Regulating Concurrency in Software Transactional Memory: An Effective Model-based Approach

Pierangelo Di Sanzo, Francesco Del Re, Diego Rughetti, Bruno Ciciani, Francesco Quaglia

DIAG, Sapienza University of Rome
Nowadays **multi-core/processor systems** have become mainstream platforms.

Hardware performance continues to improve mainly due to:

- even more cores in a single processor
- even more processors in a single machine

**Single-threaded/process applications** can not take advantage of such a hardware performance improvement.

need for **multi-threaded/process applications**
The synchronization problem in concurrent applications: code sections accessing shared data may have to be synchronized (e.g. critical sections)

- using traditional synchronization techniques (i.e. locks, semaphores, monitors, ...) is not easy
- fine-grained synchronization is a time-consuming task

• pitfalls for programmers: deadlocks, livelocks, priority inversions, code composition is complex, scalability issues, ...
• debug is complex

Transactional Memories (TM):

• programming paradigm for multi-core/processor systems
• simplifies the development of parallel and concurrent applications

Key idea: hide away synchronization issues by using transactions

Diagram:

- Application
- STM layer
- Hardware

---

HPDCS Research Group

SAPIENZA Università di Roma
Using TM: code example (queue pop)

```c
...
  elem_t* elemPtr;
  ...
  TM_BEGIN();

  long pop    = (long)TM_READ(queuePtr->pop);
  long push   = (long)TM_READ(queuePtr->push);
  long capacity = (long)TM_READ(queuePtr->capacity);

  long newPop = (pop + 1) % capacity;
  if (newPop == push) {
    elemPtr = NULL;
  } else {
    void** elements = (void**)TM_READ(queuePtr->elements);
    elemPtr = (pair_t*) TM_READ(elements[newPop]);
  }
  TM_WRITE(queuePtr->pop, newPop);

  TM_END();
...
```

The underlying transactional memory layer takes care of ensuring **atomic and isolated executions of transactions**.
Transactional Memories: How Many Threads?

![Speedup curve diagram]

- **Concurrency level too low**: performance is penalized due to limitation of parallelism and underutilization of hardware resources.
- **Optimal concurrency level**.
- **Concurrency level too high**: loss of performance due to excessive data contention and consequent transaction aborts and re-runs.
Identifying the optimal concurrency level...

The optimal concurrency depends on:
application logic, workload profile, hardware architecture, ...

Additionally, the optimal concurrency level may change depending on the application execution phase.

**phase 1**
Speedup curve

optimal concurrency level: 10

**phase 2**
Speedup curve

optimal concurrency level: 8

**phase 3**
Speedup curve

optimal concurrency level: 14
Goal and Proposed Solution

Goal:

enabling TM platforms to **self-regulate the concurrency level**

How:

1) a *parametric performance model* of TM applications is used to estimate the throughput of an application as a function of the:
   - concurrency level (number of concurrent threads)
   - the current workload profile of the application

2) *regression analysis* is exploited to customize the parametric performance model for a specific TM system (application + hardware platform)

3) a *controller* is integrated with the TM platform. At run-time, the controller exploits the customized model in order to decide, on basis of the observed workload profile, the number of concurrent threads to keep active
The TM Parametric Performance Model

The parametric performance model estimates the transaction abort probability of transaction \( p_a \) as a function of:

- the average size of the transaction read-set \( (r_s) \)
- the average size of the transaction write-set \( (w_s) \)
- the average execution time of committed transaction runs \( (t_t) \)
- the average execution time of code blocks outside of transactions \( (t_{ntc}) \)
- the read/write affinity \( (r_{wa}, \text{i.e. the probability that an object read by a transaction is also written by other transactions}) \)
- the write/write affinity \( (w_{wa}, \text{i.e. the probability that an object written by a transaction is also written by other transactions}) \)
- the number of concurrent threads \( (k) \)

\[
p_a = f(r_s, w_s, r_{wa}, w_{wa}, t_t, t_{ntc}, k)
\]
Results from analytical modeling studies of transactional systems (e.g. \([1,2]\)):

\[ p_a = 1 - e^{-\alpha} \]

where \( \alpha \) is a complex function which is hard to identify unless making strong assumptions on workload profile, operations’ execution speed, data contention, etc. This may cause high prediction error with real applications.
Proposed approach:

- Simulation has been used to determine a parametric expression which captures the shape of the curve of the function $p_a$ depending on the workload profile.

- The parametric expression has been validated using data achieved by executing TM applications on real systems.

Basic equation:

\[ p_a = 1 - e^{-\rho \cdot \omega \cdot \phi} \]

- parametric function of $rs$, $ws$, $rw_a$, $ww_a$,
- parametric function of $t_t$, $t_{ntc}$,
- parametric function of $k$.
Model Construction Approach

Case of $\rho$: expressing $\rho$ as a function of $w_s$ and $w_w$

fitting function:

$$[c \cdot (ln(b \cdot w_s + 1)) \cdot ln(a \cdot w_w + 1)]^d$$

fitting parameters: $a, b, c, d$
Model Construction Approach

fitting function: \[ [c \cdot (\ln(b \cdot w_{s_s} + 1)) \cdot \ln(a \cdot w_{w_a} + 1)]^d \]

Error evaluation with respect simulation data:

- fitting parameters \((c, b, a, d)\) calculated through regression analysis (using 40 randomly selected workload profiles)
- average error (using 80 randomly selected workload profiles while varying \(w_{w_a}\) and \(w_{s_s}\)): 5.3%
Model Construction Approach

The same approach for \( \omega \) and \( \Phi \), ...

Finally

\[ p_a = 1 - e^{-\rho \cdot \omega \cdot \Phi} , \text{ where:} \]

expression of \( \rho \):

\[ [c \cdot (ln(b \cdot ws_s + 1)) \cdot ln(a \cdot ww_a + 1)]^d + \]
\[ + [e \cdot (ln(f \cdot rw_a + 1)) \cdot ln(g \cdot rs_s + 1) \cdot ws_s]^z \]

expression of \( \omega \):

\[ h \cdot (ln(l \cdot (k - 1) + 1) \]

expression of \( \Phi \):

\[ m \cdot ln(n \cdot \theta + 1) , \text{ where} \quad \theta = \frac{t_t}{t_t + t_{ntc}} \]
Model Construction Approach

Final average error with respect to simulation data: 4.8%

Transaction abort probability vs. workload profiles

- Achieved by simulations
- Predicted by the model
Model Validation with a Real System


Hardware: HP ProLiant server with 2 AMD OpteronTM6128 Series Processor, 8 cores per CPU (for a total of 16 cores), 32 GB RAM, Linux kernel version 2.7.32-5-amd64.

- Evaluation of the extrapolation capability of the model: for each application, three regression analysis have been performed using three different sets of measurements. Each set of measurements includes 80 samples gathered observing the application running with:
  - 2 and 4 concurrent threads (first set)
  - 2, 4 and 8 concurrent threads (second set)
  - 2, 4, 8 and 16 concurrent threads (third set)

Results (while varying application workload profiles and the number of threads between 2 and 16)

abort probability prediction error (variance in brackets)

<table>
<thead>
<tr>
<th>application</th>
<th>2/4 threads</th>
<th>2/4/8 threads</th>
<th>2/4/8/16 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vacation</td>
<td>2.166% (0.00089)</td>
<td>1.323% (0.00028)</td>
<td>1.505% (0.00032)</td>
</tr>
<tr>
<td>Kmeans</td>
<td>18.938% (0.09961)</td>
<td>2.086% (0.00100)</td>
<td>2.591% (0.00109)</td>
</tr>
<tr>
<td>Yada</td>
<td>2.385% (0.00029)</td>
<td>2.086% (0.00016)</td>
<td>2.083% (0.00022)</td>
</tr>
</tbody>
</table>
Comparison with a Neural Network-based Performance Model

Evaluation of the extrapolation capability with respect a neural network-based model (proposed in [5])

| Set of samples | regression analysis for the parametric model | training phase for the neural network | comparison of the prediction accuracy |

Results for Yada benchmark (using the first set of samples)

- **Model prediction accuracy**
  - Average error: 2.385%

- **Neural network prediction accuracy**
  - Average error: 17.3%
Self-regulating the Concurrency Level

Enabling STM to self-regulate the concurrency level:

architecture of CSR-STM (Concurrency Self-Regulating STM)
available at URL http://www.dis.uniroma1.it/hpdcs/CSR-STM.tar
Self-regulating the Concurrency Level

Periodically, the controller uses the performance model to calculate the expected abort probabilities $p_{a,k}$ while varying the number of concurrent threads $k$, i.e.:

$$\{(p_{a,k}), 1 \leq k \leq max_{thread}\}$$

hence, the controller keeps active $m$ threads, where $m$ is the value of $k$ such that the application throughput, i.e.

$$\frac{k}{w_{t,k} + t_{t,k} + t_{ntc,k}}$$

is maximized

average transaction wasted time:

average execution time of committed transaction runs:

average execution time of code blocks outside of transactions:

It can be calculated using the average number of transaction re-runs:

$$p_{a,k}/(1-p_{a,k})$$

Hardware scalability model used for validation [14]:

$$C(k) = 1 + p \cdot (k - 1) + q \cdot k \cdot (k - 1)$$
Comparison Between CSR-STM and Tiny STM

- Vacation: `input: -n4 -q60 -u90 -r32768 -t4194304 -p800000`
- Kmeans: `input: -m10 -n10 -t0.0005 -i random-n65536-d32-c16.txt`
- Yada: `input: -a20 -ltimeu100000.2`
Comparison with Other Approaches

- Analytical models of concurrency control protocols for transactional systems (e.g. [1,2,6,7,8]):
  - strong assumptions on workload profile, operations' execution speed, data contention, etc → high prediction error with real applications

- Interpolation using different kind of functions (e.g. polynomial, rational, logarithmic functions) [8]:
  - workload profile is not accounted → variation of the optimal concurrency level along the execution of the application can't be captured

- Machine learning-based approach (e.g. [5]) → low extrapolation capability (vs. the parametric performance model-based approach)

- Pro-active transaction scheduling schemes (e.g. [10,11,12])
  - based on heuristic schemes → require evaluating suitable heuristics and tuning a set of thresholds depending on the application workload
Thank you

References (1/2)


References (2/2)


