Inconsistency Robustness for Scalability in Interactive Concurrent-Update In-Memory MOLAP Cubes

David Ungar, Doug Kimelman, and Sam Adams
IBM Thomas J. Watson Research Center
{davidungar,dnk,ssadams}@us.ibm.com

Abstract

We address inconsistency in large-scale information systems. More specifically, we address inconsistency arising out of non-determinism which we introduce, in place of synchronization, in an effort to achieve scalability on massively-parallel low-latency systems. To the extent that an application remains acceptable to its end users with respect to their business needs, we aim to permit – or no longer attempt to prevent – inconsistency relating to the order in which concurrent updates and queries are received and processed. Our long-term goal is to find algorithms that let us probabilistically bound the duration and amount of inconsistency, yet achieve scalability by using a minimum of synchronization, if any. In our initial experimentation, we have chosen a problem drawn from the general field of Online Analytical Processing ("OLAP"): interactive concurrent-update in-memory MOLAP cubes. As input values are updated, our system allows the invalidation of previously-cached computations to proceed in parallel *and unsynchronized* with the computation of new values to be cached. This lack of synchronization sometimes results in caching *stale* values. Such values are already out of date and hence inconsistent with the values of the inputs on which the computation is based. We introduce the notion of a *freshener:* a thread that, instead of responding to user requests, repeatedly selects a cached value according to some strategy, and recomputes that value from its inputs, in case the value had been inconsistent. Experimentation with a prototype showed that on a 16-core system with a 50/50 split between workers and fresheners, fewer than 2% of the queries would return an answer that had been stale for at least eight mean query times. These results suggest that tolerance of inconsistency can be an effective strategy in circumventing Amdahl's law.

1. Introduction

We are investigating inconsistency robustness as a means of achieving scalability on massively-parallel systems. Our initial focus is on computationally demanding business analytics applications, with an eye towards scaling to systems with petabytes of memory and tens of thousands of processor cores. These applications will need to respond rapidly to high-volume high-velocity concurrent request streams containing both queries and updates. We believe that synchronization will become prohibitively expensive when scaling to these levels of parallelism. Even if synchronization operations themselves were to remain efficient, Amdahl's law would limit the scalability of performance [2]. Thus, unlike others who seek the smallest feasible amount of synchronization to guarantee consistency, we take the extreme position of eliminating synchronization entirely. Without synchronization, errors will arise from data races. We propose to repair errors instead of preventing them. In other words, such a system will sometimes deliver inconsistent results. The key to usability will lie in bounding the frequency and magnitude of the errors, and the longevity of the inconsistent results.

Business analytics broadly is a source of ever more challenging performance and scalability requirements. Within business analytics, financial performance management is one example of a demanding business application domain. On a regular basis, e.g. quarterly, enterprises engage in financial planning, forecasting, and budgeting. End-users explore the enterprise's finances, compare plans to actual results, construct financial models, and experiment with what-if scenarios, refining the plan in an iterative fashion. As deadlines approach in a globally distributed enterprise, thousands of users could be collaborating remotely. Some users review continually-updated high-level consolidated financial performance measures aggregated from tens of gigabytes of data. Other users perform localized detailed data entry and review, continually refining the data as they strive to meet targets. Ultimately, they all converge to a finalized plan.

One advanced technology that can support this kind of financial performance management scenario is: interactive (or “real-time”) concurrent update (or “write back”) in-memory MOLAP cube technology, such as that provided by IBM Cognos TM1 [3]. Distinguishing features of this technology vs. other forms of Online Analytic Processing (“OLAP”) technology, such as ROLAP, HOLAP, and others described in [1], include: users can change previously-entered cube data values on the fly and can enter new data values into the cube; those changes will be reflected in computed or aggregated data values whenever the user next requests that data values be recalculated; query response time is sufficient for interactive use; and, multiple users can access a set of linked cubes concurrently. To support these features, the implementation keeps all data in memory, does not pre-compute aggregations or materialized views, calculates and caches aggregated and computed values on demand, and employs highly efficient and compact data structures to represent the highly multidimensional and sparse data sets.

This style of cubing is one of the problem areas we have chosen for prototyping and experimentation, to apply, refine, and validate our approach to inconsistency robustness for scalability. Although our initial work is specific to this style of OLAP cubes, and only constitutes initial steps in that direction, our results are encouraging, and we feel that it is likely that ultimately our work will generalize to other application domains.

The rest of this paper is organized as follows: First we review some of the basic concepts of our style of OLAP cubes (throughout the paper we will adopt terminology driven by the style of cube we are implementing). We then describe our techniques, our prototype, and our experimental results. We present four experiments that investigate: scalability, the creation of inconsistent results, the remediation of inconsistent results, and a whole system combining creation and remediation. Finally, we briefly discuss related work, point out some next steps in our work, and present our conclusions.

2. Cube Concepts

From the perspective of an end-user who is working interactively with our style of cube, the cube is somewhat like a multidimensional spreadsheet.

Figure 1 [4] depicts a simple cube. There is a Time dimension (vertical), a Product Category dimension (depth), and a “Measure” dimension (horizontal). Each dimension has an axis consisting of a number of values, such as “January”, “February”, etc. for the Time dimension. In an n-dimensional cube, each coordinate n-tuple in the Cartesian product of all of the axes, e.g. (Sales Amount, January, Diesel), uniquely identifies a cell of the cube, and each cell of the cube can hold a data value. Some cells hold values entered by a user, and other cells hold values that are computed according to formulae and aggregation structures that are part of the definition (or “schema” or “metadata”) of the cube, as we discuss in greater detail below. Further, in practice, cubes tend to be very sparse – many cells hold no value.

2.1 Hierarchical Axes

Rather than the axes of a cube being simply linear, the values along each axis of a cubes can in fact be arranged into a hierarchical structure.

Figure 2 [5] depicts a different cube. It has three dimensions: Time, Source, and Route. (Measures are depicted In Figure 2 by vertical subdivision of each cell. With some OLAP systems, measures might in fact be implemented by a fourth dimension, in others they might not. That discussion is beyond the scope of this paper.) A hierarchy is shown in Figure 2 for the values of each dimension’s axis. It is entirely likely that queries will occur requesting the value of a cell whose coordinate in at least one dimension is not a

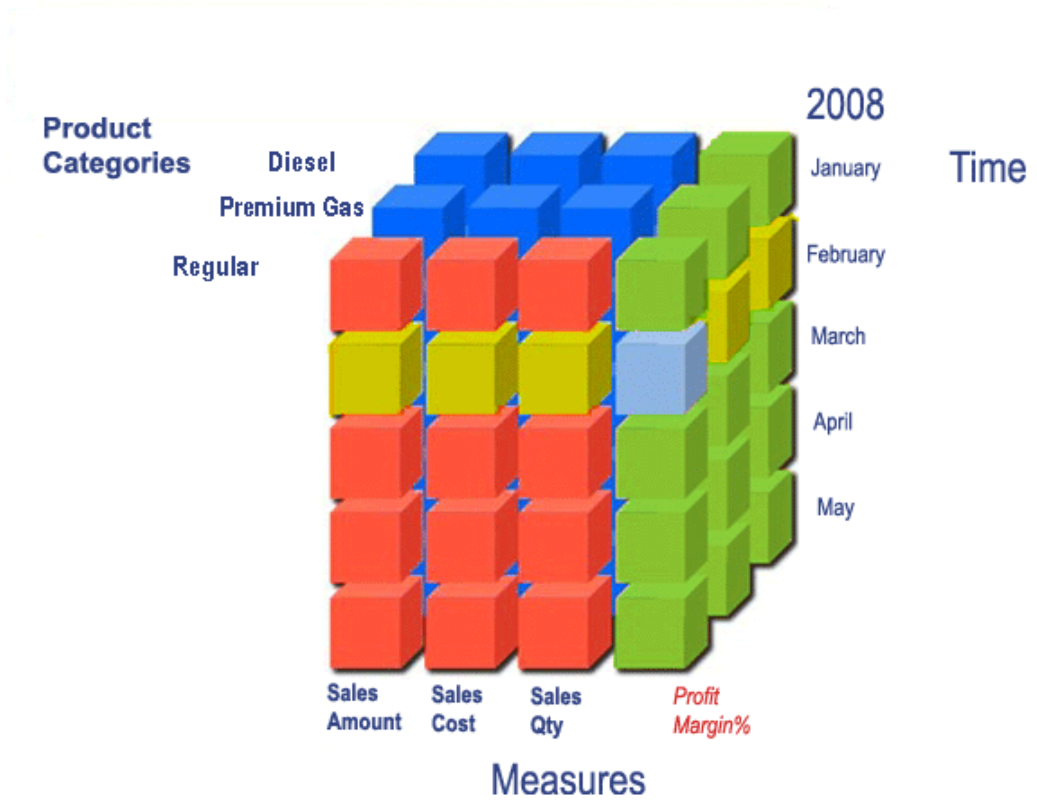


Figure 1: A simple cube [4].

leaf of that dimension's axis hierarchy. The hierarchies generally imply "aggregation" or "consolidation" – the value corresponding to an internal node of an axis hierarchy is often the (possibly weighted) sum of the values corresponding to the child nodes of the internal node. For example, the value of the cell (Route = air, Time = 1st half, Source = Western Hemisphere, Measures = Packages) would be 8916, the sum of the values 3056, 4050, 600, and 490 – those being the values of the cells having coordinates "1st quarter" and "2nd quarter" (the children of "1st half") in the Time dimension and "North America" and "South America" (the children of "Western Hemisphere") in the Source dimension. The definition of the cube axes specifies the aggregation corresponding to each dimension.[1]

Another way in which a cube definition specifies computation based on the structure of the cube is with formulae.

Figure 3 shows a formula for yet another example cube – one having a dimension named "Measure", as well as dimensions "Date", "Fishtype" and "Market". The Measure dimension axis has values including: "Purchase Cost - LC", "Quantity Purchased - Kgs", and "Price/Kg - LC" (presumably this cube contains data concerning purchases of fish at various times in various markets). The formula in Figure 3 specifies, among other things, that the value of the cell at coordinates

(Measure = "Purchase Cost – LC", Date = "January", Fishtype = "Cod", Market = "Fairbanks")

would be calculated by evaluating the expression that multiplies the value of the cell at coordinates

[1] In fact, a number of different hierarchies can be specified for each dimension, each providing a different aggregation structure for the leaf values of the axis. We address that generalization in follow-on work.

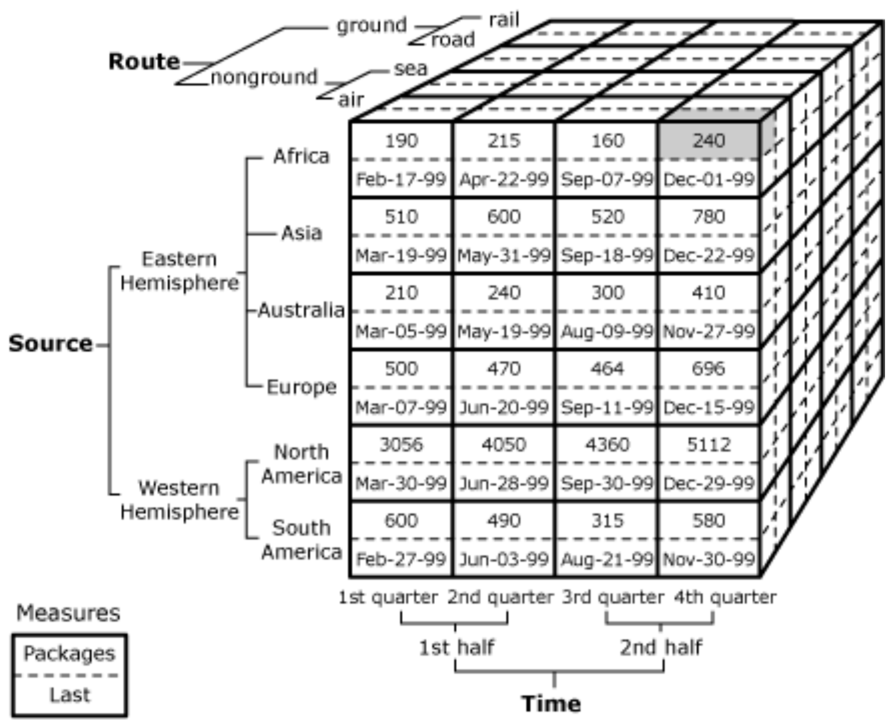


Figure 2: A cube with hierarchical axes [5].

['Purchase Cost - LC'] = N: ['Quantity Purchase - Kgs'] * ['Price/Kg - LC'];

Figure 3: A rule file specifying cube calculations.

(Measure = “Quantity Purchased – Kgs”, Date = “January”, Fishtype = “Cod”, Market = “Fairbanks”)

by the cell at coordinates

(Measure = “Price/Kg - LC”, Date = “January”, Fishtype = “Cod”, Market = “Fairbanks”).

In other words, for a given date, fishtype, and market, a purchase cost in local currency is obtained by multiplying the quantity purchased by the unit price. The formula concerns a hyperslice, not just a single cell, and the system fills in specific coordinates appropriately to calculate the value of a given cell of the hyperslice. For the sake of brevity, we omit discussion of many of the subtleties of the formula in Figure 3, as well as a more comprehensive discussion of computation specification for cubes. Please see [1] for much more detail.

In the interests of space, we also omit discussion of decomposing cubes into distinct (typically lower-dimensional) linked cubes, and discussion of the various ways in which a user can view and manipulate a multidimensional cube via a two-dimensional spreadsheet-like user interface, with actions including: slicing, dicing, stacking, and pivoting, and rolling up and drilling down. Those topics are not germane to the discussion within this paper. Again, please see [1] for a detailed discussion of these topics and many more.

The key point here is that both hierarchies and formulae give rise to flows of data within the cube. In fact, in some sense, the cube can be regarded as a quite regular array of interwoven and intersecting data flow networks.

2.2 OLAP Implementation Issues

The computation arising out of aggregation and formula evaluation can be a bottleneck for an application based on our style of cubes, with large amounts of data, large numbers of users, and high volumes of read and write requests. In order to optimize computation, a cube implementation can cache the values of computed cells. In any case, ever growing workloads have produced a desire to harness multicore CPUs and clusters. The combination of caching and parallelism has led to a need for synchronization, which in turn can limit scalability. We believe that this situation represents an opportunity for an implementation of our style of cubes that can harness more parallelism by tolerating a bounded amount of inconsistency.

3. Experimental Methodology

In order to investigate whether an inconsistency-tolerant approach to our style of cubes might be fruitful, we have built a prototype implementation with which to experiment.

3.1 Prototype

Our prototype models a cube as an unordered collection of cells, each represented by a Smalltalk object. A cell contains its coordinates in the cube, represented by a tuple locating the cell in N-space. For example, in a three-dimensional fish-market cube, the cell referring to the amount of cod sold on April 15, 2000 would have a tuple with three coordinates: (fish = /cod, measure = /amount, date = /2000/Q2/April/15). Each element of the tuple represents a location in one of the dimensions of the cube, and, since each dimension is in general hierarchical, each coordinate is represented by a sequence giving a path from the root of the hierarchy. In our example, the coordinate sequence "/2000/Q2/April/15" represents a path through the hierarchical *date* dimension down to a leaf of the axis of the dimension. This example cell would hold a number that was entered by a user, and so we call it an *entered (leaf)* cell. In contrast, the cell containing the aggregate amount of cod sold in the year 2000 would have coordinates (fish = /cod, measure = /amount, date = /2000). It has a date coordinate that is not a leaf. This latter cell is called a *computed* cell, because when queried for its contents, that cell will return a *computed* value: the sum of all of the entered cells containing amounts of cod for 2000.

We further distinguish between three kinds of computed cells:

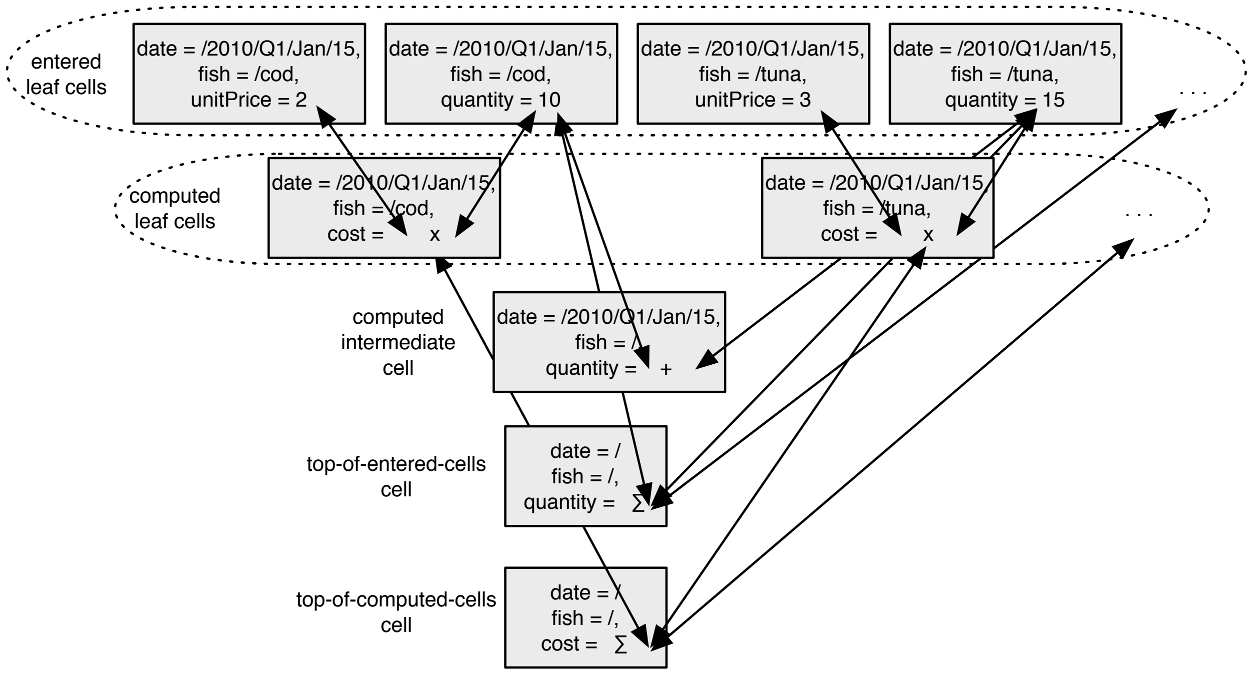
1. *top cells,* which are cells at the top of the hierarchy in every dimension except the measure dimension;
2. *intermediate cells,* which are computed cells in intermediate levels of a hierarchy; and
3. *leaf cells,* which are computed and entered cells at the bottom level of a hierarchy. At this level, each coordinate represents a leaf in its dimension. These include *computed* leaves, which are possible as the result of formulae.

In our particular cube, there are two sorts of top cells: *top-of-entered-cells* are the two top cells that represent the sum over all time and all fish of either the amount of a sale or the unit-price of a sale,[2] and the (single) *top-of-computed-cells* cell, which sums the total cost of a sale over all time and all fish. In the first case, the arguments to the summation are simply quantities entered by the user, but in the second case, they are products of pairs of unit-price and quantity cells (as we illustrate in greater detail below). Thus we differentiate cells in two different ways: there is a dichotomy between *entered* and *computed* cells, where computed cells may either be aggregates or formulae results; and we speak of *leaf*, *intermediate*, and *top* cells. Table 1 below gives examples of these cells.

2 Of course, the sum over all time and all fish of the unit-price of a sale might not be useful. The potential for useless aggregation is a well-known aspect of designing cube applications and is discussed in the literature.

Table 1. Examples of various kinds of cells in our prototype				
our term for this kind of cell	contents	fish coord.	date coordinate	measure coordinate
entered leaf	contents entered by user	/cod	/2002/Q2/April/15	/unitPrice
computed leaf	For cod on April 15, unitPrice * quantity	/cod	/2002/Q2/April/15	/cost
(computed) intermediate cell	total cost of all fish in April	/	/2002/Q2/April	/cost
top of entered cells	total quantity of all fish for all time	/	/	/quantity
top of computed cells	total cost of all fish for all time	/	/	/cost

The representation employed by our prototype implementation includes both forward and backward pointers so that each computed cell can find all of the cells that its contents depend on, and so that each cell can find all of the computed cells depending on it. (See figure 4). When adding an aggregation (computed) cell, our prototype links it to all of the entered cells on which it ultimately depends, rather than just link it to immediate lower-level aggregations. This choice simplified the implementation, especially with respect to the addition of newly materialized intermediate-level cells. Its effect on performance will vary with usage patterns.



entered leaf cells
date = /2010/Q1/Jan/15, fish = /cod, unitPrice = 2
date = /2010/Q1/Jan/15, fish = /cod, quantity = 10
date = /2010/Q1/Jan/15, fish = /tuna, unitPrice = 3
date = /2010/Q1/Jan/15, fish = /tuna, quantity = 15
. . .
computed leaf cells
date = /2010/Q1/Jan/15, fish = /cod, cost = x
date = /2010/Q1/Jan/15, fish = /tuna, cost = x
. . .
computed intermediate cell
date = /2010/Q1/Jan/15, fish = /, quantity = +
top-of-entered-cells cell
date = / fish = /, quantity = ∑
top-of-computed-cells cell
date = / fish = /, cost = ∑

Figure 4: Cells and interconnections in our prototype cube.

Since a cube may contain many computed cells that are not of interest at any given time, it seems reasonable to compute the contents of any computed cell *lazily;* that is, to defer the computation until the value is needed by a user. Furthermore, since a user may examine the value of a computed cell many times before there is a change to the contents of any of the cells on which that computed cell depends, it also seems reasonable to cache the contents of a computed cell. Thus, a serial implementation, or a naive unsynchronized parallel implementation of our OLAP cube includes the following algorithms, as illustrated in Figure 5: When a computed cell is asked for its contents, if the value is not already cached, the cell computes the value, caches it, and returns it. When the contents of an entered cell are changed, the cell empties (i.e. "invalidates") the caches of all of the computed cells that (transitively) depend upon it. (The invalidation propagation need not continue transitively beyond a computed cell that is already invalid.)

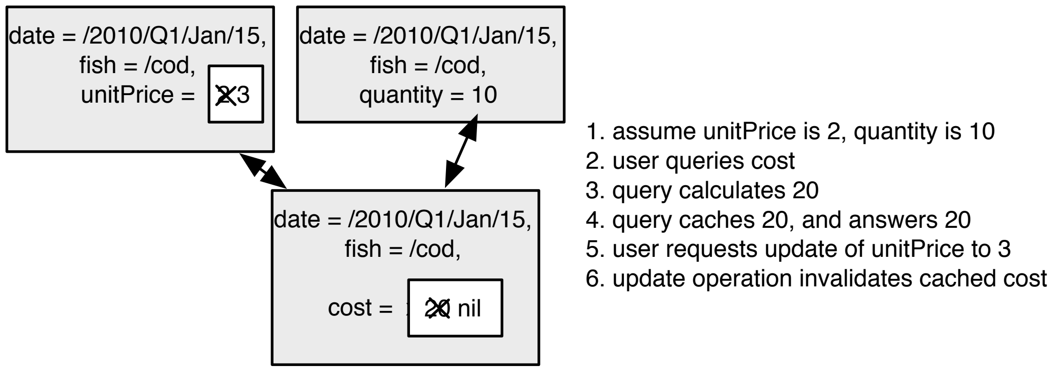


Figure 5: Scenario in which the user changes the price from 2 to 3 at step 5

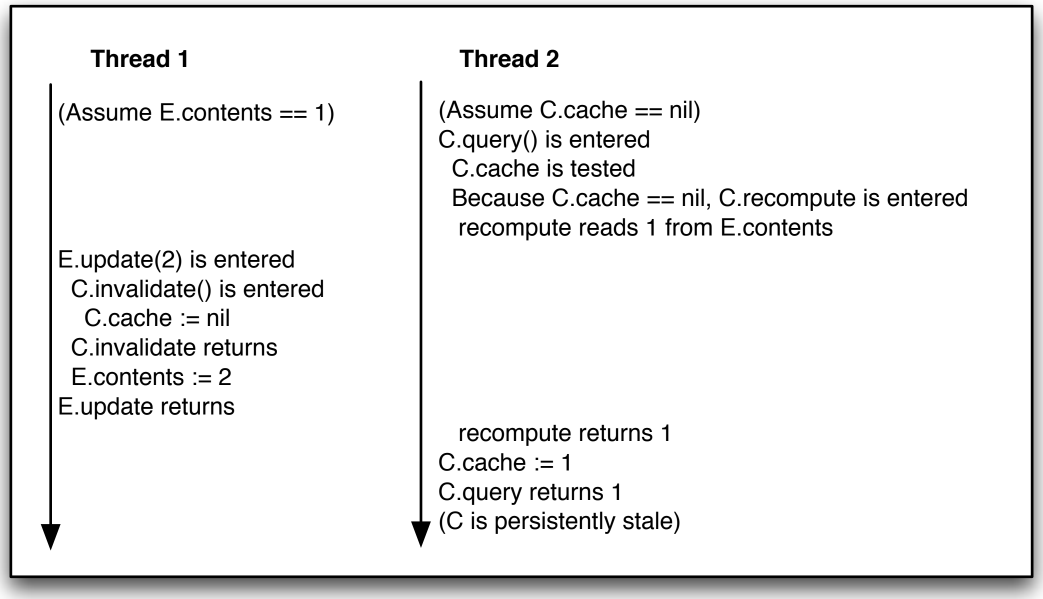
If queries and updates can happen concurrently, it is possible in the absence of synchronization for a computed cell to cache a result that is *inconsistent* with the source data (as we show in greater detail below). For instance, a computed cell could return the value 3 even though both of the entered cells it aggregates contain the value 1, possibly because one of those entered cells previously contained the value 2. What's worse is that this inconsistency could persist forever. We illustrate with a simplified example consisting of an entered cell E, and a computed cell C, where C's value is supposed to be the same as E's value. We assume two public operations, query, which returns the cell's value, and, for entered cells only, update, which sets the cell's value to its argument. Here is the pseudo-code for our cells:

```
class EnteredCell extends Cell {
   private int contents;
   int query() { return contents; }
   void update(int x) {
      for all dependent cells, c:
         c.invalidate();
      contents := x; // (The problem does not essentially change if this line is at the top of this function.)
   }
}

class ComputedCell extends Cell {
   private int cache := nil;
   void invalidate() { cache := nil; }
   private int recompute() {
      // recompute my contents by querying the cells I depend upon
      ...
   }
   int query() {
      if (cache == nil)  cache := recompute();
      return cache;
   }
}
```

Figure 6: Pseudo-code for Naive Cube

Although this pseudo-code is quite natural for serial operation, it will not work reliably when one thread's query to computed cell C overlaps another thread's update to entered cell E: Consider what happens if the query of C's value causes its recomputation, said recomputation reads E's value before it is updated, and the invalidation of C's cache caused by updating E's value occurs *during* the recomputation of C, i.e. before the end of the recomputation. In that case, the recomputation will be based on E's old value, yet the result will be stored in C's cache, as if it were current. Since C's cache will seem to be current, all future queries to C (assuming no more updates to E), will yield an obsolete, inconsistent result. This scenario is exemplified in Figure 7 below:



```
Thread 1                           Thread 2

(Assume E.contents == 1)           (Assume C.cache == nil)
                                   C.query() is entered
                                     C.cache is tested
                                     Because C.cache == nil, C.recompute is entered
                                      recompute reads 1 from E.contents
E.update(2) is entered
  C.invalidate() is entered
    C.cache := nil
  C.invalidate returns
  E.contents := 2
E.update returns
                                      recompute returns 1
                                   C.cache := 1
                                   C.query returns 1
                                   (C is persistently stale)
```

Figure 7: Sequence producing a persistently stale result

In the problematic scenario above, cell E is changed from 1 to 2, but (unless there are other invalidates of C), C.query() will continue to return the inconsistent stale value of 1. We call this unbounded inconsistency *persistent staleness.*

Standard design practice has been to eliminate this inconsistency with a variety of locking schemes: for example a user who desired to modify entered data would obtain a lock that would prevent others from querying or updating the contents of computed cells. Such a solution limits scalability.

We are examining approaches that tolerate but limit inconsistency. The rest of the paper presents the results of four experiments: one that verifies the scalability advantages of inconsistency, a second that examines an optimization to reduce the creation of inconsistency, a third that examines a first attempt to limit the longevity of inconsistency, and a fourth that combines reduction of inconsistency creation with limitation of inconsistency longevity. Each experiment has been conducted on both the Mac and the Tilera platforms.

In addition to problems arising out of leaf cell value changes, there is also an issue of dynamic structural changes within the representation of the cube. For example, when the user adds values for a new time period, the cube must grow and the data-propagation constraints must adapt. Dynamic structural changes are also required when previously unpopulated cells of a sparse cube are filled in. These issues lie outside the scope of the present work.

4. Experiments and Results

We have only just begun to investigate the utility of achieving scalability through tolerating inconsistency. Accordingly, the data from the experiments may be regarded as suggestive, but likely not dispositive. In this paper, as we indicated above, we report on four experiments: one on scalability, one on inconsistency creation, one on inconsistency remediation, and one that combines elements of the previous two.

Our experimental framework proceeds as follows:

- A configuration of independent variables is selected, for example, setting the number of worker threads,
- a new cube is generated with random initial values,
- the test code is run for five seconds,
- the dependent values are recorded.

This process is repeated for different configurations of independent variables.

Because random numbers were involved, each experiment was run twice to get a feel for the potential variation, and each graph below plots the results from both runs.

The platform for our experiments consisted of two distinct machines and a Smalltalk runtime:

- *Hardware.* All of our experiments ran the same code on two different machines: 1) an 8-core, 16 hyperthread Mac Pro [6] with two 2.94 GHz Intel Xeon 5500 series processors, and 2) a 64-core Tilera TILE64^{TM} CPU [7] with a 700 MHz clock, and 64KB Level-2 combined I and D cache. Given the Tilera's slower clock speed and smaller caches, it is no surprise that we observe its single-core performance to be significantly lower than the Mac's. The Mac Pro's Intel CPU represents a *multicore* design, with a relatively small number of cores, large caches, and extensive hardware support for cache-coherency. The Tilera CPU represents a *manycore* design, with far more cores, smaller caches, and less hardware support for cache coherency. We report on both systems in order to assess the results on both styles of architecture.
- *Runtime System.* Our prototype is written in Smalltalk and runs on the RoarVM [8, 9], a Smalltalk virtual machine that has been written to exploit the parallelism of multi- and many-core CPUs. When there are enough hyperthreads (or cores) available, each Smalltalk thread runs concurrently with the others. Each thread runs in a common, shared object heap. The RoarVM provides a primitive operation that returns a fine-grained time value on each machine. On the Mac, we use the Microseconds API, on the Tilera, we use the hardware cycle counter facility.

4.1 Scalability Experiments

The scalability experiments examine the scalability advantages of inconsistency. Scalability is compared between an algorithm that permits inconsistency and one that does not, with the number of worker threads varying between one and the number of cores on the machine. For the scalability experiments, we arbitrarily chose the following mixed workload:[3]

[3] At this stage in our work, we make no claims about the representativeness of our workloads. We are in the process of obtaining workloads representative of production usage of OLAP engines.

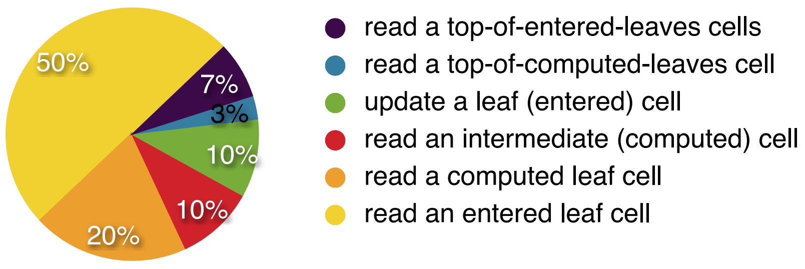


Figure 8: The workload used for measuring scalability

For each run, we measured the total number of queries and updates performed in each five-second run. In the four scaling graphs below, we plot the number of operations completed in five seconds, summed across all worker threads, vs. the number of worker threads employed. On each machine, we limit the maximum number of threads to be the number of available cores. Although the Mac Pro supports hyperthreading, the additional complexity of thread scheduling and interference would limit our ability to interpret those results.

Figure 9 compares the scaling of the unsynchronized nave implementation (allowing unlimited inconsistency) with a simple locking implementation that allows multiple readers or one writer at a time (sacrificing scalability in order to achieve total consistency). Each graph shows four runs of the unsynchronized cube, shown as solid lines, and four runs of the locking cube, shown as dotted lines. As expected, the locking implementation hits a scaling wall beyond four cores on the Mac and eight cores on the Tilera. Also as expected, the Tilera system is slower, core for core.

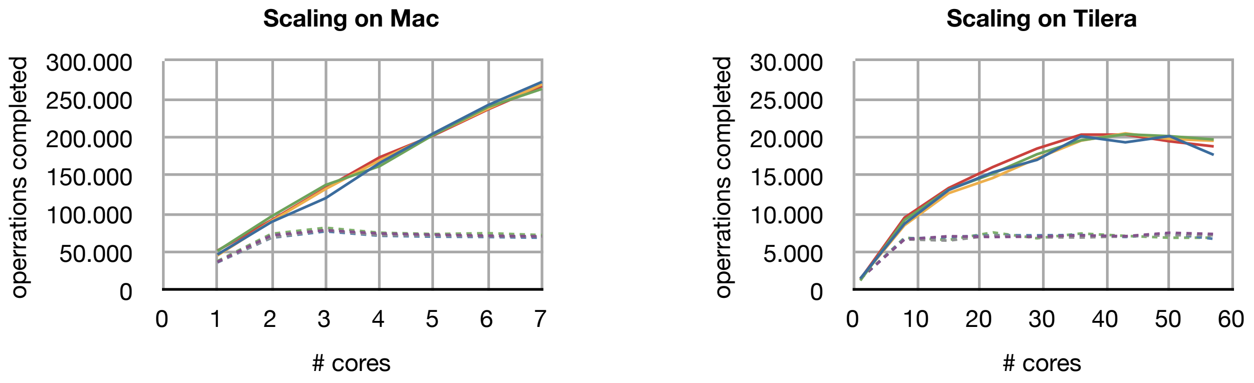


Figure 9: Cube scaling

From these measurements, we see no surprises: a fully synchronous system does not scale, but one that does not synchronize does scale. We have not yet investigated why the Tilera-hosted system plateaus around 40 threads.

4.2 Inconsistency Creation Experiments

The inconsistency creation experiments examine the rate of creation of inconsistent results, as the number of worker threads increases. For the purposes of this experiment, each worker thread splits its operations 50/50 between changing the contents of a randomly-chosen entered cell and querying the (cached) contents of an intermediate cell. As described above, the intermediate cell's cache is invalidated whenever a cell upon which it depends is changed.

For this experiment, we are most concerned with the creation of *persistent staleness,* that is the situation described above in which a computed cell's cache is left with its *valid flag* set, but contains an obsolete result which may never be corrected. We test an optimization to the basic code outlined above — any thread that invalidates a cache leaves behind a “breadcrumb”, so that if a cache has been invalidated while being recomputed, the *valid flag* is not set when the recomputation completes. Figure 10 illustrates the code including the optimization, which is shown in italics. (The *valid flag* is represented by a non-nil value in the cache.)

```
class EnteredCell extends Cell {
    private int contents;
    int query() { return contents; }
    void update(int x) {
        for all dependent cells, c:
            c.invalidate();
        contents := x;
    }
}

class ComputedCell extends Cell {
    private int lastThread := nil;
    private int invalidate() { cache := nil; lastThread := thisThread(); }
    private int recompute() {
        lastThread := thisThread();
        // recompute my contents by querying the cells I depend upon
        ...
    }
    private int cache := nil;
    int query() {
        if (cache == nil)  x := recompute();
        if (lastThread == thisThread()) cache := x;
        return x;
    }
}
```

Figure 10: Pseudo-code including staleness creation optimization

Persistent staleness can still occur, if an invalidation occurs between the time ComputedCell.query() compares lastThread to thisThread() and the time it sets cache := x, because we did not lock / synchronize at that point. But because this is a much smaller window in time, we expect this to occur much less frequently than the “race” that causes persistent staleness in the unoptimized code. This experiment measured the frequency at which the computed cell query operation returned a permanently stale value. Figure 11 shows these results. Each graph shows four runs without the optimization described above, depicted as dashed lines, and four runs with the optimization, depicted as solid lines.

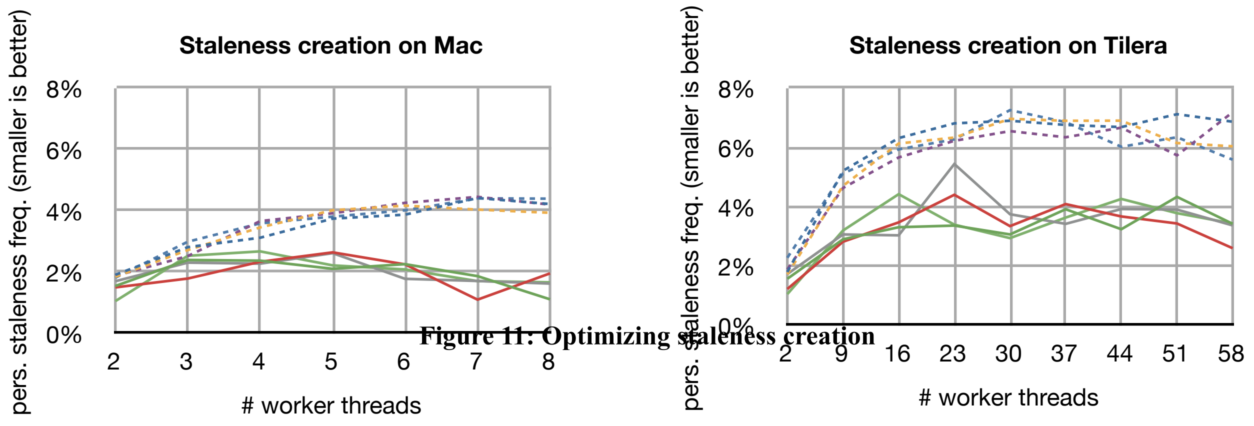


Figure 11: Optimizing staleness creation

As expected, adding worker threads generally increases the rate of creation of inconsistent (i.e., stale) results. Also, the rate of staleness creation is reduced by the optimization of not marking the cache valid if another thread has invalidated it or begun recomputing it since the time that we began our recomputation.

4.3 Inconsistency Remediation Experiments

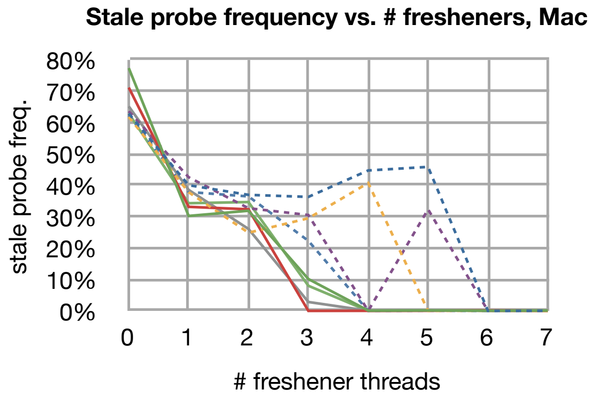
Our next experiment represents an attempt to remediate the inconsistency of results. We devote a number of threads to repeatedly selecting a cell, according to some strategy, and recomputing its value just in case it is a persistent stale value. We call these threads *freshener* threads. For this experiment, each worker thread uses half of its requests to change entered data *without invalidating dependent caches.* By omitting the invalidation, we create conditions under which inconsistent results are profligately created "artificially". This omission produces larger numbers of inconsistent results so that we can study remediation more effectively. The other half of each worker's requests are spent reading (and recomputing, if necessary) the values of randomly-chosen intermediate cells.[4] On the Mac, there is one worker thread while on the Tilera there are four, and on each machine we ramp up the number of freshener threads until all cores are in use.

What is measured is the frequency and duration of staleness of query results. For the purposes of this experiment, staleness is defined as follows:

- If a query result matches the result of a recomputation, there is no staleness.
- If a query result does not match the result of a recomputation, its staleness is the difference between the current time and the most recent time at which there was an update to an entered cell on which the computed cell depends.

We introduce a varying number of freshener threads that recompute and restore cache contents. By recomputing and restoring caches, even valid caches, the fresheners remove persistent staleness. In one strategy, each freshener chooses a cell to recompute at random, in the other it uses a round-robin pattern.

Figure 12 presents the frequency data: what proportion of queries returned a stale result. (Smaller is better in all graphs below.) Each graph shows four runs with random fresheners, depicted as dashed lines, and four runs with round-robin fresheners, depicted as solid lines.



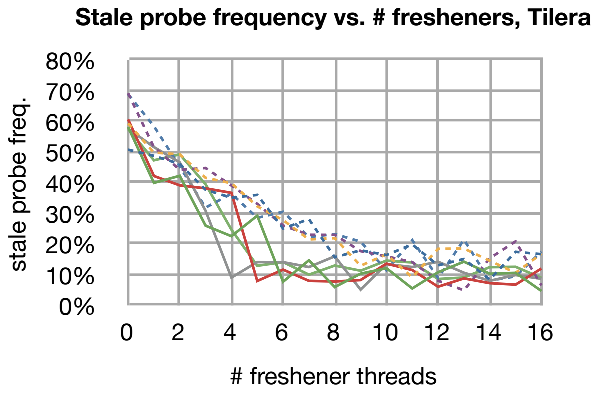


Figure 12: Reducing stale probe frequency with fresheners

[4] Top cells are very few in number with many cells feeding them; computed leaves are numerous with only two cells feeding each. We used intermediate cells in an attempt to measure a more typical case.

As expected, without any fresheners at all, the stale probe frequency is very high, in fact, one would expect it to be 100% in steady-state. As hoped-for, adding freshener threads generally reduces the frequency of stale results. The round-robin policy performs better than the random policy, perhaps because each freshener is guaranteed to hit every computed cell in a bounded amount of time.

Next, Figure 13 shows the staleness duration, normalized to the mean probe time of each run. The medians and 90%iles are taken only from the *stale* probes, not the complete population of probe results. As above, each graph shows the results of four runs with random fresheners, depicted with dashed lines, and the results of four runs with round-robin fresheners, depicted with solid lines.

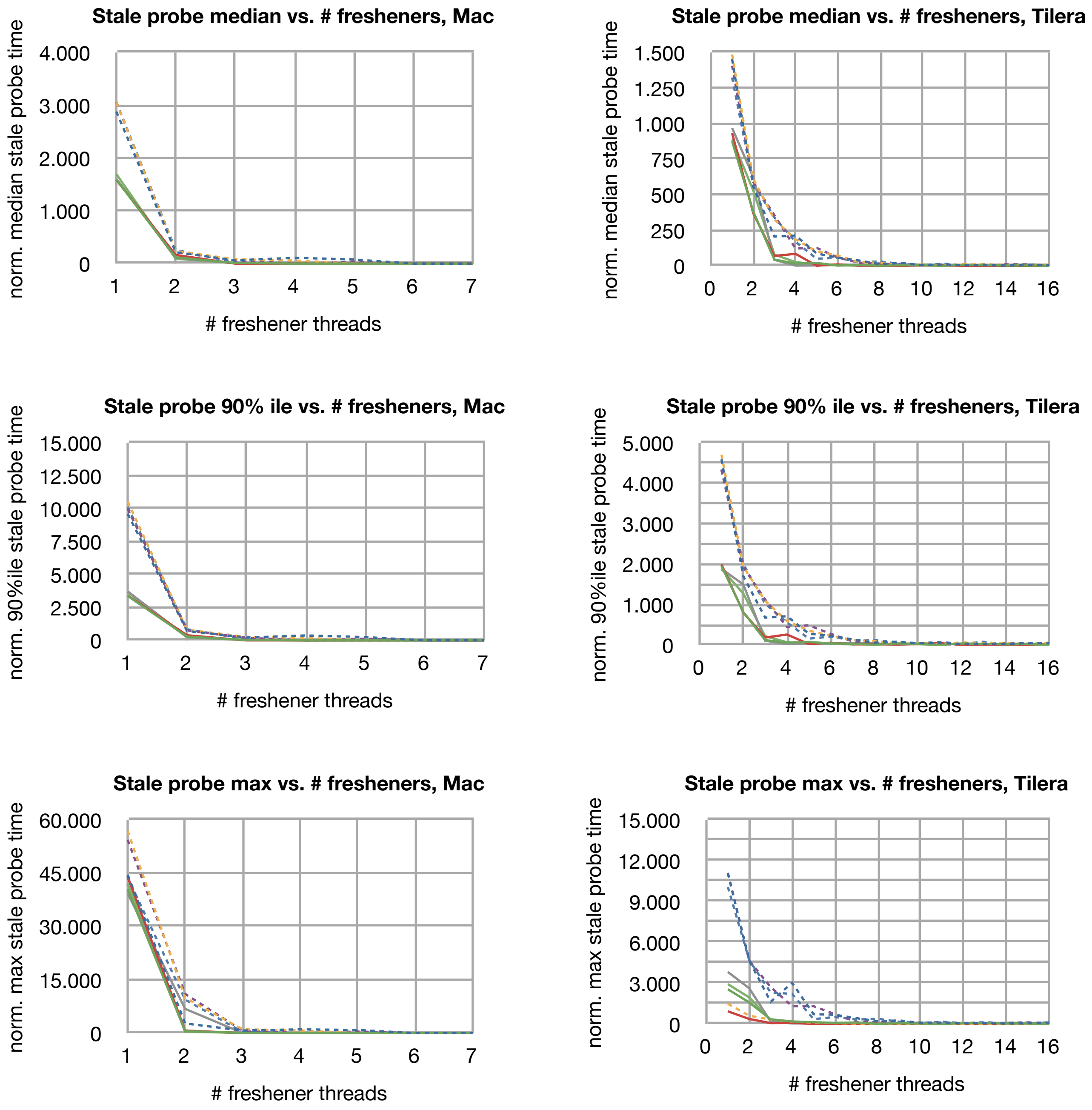


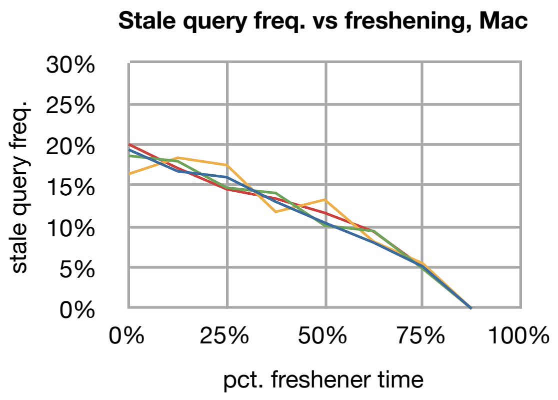
Figure 13: Staleness duration graphs

As expected, on either platform adding fresheners generally shortens stale times, with the round-robin strategy shortening them more quickly. The above graphs of median and 90%ile staleness for the Mac show an interesting anomaly: In Run 2, with random fresheners (shown by the yellow squares), the staleness increased from five to six to seven threads. As in the Mac data, the Tilera runs also include an anomaly: This one occurs for the Run 1 round-robin case at 13 fresheners. This sort of oddity is exactly why we report on more than one run. One would expect that over the course of five seconds, random effects would average out. But they do not always seem to do so. We do not yet know the cause of such effects, whether it be bad luck, garbage collection pauses, or something else.

4.4 Putting it together: Creation Plus Remediation

For the final experiment, we combined the best recomputation policy (the one with invalidation and the optimization described above, and *not* the artificial staleness creation of the previous experiment), with the best freshening policy (round-robin). The experiment, run on both platforms, employed a fixed number of cores, while varying the split between worker threads and freshener threads. In particular, on the eight-core Mac, the experiment varied the number of workers from one to eight, with the number of fresheners consequently varying from seven to zero. On the Tilera, time constraints limited us to 16 cores so the experiment varied the number of workers from one to 16, with fresheners varying from 15 to zero. For both platforms, the x-axis is reported in terms of the percent of cycles devoted to freshening: zero for no fresheners, 50% for 8 workers and 8 fresheners (on Tilera), 87.5% for 1 worker and 7 fresheners. Rather than attempt to model a realistic workload, each worker spent half of its operations updating an entered value (chosen at random), and the other half querying an intermediate cell.

The results of the queries were measured as in the previous experiments. The graphs in Figure 14 below report the stale probe frequency, the fraction of query results that were stale.Each graph shows the results of four runs.



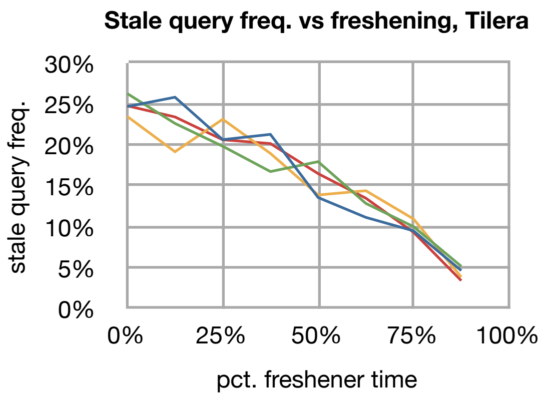


Figure 14: Frequency of stale query results vs. portion of cycles spent on freshening

Figure 15 addresses the staleness duration, normalized as before to the mean probe times for each run.Each graph shows the results of four runs.

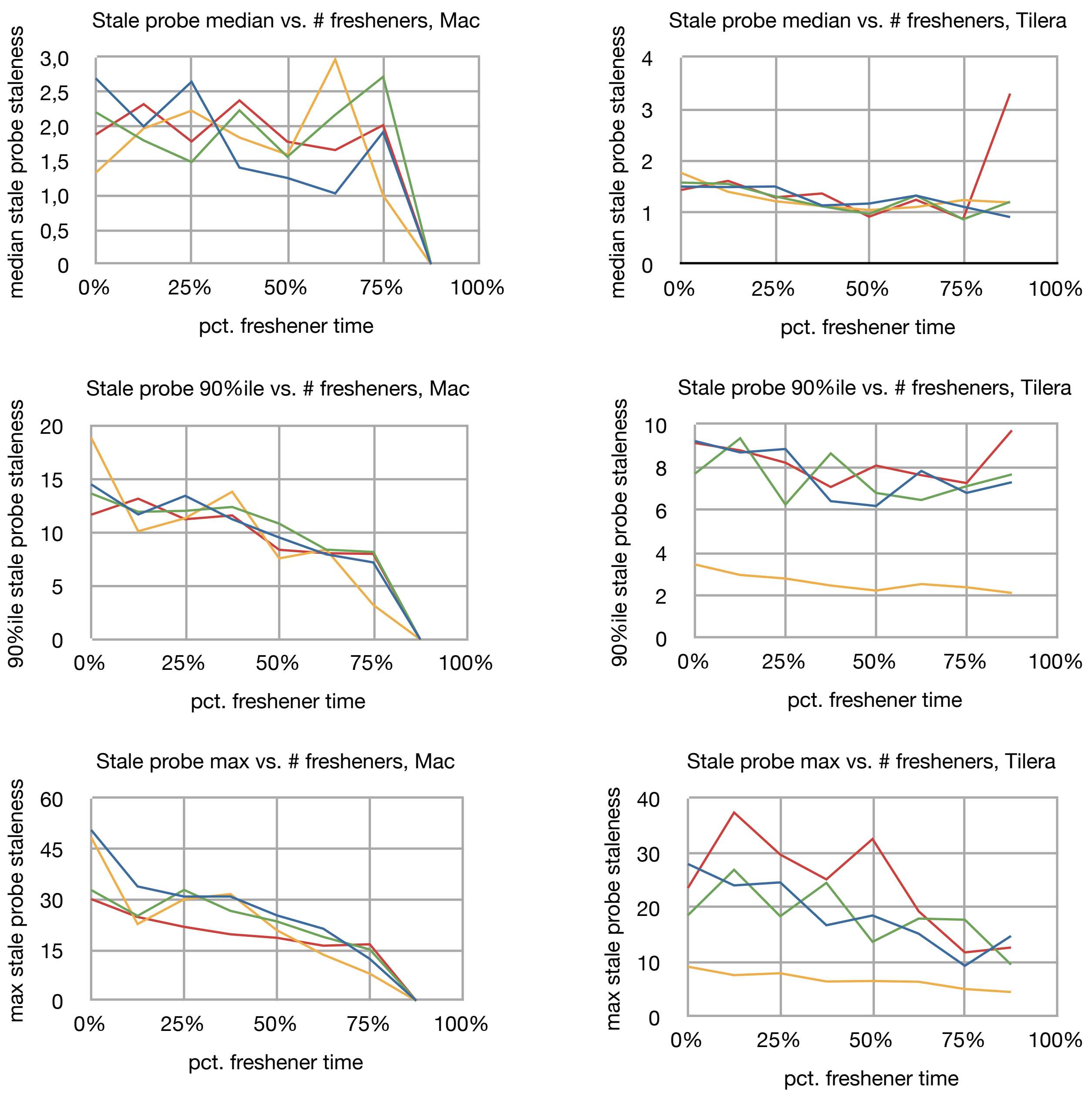


Figure 15: Staleness of stale query results vs. portion of cycles spent on freshening

Several observations follow from these results:

- The disparity between platforms of maximum staleness duration in some cases is striking.We have no explanation at present.
- Spending more time on freshening does not in general decrease the median or 90%ile staleness duration. For example, on Tilera, moving from 12.5% freshener cycles to 87.5% made no appreciable difference in the median time of the stale results (for three of the four runs) returned from stale probes. (It did decrease the frequency of stale results of probes from 24% to 4%, as shown in the frequency data above.)
- To repeat, these data are taken only from the distribution of stale results. For example, on Tilera, with a 50/50 split between workers and fresheners, the frequency graph indicates that under 20% of the probes

would return stale results. The 90%ile graph above shows that of these 20%, only 10% would be staler than 8 probe times. Thus, only 2% of the probes would return results staler than 8 probe times.

5. Summary and Discussion of Experimental Results

On each of two platforms – an eight-core Intel-based Mac Pro and a 64-core Tilera microprocessor – we measured four effects: the effect of a simple synchronization scheme on scalability; the effect of a straightforward optimization on the rate of stale result creation; the effect of a round-robin vs. a random freshening policy on the frequency and duration of stale results; and finally the effect of varying the proportion of cycles dedicated to freshening when combining the optimized creation algorithm with the round-robin fresheners.

On the Mac Pro, adding synchronization reduced scalability to four threads instead of seven, on Tilera it limited scalability to eight, instead of 35. This result supported our thesis that synchronization can limit scalability. (But see Threats to Validity below.) These results are pessimistic for synchronization, but we suspect that even cleverer synchronization schemes will limit scaling at some point, and we do expect systems with very large numbers of cores in the not-too-distant future.

On both of our platforms, the straightforward optimization succeeded in reducing the rate of creation of stale results. For example, on Tilera, it yielded almost a factor of two reduction when at least 30 worker threads were running. Only 4% of the cached results were stale.

On both of our platforms, adding round-robin freshener threads reduced the frequency and duration of stale results. Random freshener threads worked, too, but not as well as the round-robin ones. This experiment used a mutated update algorithm from which the cache invalidation had been eliminated, in order to create far more stale results than even the simple update algorithm used in the prior experiment. Even so, the round-robin freshener on Tilera limited the frequency of queries returning stale results to under 8%, once at least five freshener threads were used with the four (mutated) worker threads.

The final experiment was designed to illustrate the results of varying how many of a fixed number of cores were devoted to freshening rather than working. Combining the frequency with the duration data suggests that in a 16-core system with a 50/50 split between workers and fresheners, fewer than 2% of the queries would return an answer that had been stale for at least eight mean query times.

6. Next Steps

These experiments constitute our initial work in this direction, and the results to date have been encouraging. Next steps in furthering this work will include:

- Comparing to synchronized implementations that employ known more-efficient schemes.
- Running more realistic workloads. One question to be asked is: What would the freshening period for each cell be, in a cube with dozens of gigabytes of data, even with fresheners running on hundreds of cores? Another question is: Could we provide guidelines for setting freshener control parameters according to workload, or could such parameters be set dynamically by an adaptive system?
- Implementing our prototype in optimized C or C++, rather than atop an interpreter for Smalltalk.
- Using high-resolution timing consistently for all measurements across all platforms.
- Determining how suitable the architecture of our simple prototype would be for OLAP product settings.

7. Related Work

Kiviniemi et al. reduce the amount of recomputation on an OLAP cube by avoiding recalculation when the result would lie within a user-specified numeric tolerance [10]. Their mechanism reduces the amount of computation required, but does not attack the inherent conflict between scalability and synchronization, as they still make deterministic guarantees to the user.

PrediCalc [11] also deals with inconsistency in spreadsheets, but a different sort of inconsistency. That system deals with inconsistency that arises when for example, a user manually enters a computed value. Resolving inconsistencies created by users is a different task than that of resolving inconsistencies arising from race conditions in the code.

Epsilon Serializability [12, 13] was a technique developed to allow more concurrency, particularly for distributed databases, by allowing for a bounded amount of error to propagate. Unlike our work, the error was strictly bounded numerically.

Our work is partly inspired by the research of Martin Rinard. He has shown that, for some programs, much computation can be omitted or perturbed without creating unacceptable results [14].

8. Conclusions & Future Work

When is it right to give a wrong answer? When the wrongness is tolerable and permits the exploitation of the inexpensive, massive parallelism that future platforms will provide.

We believe that synchronization will become prohibitively expensive when scaling to these levels of parallelism. Thus, unlike others who seek the smallest feasible amount of synchronization to guarantee consistency, we take the extreme position of *eliminating synchronization entirely.* Errors arising from data races will be addressed by *repair* as opposed to prevention. In the experiments described here, races sometimes deliver a kind of inconsistent results that we call *stale* results. The key to usability will lie in ultimately bounding the frequency and longevity of such results. Our contributions to date include a technique we call *fresheners,* which reduce the frequency and longevity of stale results. Although this initial work is specific to a particular class of OLAP cubes, it is likely to generalize to other domains.

Our long-term goal is to find algorithms that let us probabilistically bound the duration and amount of inconsistency, while achieving scalability by eschewing synchronization. We have built a prototype that implements a kernel abstracted from a particular kind of OLAP cube engine, and we have used that prototype to show that permitting inconsistency can improve scalability. A simple optimization, tracking overlapping cache invalidation with recomputation, can help reduce the rate at which inconsistency is created. In order to reduce to tolerable levels the duration of the inconsistency that is created, we introduce *fresheners.* A freshener is a thread that, instead of responding to user requests, repeatedly selects a cached value according to some strategy, and recomputes that value from its inputs, in case it had been inconsistent. A round-robin policy for fresheners outperforms a random selection policy. Experimentation with our prototype suggested that on a 16-core system with a 50/50 split between workers and fresheners, fewer than 2% of the queries would return an answer that had been stale for at least eight mean query times. These results suggest that consistency can be traded off for scaling.

We may need a paradigm shift, away from programs that always produce consistent results, towards nondeterministic algorithms that exhibit a tolerable level of inconsistency.

We feel that we have only scratched the surface of allowing systems to scale by managing inconsistency. We expect to find better algorithms both for reducing the creation of inconsistent results and for returning inconsistent results back to consistency. For example, it may prove efficacious to guide fresheners to the stalest caches by using timestamps in a pheromone-based algorithm. We need to measure these improvements in runs against workloads that are representative of production usage, and in these scenarios we hope to achieve scalability far beyond what it possible with synchronized systems, while keeping staleness to a level that is tolerable by end-users of applications of interest. Nonetheless, our results to date suggest that tolerance of inconsistency can be an effective strategy in circumventing Amdahl’s law.

Acknowledgements

Mark Wegman has offered many helpful suggestions for improving both this paper and the work. In particular, it was he who first suggested that the round-robin discipline might merit investigation. Manny Perez has been exceedingly generous in making time to discuss OLAP with us.

Bibliography

[1] Thomsen, E. *OLAP Solutions: Building Multidimensional Information Systems*. Wiley, 2002.

[2] Hill, M. D. and Marty, M. R. Amdahl's Law in the Multicore Era. *IEEE Computer*, July (July, 2008 2008).

[3] IBM Cognos TM1, http://www-01.ibm.com/software/data/cognos/products/tm1/. Accessed February 2011.

[4] datanova Business Intelligence, http://datanovasoftware.com/petroplus_en.html. Accessed February 2011.

[5] BI SSAS - What is an OLAP cube?, http://biresort.net/blogs/pedrocgd/archive/2009/06/04/bi-ssas-what-is-a-cube.aspx. Accessed February 2011.

[6] Apple Corp. Mac Pro (Early 2009) - Technical Specifications, http://support.apple.com/kb/SP506. Accessed March 2011.

[7] Tilera Corp. *TILE64 Processor - Product Brief*. 2008.

[8] Marr, S. RoarVM - The Manycore SqueakVM, http://github.com/smarr/RoarVM. Accessed March 2011.

[9] Ungar, D. and Adams, S. S. Hosting an object heap on manycore hardware: an exploration. In *Proceedings of the DLS'09*. ACM, 2009.

[10] Kiviniemi, J., Wolski, A., Pesonen, A. and Arminen, J. Lazy Aggregates for Real-Time OLAP. In *Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery (DaWak'99)*. 1999.

[11] Kassoff, M., Zen, L.-M., Garg, A. and Genesereth, M. PrediCalc: A Logical Spreadsheet Management System. In *Proceedings of the 31st VLDB Conference*. 2005.

[12] Kamath, M. and Ramamritham, K. Performance Characteristics of Epsilon Serializability with Hierarchical Inconsistency Bounds. In *Proceedings of the Ninth International Conference on Data Engineering*. 1993.

[13] Pu, C. *A Brief Tutorial on Epsilon Serializability and a Survey of Recent Work*. Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

[14] Rinard, M., Cadar, C. and Hguyen, H. H. Exploring the Acceptability Envelope. In *Proceedings of the 2005 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2005.