Uri Dekel
Brett Meyer

Cache-Sensitive Optimization of Immutable Graph Traversals

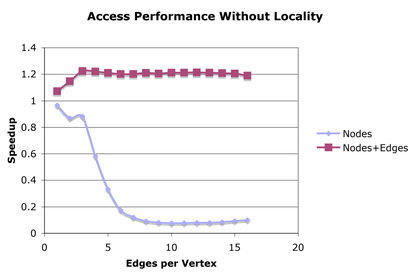
15-745 Spring 2006
Advanced Optimizing Compilers

Experimental Setup

- We conducted experiments to determine sources and degree of speedup
- Randomly generated graphs
- Graphs fit in L2 with fixed memory footprint
  - Minimizes impact of VM
- Microbenchmarks and one "real-world" application

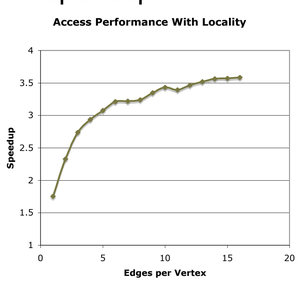
Effect of Code Transformation

- Break spatial locality with padding
- Slowdown accessing just nodes
  - Optimized graph suffers greater number of conflict misses in repeated traversals
- 1.2x speedup accessing nodes and edges



Access Performance With Locality

- Edge locality improves execution time significantly
- Edges are allocated sequentially, and accessed sequentially
- Up to 3.5x speedup



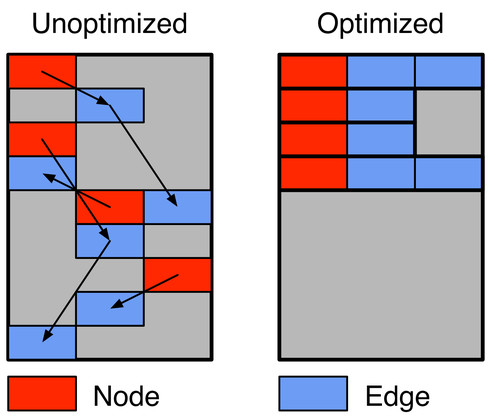
Why Optimize Graphs?

- Graphs are common data structures in many applications
- Graphs are typically collections of pointer-connected objects
  - Traversal produces long series of dependent cache misses
- Reduce execution time by restructuring graphs to
  - Improve locality
  - Break sequences of dependent misses

How Should Graphs Be Optimized?

- Optimized for traversal locality
- Key technique: flatten the graph *after* its topology is *"fixed"*
- Flatten pointers into arrays
  - Group edges with nodes
  - Group nodes with nodes
- Detect traversal pattern, and at run-time adaptively flatten
  - True detection beyond scope
  - We group edges with nodes

Optimized Graph Implementation



- Very simple transformation
- Transform data
  - Allocate an integer array for each node, containing node and all edges
  - Nodes and edges allocated in same order as in original graph
- Transform code
  - Pointer dereferencing replaced with indexing into an array

Conclusions

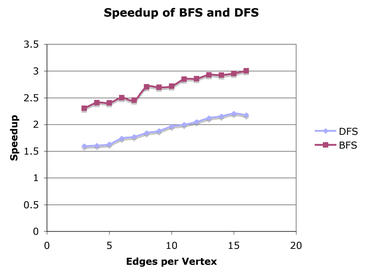
- Graph traversal is a source of long sequences of dependent cache misses
- Simple graph optimization leads to significant speedup
  - Improve locality
  - Eliminate pointers
  - Greater improvement for heavily connected graphs
- Speedup of up to 2x

Future Work

- Compile-time traversal analysis
  - Compiler can detect simple access patterns, provide hints for run-time reorganization
- On-line traversal profiling and optimization
  - DFS breaks "node-edge-edge ..." locality, perhaps reorganize to optimize for DFS

Optimizing DFS and BFS

- DFS recursively visits nodes, traversing one edge at a time
  - Breaks edge locality
- BFS accesses a node, then all its edges to place nodes on worklist
  - Leverages edge locality



- DFS speedup similar to accessing objects with no locality
  - As edge-node ratio increases, greater temporal locality in optimized version
  - Up to 2.25x speedup
- BFS speedup similar to accessing objects with locality
  - As edge-node ratio increases, greater spatial locality in optimized version
  - Up to 3x speedup

Optimizing A*

- A* search has similar access behavior to BFS
- As edge-node ratio increases, greater spatial locality
- Up to 2x speedup