Big Wins With Small Application-aware Caches [*]

Julio C. López [†]
jclopez@cs.cmu.edu

David R. O'Hallaron [‡]
droh@cs.cmu.edu

Tiankai Tu [§]
tutk@cs.cmu.edu

ABSTRACT

Large datasets, on the order of GB and TB, are increasingly common as abundant computational resources allow practitioners to collect, produce and store data at higher rates. As dataset sizes grow, it becomes more challenging to interactively manipulate and analyze these datasets due to the large amounts of data that need to be moved and processed. Application-independent caches, such as operating system page caches and database buffer caches, are present throughout the memory hierarchy to reduce data access times and alleviate transfer overheads. We claim that an application-aware cache with relatively modest memory requirements can effectively exploit dataset structure and application information to speed access to large datasets. We demonstrate this idea in the context of a system named the tree cache, to reduce query latency to large octree datasets by an order of magnitude.

1. INTRODUCTION

Abundant computational resources and advances in simulation techniques allow scientists to generate increasingly larger datasets. Users can cheaply store these datasets as the price of storage per MB continues to decline. For example, seismologists affiliated with the Southern California Earthquake Center (SCEC) [25] generate large datasets from simulation and seismic sensors. The sizes of these datasets range from a few gigabytes to terabytes [1].

In order to extract meaningful information out of these large datasets, scientists need to *interactively* handle and transform the data into a simpler form that is easier to understand. Tools, such as the CVM service [16] and the Grid Visualization Utility [8], allow scientists to query these datasets and discover features of interest in the data. For example, through the CVM service users generate images and explore the SCEC's 3D Community Velocity Model for Southern California (CVM) [17].

Ideally, scientists should be able to interactively analyze large datasets from their *desktop computers* whether the datasets are stored at a local or at a remote location. Supporting interactive access to these datasets is challenging because, as dataset sizes grow, query latency increases due to the large amount of data movement between various levels of the memory hierarchy [31].

Years of research have produced many general approaches to reduce access latency to large datasets by reducing I/O overhead and pooling multiple I/O and computing resources in tightly coupled systems to reduce processing time. Application-independent system-level caches, such as database buffers and operating system page caches, are the norm in modern computer systems. While these mechanisms do a good job reducing access latency, they are not sufficient for certain interactive applications. For example, the CVM service, using state of the art techniques implemented in the CMU etree library [28, 29], often takes in the order of minutes to satisfy a request for an image. Ideally, it should take in the order of a few seconds to satisfy a user's request. The question is: *How can we reduce access time to large dataset even further in order to support interactive applications?*

Our approach is to use a small application-aware cache to speed up access to large datasets in interactive applications. The main idea is to set aside a relatively small portion of the system caches memory and use it to implement a cache that exploits dataset-specific structure and application-level information to reduce query latency. This idea has been used in other contexts, such as databases [9] and distributed object systems [10, 14]. Section 3 explains in more detail.

As a proof of concept we have implemented this idea in a system called the *tree cache*. The tree cache reduces query latency to large octree datasets by implementing the following application-level techniques: fine-level caching of individual octants, approximate-value queries, and query reordering.

We evaluate the tree cache in the context of queries to the SCEC's CVM dataset. Our evaluation shows that the tree cache reduces average query time by an order of magnitude over the case when only system-level caches are used.

[*]This work is sponsored in part by the National Science Foundation under Grant CMS-9980063, in part by a subcontract from Southern California Earthquake Center as part of NSF ITR EAR-01-22464, and in part by a grant from the Intel Corporation.

[†]Department of Electrical & Computer Engineering, Carnegie Mellon University, Pittsburgh, PA USA

[‡]Computer Science Department and Department of Electrical & Computer Engineering, Carnegie Mellon University, Pittsburgh, PA USA

[§]Computer Science Department, Carnegie Mellon University, Pittsburgh, PA USA

SC2004 November 6-12, 2004, Pittsburgh, PA USA
0-7695-2153-5/04 $20.00 (c) 2004 IEEE

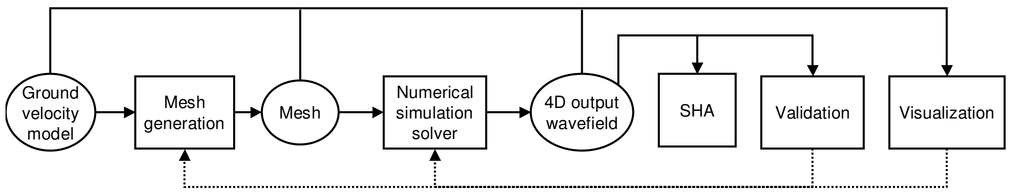

Figure 1: Physical simulation process

The rest of this paper is organized as follows: Section 2 presents the motivation for this work. Section 3 presents the previous work that the tree cache builds on. Section 4 provides background information about octrees. Section 5 describes our application-aware cache, the tree cache. Section 6 evaluates the effectiveness of the tree cache library.

2. APPLICATION CONTEXT

The tree cache is motivated by the desire to support data analysis in the SCEC Community Modeling Environment (CME) [25]. The goal of the CME is to enhance the understanding of how the Earth is structured and how the ground shakes during strong earthquakes. To achieve this goal, the CME effort is developing a common framework for ground motion simulation.

Figure 1 sketches the methodology for the physical simulation process in the CME. The boxes represent processes and the ovals correspond to datasets. An input *3D ground velocity model* contains properties of the ground for a region of the earth. A *mesh generation* produces a discrete *mesh* from the velocity model. A *numerical solver* simulates the propagation of waves the ground during an earthquake and outputs a *4D wavefield* dataset.

The analysis of these datasets has great value for scientists and the community in general. *Seismic hazard analysis (SHA)* performs risk evaluation for a given region based on output wavefields. The *validation* task compares the output wavefields against actual readings from seismographs. *Visualization tools* create visual representations of the datasets allowing users to find features of interest. The feedback from the validation task and visualization tools allows scientists to fine tune their models and simulations. Below, we describe two visualization tools, which are our motivating applications. They are the CVM service and the grid visualization utility (GVU).

2.1 The CVM service

The 3D community velocity model for Southern California (CVM) [17] is a dataset widely used in CME physical simulations. As part of the CME effort, we developed a capability named the CVM service that allows scientists to query the CVM dataset remotely. This service enables users to generate meshes and images of the CVM dataset through a Web browser.

The CVM dataset describes the entire Los Angeles basin with a spatial resolution of about 50 meters. It covers a 100km $\times$ 100km $\times$ 50km volume of the ground. A point at a position (x, y, z) has associated properties such as the density of the ground at that point. Figure 2 shows a vertical cross-section of the LA basin. The units for the X and Y axes are meters. The Y axis indicates the depth in the basin. The color of a point in the image corresponds to the value of the wave propagation speed ground property for that point. The density of the ground determines the wave propagation speed. Notice two interesting properties of the dataset: (1) the density of the ground can vary by several orders of magnitude; (2) large portions of the dataset are homogeneous, especially deep in the Earth. A 3D matrix representation of the CVM model would have approximately 4000 million cubes, requiring about 16 GB per stored attribute (e.g., wave velocity, ground density, etc). This dataset is represented as an octree [21] and accessed using the CMU etree library [28, 29] described in Section 4.2. This representation exploits the homogeneity of contiguous regions in the dataset. The octree representation of the CVM model has 71 million cubes, which requires 900 MB for the structure representation and 284 MB per attribute stored.

When submitting requests to the CVM service, users specify parameters such as the desired resolution and region of interest. These parameters directly affect the response time. For example, processing requests for 2D images can take in the order of minutes to even tens of minutes. This lack of responsiveness is due to the large amount of data needed to be accesses in order to satisfy a request. This motivated us to develop mechanisms to speed up the CVM service.

2.2 Visualization of 4D wavefields

The 4D wavefield datasets describe the wave propagation over time in the simulated region. For each time step the numerical solver records various attributes for all the mesh nodal points. These attributes include wave-velocity components (v_x, v_y, v_z) and optionally the wave amplitude. According to the estimates provided by the SCEC/CME working group the output dataset sizes for finite difference simulations are in the range of a 4 GB to 4 TB depending on the degree of down-sampling both in the time and space domains.

Researchers at the University of Southern California's Information Science Institute (USC/ISI) are developing tools to visualize output wavefield datasets as part of the grid visualization utility project (GVU) [8]. A pre-processing step samples the dataset at the finest available granularity and aggregates multiple fine-grained points to create a coarser version of the dataset. The visualization tool operates on the coarse dataset to allow user interaction.

3. RELATED WORK

Many approaches have been proposed to access and query large datasets. Here we present previous work that our approach builds on.

Computer systems caches are commonly used in the memory hierarchy [4], in distributed file systems [18, 22], the web [15] and others, to speed up data transfers between system elements with different speed characteristics. Examples include disk caches, OS caches [23] and database buffer managers [26]. Data is retrieved from disk in relatively large-size units as a prefetching mechanism

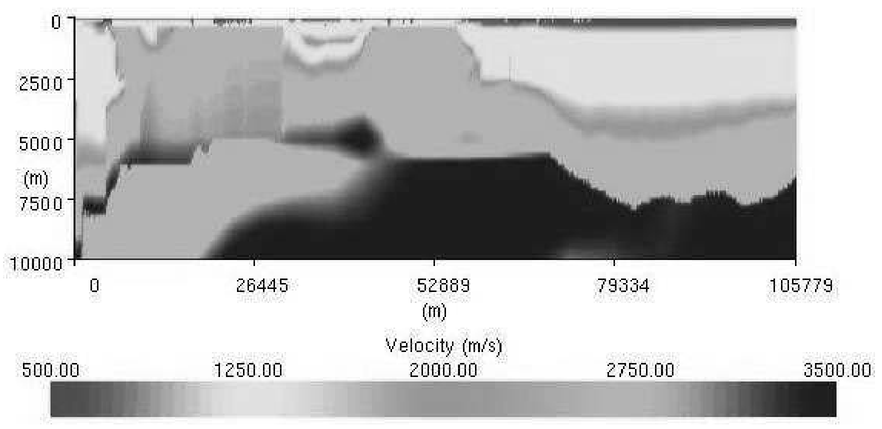

Figure 2: Sample vertical cross-section of the LA basin produced by the CVM service

to decrease the transfer overhead. System level caches store these large-size data retrieval units at the granularity of a memory page or multiple disk blocks [5]. Often, in our applications a small fraction of the items in each data retrieval unit is used to satisfy a set of queries.

There exist various indexing schemes to reduce the overhead involved in searching for a small number of records in a large database [24]. The most widely used indexing scheme in modern database systems is the B-tree and its derivatives [7]. Vitter provides an extensive survey on data structures and algorithms to access large datasets stored on disk [31].

Various systems use the database indexing mechanisms to access large multi-dimensional datasets. ADR/DataCutter[3] is a middleware infrastructure based on R-trees[13] to store and retrieve large multi-dimensional spatial datasets. Similarly, various approaches in the scientific visualization community map and match dataset and work units to disk I/O blocks and use well-known indexing schemes to alleviate the I/O bottleneck[2, 27, 6, 30]. These approaches rely on standard database buffer managers.

Using application-level information to improve cache performance has been used in various contexts. Databases use tuple caching [11] to maintain individual tuples rather than entire pages in the client cache. In semantic caching [9] the client manages the cache as a collection of semantic regions and remainder queries. Remote mobile object systems, such as Thor [10], cache individual objects instead of pages at the client side. Component-based systems use customized views to cache parts of a component instead of whole components [14].

4. OCTREES AND THE ETREE LIBRARY

This section describes key features of the octree dataset representation and the etree library that used by the tree cache to reduce query latency.

4.1 Octrees

Octrees are hierarchic data structures used in many domains to represent spatial data [21]. In particular, the CMU Quake project uses octrees in the physical simulation process to represent ground velocity models, meshes and output wavefields [1, 29].

For simplicity, we use *quadtrees*, the 2D counterpart of 3D octrees,

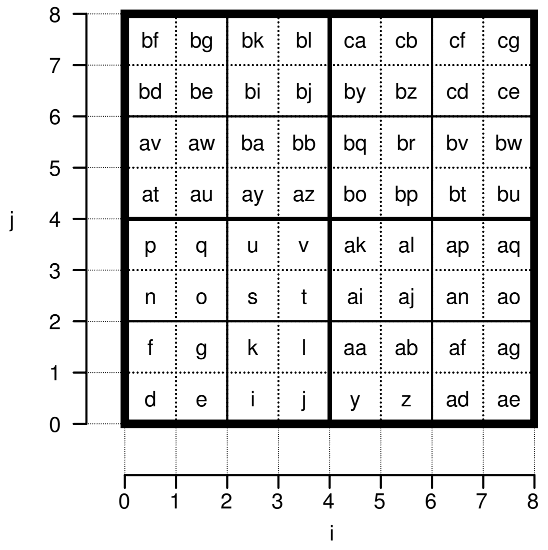

Figure 3: Domain representation

to explain key properties that also apply to higher dimension structures. A quadtree represents a 2D region of space by recursively dividing each region into 4 smaller regions, or *quadrants*, until a desired resolution is achieved. Figure 3 shows a sample 8×8 rectangular domain (heavy line) divided into 4 smaller 4×4 quadrants. We apply this process recursively until we have 1×1 quadrants.

Figure 4 shows the equivalent tree representation for this domain. Each *node* in the tree corresponds to a quadrant in the domain, and its child nodes correspond to the subdivisions of the quadrant. For example, nodes (b) and (bm) correspond to the 4×4 quadrants. *Interior nodes* are nodes with descendants, e.g., (b). *Leaf nodes* have no descendants, e.g., (d). The set of *ancestors* for a node n is composed by its parent (i.e., immediate ancestor) and its parent's ancestors. Each node in the tree has an associated level l. The level of the *root node* is 0, and a node's level is equal to its parent level plus 1. *Max-level* is the maximum level of any node in the tree, 3 in this example. The node level encodes the quadrant's size $(d \times d)$, where $d = 2^{(\text{max-level}-l)}$.

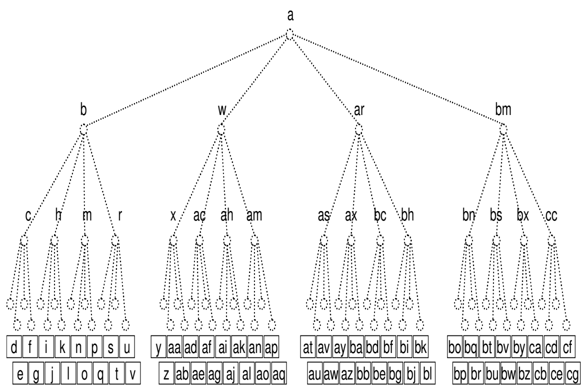

Figure 4: Tree representation

Figure 3 shows a *complete* quadtree where the domain is divided to the finest resolution. For many applications, homogeneous sibling nodes can be *aggregated* into a single parent node according to a data-specific criteria. Figure 5 shows a (8×8) domain, where various nodes have been aggregated into their respective parent nodes. E.g., node (bm) corresponds to an aggregation of all of its descendants.

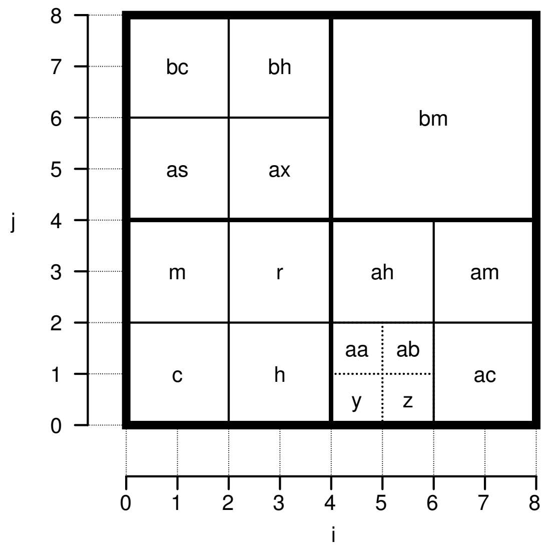

Figure 5: Aggregated quadtree

A *linear quadtree* representation [12] captures the structured of a quadtree by assigning a key to each quadrant. A quadrant's key implicitly encodes the quadrant's location and size, allowing the mapping and storage of a quadtree in a flat structure such as a 1D array. In our implementation we use keys of the form (i, j, l) to uniquely identify a quadrant. (i, j) is the coordinate of the quadrant's lower-left corner in a regular grid at the finest resolution. l is the quadrant's level in the tree. For example, use an 8×8 regular grid as a coordinate system for the domain shown in Figure 3. Then, assign each quadrant the coordinates of its lower-left corner in the grid. Notice that in various instances a child node (the lower-left quadrant of a larger quadrant) has the same grid coordinates as its ancestors. The level in a quadrant's key disambiguates this situation and also encodes the quadrants size. For example, the key for quadrant (ay) is $(2, 4, 3)$ and its parent's (ax) is $(2, 4, 2)$.

euclid3/libsrc/etree.h

```
typedef enum {
    ETREE_INTERIOR = 0,
    ETREE_LEAF
} etree_type_t;

/*
 * - (x, y, z, t) is lower left corner
 * - t is the time dimension for 4D etrees
 * - level is the octant level
 * - type is ETREE_LEAF or ETREE_INTERIOR
 */
typedef struct etree_addr_t {
    etree_tick_t x, y, z;
    etree_tick_t t;
    int level;
    etree_type_t type;
} etree_addr_t;
```

euclid3/libsrc/etree.h

Figure 6: Etree address structure

4.2 The etree library

The CMU etree library [28, 29] provides a capability to access large octree datasets stored on disk. The etree library represents a spatial dataset as an octree using linear quadtree representations and efficiently stores the data in a *B-tree* indexing structure [7].

Applications manipulate datasets as octrees, possibly exploiting the hierarchical data representation. The etree API functions allow applications to perform various operations on octrees, such as, search, insert, delete and update nodes. The library uses a linear quadtree representation in the API to refer to individual quadrants. When referring to a quadrant, applications specify a linear key of the form (x, y, z, level) in a `etree_addr_t` structure (Figure 6).

In order to provide efficient access to the data, the library maps the octree structure to a B-tree index. Internally, the library converts the `etree_addr_t` to a locational code [12], which is a variant of the *Morton code* [19]. The locational code is used as a key to store and search a quadrant in a B-tree structure. The total ordering produced by the locational and Morton codes is known as *z-ordering* or *Peano curve* [20]. This ordering is known to have good spatial clustering properties.

5. THE TREE CACHE

The tree cache is a user-level C library that exploits application-specific information to speed up queries to large octree datasets. It implements a set of techniques to avoid performing expensive data fetches when possible. These techniques include (1) *fine-level caching of individual octants*, (2) *approximate-value query*, and (3) *query reordering*,

Figure 7 shows an overview of the tree cache. When the tree cache receives a request, it looks the octant up in the cache. If the octant is not found, the cache fetches the node data through the data access interface. The etree library provides the data access method for locally stored octree datasets. The Remote Cache Protocol (RCP) provides access to datasets stored at remote locations. This design allows the instantiation of the cache in various scenarios (client application, proxy, server) using the same implementation. In addition, this design allows applications to uniformly access datasets whether they are stored at a local or remote location.

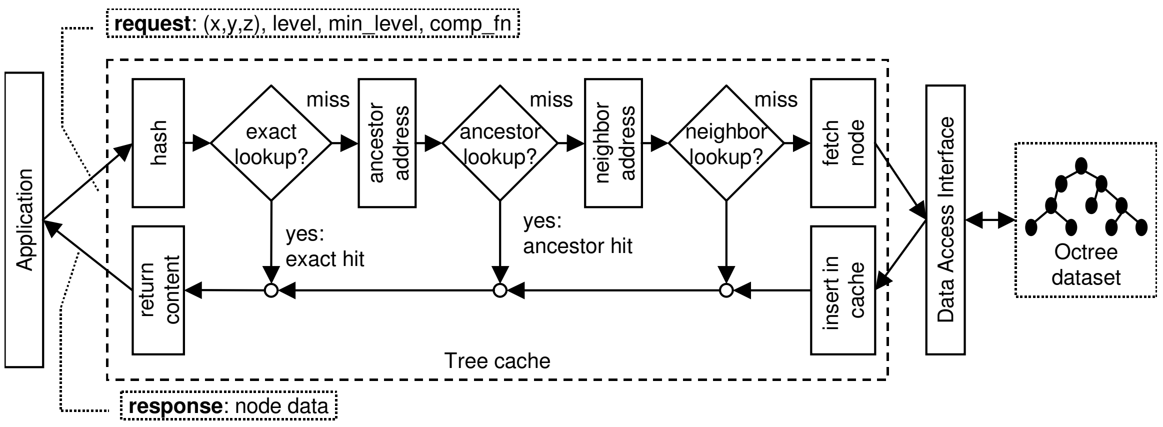

Figure 8: Tree cache lookups

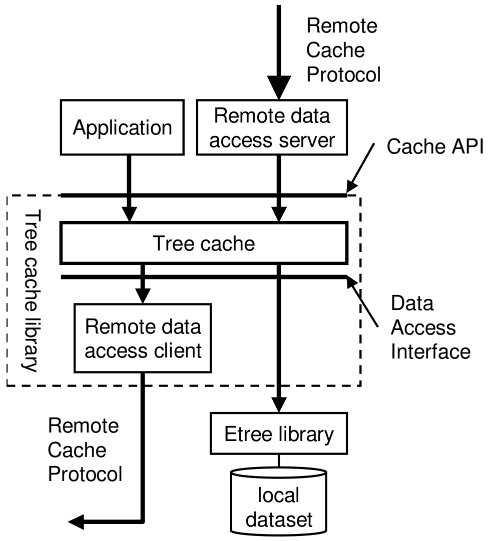

Figure 7: Tree cache overview

Fine-level caching of individual octants. The tree cache caches individual octree nodes (i.e., octants). Nodes are identified by their (x, y, z, level) coordinates. These fixed-length linear keys make cache lookup consistent, uniform and fast regardless of the object requested by the application, e.g., line, plane, volume, since the cache receives only a sequence of requests for nodes. Sharing of cached objects across requests is straightforward.

Figure 8 illustrates the steps that occur during cache lookups. The application requests a node, specifying the node's (x, y, z, level) coordinates. The cache hashes the node coordinates to performs an *exact lookup*, comparing the node's spatial address with the addresses of nodes stored in the cache. If there is a match we have an *exact hit*, otherwise, we have an *exact miss*.

Approximate-value query. The approximate-value query technique decreases mean latency for queries and enables extended functionality for applications, e.g., multi-resolution queries. This technique requires that not only leaf nodes but also the interior nodes are stored in the octree dataset. On an exact miss, the cache performs an *ancestor lookup* by iteratively computing ancestor's coordinates and looking them up until either an ancestor is found or an application-specified minimum node level ≥ 0 is reached. Since an interior node is an aggregate representation of its descendants, when an ancestor is found the cache invokes an application-specified function to determine whether the ancestor satisfies the application requirements. When an suitable ancestor node is found we have an *ancestor hit*, otherwise, we have an *ancestor miss*.

Query reordering. The order in which octants are retrieved from the dataset influences the query response time when the pages containing these octants are not in the OS cache, nor in the database buffer. The goal is to reduce query latency by exploiting the spatial locality produced by etree's storage representation on disk. Fetching the missing octants in the same order they are stored increases the probability of a request being satisfied from the database cache, thus reducing disk accesses and query latency.

6. EVALUATION

Our evaluation intends to answer the following question: *What is the query latency reduction obtained with an application-aware cache?* We evaluate the effectiveness of various tree cache techniques when querying the CVM dataset (Section 2.1). In particular, we look at the following techniques: (1) fine-level caching, (2) approximate-value query, and (3) query reordering.

Experimental Setup: The CVM dataset for these experiments contains both leaf and interior nodes and its size on disk is 10 GB (See Figure 9).

Leaf octants	71,041,024
Interior octants	10,148,729
Total number of octants	81,189,753
Payload size	100 B
Total storage requirement	10 GB

Figure 9: CVM dataset characteristics

We used 3 query traces that are representative for the queries performed by a user to the CVM service during an interactive session. Each trace is divided into a series of steps. Each step corresponds to a request for an image in a zoom-in, pan, zoom-out sequence. The first step in a trace corresponds to a request for a large ROI at low resolution. Following requests are for smaller regions at

Trace name	Steps	# Points
Vertical	5	15.440
Horizontal	10	91.190
Volume	10	364.168

Figure 10: Query traces characteristics

higher resolutions. Figure 10 shows the total number of points and steps for each number trace. The first two traces correspond to 2D requests for vertical and horizontal slices respectively. The third trace corresponds to requests for 3D volumes.

Parameter	Values
Tolerance	0 (exact), 0.0001 (approx)
Num. entries	0, 16K, 32K, 64K, 128K
Trace	vertical, horizontal, volume
Order	random, xyz, z-order

Figure 11: Parameter values for the experiments

Each experiment using a particular trace is divided in 3 phases: The first phase is the *warmup phase* and is composed by the queries for a given trace (e.g., 10 requests for the volume trace in the order they appear in the trace). This phase warms up the tree cache and the database buffer. The second phase is the *pollution phase* and consists of 30 unrelated query steps, with a total of 86962 points. The third phase is the *query phase*, it consists of the queries for the trace, i.e., the same requests performed in the warmup phase in the order they appear in the trace.

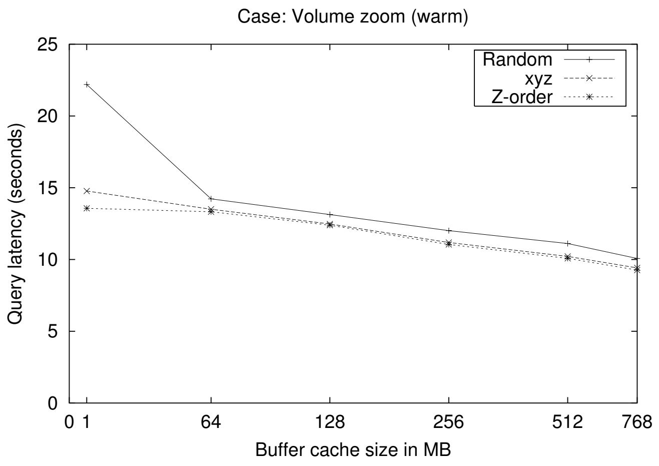

Figure 12: Latency for the query phase in the volume trace without a tree cache

We used the values shown in Figure 11 for the tolerance, number of cache entries, trace type and query order parameters. To establish how query order affects query latency, we performed queries with different orders as follows: we reordered the points within a step (request) of a trace for all steps in that trace, both in the first (warm-up) and last (query) phases. We used three orders: *random, xyz, z-order*. In the random order we randomized the query order using standard C functions. The xyz order corresponds to the standard order used by the application where the X coordinate varies the fastest and Z the slowest. In z-order the points are in the ascending order given by the scalar value of their corresponding locational code, i.e., z-order.

All experiments executed with a warm OS cache. Each experiment execution started with a cold database cache and a cold tree cache, which were warmed up after the first phase of the query trace. We measured query latency for each phase. We performed these experiments on a PIII 1 GHz machine with 3 GB memory and a Ultra SCSI 160 controller and disk, running the Linux 2.4.20 kernel. We reserved 640 MB for the database buffer cache, and the required memory for the tree cache was drawn from the OS-managed page cache. The size of the memory required for the tree cache varied from 0 to 10.5 MB.

Entries	Exact (sec)			Approximate (sec)		
(K)	rand	xyz	z-ord	rand	xyz	z-ord
0	10.31	9.66	9.57	10.28	9.63	9.52
16	10.46	9.71	9.60	9.89	9.30	9.12
32	8.88	8.16	8.06	7.43	6.80	6.71
64	4.98	4.63	4.55	3.19	2.93	2.81
128	0.89	0.83	0.78	0.12	0.07	0.02

Figure 13: Query phase latency in seconds for the volume trace

Effectiveness of application-independent caching. First, we want to determine the resulting query latency when only the system level caches are used and use this as a baseline. We compare the observed query latency for various sizes of the database buffer cache. Figure 12 shows the query latency without tree cache for the volume trace. The X axis is the database buffer size in megabytes. The Y axis is the average query latency in seconds. Notice that diminishing returns are obtained as the size of the database buffer increases.

Effectiveness of application-aware caching. To measure the effectiveness of the tree cache, we want to compare the elapsed time for queries with and without the tree cache. Figure 13 contains the average elapsed time for the volume trace when both the database and OS caches are warm. The other two query traces produce similar results and are not shown here. The first row in Figure 13 (zero entries) corresponds to the query latency with no tree cache, and is the baseline case.

Figure 14 shows the elapsed time vs. tree cache size for the data in Figure 13. The units for the X axis are the cache size in number of entries. The Y axis is the average query-phase elapsed time. Each line corresponds to a set of experiments with a fixed tolerance value (exact vs. approximate) and query ordering. The average query latency decreases as the tree cache size increases. Once the tree cache is large enough, most requests are quickly served from the tree cache, reducing the query latency by an order of magnitude.

Discussion. We are interested in knowing how each tree cache technique contributes to the latency reduction. To determine the contribution of fine-level caching, consider how the presence of a tree cache affects query latency for exact queries. The top three lines labeled *exact random, xyz and z-order* in Figure 14 show the query latency for exact queries. Clearly, fine-level caching pays dividends by reducing the latency by up to an order of magnitude compared to the case when no fine-level caching is used (left-most point of the top three lines).

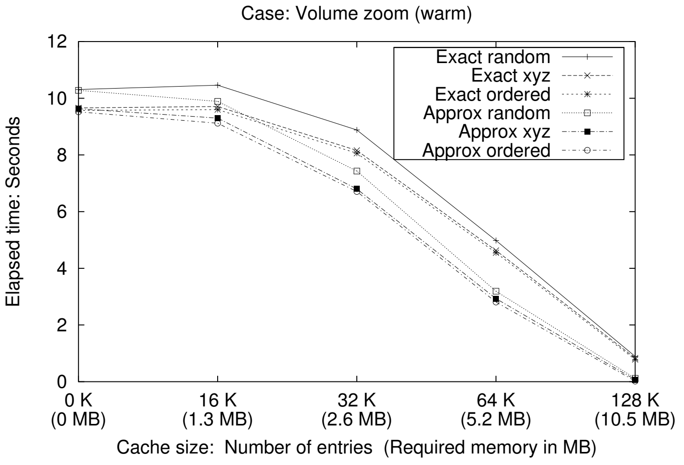

Figure 14: Latency for the query phase in the volume trace

To determine the effectiveness of the approximate-value query technique, we compare the latency obtained for exact queries against the latency obtained for approximate queries for a given cache size. The bottom three curves in Figure 14 (*approx. random, xyz and z-order*) correspond to the latency observed for approximate-value queries. For a given cache size, the query latency is consistently lower for approximate-value queries. For example, for 64K entries, exact queries have a 40% latency reduction, whereas approximate queries have a 50% reduction. Overall, approximate-value queries contribute an additional 10% reduction in query latency. Once the cache is large enough (128K entries in this case) the query latency for approximate-value queries and exact queries is the same. The use of approximate-value queries allows interactive applications to perform tradeoffs between query latency, accuracy and memory requirements.

To determine the effect of query reordering, we compare the observed latency for traces with different orders. In Figures 13 and 14 we can see that, although queries in z-order have lower latency, the difference is not significant. Reordering query points does not provide an additional benefit when the OS page cache is warm. Once the B-tree pages are in the OS page cache, the access cost for any of those pages is approximately the same. We expect query reordering to result in lower query latency when the requested items are not in the database buffer nor in the OS page cache.

In summary, our evaluation indicate that the use of a small application-aware cache can effectively reduce the average query time up to one order of magnitude over the case when only system level caches.

7. CONCLUSIONS

Using application-level information and dataset-specific structure in a small application-aware cache reduces query latency to large datasets. As dataset sizes grow it becomes more important to maintain low query latency in order to support interactive exploratory tools. Our evaluation shows that the tree cache reduces query latency by one order of magnitude over the case when only system level caches are used. It is able to do so by exploiting the structure of octree datasets and allowing applications to relax the accuracy requirements of the queries to the dataset.

The results of the tree cache evaluation are encouraging. In the near future we will implement other techniques in the tree cache, such as approximate-distance queries, and octant prefetching. In approximate-distance queries, the application relaxes the accuracy constraints allowing a query to be satisfied with a cached octant that is close to the requested octant (i.e., a neighbor). The idea behind octant prefetching is to batch requests for missed octants in a single transaction. We expect this technique to be very effective in reducing access time to remote datasets, as access time to remote datasets is dominated by the round trip time between the local and remote hosts. Batching multiple transactions in a single requests serves as a prefetching mechanism and amortizes the round trip time cost.

8. REFERENCES

[1] V. Akcelik, J. Bielak, G. Biros, I. Ipanomeritakis, A. Fernandez, O. Ghattas, E. Kim, J. López, D. O'Hallaron, T. Tu, and J. Urbanic. High resolution forward and inverse earthquake modeling on terasacale computers. In *Proceedings of Supercomputing SC'2003*, Phoenix AZ, USA, Nov 2003. ACM, IEEE. Available at `www.cs.cmu.edu/~ejk/sc2003.pdf`.

[2] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proc. of the Symp. on Parallel Visualization and Graphics*, pages 97–104. IEEE, ACM Press, 1999.

[3] M. Beynon, R. Ferreira, T. M. Kurc, A. Sussman, and J. H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Symp. on Mass Storage Systems*, pages 119–134. IEEE, 2000.

[4] R. E. Bryant and D. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.

[5] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-grained sharing in page server database systems. In *Proc. ACM SIGMOD Conf.* ACM, 1994.

[6] Y.-J. Chiang, R. Farias, C. T. Silva, and B. Wei. A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids. In *Proc. Symp. on Parallel and Large-data Visualization and Graphics*, pages 59–66. IEEE, IEEE Press, 2001.

[7] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 2(11):121–138, 1979.

[8] K. Czajkowski, M. Thiebaux, and C. Kesselman. Practical resource management for grid-based visual exploration. In *High Performance Distributed Computing HPDC'01*, 2001.

[9] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proc. 22th Int. Conf. on Very Large Data Bases*, pages 330–341, Bombay, India, Sep 1996. Morgan Kaufmann.

[10] M. Day, B. Liskov, U. Mahashwari, and A. C. Myers. References to remote mobile objects in thor. *ACM Letters on Programming Languages and Systems*, 2(1–4):115–126, Mar–Dec 1993.

[11] D. DeWitt, P. Futtersak, D. Maier, and F. Velez. A studey of three alternative workstation-server architectures for object-oriented databases. In *Proc. VLDB Conf.*, 1990.

[12] I. Gargantini. Linear octree for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20(4):365–374, 1982.

[13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. Intl. Conf. on Management of Data*, pages 47–57, Boston, MA, Jun 1984. ACM SIGMOD.

[14] A. Ivan and V. Karamcheti. Using views for customizing reusable components in component-based frameworks. In *Proc. 12th High Performance Dist. Computing (HPDC'03)*, Seattle, WA, Jun 2003.

[15] G. Lai, M. Liu, F.-Y. Wang, and D. Zeng. Web caching: architectures and performance evaluation survey. In *Conf. on Systems, Man, and Cybernetics*, pages V5: 3039–3044. IEEE, Oct 2001.

[16] J. Lopez, T. Tu, and D. O'Hallaron. CVMs: Community Velocity Model service. `http://cvm.cs.cmu.edu`, 2002.

[17] H. Magistrale, R. Graves, and R. Clayton. A standard three-dimensional seismic velocity model for southern California: version 1. *EOS Transactions AGU*, 79:F605, 1998. `www.scecdc.scec.org/3Dvelocity/3Dvelocity.html`.

[18] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A distribute personal computing environment. *Communications of the ACM*, Mar 1986.

[19] G. M. Morton. A computer oriented geodetic database and a new technique in file sequencing. Technical report, IBM, Ottawa, Canada, 1966.

[20] H. Sagan. *Space Filling Curves*. Springer, 1994.

[21] H. Samet. *Applications of Spatial Data Structures: Computer Graphics Image Processing and GIS*. Addison-Wesley, 1989.

[22] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, Apr 1990.

[23] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. Addison-Wesley, 3rd edition, 1991.

[24] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, 4th edition, Oct 2001.

[25] Southern California Earthquake Center. Community velocity model (SCEC/CME). `www.scec.org/cme`.

[26] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, Jul 1981.

[27] P. M. Sutton and C. D. Hansen. Accelerated isosurface extraction in time-varying fields. *Transactions on Visualization*, 6(2), Apr-Jun 2000.

[28] T. Tu, J. Lopez, and D. O'Hallaron. The Etree library: A system for manipulating large octrees on disk. Technical Report CMU-CS-03-174, Carnegie Mellon School of Computer Science, Pittsburgh, PA, July 2003.

[29] T. Tu, D. O'Hallaron, and J. Lopez. Etree – a database-oriented method for generating large octree meshes. In *Proceedings of the Eleventh International Meshing Roundtable*, pages 127– 138, Ithaca, NY, Sep 2002.

[30] M. van Kreveld, R. van Oostrum, C. Bajaj, D. Schikore, and V. Pascucci. Contour trees and small seed sets for isosurface traversal. In *Proc. 13th Symp. on Computational Geometry*, pages 212–219, Nice, France, Jun 1997. ACM, ACM Press.

[31] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.