

Knowledge-Based Out-of-Core Algorithms for Data Management in Visualization

David Chisnall Min Chen Charles Hansen
csdavec@swan.ac.uk m.chen@swan.ac.uk hansen@cs.utah.edu

Hypothesis

It is possible (and practical) to design a generic knowledge-based prefetching algorithm that achieves performance commensurate with algorithms specific to the rendering method.

This is science, so we need a hypothesis.

Currently two choices:

- Let the OS VM handle external memory (not very good)
- Write a problem specific handler (lots of effort, better performance)

We want an algorithm that can go into the OS and give good performance.

Knowledge Based

- Records previous interactions
- Deduces future behaviour from acquired knowledge
- Does not rely on information provided by other means

What do we mean when we say 'Knowledge Based'
No strict definition exists, this is what we mean.

Related Work

- Very Large Dataset Visualisation
 - Stockinger et al. '05, Kaehler et al. '05, Childs et al. '05, Sutton and Hansen '00, Koutsofio '99, Pharr et al. '97, Teller et al. '04 ...
- General Out-of-core
 - Vitter '01, Leutenegger '99, Cox and Elleworth '97, Denning '70, Denning '68...
- Out-of-core for Visualisation
 - Gao et al '05, Isenburg and Lindstrom '05, Pajarola '05, Castanie and Levy '05, Lindstrom and Pascucci '02, Silva et al. '02, Varadhan and Manocha '02, Farias and Silva '01, Lindstrom '00, Cignoni et al. '96, Ueng and Sikorski '97, Shen et al. '99, ...

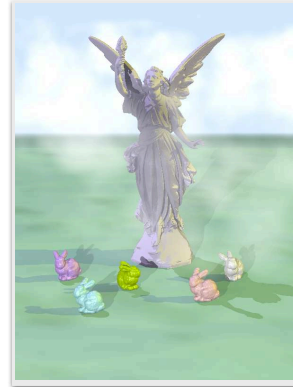
Lots of approaches have been proposed for rendering large datasets, some using explicit pre-fetching, some relying on the OS to handle external memory.

A number of general OOC approaches have been proposed. These include reducing the page size (page sizes closer to data structure sizes give better results), re-ordering the data on the disk for better linear reads, and combining LRU and LFU approaches for eviction. Some proposals suggest exporting APIs for userspace processes to control page eviction.

Most OOC algorithms in Vis are tied very closely to the rendering

Test Renderer

- Point datasets
- Ray tracing
- Octree partitioning
- Pluggable prefetching algorithm



The first step to proving our hypothesis is to try to produce a general algorithm that meets the criteria for a specific renderer. This is the rendering process we used.

Four Algorithms

| Algorithm | Acquires Knowledge | Uses Render Information |
|--------------------------|--------------------|-------------------------|
| Least Recently Used | × | × |
| Ray Driven Predictor | × | √ |
| Access Path Predictor | √ | × |
| History Access Predictor | √ | × |

We used two benchmark algorithms to assess our attempts.

LRU – Very simple approach.

RDP – Renderer-specific algorithm.

Two knowledge-based attempts.

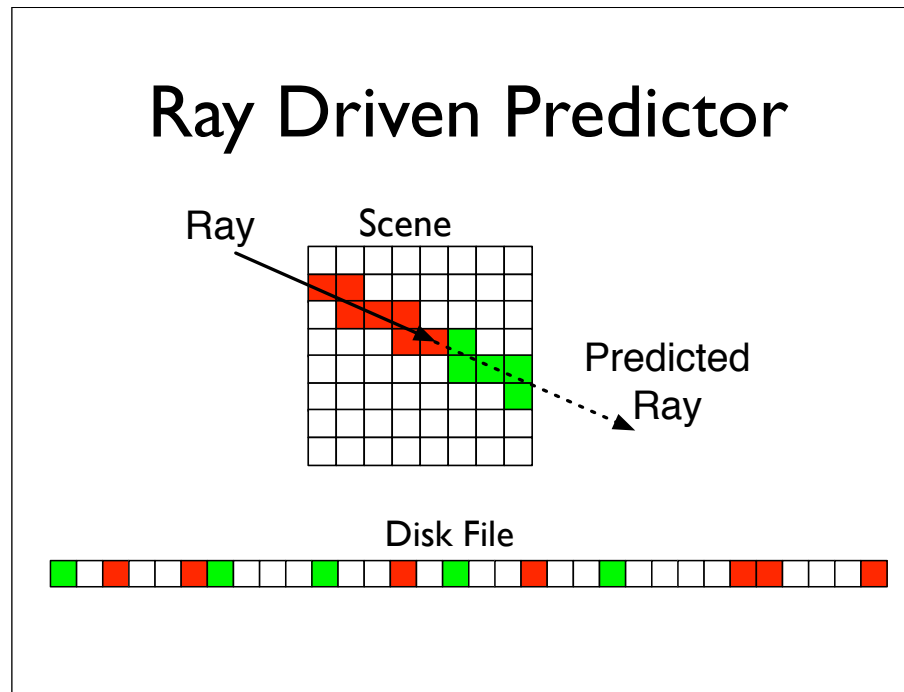
Least Recently Used

- Naive approach
- Finer grained than OS
- Demand paging

Ray Driven Predictor

- 'Understands' ray paths
- Predicts assuming infinite-length rays
- Faults at ray termination and secondary rays

Ray Driven Predictor



Example uses 2D simplification.

Predicts ray course based on previous two sample points.

Note that octree nodes may not be stored together on disk.

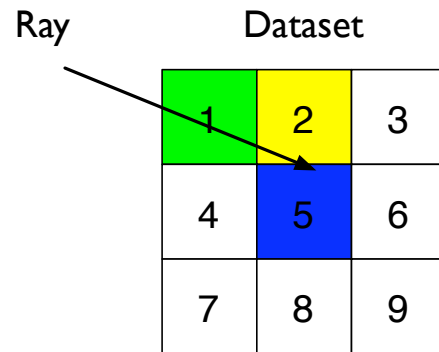
Access Path Predictor

- Assumes (predecessor, current, successor) relationship
- Stores four previous accesses
- Falls back to guessing any previous successor if the (predecessor, current, _) relationship does not hold

Complex. This should acquire all of the knowledge required for primary and secondary rays.

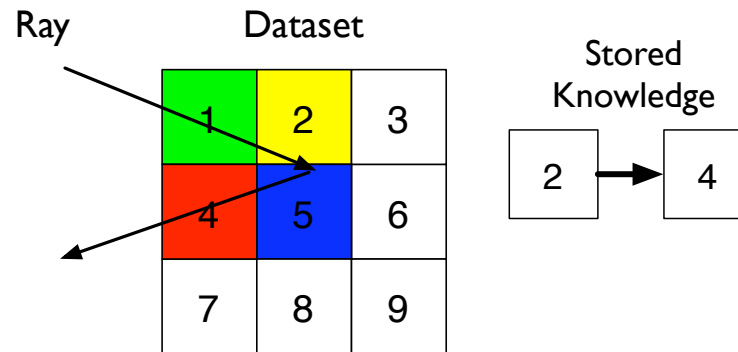
It does, however, require a fairly significant amount of CPU time spent doing the prediction.

Access Path Predictor



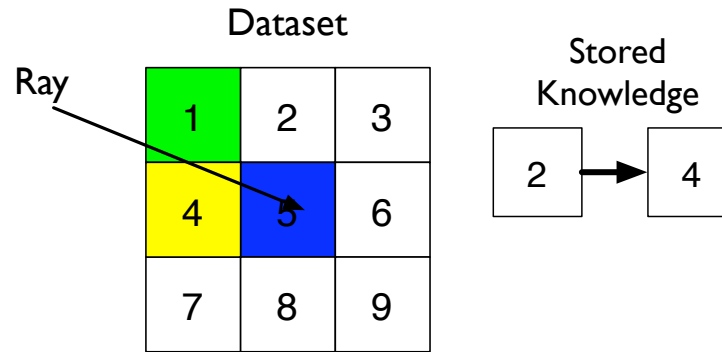
Ray enters the centre node.

Access Path Predictor



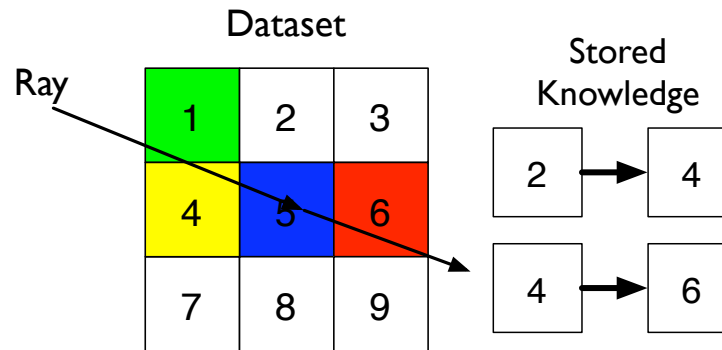
This ray hits something in the scene and rebounds.
As the ray continues, the predecessor-successor relation is recorded.

Access Path Predictor



Next ray enters from another predecessor.

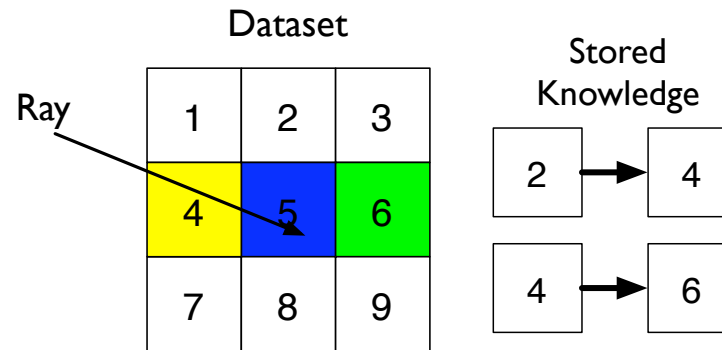
Access Path Predictor



This node continues straight through, without hitting the obstacle in the corner.

New predecessor–successor relation recorded.

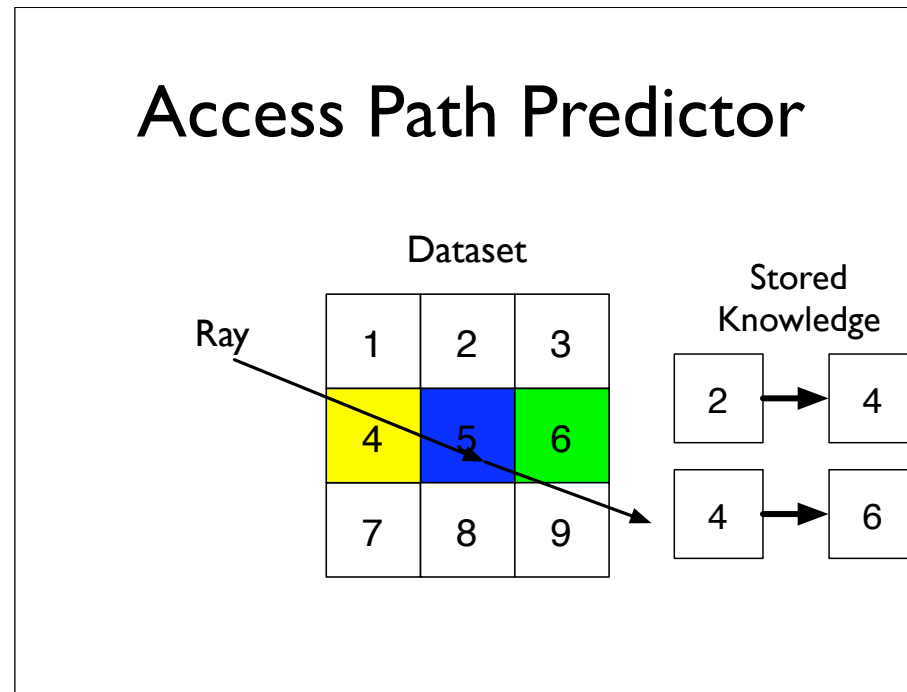
Access Path Predictor



For the third ray, the predecessor matches one in the stored knowledge.

The successor in this instance is loaded.

Access Path Predictor



The ray continues into a node that is already loaded.
And, hopefully, the same thing happens to prefetch the next node.

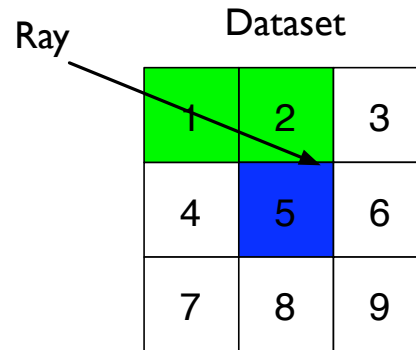
If the predecessor is unknown, this algorithm falls back to prefetching all known successors.

History Access Predictor

- Assumes repeated successor pattern
- Stores eight previous successors
- Loads all previous successors when a node is accessed

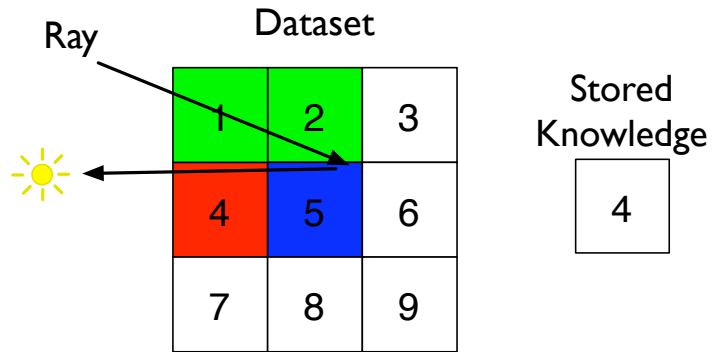
Simpler. Faster to computer. Almost as good.

History Access Predictor



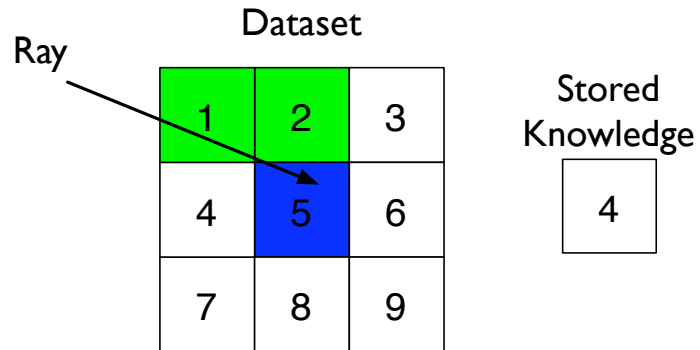
Ray enters octree node for the first time. No prefetching.

History Access Predictor



Secondary ray towards a light source leaves node, next node recorded.

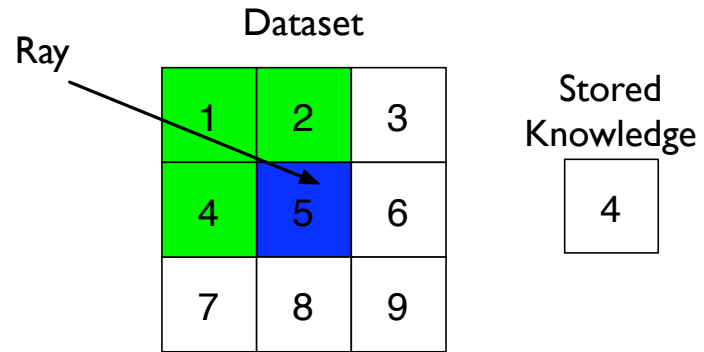
History Access Predictor



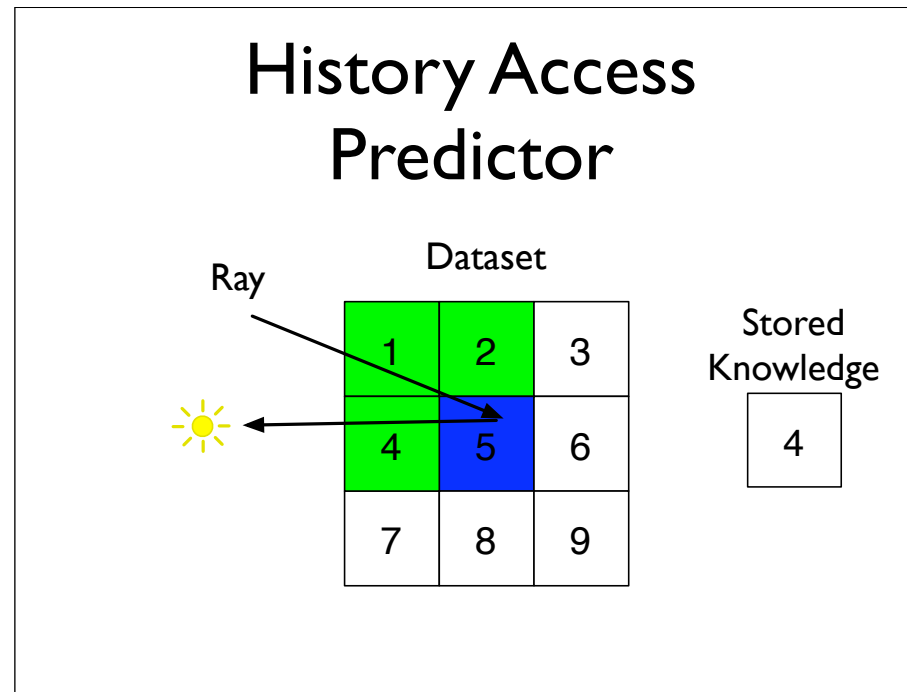
A second ray enters the node.

At this point, there is stored knowledge of a potential next node.

History Access Predictor



The node is loaded.



The secondary ray, fired towards the light source, continues without a fault.

This works well, since most rays are either towards a light source, or away from the viewpoint.

Performance Metrics

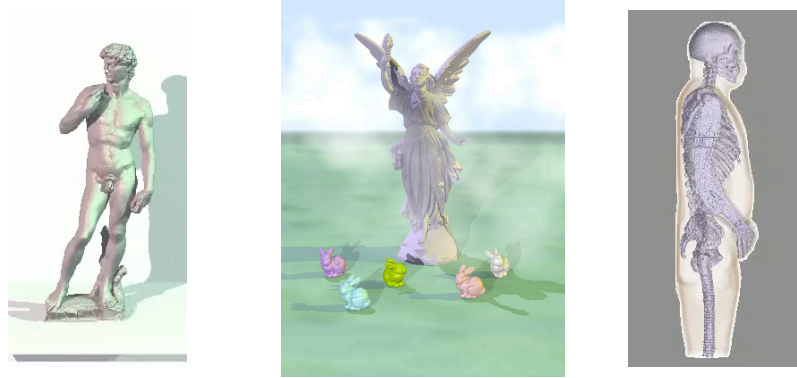
- Total disk accesses
- Cache hit rate
- Rendering time

Disk accesses indicate the number of false positives (i.e. incorrect prefetching).

Cache hit rate indicates the number of false negatives (i.e. nodes not prefetched that should have been).

Rendering time gives some indication of real-world performance. Not as relevant – aspects of our testing framework designed to ensure a fair assessment of the algorithms (instrumentation, same information provided, extra layers of abstraction) cause the performance to be lower than it could have been.

Testing Results



We tried rendering a series of different scenes. Some contained real data, some where random point sets.

Normalised Disk Reads

| | LRU | RDP | APP | HAP |
|---------|-----|---------|-----|-----|
| Best | 1 | 0.4 | 0.3 | 0.8 |
| Average | 1 | 1.8 | 0.8 | 1.9 |
| Worst | 1 | 5.1(38) | 1.4 | 4.9 |
| Rank | 2 | 3 | 1 | 4 |

The worst case for RDP of 38 was a pathological corner case (very small image, large sampling interval) 5.1 is a more representative worst-case score.

HAP scored worse than APP here because it speculatively pre-cached all potential successor nodes, rather than just the one that was most likely to be needed.

HAP and RDP have similar performance, reinforcing our hypothesis that a knowledge-based algorithm could perform as well as a domain-specific one. APP better by this metric, which was a (pleasant) surprise.

Cache Hit Rate

| | LRU | RDP | APP | HAP |
|---------|-------|-------|-------|-------|
| Best | 99.9% | 100% | 99.9% | 99.9% |
| Average | 98.9% | 99.8% | 99.8% | 99.7% |
| Worst | 91.1% | 99.3% | 99.2% | 99.0% |
| Rank | 4 | 1 | 2 | 3 |

RDP and APP have similar performance, again reinforcing our hypothesis. HAP does marginally worse, but the difference between RDP and the two knowledge-based approaches is an order of magnitude less than the difference between those three and LRU.

Normalised Timings

| | LRU | RDP | HAP | APP |
|---------|-----|------|------|------|
| Best | 1 | 0.88 | 0.87 | 0.96 |
| Average | 1 | 0.91 | 0.90 | 1.01 |
| Worst | 1 | 1.05 | 1.03 | 1.05 |
| Rank | 3 | 2 | 1 | 4 |

In most cases, RDP and HAP gave approximately a 10% speed boost even with our instrumented implementation. This should increase as the disk latency increases relative to CPU speed.

APP faired slightly worse, as we expected. We still have hopes for this algorithm in the future, if the differential between CPU and disk speeds continues to grow, but at the moment it is too expensive.

Conclusion & Questions

Both the knowledge-based algorithms shown are consistently close to, or better than RDP in cache usage.

HAP is close in terms of performance and cost, which supports our initial hypothesis.

Questions?