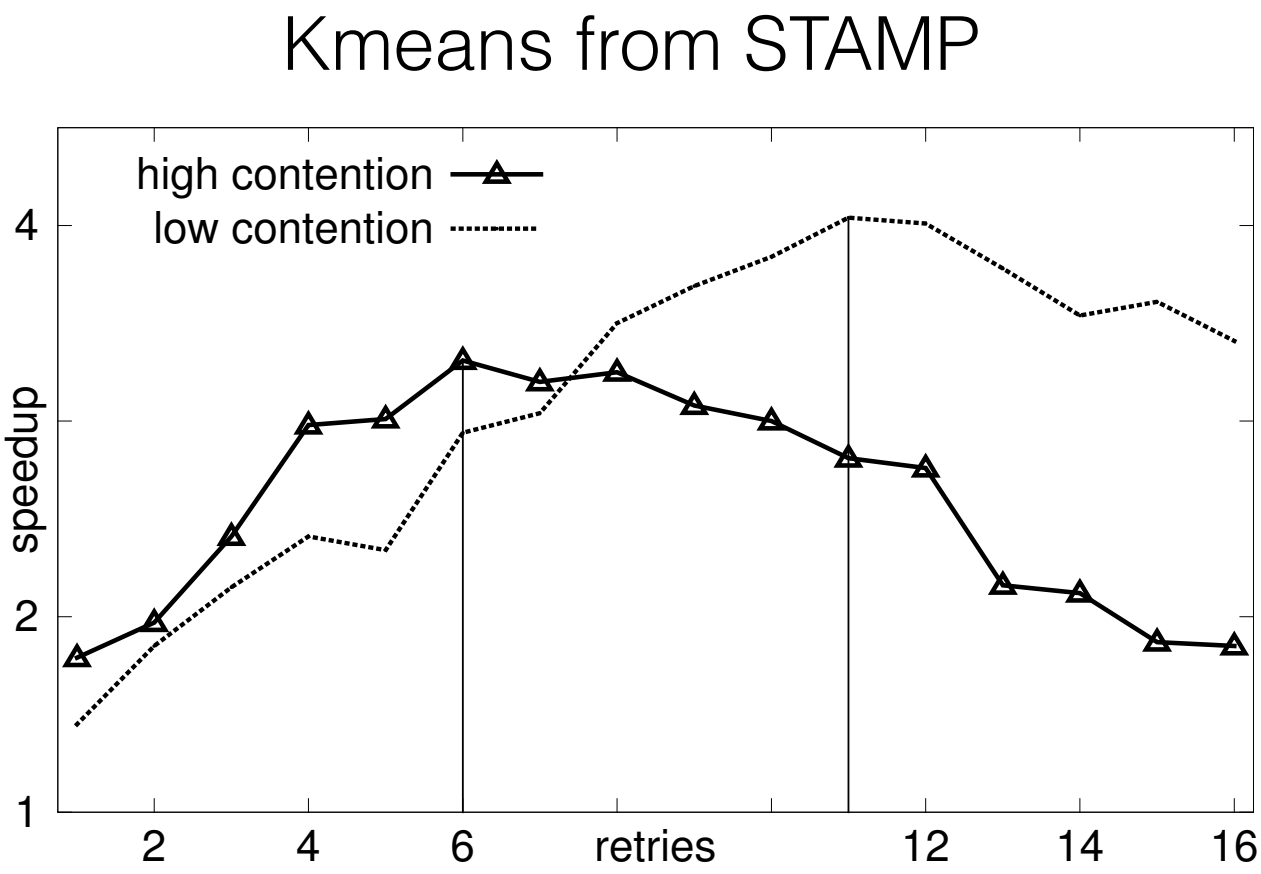
$$x_i = \frac{1}{\textit{avg. \#cycles using strategy } i}$$

- Estimate upper bound on reward of each strategy:

$$\bar{\mu}_i = \bar{x}_i + \sqrt{2\frac{\log n}{n_i}}$$

Amplify confidence bound of rarely explored levers

# How many attempts using HTM?



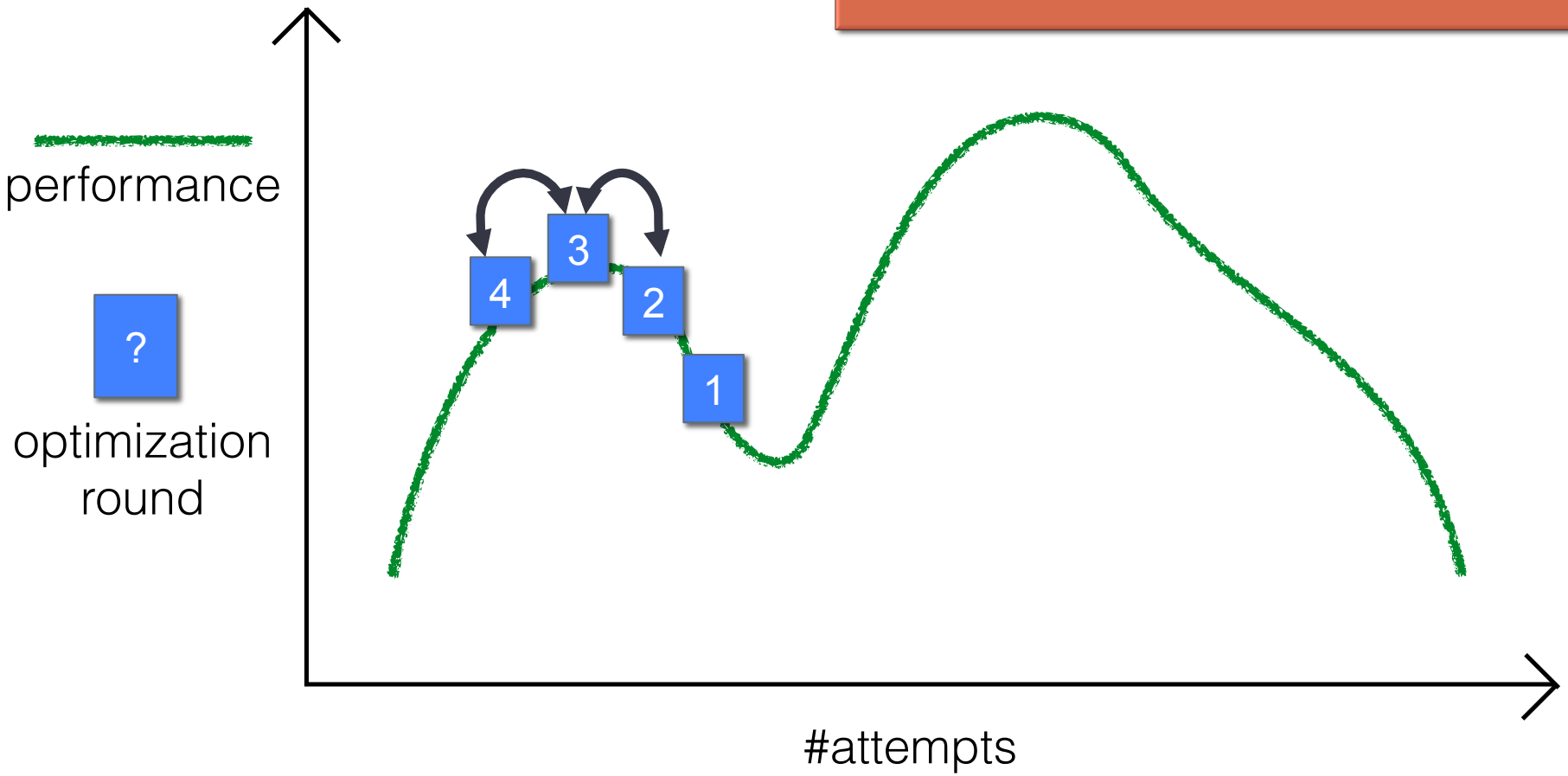

UCB not a good fit ➔ too many levers to explore!

# Gradient Descent

**Problems:**

1- unnecessary oscillations

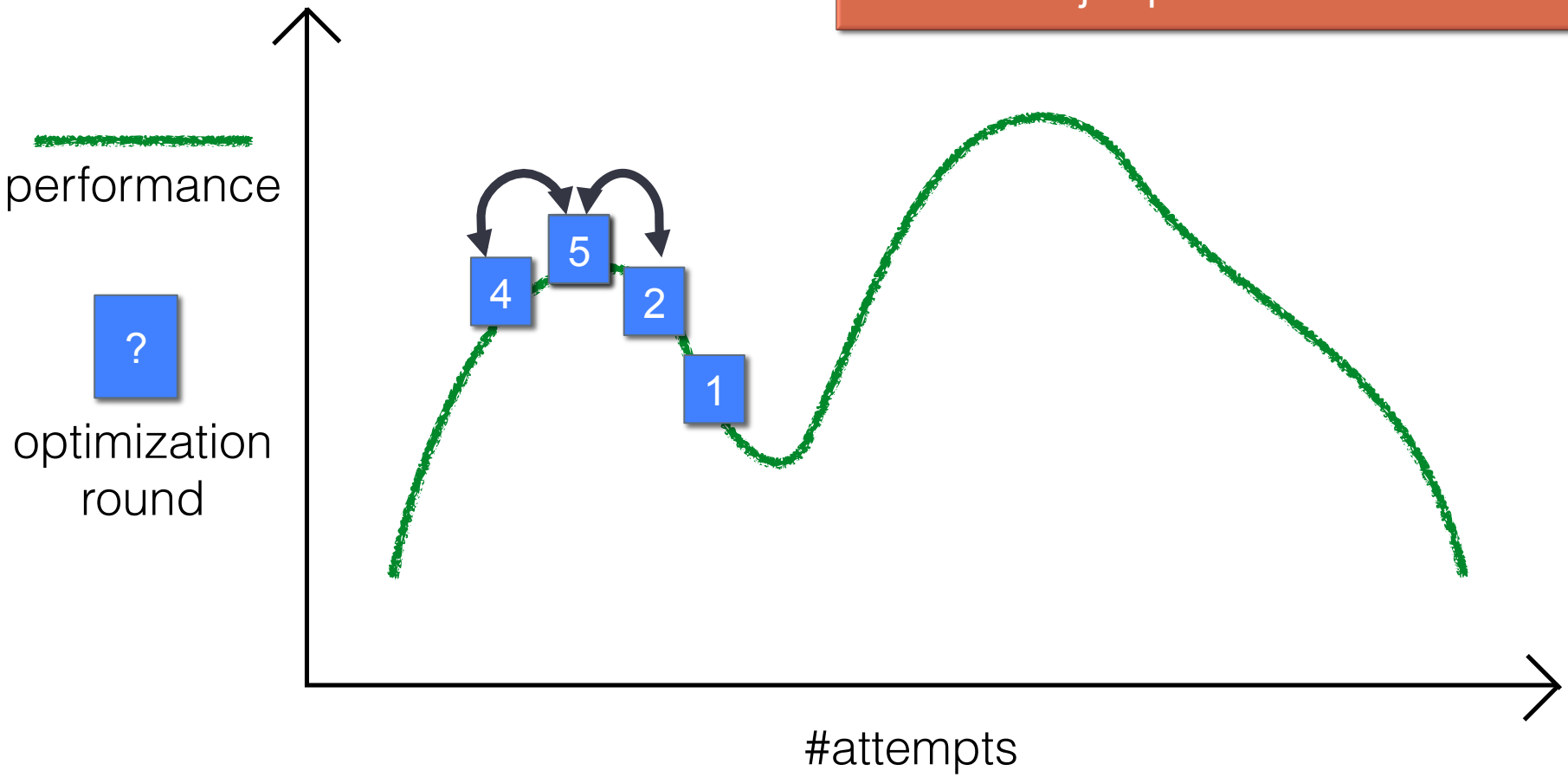
2- stuck in local maxima



# Gradient Descent

**Problems:**

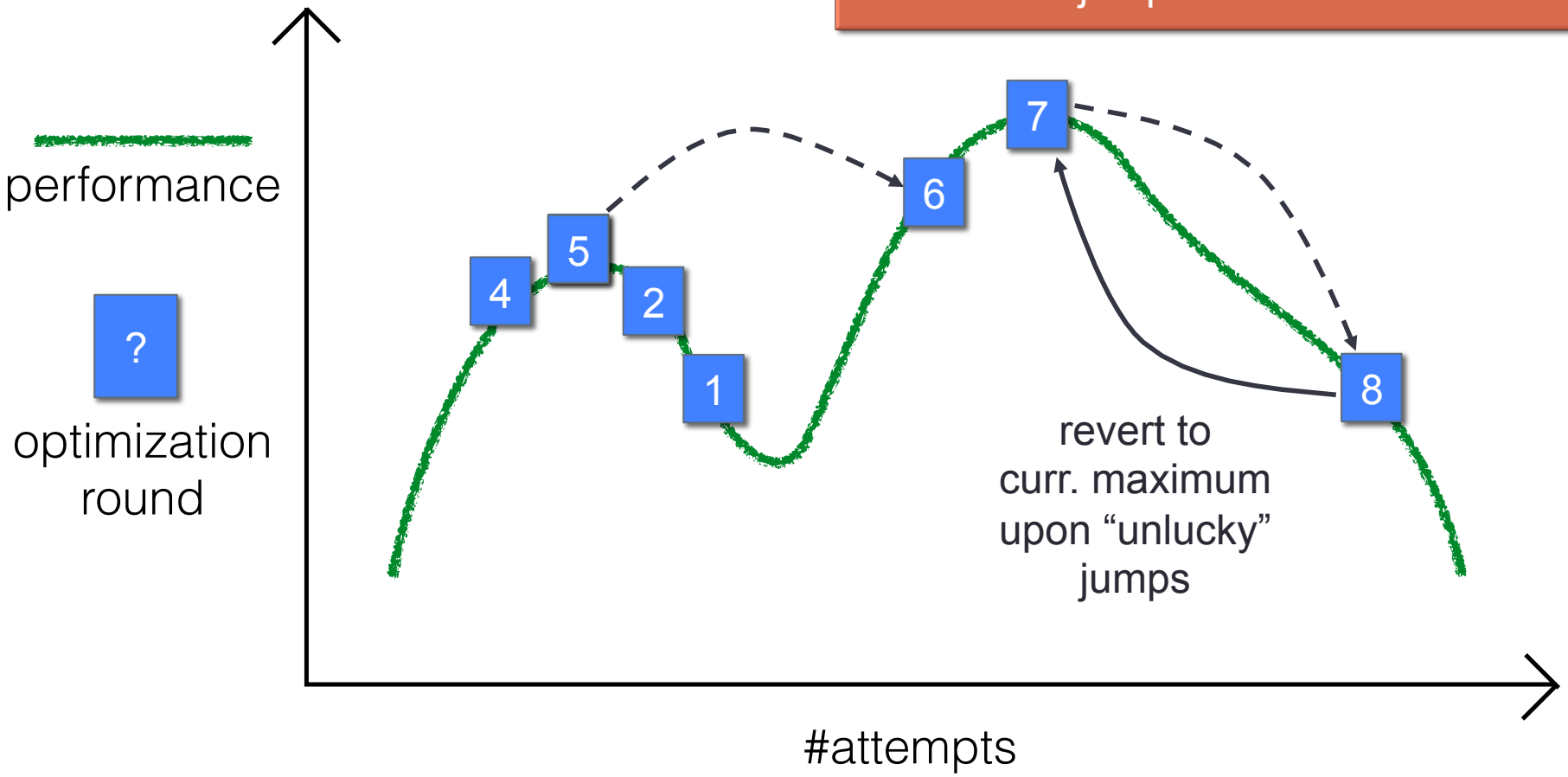
1- unnecessary oscillations
   * stabilization threshold
2- stuck in local maxima
   * random jumps



# Gradient Descent

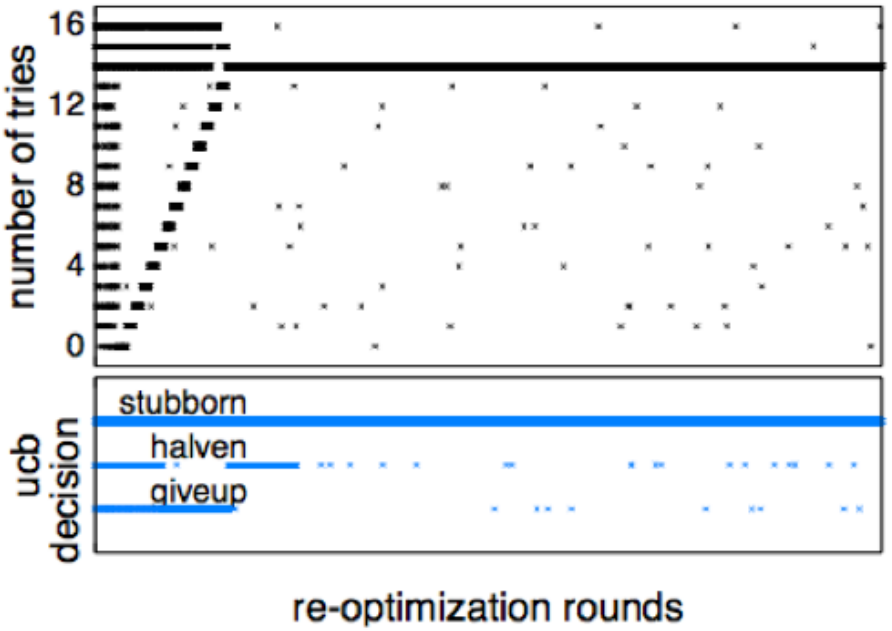
**Problems:**

1- unnecessary oscillations
   * stabilization threshold
2- stuck in local maxima
   * random jumps



# Optimizers in action

One atomic block in Yada benchmark (8 threads).




the two optimizers are *not* independent

# Coupling the Optimizers

UCB and Gradient Descent overlap in responsibilities:

- Optimize consumption of attempts upon capacity aborts
- Optimize allocation of budget for attempts

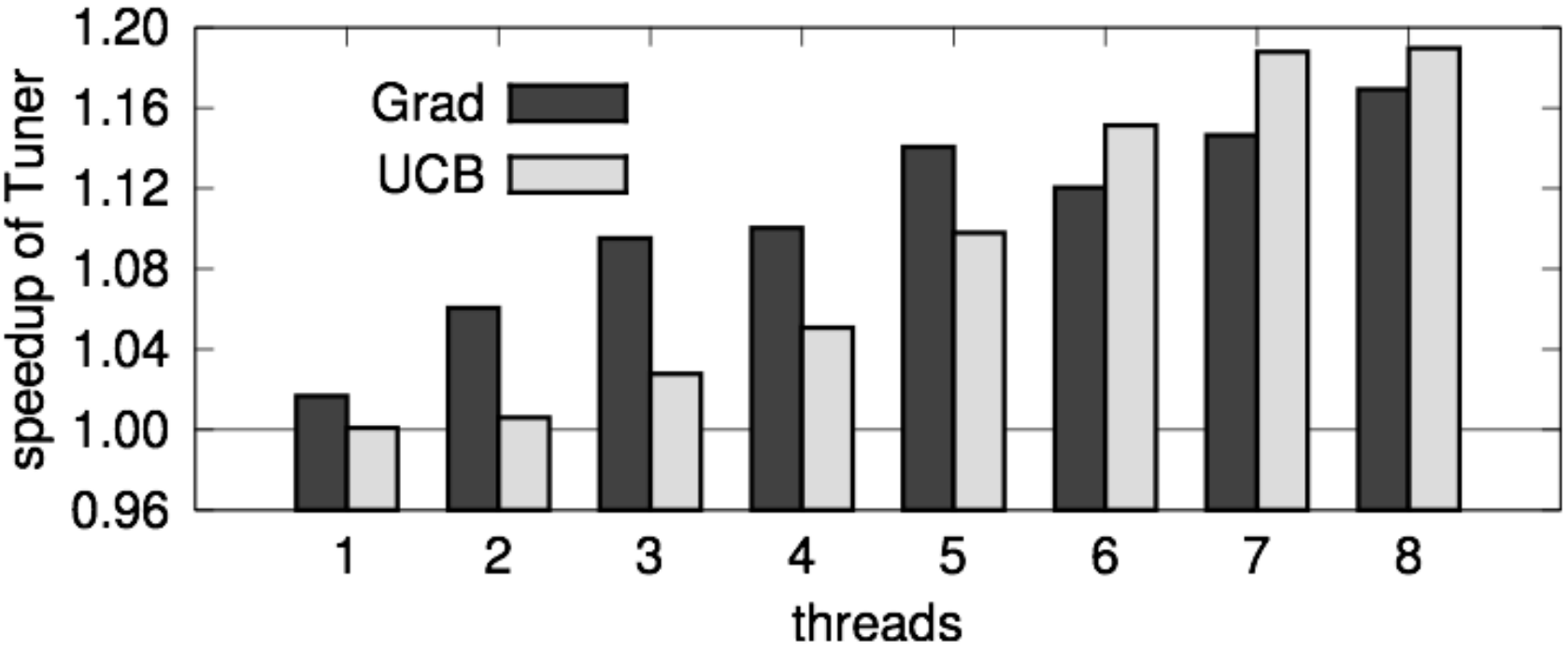
Minimize interference via hierarchical organization:

UCB rules over Grad:

- UCB can force Grad to explore with random jump
  - Direction and length defined by UCB belief
  - More details in the paper

# Coupling the Optimizers

Speedup of coupled techniques vs individual ones



# Tuner: self-tuning Intel TSX

Performance measured through processor cycles:

- RDTSC instruction, lightweight, user space

Optimization per atomic block, per thread:

- Adjusts individually in case of heterogeneous workload
- Avoids any synchronization

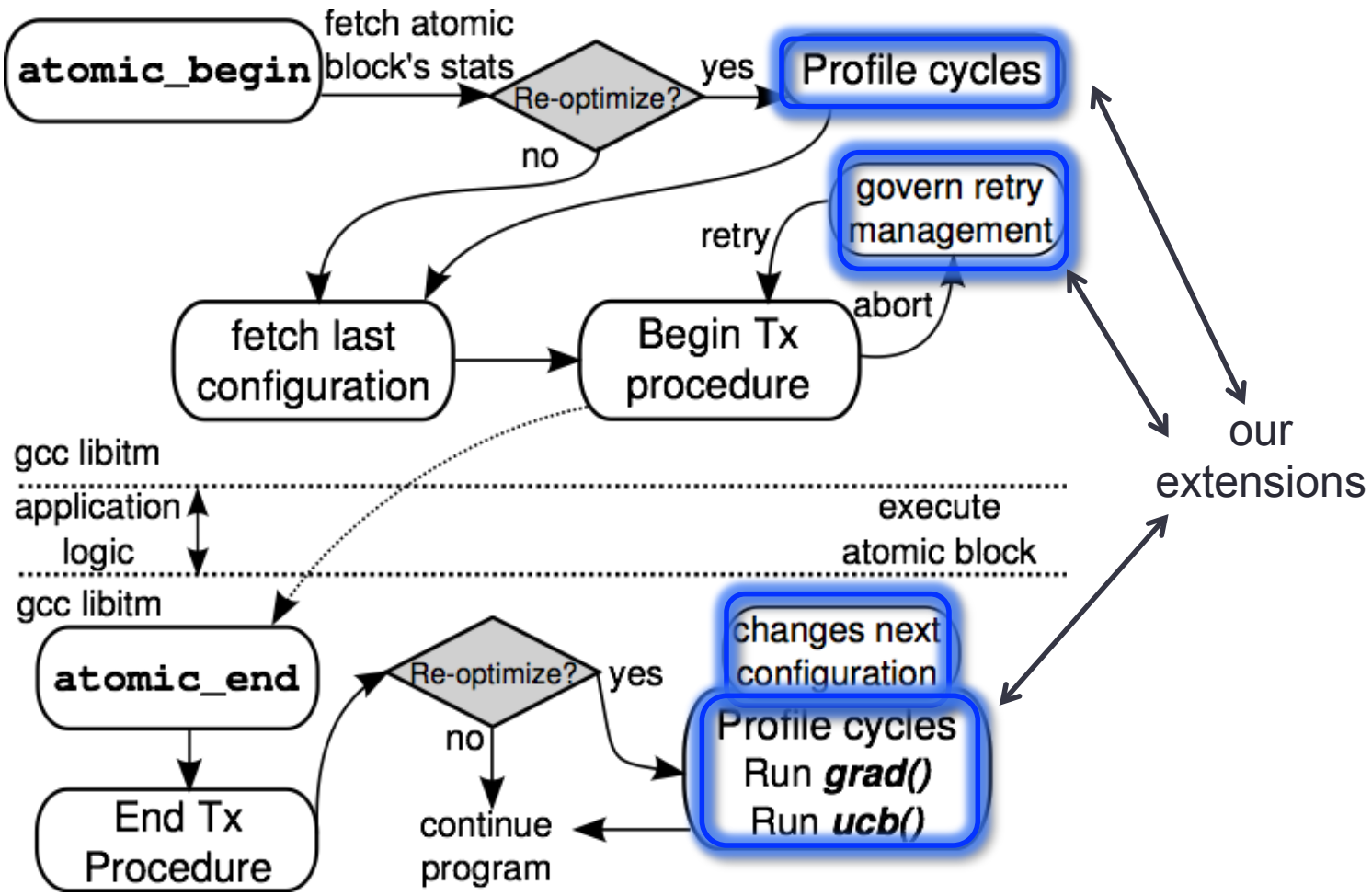
Periodic profiling and re-optimization:

- Crucial to avoid overheads shadowing improvements

# Integration in GCC

- Workload-oblivious
- Transparent to the programmer
- Lightweight for general purpose use
- Ideal candidate for integration at the compiler level

# Integration in GCC



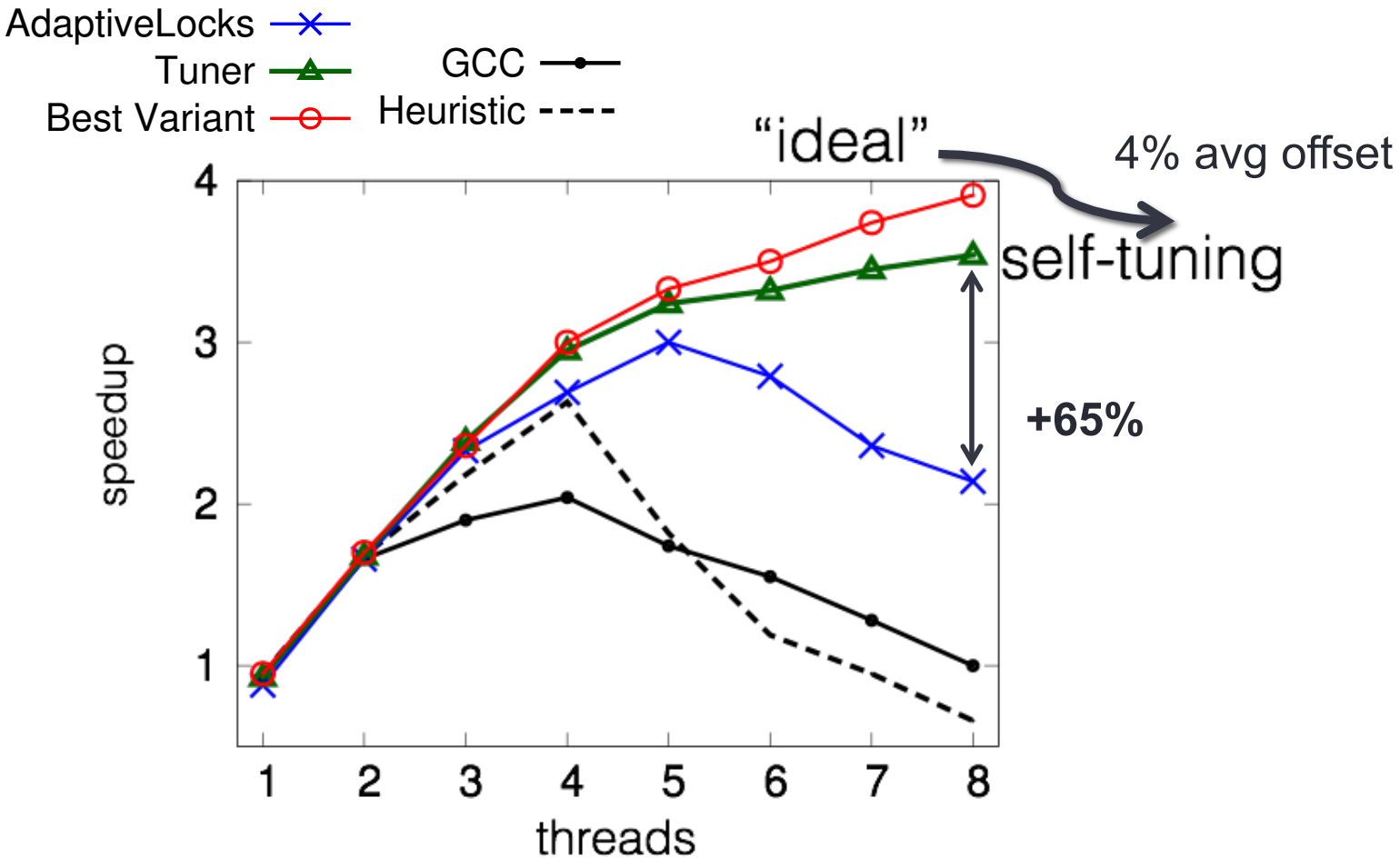
# Evaluation

Comparison across 12 benchmarks:

- Tuner
- GCC
- Heuristic
- Adaptive Locks [PACT09]

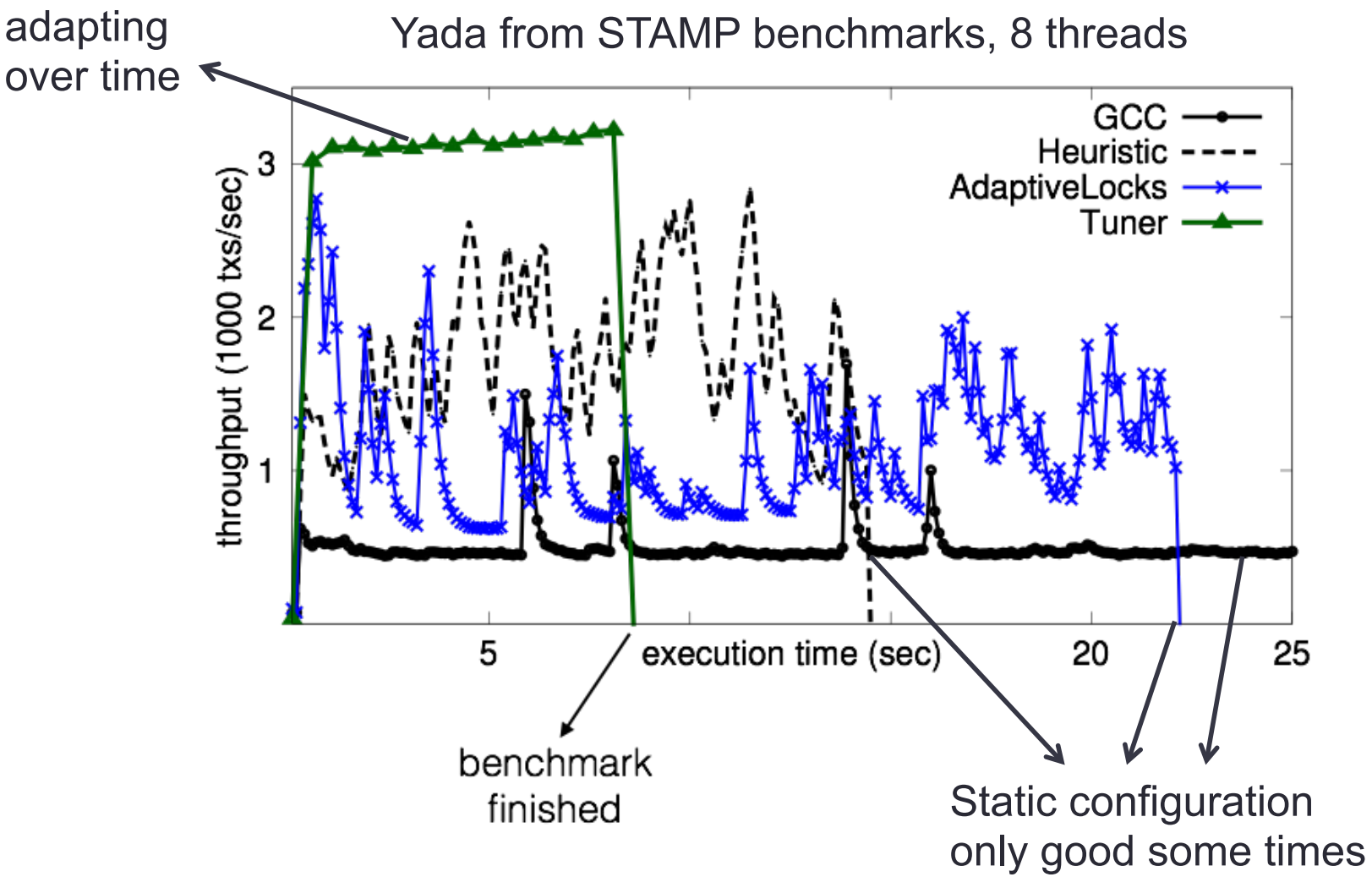
Intel Haswell Xeon with 4 cores (8 hyperthreads)

# Evaluation




Intruder from STAMP benchmarks

# Evaluation


adapting
over time
Yada from STAMP benchmarks, 8 threads
GCC
Heuristic
AdaptiveLocks
Tuner
throughput (1000 txs/sec)
3
2
1
5
execution time (sec)
20
25
benchmark
finished
Static configuration
only good some times

# Evaluation

| Algorithms | threads | | | |
|---|---|---|---|---|
| | 2 | 4 | 6 | 8 |
| GCC | | | | |
| HEURISTIC | | | | |
| ADAPTIVELOCKS | | | | |
| TUNER | | | | |
| Best Variant | | | | |

# Summary

- Hw Transactional Memory is nowadays mainstream…
- …but it needs proper tuning ➔ no-one-size-fits-all!
- 1st self-tuning solution for Intel's HTM (TSX)
- Combination of reinforcement learning techniques
- Fully transparent to the programmer via GCC integration

- **Future research directions:**
  - test on larger parallel machines ➔ available 2015 (?)
  - theoretical work on convergence of optimizers

# Thank you!

Questions?

Code available at: https://github.com/nmldiegues/tuner-icac14