

# Improving Performance through Object Fusion

Shelley Chen

Department of Electrical and Computer Engineering

Nikos Hardavellas

Department of Computer Science

Carnegie Mellon University

Pittsburgh, PA 15213

{shelleychen, hardavellas}@cmu.edu

<http://www.ece.cmu.edu/~schen1/cs745>

## 1. Introduction

Technological advancements in semiconductor fabrication, coupled with architectural innovation, have resulted in lightning-speed improvements in processor performance, which doubles every eighteen months. While memory capacity has increased commensurately with processor performance, memory access speeds improve at the glacial rate of almost 10% annually [9]. The memory wall is rising fast, and promotes memory reference behavior to the dominant factor determining overall performance of many important applications (e.g., commercial database systems spend 25% to 50% of their execution time waiting at the memory system [10]). To alleviate this problem we propose object<sup>1</sup> fusion, a compile time technique that co-locates elements of a pointer-based data structure which are likely to be accessed contemporaneously, and re-orders their fields to improve cache line utilization during traversal.

### 1.1 Motivation

To reduce the processor/memory performance gap, architects have conventionally employed a cache hierarchy [12][13]. In this model, faster but smaller cache levels keep the most recently referenced data close to the processor, while larger cache levels provide backup storage for the faster levels at the cost of slower access times. As the memory wall keeps rising, architects employ multiple levels of cache hierarchies to hide memory latency (e.g., many high-end designs now incorporate L3 board-level caches). As cache hierarchies become deeper, the distance between the processor and the lower memory levels grows. Unfortunately, the limited capacity and simple data placement mechanisms often result in high miss rates in the cache hierarchy. While superscalar engines with non-blocking caches [8] allow overlapping the data miss latency among the higher cache levels, limited available instruction-level parallelism and long miss latencies to lower cache levels often expose data cache miss latency in many important classes of applications.

High miss rates are primarily caused by poor reference locality. It is not rare for applications to bring a whole cache line worth of data into the cache in order to use only a single word, resulting in poor cache utilization. The problem of poor cache utilization is more pronounced in applications using pointer-based data structures. Data structure traversal in these applications is done via a tight loop that compares the key field of an element against

the key of the search and moves to the next element by de-referencing a pointer if a match is not found. Often the key field of the element and the pointer to the next element lie in different cache lines resulting in large footprints during search that stress the cache hierarchy. Even worse, such applications suffer from the pointer-chasing problem: the location of the next object is not known until the pointer to it is loaded along with the current object.

In general, cache hierarchies attempt to enhance program performance by exploiting its reference locality. A program's data reference locality can be further improved by (1) changing the order in which the program accesses data -- inherent locality, or (2) changing the mapping of data objects to memory addresses -- realized locality [7].

Improving a program's reference locality by changing its inherent locality is usually done via prefetching which can tolerate memory latency [11]. However, prefetching for pointer-chasing codes remains a challenging problem, and requires sufficient amount of overlapping computation to exist in order to hide the prefetch latency. Moreover, techniques that change a program's inherent locality still do not address the poor utilization of the cache. Hence, in this work we focus on improving a program's realized locality.

Optimizing a program's layout for caches to enhance its realized locality is usually done using techniques such as compression, coloring and clustering. Compression typically requires source code modification which is not practical for very big and complex applications, it is error prone, and also places an extra burden to the programmer who should solely focus on concrete and sound implementations of his algorithms. Coloring techniques reduce cache conflict misses caused by multiple blocks mapping to the same location. Clustering attempts to enhance spatial locality by co-locating data objects that are likely to be accessed contemporaneously. Clustering via source code modification obfuscates algorithms, makes source code maintenance more difficult, increases the probability of programming errors, and is not portable. Clustering via static analysis can be a more challenging problem, but its transparency makes it appealing. In this work we plan to use clustering and coloring techniques to improve the realized locality of an application.

### 1.2 Our Approach: Object Fusion

Even though pointer-based data structures challenge current techniques aiming to improve a program's reference locality, they possess the powerful property of *locational transparency* [6]: elements in a structure can be placed at different memory (and

---

<sup>1</sup> Throughout the report, we use the word objects to refer to C-like structures, as well as objects of an object-oriented language.

cache) locations without affecting the program's semantics. We can exploit this property and carefully place together objects that are likely to be accessed near in time. To achieve this, we pack a small amount of objects together within a contiguous piece of memory, forming a hyperobject. We can then ignore pointers from one object to another within the hyperobject, as we can statically calculate the address of any object it encapsulates. Pointers connecting two elements that belong to different hyperobjects are still required to maintain the integrity of the original data structure.

Objects within the hyperobject now possess a new powerful property: uniform, random access to any of their fields. This allows a compiler to analyze fields within a hyperobject completely, and to reorder them in a way that increases spatial locality without affecting a program's result. To improve data structure traversal, all search keys can be placed within a single cache line, followed by pointers to the other hyperobjects. Traversing through the elements of the hyperobject is reduced to loading the cache line with all search keys and scanning through them, fully utilizing it. In the mean time, prefetching can load the cache line of the hyperobject holding the pointers to the succeeding hyperobjects ahead in time, so that a pointer can be dereferenced if a match is not found. Prefetching of hyperobjects is an orthogonal optimization which we do not plan to investigate in this work.

As always, there is some hidden complexity. Conflicting placement requirements limit the optimization potential as only one requirement can be satisfied at a time. Also, it is not uncommon for applications to have data elements participate in multiple pointer-based data structures. Reordering one data structure may degrade performance when traversing through the elements using the other data structure. However, usually one of this data structures is most heavily used and dominates the execution time. For this project, we focus on optimizing the data structure of the dominant traversal. The user is charged with the responsibility of providing this data structure to the compiler for further analysis. Ideally prefetching could be used to identify the dominant traversal, but due to time constraints we have to limit the focus of our work.

Since we heavily rely on locational transparency, it is imperative that the elements we reorder in the address space are being accessed through pointer-based structures only. The user is charged with the responsibility of guaranteeing the safety of the optimization.

### 1.3 Prior Work

Rao and Ross proposed a technique using Cache-Sensitive Search Trees (CSS-Trees) [1]. The nodes of the tree are stored in an array, so the children of a node can be directly computed from the parent's address, eliminating the need for child pointers. Unfortunately, the insertion and deletion of nodes is not supported.

Chen, *et al.*, introduced the use of prefetching to reduce the amount of time needed to scan or search through a B+-tree data structure [2]. With multiple cache and memory banks and crossbar interconnects, cache misses can be overlapped, reducing the penalty of memory accesses. In addition, a jump-pointer array allows for continual prefetching of future nodes, so the actual leaf

nodes do not need to be dereferenced to obtain the next pointers. This technique exercises clustering of objects, where we propose to fuse the objects together into a hyperobject structure.

Class co-location is where small classes with high temporal affinity are placed in the same cache block [3]. This reduces the number of memory accesses that need to be tolerated in order to access these two objects. Unfortunately, this technique is only useful for objects that are smaller than half the size of the cache block.

Field relocation [4], also referred to as field reorganization [5], is a useful technique that can be applied to objects that are equivalent to, or larger than, the size of the cache block. Basically, the fields of a data structure that are contemporaneously accessed are placed in the same cache line, reducing the memory latency of accessing the fields.

Structure splitting is another technique that is applied to larger data structures [4]. It should be used in conjunction with the class co-location technique proposed by Chilimbi *et al.* Structure splitting divides a large data structure into two smaller data structures. One data structure contains the hot fields, or the fields that are frequently accessed. The other will contain cold data structures, or rarely accessed fields. The hot data structure contains a pointer that references the cold data structure, introducing an extra level of indirection needed in order to access the cold fields.

## 2. Object Fusion Design

As mentioned before, in this work we focus on optimizing traversals of a pointer-based data structure based on its dominant traversal method. We assume that the elements of the data structure are being accessed through pointers only. We also assume tree-like acyclic structures for which there are no pointers to the middle of the data structure. It is the user's responsibility to guarantee the safety of the optimization and to provide the dominant data structure. The user provides the compiler with the location of the search key within the elements of the data structure, the number and location of the pointers used in the traversal, and a pointer to the function describing how we move from one object to the next.

Since we focus on tree-like data structures, we partition the structure in subtrees and fuse those objects together in a contiguous block of memory. This way we optimize for random traversals, since we always use more than one object in a hyperobject we access regardless of the actual path the traversal takes. Note that if the dominant traversal method is something else (e.g., depth first search) we may choose to partition the structure in that order. However, due to time constraints we will not pursue this any further.

### 2.1 Fused Object Layout

The field reordering of the fused objects is done as follows: the first cache line of the hyperobject will hold the search keys of the fused objects. Hence, the number of objects to fuse together is determined by the maximum number of search keys that fit within a single cache line. The following cache lines will hold the pointers from the fused objects to other hyperobjects. The number of cache lines needed depends in the fan-out of the objects we fuse. The remaining cache lines will hold any other data that the

original objects have. Note that since the objects within a hyperobject are laid out in contiguous address space, pointers among them are no longer required and thus we delete them. Moving from one object to another within a hyperobject can be done via simple static address calculations. Similarly, accessing the fields of any object can again be done using simple static address calculations.

Other design points include fusing together as many objects as possible, such that the search keys and the pointers to other hyperobjects are all packed within a single cache line. That way, traversing through a hyperobject can be done by accessing only one cache line, as opposed to a minimum of two if one line is completely filled with search keys. However, note that at any pointer dereference to a hyperobject we can also prefetch the cache lines that hold the pointers to the other hyperobjects and hide the latency. Hence, our solution doesn't seem to have a significant disadvantage and we can also co-locate more objects, increasing the potential benefit of the optimization.

An alternative design point is to determine the size of the subtree in order to completely overlap the latency of a subsequent pointer dereference to another hyperobject. At this design point, keys may potentially span multiple cache lines. The subtree size can be estimated at compile time through analytical calculations, but due to time constraints we will not pursue this any further either.

In the original data structure, search keys are trivially used for searching the data structure. If an object exists, its key would be meaningful to the application traversing it. With object fusion, though, this is no longer true. In order to minimize the complexity of the traversal algorithm, a cache line is set aside for search keys, even if the subtree has fewer objects than required to fill it up. Thus, a way to determine whether a key field is valid is required. For this, we use a bitmask located at the cache line that holds the search keys. A search key is valid if and only if its corresponding bit in the bitmask is set.

Coloring techniques can be used to ensure that the highest levels of the tree-like structure are mapped to a partition of the cache in order to minimize conflicts. We propose to color the subtrees that are closest to the root (single entry point to the data structure), since some of these are likely to be used in any traversal. If time permits, we will attempt to implement coloring schemes for the final version of our technique.

## 2.2 Data Structure Operations: Traversal, Insertion, and Deletion

In order to traverse the modified data structure, a nested loop is needed. The external loop will traverse the hyperobjects, by dereferencing the pointers at the leaves of the fused subtree. The internal loop will traverse the fused elements within the hyperobject, by statically calculating the address of the field that needs to be accessed within the hyperobject. Standard compiler analysis is sufficient in order to perform the necessary code transformations of the traversal method.

Fusing objects together into hyperobjects violates their locational transparency. This gives rise to the problem of performing insertion and deletion. Fortunately, it is not difficult to construct an algorithm that supports dynamic insertion and deletion at the modified data structure. If the new element is inserted between

two fused elements in a hyperobject that has space to hold it, we simply move elements accordingly within the hyperobject to make room for the new element. If the hyperobject does not have enough space for the new element, a new hyperobject has to be allocated and the appropriate element from the current hyperobject should be moved into the newly allocated one. The new hyperobject is linked appropriately with the other hyperobjects, and then we perform the insertion of the element similarly to the former case.

Dynamic deletion follows a similar algorithm. Note that insertion and deletion may increase fragmentation of the address space. Moreover, they may require multiple memory copies which can negatively affect overall performance. Hence, our technique is more suitable to mostly-read data structures. Insertion and deletion can be supported, but the additional overhead associated with them may cancel the optimization's benefits.

If insertions and deletions are performed regularly, the compiler may insert code that periodically reorders parts of the data structure to minimize address space fragmentation and improve cache line utilization. Again, due to time constraints we do not plan to explore this any further.

## 3. Proposed Evaluation Method

In order to evaluate our proposal, we plan to apply the compile time transformations on Olden benchmarks. This benchmark suite is composed of applications that stress the memory system with pointer-based data structures. Since our transformations are not safe, we pick applications that conform to our assumptions. At this point it appears that TreeAdd, Health, MST, and Perimeter are suitable for evaluation purposes. TreeAdd sums the values stored in the nodes of a binary tree, Health simulates a health care system using doubly linked lists, MST computes the minimum spanning tree of a graph using an array of singly linked lists, and Perimeter computes the perimeter of regions in images using a quadtree structure.

Our metric is overall execution time. If it is possible to obtain exclusive access to an Alpha system for measurements on real hardware, we plan to use hardware counters and performance analysis tools like DCPI to measure the impact of our technique on performance. If we can not obtain exclusive access to real hardware, we plan to compile the benchmarks for SimpleScalar and use simulation methods for the evaluation.

## 4. Logistics

### 4.1 Research Infrastructure and Resources

In class, we have been using the MachSUIF tool to implement compiler optimizations. However, since we are working with data structures, it seems cumbersome to perform optimizations on data structures at the assembly level. Thus, we will investigate if we should use SUIF2 instead, which implements optimizations at a higher level.

The software tools we need are readily available and can be compiled for Alphas. We plan to use the ECE Bone Alpha cluster for development and evaluation. In case we need to revert to simulation methods for the evaluation, SimpleScalar is also

available. Finally, the Olden benchmarks can be compiled both natively, and for SimpleScalar use.

## 4.2 Plan of Work

### 4.2.1 Schedule

Below is the weekly schedule we plan to follow:

1. Determine appropriateness of MachSUIF or SUIF2. Select Olden benchmarks and ensure safety of optimization. Compile Olden on Alphas with SUIF. Determine whether exclusive access to an Alpha machine is possible. If not, compile Olden benchmarks for SimpleScalar.
2. Implement object fusion layout.
3. Implement data structure traversal algorithm. Prefetch cache lines holding pointers to other hyperobjects when a hyperobject is accessed.
4. Implement insertion/deletion. Major bug fixing.
5. Evaluation and minor bug fixing.
6. Write up report. If there is time, support coloring.

### 4.2.2 Milestone

By Monday, April 14<sup>th</sup>, we will have the single cache-line implementation of the object fusion technique, with support for insertion and deletion of elements, and prefetching of the hyperobject pointers.

## 4.3 Getting Started

The only work that has been done has been the research that was needed for this proposal. No implementation has been done yet.

## 5. REFERENCES

- [1] J. Rao, K. A. Ross. "Cache Conscious Indexing for Decision-Support in Main Memory." In *Proceedings of the 25<sup>th</sup> VLDB*, 1999.
- [2] S. Chen, P. B. Gibbons, T. C. Mowry. "Improving Index Performance through Prefetching." In *Proceedings of the SIGMOD 2001 Conference*, May 2001.
- [3] T. M. Chilimbi, J. R. Larus. "Using generational garbage collection to implement cache-conscious data placement." In *Proceedings of the 1998 International Symposium on Memory Management*, October 1997.
- [4] T. M. Chilimbi, B. Davidson, J. R. Larus. "Cache-Conscious Structure Definition." In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999.
- [5] D. Troung, F. Bodin, A. Seznic. "Improving cache behavior of dynamically allocated data structures." In *International Conference of Parallel Architectures and Compilation Techniques*, October 1998.
- [6] T. M. Chilimbi, M. D. Hill, J. R. Larus. "Cache Conscious Structure Layout." In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999.
- [7] T. M. Chilimbi. "Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality." In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, June 2001.
- [8] D. Kroft. "Lockup-free instruction fetch/prefetch cache organization". In *Proceedings of 8<sup>th</sup> Annual International Symposium on Computer Architecture*, May 1981.
- [9] D. A. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keaton, C. Kozyrakis, R. Thomas, K. Yelick. "A case for intelligent RAM." In *IEEE Micro*, pp. 34-44, April 1997
- [10] A. Ailamaki, D. J. DeWitt, M. D. Hill, D. A. Wood. "DBMSs on a Modern Processor: Where Does Time Go?" In *Proceedings of VLDB Conference*, 1999.
- [11] C-K. Luk, T. Mowry. "Compiler-based prefetching for recursive data structures." In *Proceedings of 7<sup>th</sup> International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [12] A. J. Smith. "Cache Memories." In *ACM Computing Surveys*, 14(3):473-530, 1982.
- [13] M. V. Wilkes. "Slave memories and dynamic storage allocation." In *IEEE Transactions on Electronic Computers*, pp. 270-271, April 1965.
- [14] D. Callahan, K. Kennedy, A. Poterfield. "Software Prefetching." In *Proceedings of the 4<sup>th</sup> International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.