SELF-TUNING HTM

Paolo Romano

Based on ICAC'14 paper

N. Diegues and Paolo Romano

Self-Tuning Intel Transactional Synchronization Extensions

11th USENIX International Conference on Autonomic Computing (ICAC), June 2014

Best paper award

Best-Effort Nature of HTM

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

Need for a fallback path, typically a lock or an STM

When and how to activate the fallback?

- 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

Is static tuning enough?

Focus on single global lock fallback

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

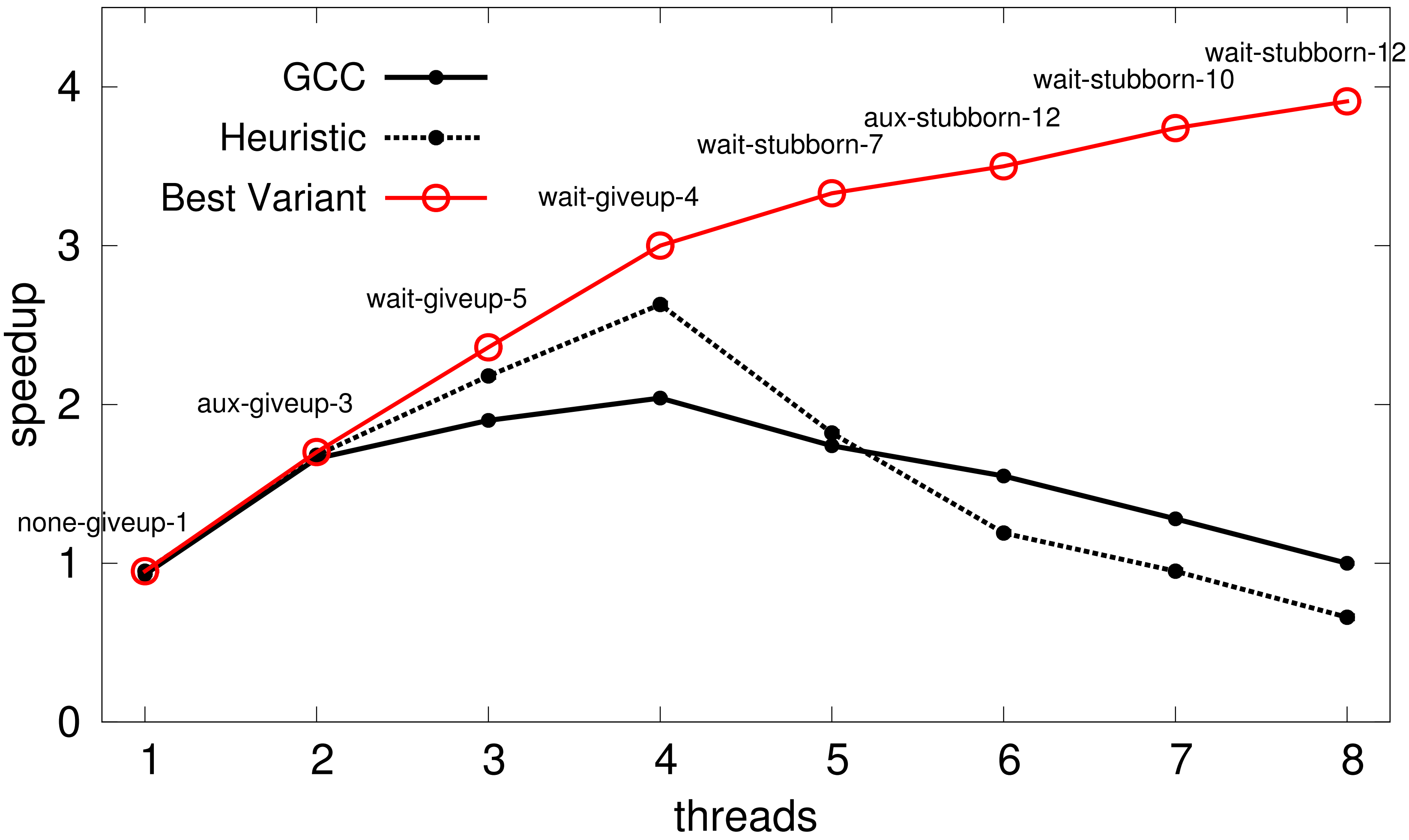
Speedup with 4 threads (vs 1 thread non-instrumented)

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

room for improvement

Intel Haswell Xeon with 4 cores (8 hyperthreads)

No one size fits all



Intruder from STAMP benchmarks

Are all optimization dimensions relevant?

- 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

- **aux** and **wait** perform similarly
- When **none** is best, it is by a marginal amount
- Reduce this dimension in the optimization problem

Self-tuning design choices

3 key choices:

- $\boxed{\text{How should we learn?}}$
- At what granularity should we adapt?
- What metrics should we optimize for?

How should we learn?

- **Off-line learning**
  - test with some mix of applications & characterize their workload
  - infer a model (e.g., based on decision trees) mapping:

    workload ➔ optimal configuration
  - monitor the workload of your target application, feed the model with this info and accordingly tune the system

- **On-line learning**
  - no preliminary training phase
  - explore the search space while the application is running
  - exploit the knowledge acquired via exploration for tuning

How should we learn?

- **Off-line learning**
  - **PRO:**
    - no exploration costs
  - **CONs:**
    - initial training phase is time-consuming and “critical”
      - accuracy is strongly affected by training set representativeness
    - non-trivial to incorporate new knowledge from target application

- **On-line learning**
  - **PROs:**
    - no training phase ➔ plug-and-play effect
    - naturally incorporate newly available knowledge
  - **CONs:**
    - exploration costs

reconfiguration cost is low with HTM
➔ exploring is affordable

Which on-line learning techniques?

Uses 2 on-line **reinforcement learning** techniques in synergy:

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

- Key features:
  - both techniques are extremely lightweight ➔ practical
  - coupled in a hierarchical fashion:
    - they optimize non-independent parameters
    - avoid ping-pong effects

Self-tuning design choices

3 key choices:

- How should we learn?
- At what granularity should we adapt?
- What metrics should we optimize for?

At what granularity should we adapt?

- **Per thread & atomic block**
  - **PRO:**
    - exploit diversity and maximize flexibility
  - **CON:**
    - possibly large number of optimizers running in parallel
      - redundancy ➔ larger overheads
      - interplay of multiple local optimizers

- **Whole application**
  - **PRO:**
    - lower overhead, simpler convergence dynamics
  - **CON:**
    - reduced flexibility

Self-tuning design choices

3 key choices:

- How should we learn?
- At what granularity should we adapt?
- What metrics should we optimize for?

What metrics should we optimize for?

- Performance? Power? A combination of the two?

- Key issues/questions:
  - **Cost** and **accuracy** of monitoring the target metric
    - Performance:
      - RTDSC allow for lightweight, fine-grained measurement of latency
    - Energy:
      - RAPL: coarse granularity (msec) and requires system calls

  - How correlated are the two metrics?

Energy and performance in (H)TM: two sides of the same coin?

- How correlated are energy consumption and throughput?
  - 480 different configurations (number of retries, capacity aborts handling, no. threads) per each benchmark:
    - includes both optimal and sub-optimal configurations

Benchmark	Correlation	Benchmark	Correlation
genome	0.74	linked-list low	0.91
intruder	0.84	linked-list high	0.87
labyrinth	0.82	skip-list low	0.94
kmeans high	0.76	skip-list high	0.81
kmeans low	0.92	hash-map low	0.98
ssca2	0.97	hash-map high	0.72
vacation high	0.55	rbt-low	0.95
vacation low	0.74	rbt-high	0.73
yada	0.77	*average*	0.81

Energy and performance in (H)TM: two sides of the same coin?

- How suboptimal is the energy consumption if we use a configuration that is optimal performance-wise?

Benchmark	Relative Energy	Benchmark	Relative Energy
genome	0.99	linked-list low	1.00
intruder	1.00	linked-list high	1.00
labyrinth	0.92	skip-list low	1.00
kmeans high	1.00	skip-list high	0.98
kmeans low	1.00	hash-map low	0.99
ssca2	1.00	hash-map high	0.99
vacation high	0.99	rbt-low	1.00
vacation low	1.00	rbt-high	1.00
yada	0.89	*average*	0.98

(G)Tuner

Performance measured through processor cycles (RTDSC)

Support fine and coarse grained optimization granularity:

- **Tuner:** per atomic block, per thread
  - no synchronization among threads
- **G$_{\text{(lobal)}}$-Tuner:** application-wide configuration
  - Threads collect statistics privately
  - An optimizer thread periodically:
    - Gathers stats & decides (a possibly) new configuration

→ Integrated in GCC

Periodic profiling and re-optimization to minimize overhead

Evaluation

RTM-SGL

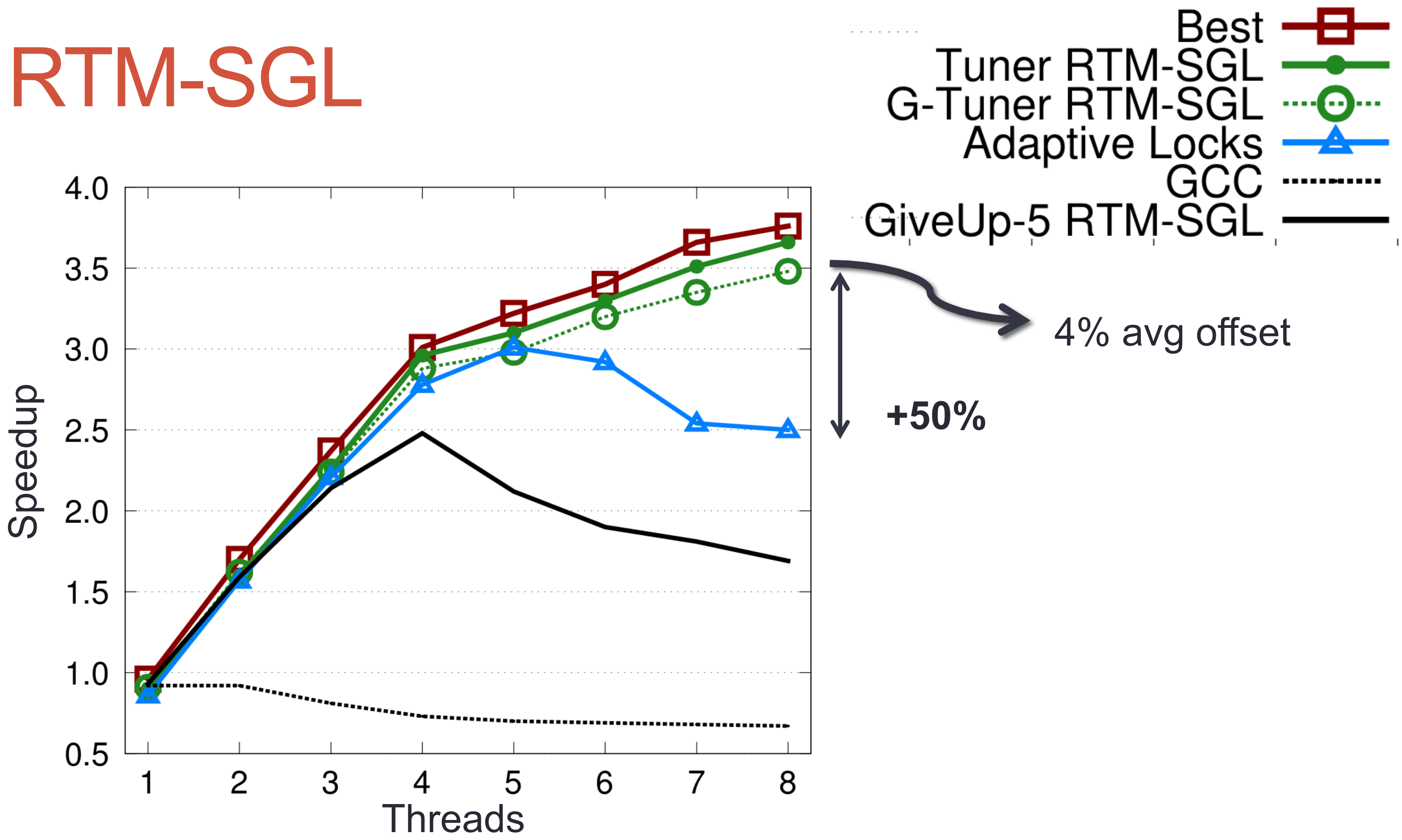
- Idealized “Best” variant
- Tuner
- G-Tuner
- Heuristic: GiveUp-5
- GCC
- Adaptive Locks [PACT09]

RTM-NOrec

- Idealized “Best” variant
- Tuner
- G-Tuner
- Heuristic: GiveUp-5
- NOrec (STM)

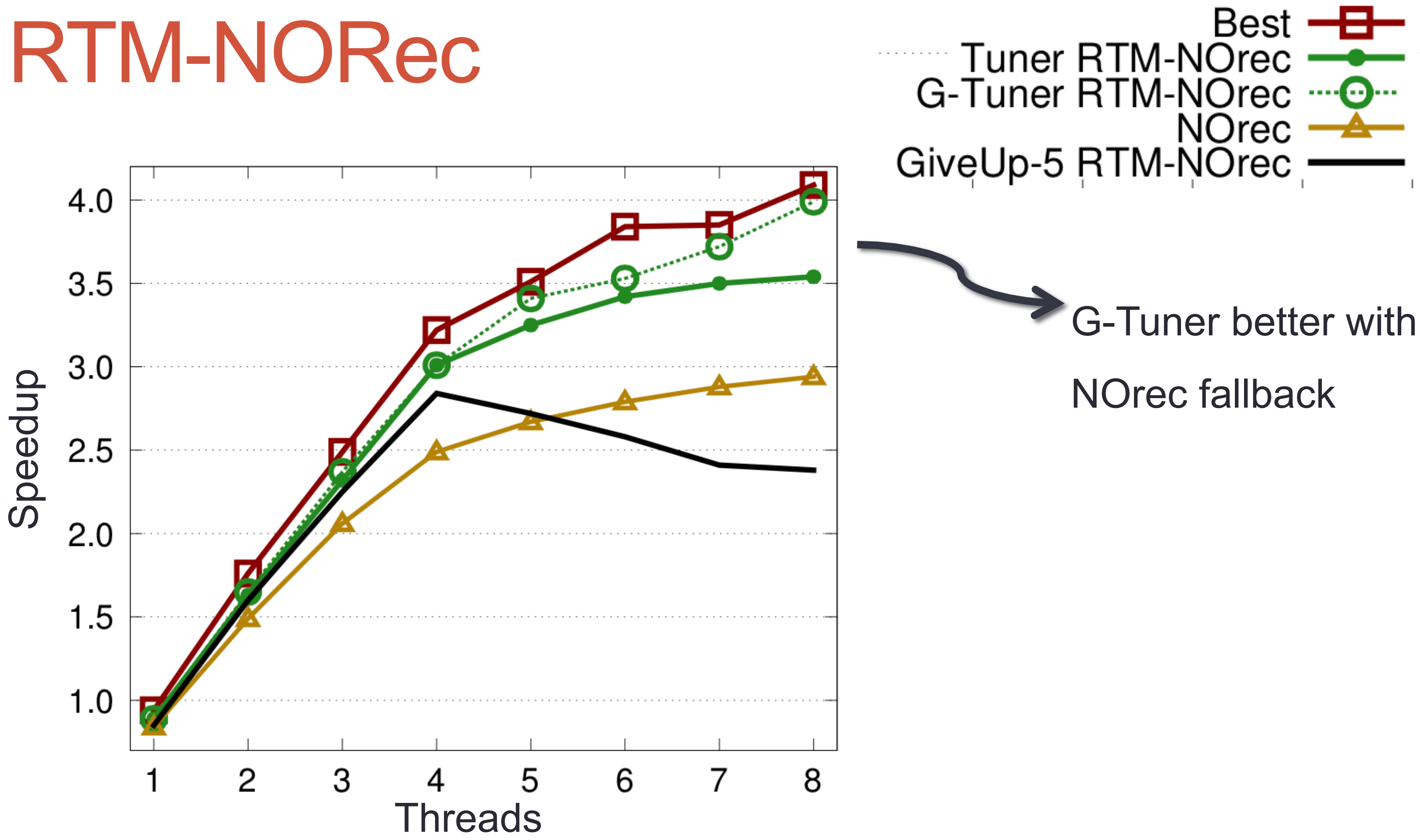
Intel Haswell Xeon with 4 cores (8 hyper-threads)

RTM-SGL



Intruder from STAMP benchmarks

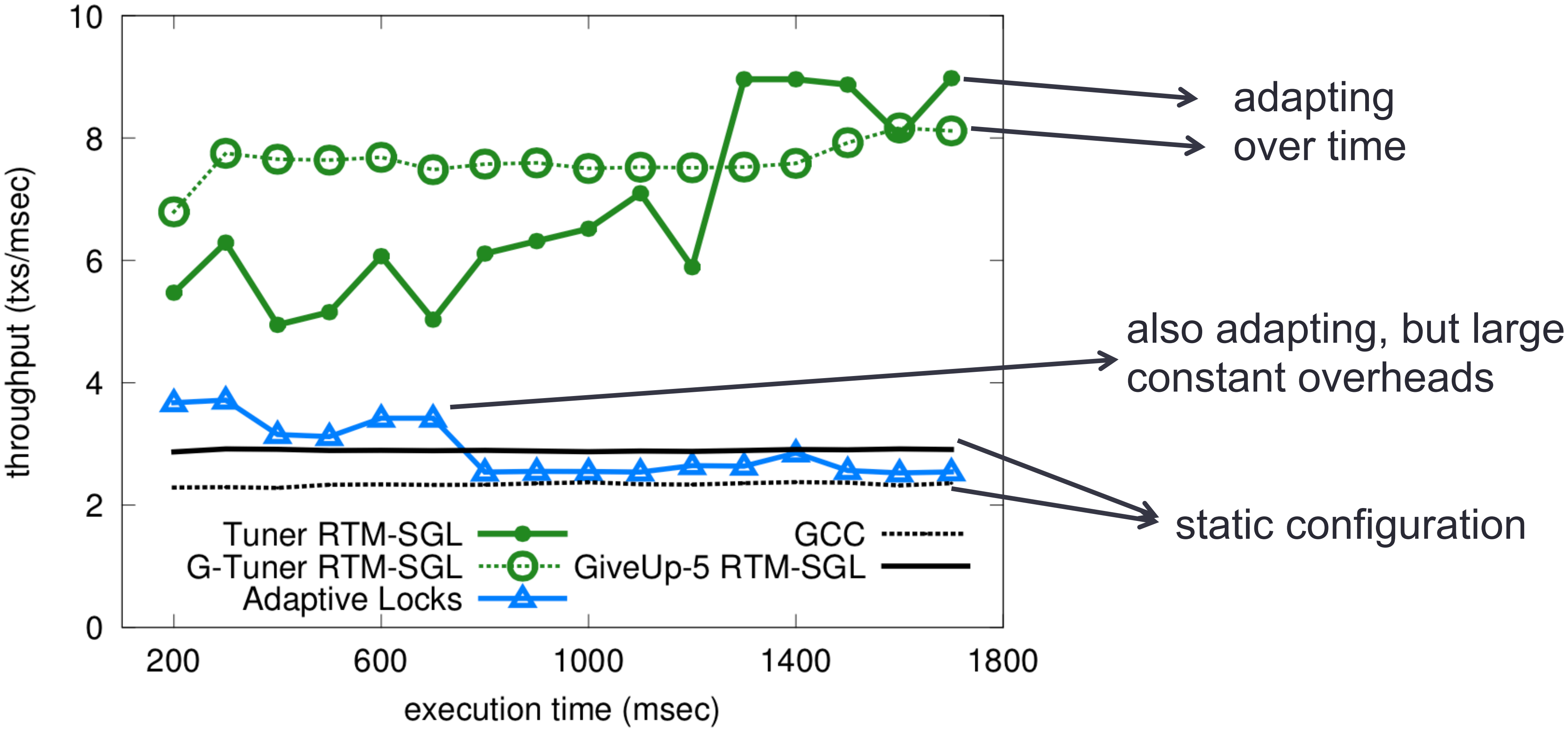
RTM-NORec



G-Tuner better with NOrec fallback

Intruder from STAMP benchmarks

Evaluating the granularity trade-off



Genome from STAMP benchmarks, 8 threads

Take home messages

- Tuning of fall-back policy strongly impacts performance
- Self-tuning of HTM via on-line learning is feasible:
  - plug & play: no training phase
  - gains largely outweigh exploration overheads
- Tuning granularity hides subtle trade-offs:
  - flexibility vs overhead vs convergence speed
- Optimize for performance or for energy?
  - Strong correlation between the 2 metrics
  - How general is this claim? Seems the case also for STM

Thank you!

Questions?