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

Uri Dekel and Brett Meyer

ABSTRACT

This project tries to improve the cache-locality of programs which frequently traverse an immutable graph data structure. In our approach, the compiler generates code to fold the inherently noncontiguous representation of vertices in a mutable graph into a more contiguous representation possible only for immutable graphs. In addition to this internal reorganization, the generated code will also try to change the external organization of vertices that are likely to be accessed sequentially in order to increase the likelihood of them residing a single cache line. Our implementation plans include laying the infrastructure, and testing a simple clustering algorithm.

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 same 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 anticipating 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. At present, few architectures offer programmers the means to control prefetching into the memory cache.

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 types 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 then 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 an intuition (which follows personal experiences), that most graph traversals take place after the graph has been created and its structure stabilized and becomes immutable For instance, in many compilers we have different optimizations that 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. At present, however, few programs do this explicitly, because of the significant required effort. 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

(A more thorough survey of previous work will be presented in the actual project report)

The idea of reorganizing data to improve cache-performance is not new. For example, Chilimbri and Larus [2, 3] 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 [1] 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 also forces us to pose many limitations on the programs that could be optimized, and in particular to require direct input from the programmer. 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.

Second, we assume that graphs are represented using a structure which will be described below. 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. Traversal of edges must take place via appropriate library functions, etc. Also, no modifications may be made to the structure of an immutable graph, although changes to key and data fields are allowed.

Third, 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.

2.2 Source representation of graphs

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, where edge objects maintain edge properties and the identifiers or pointers to the vertex objects, and vertex objects maintain the properties of each vertex. The other representation does not use edge objects at all. Instead, each node maintains a collection of its neighboring nodes, possibly with properties for the edges that connect them. In this project we are going to focus on this latter sub-representation, optimizing programs which utilize vertices that maintain their own connections to other vertices. 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

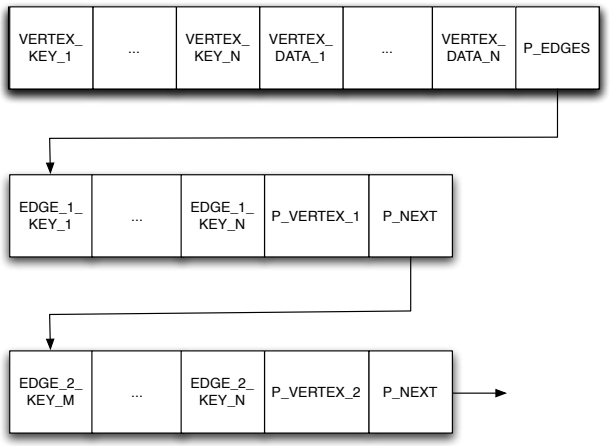

Figure 1: Layout of a vertex in a mutable graph

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 Optimized graphs

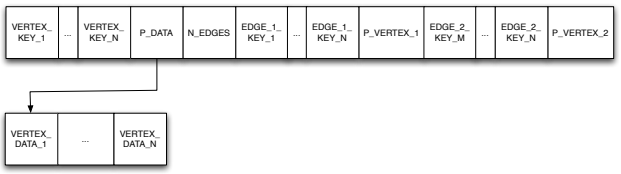

Figure 2: Layout of a vertex in an immutable graph

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.

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

2.4 Evaluation model

We will create benchmark programs that utilize a variety of common graph algorithms. We will then produce comparisons of the performance with our two optimizations (flattening vertices, and placing related vertices on the same cache lines). In addition to measuring performance, we will ensure that outputs are the same.

2.5 Platform and language

At present, we have not yet decided on a specific language which we will optimize, nor on the exact compiler infrastructure we will use. Evaluating different options is a natural first step in our plan.

3. PLAN

At present, no work has been done beyond the writing of this proposal. The following steps will be carried out:

- Decide on an implementation language and compiler infrastructure
- Define specifics of user input, including exact source structure and function call.
- Define (in writing) specific transformations that must take place (e.g., data field accesses)

- Implement compiler extension to identify method call and source structure
- Implement compiler extension to generate immutable form
- Apply necessary translations to ensure rest of program utilizes immutable form
- Run benchmarks on programs with immutable forms
- Modify immutable form generation exception to try and place multiple vertices on same memory page.
- Run benchmarks again.

4. REFERENCES

[1] S. Chen and N. Hardavellas. Improving performance through data object fusion - final project report for 745 course, 2003.

[2] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 13–24, New York, NY, USA, 1999. ACM Press.

[3] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 37–48, New York, NY, USA, 1998. ACM Press.