SELF-TUNING INTEL TSX

Nuno Diegues and Paolo Romano
Multi-cores are now ubiquitous

Parallel programming is complex

Transactional Memory abstraction

Hard to get right:
- fine-grained locks
- deadlocks
- correctness

now available in commodity hardware

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

Programmer identifies atomic blocks
Runtime implements synchronization

Intel TSX

x86 instruction set extension: xbegin, xend, xabort…
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
go to code // fast-path
else xabort // fall-back in use
if (shouldRetry()) // retry policy
go to start
else
    acquire(lock) // fall-back

code:
application logic

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

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)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GCC</th>
<th>Heuristic</th>
<th>Best Tuning</th>
</tr>
</thead>
<tbody>
<tr>
<td>genome</td>
<td>1.54</td>
<td>3.14</td>
<td>3.36 wait-giveup-4</td>
</tr>
<tr>
<td>intruder</td>
<td>2.03</td>
<td>1.81</td>
<td>3.02 wait-giveup-4</td>
</tr>
<tr>
<td>kmeans-h</td>
<td>2.73</td>
<td>2.66</td>
<td>3.03 none-stubborn-10</td>
</tr>
<tr>
<td>rbt-l-w</td>
<td>2.48</td>
<td>2.43</td>
<td>2.95 aux-stubborn-3</td>
</tr>
<tr>
<td>scca2</td>
<td>1.71</td>
<td>1.69</td>
<td>1.78 wait-giveup-6</td>
</tr>
<tr>
<td>vacation-h</td>
<td>2.12</td>
<td>1.61</td>
<td>2.51 aux-half-5</td>
</tr>
<tr>
<td>yada</td>
<td>0.19</td>
<td>0.47</td>
<td>0.81 wait-stubborn-15</td>
</tr>
</tbody>
</table>

Intel Haswell Xeon with 4 cores (8 hyperthreads)
Why Static Tuning is not enough

Intruder from STAMP benchmarks

- GCC
- Heuristic
- Best Variant
  - none-giveup-1
  - aux-giveup-3
  - aux-stubborn-12
  - wait-stubborn-7
  - wait-stubborn-12
  - wait-stubborn-10

threads

speedup
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

Strategy:
- giveup
- half
- stubborn

Lever A
Lever B
Lever C

Reward:

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

- Estimate upper bound on reward of each strategy:

  $$\bar{\mu}_i = \bar{x}_i + \sqrt{2 \frac{1}{n_i} \log n}$$

Amplify confidence bound of rarely explored levers.
How many attempts using HTM?

Kmeans from STAMP

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

Problems:
1- unnecessary oscillations
2- stuck in local maxima

- unnecessary oscillations
- stuck in local maxima

Grad

1- unnecessary oscillations
2- stuck in local maxima

Problems:
1- unnecessary oscillations
2- stuck in local maxima

Grad

1- unnecessary oscillations
2- stuck in local maxima

Problems:
1- unnecessary oscillations
2- stuck in local maxima

Grad

1- unnecessary oscillations
2- stuck in local maxima

Problems:
1- unnecessary oscillations
2- stuck in local maxima

Grad

1- unnecessary oscillations
2- stuck in local maxima

Problems:
1- unnecessary oscillations
2- stuck in local maxima

Grad

1- unnecessary oscillations
2- stuck in local maxima

Problems:
1- unnecessary oscillations
2- stuck in local maxima

Grad

1- unnecessary oscillations
2- stuck in local maxima

Problems:
1- unnecessary oscillations
2- stuck in local maxima

Grad
Gradient Descent

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

Graph:
- Performance vs. #attempts
- Optimization round

Questions:
- What is the relationship between performance and #attempts?
- How do optimization rounds affect performance?
Gradient Descent

**Problems:**
1. unnecessary oscillations
   * stabilization threshold
2. stuck in local maxima
   * random jumps

**Graph:**
- Performance vs. #attempts
- Optimization round

- Yellow arrow indicates revert to curr. maximum upon "unlucky" 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

![Graph showing speedup of coupled techniques vs individual ones for Grad and UCB across different threads.](image)
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

```
atomic_begin

fetch atomic block's stats

Re-optimize?

yes

Profile cycles

no

govern retry management

retry

abort

fetch last configuration

Begin Tx procedure

gcc libitm
application logic

gcc libitm

atomic_end

End Tx Procedure

Re-optimize?

yes

Profile cycles

Run \texttt{grad()} 

no

continue program

Run \texttt{ucb()}
```

our extensions
Evaluation

Comparison across 12 benchmarks:

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

Intel Haswell Xeon with 4 cores (8 hyperthreads)
Evaluation

AdaptiveLocks ➤ Tuner ➤ GCC ➤ Heuristic ➤ Best Variant ➤ “ideal”

+65%

4% avg offset

self-tuning

Intruder from STAMP benchmarks

AdaptiveLocks Tuner GCC Heuristic
Best Variant

Figure 8: Speedup of different approaches to tune TSX relative to sequential executions in all benchmarks.

Intruder from STAMP benchmarks
Evaluation

Yada from STAMP benchmarks, 8 threads

Static configuration only good some times

benchmark finished

adapting over time

throughput (1000 txs/sec)

execution time (sec)
## Evaluation

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>GCC</td>
<td></td>
</tr>
<tr>
<td>HEURISTIC</td>
<td></td>
</tr>
<tr>
<td>ADAPTIVELOCKS</td>
<td></td>
</tr>
<tr>
<td>TUNER</td>
<td></td>
</tr>
<tr>
<td>Best Variant</td>
<td></td>
</tr>
</tbody>
</table>
Summary

- Hw Transactional Memory is nowadays mainstream…
- …but it needs proper tuning ➔ no-one-size-fits-all!
- 1\textsuperscript{st} 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