Motivation

\[ 2 + 2 = 4 \]
\[ 2 + 2 = 4.2 \]
\[ 2 + 2 = 3.9 \]
Motivation (for the new reality!)

- **Processor Evolution**
  - More Cores
  - Increasing density

- **Problem**
  - Power Consumption

- **“Trivial” Solutions**
  - (A) Disable part of components
  - (B) Execute at lower voltage

- **Approximate Computing:**
  - Design relaxed accuracy components
  - Expose Errors to Application-level
  - Fits well for some apps (e.g. image processing)

---

But...

- **Different conditions** results in different error rates
- **Different applications** have different sensitivity to errors
- **Design of Adaptive Tools** to control the 2-way variability

**We propose:**
- Trade Parallelism for Accuracy
- Dynamically determine number of replicas needed to improve results
- Result combining tool for DSS workload
Increasing Accuracy with Parallelism

Where was CF hosted until 2011?
(A) Cagliari, Italy
(B) Bertinoro, Italy
(C) Ischia, Italy
(D) Paphos, Cyprus

A, B, D
Correct Answer? Ask Philosophers, Sociologists...

Contributions

- Proposal of **QoR a metric** that quantifies the quality of the results of a DSS query
- Analysis of **query result sensitivity** to relaxed correctness
- Proposal of a **dynamic adaptive technique** that is implemented as a tool
- Evaluation of the proposed technique using a **real DSS workload** on a **real DBMS** for different fault rate scenarios
Related Work

- Relaxed Correctness
  - Stochastic, Approximate, Inexact Computing [CF’12]
  - Metric of Correctness
  - Applications detect and correct/tolerate errors
- Disciplined Approximate Programming Model [ASPLOS XVII]
- Fault-tolerance:
  - techniques to overcome transient faults: replication, redundancy
- Algorithmic changes to applications
- (Mixed Precision)
- Autonomic Computing

Our Work: Combination of the above for Database Workloads - Adaptive dynamic combination for different hardware fault rates and query sensitivities

Modelling Approximation

- Disciplined Approximate Computing Model
- Localized Fault injection in PostgreSQL
  - ExecQual function
  - Flipping result: generates False Positives and False Negatives
  - Normal distribution: based on random function
  - Error Rate = Frequency that a tuple is evaluated incorrectly
  - Controlled error rate: threshold
  - Emulates also storage errors
Quality of Result Metric (QoR)

- A query result may:
  - Include tuples that do not belong (False Positive)
  - Exclude tuples that should belong (False Negative)
  - Include tuples with wrong values in fields

\[
QoR = \frac{\#TP}{\#ANS} \times \frac{\#TP}{\#RES} \times \left(1 - \frac{\sum |ERR|}{\#RES}\right)
\]

- **AccuracyTP**: how many of the true tuples are present (\#ANS is number of tuples in correct answer)
- **AccuracyFP**: how the result is affected by false results (\#RES is \#TP+\#FP)
- **AccuracyOwn**: how the values in fields are affected (ERR is distance of value from correct)

**Answer:**
- <Pedro, UCY, 2>
- <Andreas, UCY, 1>

**Result:**
- <Pedro, UCY, 2>

\[
QoR = \text{AccTP} \times \text{AccFP} \times \text{AccOwn}
\]

\[
= 1/2 \times 1/1 \times 1 = 50\%
\]

Workload Sensitivity

TPC-H 100MB
PostgreSQL
Error Rate = 1 in 100,000

- 0.00001
- Q1
- Q2
- Q3
- Q4
- Q5
- Q6
- Q7
- Q8
- Q9
- Q10
- Q11
- Q12
- Q13
- Q14
- Q15
- Q16
- Q17
- Q18
- Q19
- Q20
- Q21
- Q22

EECS Seminar on Optimization and Computing, NTNU, 30/09/2014
Workload Sensitivity

TPC-H 100MB
PostgreSQL
Error Rate = 1 in 10,000

Workload Sensitivity

TPC-H 100MB
PostgreSQL
Error Rate = 1 in 1,000
Workload Sensitivity

• **Intolerant:** Q21
• **Highly Sensitive:** Q2, Q18
• **Insensitive:** Q1, Q14

EECS Seminar on Optimization and Computing, NTNU, 30/09/2014
Workload Sensitivity: Breakdown

- Q21 (Intolerant): ErrorOwn (errors in aggregate values)
- Q2 (Highly Sensitive): ErrorTP (reduced correct tuples)
- Q18 (Highly Sensitive): ErrorFP (many extra tuples)
- Q1, Q14 (Insensitive): Simple queries without nested operations

Nested Queries ➔ Complex ➔ More Operations ➔ Sensitive

Combining Algorithm

```c
CombineResults( query ) { 
query' = cleanAggregate( query )
results = execute n instances of query'
uniqueTuples = frequency( results )
combineResult = combine( uniqueTuples, x )
newResult = aggregate( combineResult )
return newResult
}
```

- Trade parallel resources for accuracy: combine inaccurate results
- Remove Aggregate, Filter out results with smaller frequency (not as probable to belong to result), Apply Aggregate
Combining Implementations

- **n** replicas, threshold **x** for combining results
- **Static** Combination of Results
  - \( x = \text{ceil}(\frac{n}{2}) \); \( n=3, 5, 7 \)
  - 2-of-3, 3-of-5, 4-of-7
- **Adaptive Dynamic** Combination of Results
  - **Real life**: Not possible to know query sensitivity and failure rate ahead of execution
  - Best \((n, x)\)
  - Execute 5 replicas and obtain 10 first results,
    - apply 3-of-5
    - Observe how far #unique tuples is from 10
    - Use heuristic to get the \( x\)-of-\( n \)

Adaptive Combine Tool

**Execution:**

1. Remove the aggregate operation from the query.
2. Send 5 replicas of the non-aggregate query to the system and execute them to obtain the first 10 tuples.
3. Collect the 5 results, combine them, and decide on the number of replicas \( n \) to be used to complete the execution (1, 3, 5, or 7)
4. Continue the execution of the query in the \( n \) replicas
5. Collect the results from the \( n \) replicas, combine the results, and use the corresponding voting scheme to filter the results
6. Apply the aggregate function to the combined filtered result
Experimental Setup

- Real DBMS: PostgreSQL v9.2.4
- Fault-rates: 1 in 100000 (0.00001), 1 in 10000 (0.0001), 1 in 1000 (0.001), 1 in 100 (0.01) and 1 in 10 (0.1)
- Workload: TPC-H 22 queries
  - Characterization: 20 queries
  - Evaluation: 10 representative queries: Q1, Q2, Q4, Q5, Q6, Q7, Q8, Q9, Q12, and Q13
- Input Data Set: SF=0.1 or 100MB
- Parts of Tool implemented as ‘C’ code
- System: Intel Core 2 Duo 1.4GHz, OS X v 10.9.1

QoR with Static Combining

- Original (no replica), 2-of-3, 3-of-5, 4-of-7, OracleBest (x-of-n)
- Combining results in increase of QoR (QoR > 80%)
- OracleBest not needed unless sensitive Qs and high failure rate
**Evaluation of Adaptive Technique**

- **Goal:** Tradeoff between QoR and Parallelism
- **Assumption:** QoR > 80%

- **Evaluation:**
  - 78% same decision as Oracle
  - 14% used more replicas (over achieved QoR)
  - 8% needed to use more replicas (under achieved QoR)

- **Query “Satisfaction”**
  - 80%+ of queries with QoR > 80% up to 1 in 100 error rate
QoR and Parallelism

- **Adaptive**:
  - increased avg QoR from 68.9% to 79.3% with a parallelism degree of 55.6%
  - achieved QoR within 7% of Oracle
  - gained 22.3% in parallelism from Static
  - correctly selects strategy 78% of the times

Degree of Parallelism (Par):
- 1 replica = 100%
- 3 replicas = 33.3%
Average 10 Qs and 5 fault scenarios

Conclusions

- **Motivation**:  
  - Handling errors at Application-level
  - Exploiting Approximate Computing for DSS Workloads

- **Contributions**:  
  - Quality-of-Result Metric
  - Sensitive Analysis
  - Fault-injection on a real DBMS (PostgreSQL)
  - Testing using real DSS workload (TPC-H)
  - Adaptive Dynamic Scheme to account for query sensitivity and fault rate & Tool

Adaptive Combining increased QoR to almost best (Oracle) with gains in degree of Parallelism compared to alternative Static schemes and no a-priori knowledge
Thank you!

CASPER GROUP
Computer Architecture Systems Performance Evaluation Research
www.cs.ucy.ac.cy/carch/casper