Cache-sensitive optimization of immutable graph traversals (CS745 Project Report)

Uri Dekel and Brett Meyer

ABSTRACT

This project strives to demonstrate the potential benefits of improving the cache locality of programs which frequently traverse an immutable data structure. In our approach, the compiler generates code to fold the inherently noncontiguous representation of each node in a mutable structure into a more contiguous representation possible only for immutable structures. For instance, a mutable graph structure in which nodes maintain linked lists of edges could be transformed into a structure where each node maintains its edges internally in a fixed array. Such a representation yields improved performance due to better cache locality and a decrease in pointer dereferencing operations. It also opens the door to external reorganizations of the nodes.

This report presents results on potential improvements to manually optimized programs, and discusses the feasibility and usefulness of implementing such optimizations in compilers. Our focus is limited to the example of graphs.

1. INTRODUCTION

1.1 Background

Memory access is a significant bottleneck in modern computing architectures, as memory latency may be longer than a processor cycle by orders of magnitude. To alleviate this problem, at least one level of caching is used to bypass these expensive accesses by retrieving the frequently accessed data from a much faster but smaller cache memory.

The effectiveness of caching arises from the principle of locality, which implies that if a certain object is accessed, then the probability of accessing the same object in the near future increases. Thus, if we just accessed a variable in main memory, then a subsequent access may be less expensive as the variable has been placed in the cache during the first access. If it is still there, then we have a 'cache hit'. However, since cache space is limited, an object may be evicted in deference to others, resulting in an eventual cache miss.

Another implication of the principle of locality is that access to one object increases the probability of a subsequent access to a related object. This implication is the basis for prefetching, which brings the necessary object from the long-latency medium before it is absolutely required, in anticipation of its possible future need. Prefetching is practical in many situations when the number of relevant objects is roughly linear in the number of active objects. For instance, modern processors assume that if an instruction is executed, so will the next few following instructions, and prefetch accordingly. However, prefetching is less practical for optimizing specific program behaviors, since it requires hardware and software support for recognizing these opportunities. Unfortunately, hardware support for pre-fetching is limited to extremely simple memory access patterns, reducing its usefulness as a general technique for memory performance optimization.

Nevertheless, an inherent property of cache memory implementations allows programs to indirectly leverage prefetching on a small but extremely effective scale: Cache memory is not at the granularity of a single object. Instead, it consists of lines of contiguous memory space which could accommodate multiple objects, and are manipulated as one unit. Thus, a memory organization that stores an object and its related objects contiguously increases the likelihood that when the first object is accessed and its memory block brought into the cache, one of the related objects will be in the same cache line. For instance, in languages like C, arrays are allocated as contiguous blocks of memory. A sequential scan of the array will thus trigger many cache hits, since the cache line containing the first member contains several of its successors, etc.

Unfortunately, many common data structures are not linear and use pointers to connect objects scattered around memory. This nonlinear structure precludes us from naively benefitting from such implicit prefetching. Nevertheless, not all is lost, since pointers offer memory transparency, in the sense that the exact location of an object in memory is unknown to the programmer. There is therefore nothing to prevent the compiler or runtime system from shuffling objects in memory to achieve a similar effect. Accomplishing this is difficult, and demands specialized treatment of the program and explicit understanding of its behavior and structure. However, optimizing the memory behavior of certain data structures can yield significant performance gain in certain applications.

1.2 Goal and Approach

The goal of this project is to optimize the cache-hit rates for programs which perform frequent traversals of graph

data structures. Graphs play central roles in many applications in computer science, from compilation to mapping algorithms to games. Unlike data structures whose primary goal is to provide efficient access to information, the essence of a graph's data is in its structure. Thus, whereas programs are likely to traverse short paths in data structures like trees or tries, graph data structures are often traversed in their entirety, increasing the number of memory accesses.

Improving the memory behavior of graphs is extremely challenging, because they do not have the inherent organization of structures such as binary search trees. Instead, each node can stand by itself or be connected to an unlimited number of nodes, making optimization choices more difficult. Furthermore, graphs can be cyclic and their edges are not ordered, making traversal less predictable than for some other data structures. Nevertheless, we plan to provide infrastructure for optimization which could be leveraged to accommodate a variety of memory organization techniques.

Our motivation is based on the intuition that most graph traversals take place after the graph has been created and its structure stabilized and becomes immutable For instance, in many compilers different optimizations passes traverse the same control-flow graph of a program after it has been calculated from the intermediate representation. Similarly, once a route map has been built, many traversals may take place to seek optimal paths. Thus, we argue that graphs are mutable as they are being constructed, but are effectively immutable at the time of traversal.

Mutable graphs must accommodate uncertainty and are not memory efficient. For instance, they use non-fixed collections such as linked lists to maintain the adjacencies of each vertex. Immutable graphs, on the other hand, can be organized more efficiently. First, the collections are now fixed in their size, reducing indirection and allowing them to be stored in new ways. Second, it is now possible to reorganize the memory ordering of vertices to increase the chances of multiple vertices being stored on the same cache line. This would be particularly useful if nodes that are accessed close to one another in a graph traversal would be on the same line.

A way to optimize programs that make frequent traversals of the same graph would be, once the graph has been created, to create an immutable version using a more efficient representation, and then have all traversals utilize this immutable version.

The goal of our project is to automate this optimization (as much as possible) by having the compiler make the necessary modifications to the program, with limited explicit input from the user.

1.3 Past work

The idea of reorganizing data to improve cache-performance is not new. For example, Chilimbri and Larus [**?**, **?**] used structure splitting to separate 'hot' and 'cold' regions of an object into different locations. We apply a similar strategy to separate fields relevant for traversal from those storing additional data, thus creating more compact traversal-specific representations of the vertex that can be packed more efficiently.

In the spring 2003 offering of the 745 course, Chen and Nikos Hardavellas [**?**] worked on a similar project, packing nodes of a binary tree structure into hypernodes. The binary tries used small nodes that allowed entire subtrees or tree slices to be stored.

Our project deals with graphs which are less organized, and therefore necessitate other optimization strategies. In particular, we fold linked lists, and use heuristics to try and place nodes together on the same lines.

2. IMPLEMENTATION DETAILS AND LIMITATIONS

The scope of this class project limits the comprehensiveness of our solution. It forces us to pose many limitations on the programs that could be optimized, and in particular to require direct input from the programmer. Within the alloted time we were not able to implement a transformation system in a working compiler. Instead, we created programs and optimized them by hand, and carried out a series of experiments to demonstrate the potential benefits of this approach. We describe these transformations in detail, making future implementation efforts straightforward.

In this section we describe our approach in detail, along with the requirements and limitations.

2.1 Requirements from the source program

First, we assume the use of a simple C-like language that does not use a garbage collector or runtime system to organize memory, and which supports pointers to actual memory locations. For experimental purposes, we used standard ANSI-C which could be compiled with the GCC compiler.

Second, we assume that graphs are represented using a structure which will be described below.

Third, restrictions are in place on how vertices are accessed, and it is the responsibility of the programmer to adhere to them. In particular, vertices and edges should only be accessed using appropriate pointers: one cannot maintain a pointer to internal fields. Ideally, we would have required all accesses to fields to occur via a set of library functions, which could then simply be swapped to work with the optimized representation, with limited changes to the source program. Unfortunately, since ANSI-C does not support method inlining and it is common to access structures directly, we must make explicit transformations on the source program and define certain restrictions. In addition, no modifications may be made to the structure of an immutable graph, although changes to key and data fields are allowed.

Fourth, after a graph has become effectively immutable, we expect the user to issue a function call to a predefined library function, specifying the pointer to the graph, and some details about the architecture and the expected traversal. At compile time, the compiler will recognize this function call as the point from which the given graph has become immutable. It will generate the necessary code to build the immutable graph, and then replace all subsequent operations having to do with the graph. We believe that static analysis could help identify situations where objects have become immutable, and thus preempt the need for explicit guidelines from the programmer. We will elaborate on this issue when we discuss the usefulness of the approach, but it is outside the scope of our work.

2.2 Representation of mutable graphs

We now proceed to describe the structures of the source mutable graph and of the optimized immutable graphs. This discussion is general and describes our approach if it was to

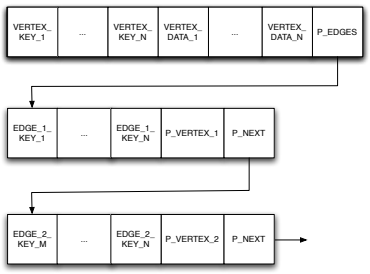

Figure 1: Layout of a vertex in a mutable graph

be implemented for actual user programs. For our concrete experiments we used slightly simpler structures, which will be described in the appropriate section.

Programs typically represent graphs using adjacency matrices or as a network of independent referencing vertices. Within the latter representation, two common sub-representations prevail: The first maintains separate collections of vertices and edge objects, while in the second each vertex maintains a collection of its incident edges and these are only available via that vertex. In both representations, edge objects maintain edge properties and the identifiers or pointers to the vertex objects, and vertex objects maintain the properties of each vertex. In this project we are going to focus on the second sub-representation, optimizing programs where vertices maintain their own connections to other vertices via edges. We assume that all graphs are directed.

More specifically, we restrict ourselves to the representation which we shall now describe and which is illustrated in Figure 1. In this representation, an object representing a vertex with two neighbors is laid in memory as follows: First, there are several key fields, explicitly indicated by the user. These key fields contain data that is crucial for traversal of the tree or the calculation which this traversal serves, such as weights, markings, identifiers, etc.. Second, there are data fields, also indicated by the user. These contain data that is not crucial for the traversal, such as objects associated with the nodes. Third comes the data structure for holding the adjacency list. In this project, we assume that vertices maintain their peers using a linked list, and that the contiguous Vertex object ends with a pointer to the first member of that list. Each element in the peer list represents an edge and is stored contiguously. It contains several key fields of that edge that may be used for traversal purposes such as weights or markings, and a pointer to the node.

2.3 Representation of optimized graph

Following our optimization, the vertex would be organized as follows (Figure 2): The object begins with the vertex key fields since these are essential for traversal. Then comes a pointer to a separate object which contiguously maintains the data fields. Since these fields are rarely used in traversal, they are stored elsewhere with a level of indirection. The compiler will generate the necessary code to access values in these fields. After this pointer comes a single byte representing the number of edges associated with the vertex. It is followed by the records for each incident vertex or edge: several edge property fields followed by a pointer to the incident vertex.

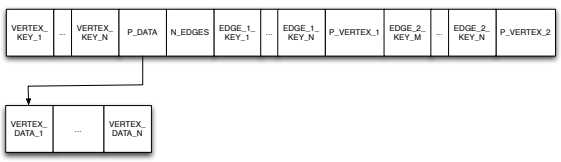

Figure 2: Layout of a vertex in an immutable graph

The primary advantage of this representation is that it allows us to eliminate the levels of indirection associated with obtaining the incident edges for the given vertex. Such indirection in the original layout is necessary since the number of edges is mutable; the problem is even more severe if we use a linked list rather than a dynamically allocated array.

A second advantage of this representation is that it allows us to move the "heavy" data fields away and maintain a compact representation of those fields needed for traversal. This may allow us to store multiple vertices on the same cache line and even more on the same virtual memory page.

To see why this is possible, consider graphs used for calculating shortest paths, spanning trees, and similar measures. To accommodate a variety of such algorithms, let every vertex and every edge have two key fields: one byte for a weight, and one byte for a marking. Thus, the base vertex object (without edges) would consist of 7 bytes (2 for keys, 4 for the data pointer, 1 for the number of edges). Similarly, each edge would add 6 bytes (of which, 4 are the pointer to the edge). Thus, a vertex with two neighbors would consume 19 bytes, while one with four neighbors would consume 31. If nodes are stored consecutively, it is thus quite possible to store two or even three nodes on the same cache line.

We note that the layout described above is not the most optimal possible packing of graph vertices, since each pointer to a neighboring vertex requires 4 bytes. We could, in theory, allocate all vertices as one contiguous memory block, and then replace vertex pointers with indices into this block. This, however, not only introduces more complexities, but also makes it more difficult to make an immutable graph mutable again without creating a new graph structure. We note that our optimized structure allows the removal and redirection of edges and the creation of new vertices, all without a need to reallocate any object.

In summary,the performance gain of the optimized graph structure will come from two sources: First, folding the linked list of connected nodes reduces the levels of indirection, and the number of disparate memory regions that are accessed as all the adjacencies of a vertex are traversed. Second, splitting the structure and folding indirection allows us to obtain vertex representations that occupy less than half a cache line, and we will try to pack nodes accordingly to increase cache hits. Performance hits originate in the one-time creation of the optimized graph, and in the extra level of indirection for accessing

3. REPRESENTATION FOR OUR EXPERIMENTS

In this section we describe in detail the representation used for our experimental test programs. The complete source code of these programs will be attached to this report,

however, we present important excerpts here. To simplify the experiments, we made several assumptions or restrictions over the ones presented in the previous section.

1. We assume that the pointers to all nodes in the original graph will be stored in one array, and have an integer serial number stored in an internal field that correspond to their index in this array. This allows us to build the optimized graph as an array of pointers to optimized nodes that follow the same order. Without this serial field, it would be difficult to determine which node in the original graph each edge connects to and building the new graph will take a long time. In addition, this allows us to replace the pointer to the target node in the mutable graph with a serial number of the target node in the optimized graph.

2. We assumed that all fields are simple integer types; this allows us to allocate the optimized nodes as simple arrays of integers and simplifies arithmetics; it also ensures that our code will work on all platforms, regardless of the specific size of integers (compared to pointers).

3. Instead of supporting a variety in the number of key and data fields as proposed above, we arbitrarily determine that each node and each edge will have three "key fields": "key", "data" and "mark". We also do not perform the optimization of moving "data fields" to other locations.

4. Iteration over edges takes place using the following syntax:

```
Edge* edge=node->pEdges;
while(edge!=NULL)
{
...
edge = edge->pNext;
...
}
```

The source code defining the unoptimized graphs is presented in Figure 3. As we can see, a node starts with four integer fields: serial, key, data, and marking. For experimental purposes, we will occasionally add dummy integer slots as padding to control the number of cache lines that each node occupies. A node ends with a pointer to the first *Edge* object in the linked list. Every edge consists of three integer fields: key, data, and marking. (Edges do not have serial numbers since they are not stored in arrays). It is terminated by a pointer to the target node to which the edge connects, and a pointer to the next edge. Two library functions are used to create nodes and to connect nodes with edges.

Figure 4 shows an excerpt from the definition of optimized graphs. Each node is essentially an array of integers, allocated consecutively, so that a pointer to the node is essentially a pointer to the first cell in the array. The length of the array depends on the number of edges: following the basic fields for the node, an integer value represents the number of edges contained in this node. The contents of each edge make up the rest of the allocated space. We note that pointers to target nodes have been replaced with indices of these nodes. This allows us to maintain the optimized node

```
typedef struct Node_T {
int serial;
NodeKey nodeKey;
NodeData nodeData;
NodeMark nodeMark;
struct Edge_T* pEdges;
#ifdef PAD_SIMPLE_NODES
int temp[PAD_SIMPLE_NODES];
#endif
};
typedef struct Node_T Node;

typedef struct Edge_T {
EdgeKey edgeKey;
EdgeData edgeData;
EdgeMark edgeMark;
struct Node_T* pTarget;
struct Edge_T* pNext;
#ifdef PAD_SIMPLE_EDGES
int temp[PAD_SIMPLE_EDGES];
#endif
};
typedef struct Edge_T Edge;

/*
Creates a node with the given key and data
*/
Node* createNode(NodeKey key, NodeData data);

/*
Creates a directed edge from source to target.
Edge is attached to source.
Returns 0 for success, -1 otherwise
*/
int connectNodes(Node* source, Node* target,
EdgeKey key, EdgeData data);
```

Figure 3: Header code for simple graphs

as an array of fixed size cells, although it actually increases the cost of accesses: instead of immediately dereferencing the pointer to obtain the contents of the next node, we first consult the array of optimized nodes with the given index to retrieve the pointer to the actual node. To access individual fields of the optimized node and edges, a variety of preprocessor macros are used for readability and consistency. Nevertheless, programs can use the actual values rather than these macros.

To convert a graph into its optimized form, the programmer is expected to invoke the `optimize_simple_graph` method once the graph is considered immutable. In our manually-optimized test programs, the source graph is not destroyed and a separate immutable version is created, and the programmer can operate on either version. This allows us to perform experiments that traverse both graphs, and compare execution times. In a version that would be automatically optimized by the compiler, we will want the compiler to discover all subsequent operations on the graph, and replace operations involving the original graph with ones involving the new graph. The compiler would generate code to invoke this `optimize_simple_graph` method (which will be imported or generated automatically), and add code to destroy the original graph structure to conserve memory.

We now proceed to present the transformations which would need to take place. We write these in the form of natural text rules, which could be implemented based on the specific semantics and grammar of the compiler. Vari-

```
/* ====== Paramters for the optimized graph ====== */
#define NODE_DATA_FIELDS 3
#define EDGE_DATA_FIELDS 3
#define EDGES_OFFSET (5+OPTIMIZED_NODE_PADDING)
#define EDGE_SIZE (4+OPTIMIZED_EDGE_PADDING)
#define OFFSET_SERIAL 0
#define OFFSET_NODEKEY 1
#define OFFSET_NODEDATA 2
#define OFFSET_NODEMARK 3
#define OFFSET_NUMEDGES 4
#define OFFSET_EDGEKEY(i) (EDGES_OFFSET
+ (i)*EDGE_SIZE + 0)
#define OFFSET_EDGEDATA(i) (EDGES_OFFSET
+ (i)*EDGE_SIZE + 1)
#define OFFSET_EDGEMARK(i) (EDGES_OFFSET
+ (i)*EDGE_SIZE + 2)
#define OFFSET_EDGETARGET(i) (EDGES_OFFSET
+ (i)*EDGE_SIZE + 3)


/* Optimized nodes are essentially an int array.
  They have internal structure:
  serial (int)
  NodeKey (int)
  NodeData (int)
  NodeMark (int)
  NumEdges (int)
  [Padding]
  For each edge:
 EdgeKey (int)
 EdgeData (int)
 EdgeMark (int)
 Target index
 [Padding]
*/
typedef int* ONode;

/*
Optimize an existing graph
*/
ONode* optimizeSimpleGraph(Node** arOriginalGraph,
int numNodes);
```

Figure 4: Header code for optimized graphs

able names could be different, of course.

1. Given a variable or parameter declaration: `Node* node`, replace it with `int* node`.
2. If `node` is a variable of type `Node*` which was converted to an `int*`, apply the following transformations:
   (a) Convert `node->serial` into `node[OFFSET_SERIAL]`
   (b) Convert `node->nodeKey` into `node[OFFSET_KEY]`
   (c) Convert `node->nodeData` into `node[OFFSET_DATA]`
   (d) Convert `node->nodeMark` into `node[OFFSET_MARK]`
3. The code for iterating over edges changes as follows:
   (a) Convert `Edge* edge = node->pEdges;` into `int j=0;`
   (b) Convert `while(edge!=NULL` into `while(j<node[OFFSET_NUMEDGES])`
   (c) Convert `edge = edge -> pNext` into `j++`
4. The following changes take place to operations over edges:
   (a) Convert `edge->edgeKey` to `node[OFFSET_EDGEKEY(j)]`
   (b) Convert `edge->edgeData` to `node[OFFSET_EDGEDATA(j)]`
   (c) Convert `edge->edgeMark` to `node[OFFSET_EDGEMARK(j)]`
   (d) Convert `edge->pTarget` to `node[OFFSET_EDGETARGET(j)]`
   (e) Convert `edge->pTarget->field` to `(arrayOfNodes[node[OFFSET_EDGETARGET(j)]])[OFFSET_...]`

4. EXPERIMENTAL RESULTS

To demonstrate the benefits of our technique and try to accurately attribute performance differences to the appropriate factors, we carried a series of experiments.

4.1 Settings

We created a test program which could be controlled by multiple parameters, such as the number of nodes and edges, amount of padding to add to some data structures, and whether to execute various algorithms.

Our test program first creates a simple graph with the specified number of nodes. Then, based on a specified edges-to-nodes-ratio, it randomly connects nodes using edges. These connections are random, so that the number of edges connected to each node is not set. However, we ensure that every node has at least one outgoing edge. After this graph is created, the test program runs the code to create an optimized graph. It then proceeds to run the test section, first for the unoptimized graph, and then for the optimized graph. It produces timing information for the time it takes to pass through each. To compensate for the low granularity of the timing mechanism in C, and to ensure that performance is not affected by the original contents of the cache, the tests inside the test section are encapsulated in a loop (of a specified length), so that a test would involve, for example, running the same code a hundred times.

We devised several test functions which could be invoked by a test session. These test functions we transformed manually.

The first test is called `setAllMarks` and its purpose is to test access time to the data structure without running any actual algorithm. In these functions, an external loop iterates over all nodes in the graph (remember that they are stored in an array), and sets their mark field to a specified value. This allows us to measure access time to nodes, which is dependent on their memory organizations. Depending on a specific precompiler flag, the function also marks all the edges: an internal loop iterates over all the edges for the given node and marks them. An important property of this test program is that it does not make any function calls, thus reducing their effects on program behavior. For this reason, we expect to to be able to be able to strictly measure and compare the impact of locality in the unoptimized and optimized representations.

In addition to the `setAllMarks` function, we implemented versions of the BFS and DFS algorithms, allowing them to be started from specific nodes in the structure. The `setAllMarks` function is always used before BFS and DFS to set the mark fields. We will discuss these algorithms later.

A potential factor which could confound the effects of our transformation involves memory allocation: Our test program creates the simple graph by creating all the nodes, and then creating all the edges. Since memory is relatively free at that point and each object is relatively small, it is quite possible that several objects would be placed so close together that they would reside on the same cache line. A similar scenario could happen for the combined nodes in the optimized graph (although that may be beneficial). However, since we want to measure the effects of our transformations independent of memory allocation effects, we devised a way to minimize the number of accidental collocations by padding objects to the size of a cache line.

Following are some of the parameters which our program supports:

NUMNODES Specifies the number of nodes to be randomized in the graph.

ENRATIO The edge-node ratio determines how many edges we would have per node in the graph. Our test program is structured so that there has to be at least one edge leaving every node. Additional edges are randomized. The ratio must therefore be at least 1.

NUM_RUNS The number of times to run each test function.

PAD_SIMPLE_NODES A number of integers added to simple (unoptimized) nodes to ensure that no other object will occupy the same cache line.

PAD_SIMPLE_EDGES A number of integers added to simple edges to ensure that no other object will occupy the same cache line.

OPTIMIZED_NODE_PADDING A number added to the start index of the edges in optimized nodes, effectively creating a padding. This allows us to ensure that the edges begin on a different cache line.

OPTIMIZED_EDGE_PADDING A number added to the start index of the next (not first) edge in optimized nodes, effectively creating a padding. This allows us to ensure that each edge lies on a different cache line.

The last four parameters allow us to control potential locality. We say that an experiment is carried in *locality* mode if we do not explicitly prevent two memory allocations from sharing on the same cache line, though this does not imply that we take any special measures to ensure that they would. We say than an experiment is carried in *no-locality* mode if we ensured, via padding that no two objects (or relevant part of the objects) would lie on the same cache line. In other words, in non-locality mode each access to a previously-unaccessed object will generate a cache miss (at least in L1), whereas we cannot be sure of that in locality mode.

We conducted our experiments on an Apple Macintosh G4 running OS X. One of the reasons for choosing this architecture is its consistency in cache line size, which is 32 bytes (or 8 ints/pointers) for both the L1 and L2 caches. In addition, the G4, being an older, simpler, processor, does not perform hyperthreading or other behaviors that could affect our results, and there is less variance between processor steppings. To ensure that padding is sufficient and that we are not fighting a fetching unit that may fetch more than one cache line at a time, we pad each memory object with an additional cache line, or 32 bytes.

Our test program allows us to set the number of nodes and edges. We chose such numbers that allow us to run sophisticated enough graphs, without running out of physical memory, which could result in confounding effects from the virtual memory manager. Furthermore, we chose numbers small enough to ensure that each graph representation could be stored completely within the L2 cache. Performance difference due to locality would then be due to the fact that the processor can reduce accesses to its L2 cache, retrieving the same information from L1. For bigger graphs that do not fit in L2 cache, we expect to see similar effects as access to regular memory is necessary. However, measurements for such situations would be more difficult due to effects of paging and multi-level caching.

Because our graph optimization approach involves the reallocation of nodes and edges, it is natural to suppose that the ratio of edges to nodes, and hence the distance in memory of various allocated objects, is important. As a result, to isolate the impact of a variable edge-to-node ratio from variation in other system properties, each experiment ensures that the memory footprint is fixed; as the number of edges in the graph increases, the number of nodes in the graph decreases accordingly.

4.2 Experiments: Single access to each object

Hypothesis 1: In non-locality mode, programs that operate exactly once on each node and do not operate on edges should have similar relative running times for both unoptimized and optimized representations.

The rationale behind this hypothesis is that programs that do not operate on nodes will not be affected by the significant improvements we expect for edges due to the reduction in levels of indirection. In addition, the non-locality mode would reduce cases in which nodes (especially in the unoptimized graph) are allocated to the same cache line. The code traversing nodes in the optimized graph differs slightly from the code traversing the unoptimized graph. Since there is no access locality, any difference in performance should be due to the code transformation. Note that by operating 'exactly once' we mean once per test iteration.

Hypothesis 2: In non-locality mode, programs that operate once on every node and edge should have similar relative running times for both unoptimized and optimized representations.

Since every node and every edge are accessed exactly once, we expect the same number of overall accesses regardless of the exact ratio between edges and nodes.[1]. Having padded each memory object, there should not be two accesses to the same cache line. As in **Hypothesis 1**, any performance difference between operations on the optimized and unoptimized graphs should be due to the code transformation.

Hyptothesis 3: If locality is not prevented, programs that operate once on every node and edge should run faster on the optimized version, with the performance advantage increasing as the edge-to-node ratio increases.

The primary benefit of our optimization is that all the incident edges of a node are stored together, and as a result are more likely to fall within the same cache line if locality is

[1]Remember that we adjust the number of nodes as we increase the ratio of edges

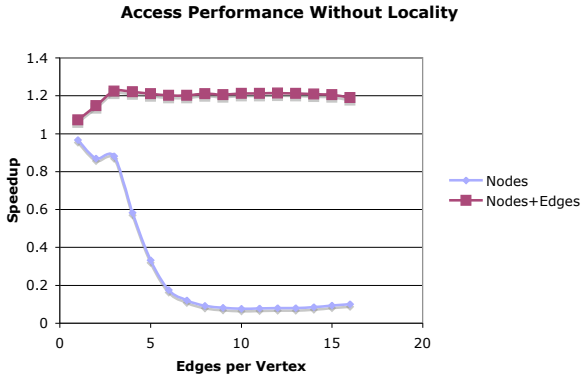

Figure 5: Experimental results for node and edge markings with locality disabled

not prevented. As a result, we expect that sequential access to the edges would cause less cache misses and thus improve performance. In the unoptimized version, each edge is likely to reside in a different location in memory, or at least on a different cache line. On the other hand, if locality is not prevented, the nodes of the original graph are more likely to be allocated on the same cache lines since at the time the memory is relatively empty and the nodes may be allocated contiguously.[2] Though these two factors conflict, we expect performance of the optimized version to improve as the edge-to-node ratio increases.

We proceeded to test the first two hypotheses using the `setAllMarks` function, once iterating only over nodes, and again iterating over both nodes and edges. We ran the program for each edge ratio from 1 to 16, and adjusted the number of nodes such that the total number of nodes and edges is always 12000, small enough to fit in L2 cache.[3] For the nodes-only test, we used a `NUM_RUNS` of 10000, and used 2500 for the nodes-and-edges test. We ran many of the trials multiple times and averaged the results, to accommodate execution time variation, which was pronounced in the nodes-only test. We present the speedup of the optimized version with respect to the unoptimized version for these tests in Fig. 5.

As we can see, they are quite different from our expectations for the first hypothesis, but fall in line with the second one.

For the programs iterating only over nodes, performance of the optimized version is slower for the optimized version than for the unoptimized version. It gets progressively worse as the edge-to-node ratio increases, bottoming at a staggering 90% performance hit as the ratio goes above 7. How can we explain this discrepancy? We believe that these results may be due to the nature of the test program in nodes-only mode, which utilizes the entire L2 cache to store the graph structure, but only traverses a small portion of this data. We apply a uniform amount of padding (at the granularity of entire cache lines) to each node and edge in both representations of the graphs. As a result, the portion of the graph accessed is likely to be mapped to a small subset of L1 cache lines. For instance, every node and edge is followed by an empty cache line, so that only "even" lines are used.

This underutilization of the L1 cache results in the significant performance hit on both implementations. However, whereas in the unoptimized graph all nodes could be allocated one after the other, this does not happen in the optimized graph, where at least one edge follows each node. Thus, the nodes of the optimized graph share an even smaller subset of L1 cache lines than do the nodes of the unoptimized graph. This explanation is supported by our observation that when the padding of the unoptimized graph is increased the traversal time also increases. When node objects in the unoptimized graph are so large that they are spaced one from the next by the same distance that separates nodes in the optimized graph, the unoptimized graph actually takes longer to traverse than the optimized graph. We argue that since this performance problem is due to the artificial strategy we chose in order to eliminate locality for experimental purposes, it should not be of great concern.

The results for the nodes-and-edges run match our hypothesis, and for ratios of more than 1, we see the optimized version consistently performing faster by about 20%. This difference is likely due to differences in the code, and in particular the fact that there is less indirection. Nevertheless, the difference is held constant as there is no benefit to locality. One may ask why we are not seeing a performance problem related to the L1 cache line issue described above. The reason is likely to be that the performance hit is equivalent for both implementations, since we now access all nodes and all edges.

We therefore test the third hypothesis, running the experiment for the nodes-and-edges version of `setAllMarks`. As we can see in Fig. 6, performance is significantly better for the optimized version, starting with a speedup of 1.75x for a ratio of 1, and leveling off at speedup of 3.5x as the ratio approaches 15. Our third hypothesis is therefore confirmed: as the ratio of edges to nodes increases, the locality in the optimized representation also increases, thereby increasing the execution time improvement possible. As there is no computation occurring per access, and edges are accessed in the optimal order, 3.5x is an excellent approximation of the maximum performance benefit possible with the optimized graph representation. Further improvement may be possible when the representation enables greater temporal locality; specifically examining this side-effect of the optimized representation is beyond the scope of this project.

4.3 Experiments: Actual graph algorithms

A method like `setAllMarks` is useful for demonstrating the limits of improvement possible with the optimized representation. However, because it traverses the entire data structure once and in a predictable way, it does not demonstrate the benefits of our approach to real world applications.

One of the potential uses we forsee for our optimization of graphs is in search-intensive programs. For instance, consider a system that must answer a large number of search queries on a relatively stable graph structure, such as a mapping system, ambulance routing systems, and the like. To demonstrate the potential benefit for such applications, we implemented several common search algorithms: breadth-first search (BFS), depth-first search (DFS), and A^*.

[2]Edges are also allocated sequentially in memory, but we randomize the edges so the edges of each node are not stored contiguously.

[3]Since we run each test multiple times, we expect all the contents to be moved to the L2 cache after the first test iteration.

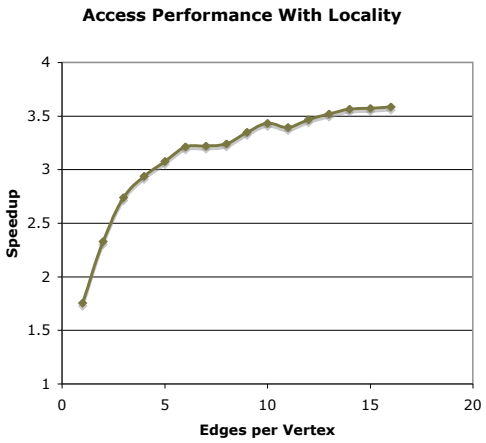

Figure 6: Experimental results for node and edge markings with locality enabled

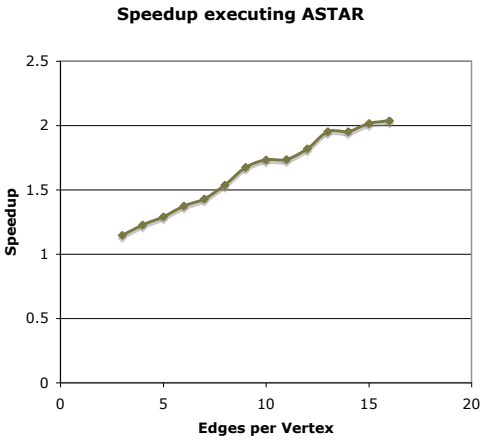

Figure 8: Experimental results for the A^* search algortihm

BFS and DFS can be implemented using recursion or a worklist. The choice of implementation may have significant performance impact, especially under our optimization. In the worklist version, more natural for BFS, as each node is explored, we iterate over all its the edges and push their target nodes into the worklist. Since our optimization reduces cache misses for accessing the edges of the same node, we expect to see a significant performance improvement. In the recursive version, more natural for DFS, we make a recursive call after exploring each edge. As a result, more time may pass between the traversals of two edges of the same node, during which relevant cache lines may be evicted. In addition, a performance hit can be expected due to the heavy use of function calls. The results of executing a worklist version of BFS and a recursive version of DFS for a variety of edge-to-node ratios are presented in Fig. 7. Note that to prevent graphs from being too sparse, we used graphs with a minimum of 4 edges per node.

As we can see from the results of Fig. 7, both algorithms exhibit a significant performance improvement for the optimized version. This improvement increases as the edge-to-node ratio increases, further supporting the usefulness of our edge-oriented optimization.

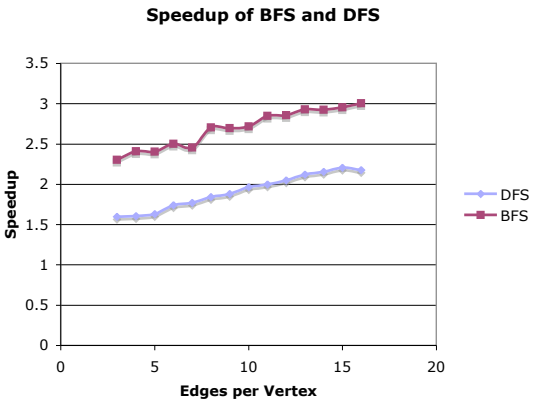

Figure 7: Experimental results for DFS and BFS

DFS exhibits performance improvements similar to that when there is no locality (Fig. 5), because DFS does not do a good job of taking advantage of edge locality: in the worst-case, only one edge per node is accessed before locality is broken by eviction. As the edge-vertex ratio increases, DFS does a better job of taking advantage of locality since the depth of the search decreases, meaning there is a higher probability that a yet untouched edge will still be in cache when the search returns to it. The optimized graph representation speeds up DFS by 1.5x to 2.1x.

BFS on the other hand, exhibits performance improvements closer to the maximum possible with spatial locality (Fig 6). As opposed to DFS, BFS visits all edges belonging to a node, taking advantage of edge locality. The performance advantage of the optimized graph increase as the edge-to-node ratio increases, as expected. The optimized graph representation speeds up BFS by 2.3x to 5x.

In addition to implementing the deterministic BFS and DFS algorithms, we implemented the more sophisticated A^* search algorithm. A^* is a heuristic for optimal graph search, and is commonly used in computer-aided design placement and route algorithms. The algorithm is more computationally heavy than either BFS or DFS, and uses a priority queue which introduces additional computational complexity and memory accesses for every node or edge accessed. As a result, we would expect A^* to not benefit from the optimized representation as much as the simpler algorithms presented above. The results of executing this algorithm for the unoptimized and optimized versions of the graph are presented in Fig. 8.

In spite of the additional overhead of the A^* search, the optimized graph algorithm provides substantial speedup, from 1-2x, compared to the original representation. Like BFS, A^* expands the search from a single node to its neighbors, taking advantage of the edge locality present in the optimized graph representation. As the number of edges in the graph increases, this locality, and thus the speedup of the optimized version relative to the unoptimized version, increases.

5. FUTURE WORK

Though the above experimentation demonstrates the ben-

efit of reorganizing graphs, compilers must be extended to minimally recognize the point at which a graph is intended to be immutable, and insert code to transform the graph, and transform all subsequent code that accesses the graph. Given a more flexible graph representation than the optimized representation presented here, a more sophisticated compilers could, go further and either perform compile-time, or static analysis, or insert instrumentation to perform run-time profiling, or dynamic analysis. Reallocating graphs for locality provides significant speedup, even when the allocation scheme (e.g., grouping edges with nodes) doesn't match the access pattern of a program (e.g., DFS). Further customizing the reallocation to the specific access patterns should yield even greater, performance, as evidenced by the difference in performance between BFS and DFS above.

5.1 Static Analysis

Static analysis must, at compile-time, determine how best to reorganize the graph in question. Structured access to graph elements, or strong typing in a high-level language, could be used to identify points in the program where graph elements are being accessed. Identifying the order in which node and edge operations occur could be leveraged at compile-time to provide hints to the run-time reallocator about whether or not nodes should be allocated together, nodes allocated with their edges, or some combination. Because of the nature of graph algorithms (which iteratively traverse graphs, making it nearly impossible to distinguish accesses between elements), static analysis can yield little other information than what the relationship is between node and edge accesses. For example, unrolling the worklist loop in our BFS implementation would reveal that for each node that is accessed, edges are accessed in a loop, indicating that edge locality may benefit the algorithm. Alternatively, unrolling a series of recursive DFS calls would reveal that for each node that is accessed, there is one edge access before nodes are accessed again.

5.2 Dynamic Analysis

Graph traversal can also be analyzed through the process of run-time profiling. As demonstrated above, the precise manner in which graphs are traversed significantly impacts the performance benefit of a particular graph representation. Because profiling could monitor and record the precise order in which individual graph elements are accessed, dynamic analysis theoretically could provide greater additional speedup than static analysis techniques alone. Dynamic analysis requires that the compile be able to reallocate an optimized graph after the optimized graph has already been created, since as we've suggested graphs typically aren't traversed until they are fully constructed. Dynamic analysis, then, would provide additional run-time information about how an already optimized graph should be reallocated *again* to further improve graph traversal latency. This clearly raises the issue of how frequently graphs should be reallocated; discussion of this issue is beyond the scope of this project.

The most straight-forward approach to run-time profiling would be to monitor the pattern of edge and node accesses, as in static analysis. Dynamic analysis, however, can precisely measure the sequence of node and edge accesses, possibly providing a more accurate picture of how the graph is being traversed. As in static analysis, this information could be used to determine how nodes and edges are allocated: whether nodes are allocated with nodes, edges with edges, nodes with edges, or any hybrid of these possibilities. Such an approach to determining how to allocate graph objects would require a more flexible representation than presented here, e.g., one that supported multiple modes of object access based on how the graph is organized in memory.

The next logical step in dynamic analysis would be to monitor and record the order in which individual graph elements are accessed, and allocate them closer to each other in memory. This is particularly useful if search algorithms tend to begin from a small subset of nodes or if a graph is sparsely connected (with little branching from node to node). Nodes and edges that are frequently accessed in a fixed order can be allocated next to each other in memory, further increasing access locality. It is obvious that this approach would also significantly improve the traversal of very large graphs that might not fit in cache. Ensuring that nodes and edges that are accessed at approximately at the same time are all available in the same page of virtual memory could significantly improve traversal latency by reducing page faults.

6. CONCLUSIONS

Graph traversal is common in many applications. Traditional graph organization is not conducive to high-performance execution, however, as traversing disparately allocated nodes and edges leads to long sequences of dependent cache misses. We have shown that graph traversal execution time can be greatly reduced through a transformation that flattens (and fixes) the data structure, removing indirection and increasing locality. Our proposed transformation reallocates a graph when the programmer indicates that it will no longer change. At this time, the graph is reallocated, nodes and edges rearranged in memory so that each node and its edges are organized as an array of integer an accessed directly. These transformations were shown to speed up the execution time of DFS, BFS, and A^* by up to 2.1x, 3x, and 2x respectively. The proposed representation could be easily integrated with a compiler, but extending the representation to more flexibly select the placement of nodes and edges could improve graph traversal latency with the support of static, compile-time analysis or dynamic, run-time analysis of application-specific graph traversals.