

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 object¹ 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

¹ 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 block 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 information about 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 no 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 tree-like acyclic data structure based on its dominant traversal method. We partition the structure in subtrees and fuse those objects together in a contiguous block of memory. Figure 1 shows an example of a tree before object fusion is performed. In the figure, all the encircled nodes can be copied into a single hyperobject.

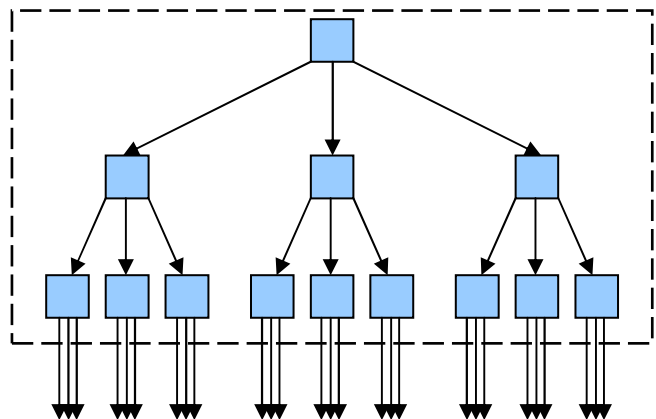


Figure 1: Tree structure. The Object Fusion optimization creates a single hyperobject node containing all the data of the encircled nodes.

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. For example, a binary tree with

32-bit search keys can be fused into a structure with 15 nodes in each hyperobject, comprising 4 complete levels. While the original structure could potentially suffer one cache miss per node access, the hyperobject structure would suffer at most one miss per four nodes accessed, thereby potentially reducing the cache misses by 75%. 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 Structure Transformation and 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.

```

struct treenode {
    <type1> Key;
    struct treenode *Next[c];
    <type2> Data;
};

```

Figure 2: Original tree node structure layout.

Figure 2 shows an example of a node data structure. Figure 3 shows the code representation of the new hyperobject data structure that our optimization creates. N is the number of tree nodes from the original data structure that are contained in the hyperobject. It is formally described with the equation below. For simplicity, a hyperobject allocates enough space for full levels of a subtree.

$$N = \max \left\{ \sum_{k=0}^H c^k \right\}, \text{ s.t. } N \leq \frac{\text{size}(\text{cacheline}) - \text{size}(\text{bitmask})}{\text{size}(\text{key})}$$

The equation above assigns to N the maximum number of nodes whose keys can fit in a single cache line along with the valid bitmask, and comprise entire levels of a subtree. For instance, in the tree of Figure 1, the number of nodes that would fit into a hyperobject would either be 4, 13, 40, and so on.

H is the height of the subtree contained within a hyperobject. The number of pointers exiting the hyperobject is c^H where c is the tree's branching factor. These represent the pointers seen exiting the dotted area in Figure 1. The valid field is a bitmask of N bits which tells us which key entries in the hyperobject are valid.

```

struct HyperObj_node {
    <type1> Key[N];
    bit valid[N];
    struct HyperObj_node *next[c^H];
    <type2> Data[N];
};

```

Figure 3: Hyperobject node after Object Fusion optimization.

The hyperobject node code representation of Figure 3 results in the data layout shown in Figure 4. K_i the key of node i contained in the hyperobject. V_i is the valid bit for node i . It is 1 if K_i is valid and 0 if K_i is invalid. P_j^i represents child j of node i . These are the pointers exiting the hyperobject, so only the pointers of the "leaves" of the hyperobject are of any significance. D_i is the data associated with node i .

K_0	K_1	K_N	V_0	V_1	...	V_N
$P_J^1 \dots P_J^C$				$P_{J+1}^1 \dots$			
$\dots P_{J+1}^C$	$P_{N-1}^1 \dots P_{N-1}^C$					
$P_N^1 \dots P_N^C$				D_1			
$D_1(\text{cont})$						D_2	
$D_2(\text{cont})$						
D_N							

Figure 4: Layout of hyperobject in the cache. Each row represents a cache line.

Since in the hyperobject the tree node keys are now all contained within a single cache line, they can all be retrieved with a single memory access, thus reducing the amount of time needed to traverse through the tree structure.

2.2 Pointer Tuple Representation

The original pointer to a tree node can now be represented as a tuple $\langle p, i \rangle$, where p is a pointer to a hyperobject and i is the node id within the hyperobject. We will refer to this tuple as "pointer tuple" in the remaining of this paper. We utilize this observation to allow the user implement arbitrary algorithms for accessing the tree. The pointer tuple is implemented via a compiler generated structure, the struct `PointerTuple`, shown in Figure 5.

```

struct PointerTuple {
    struct HyperObj_node *p;
    int i; /* NodeId */
};

```

Figure 5: Representation of original struct `treenode*` type in the transformed code.

2.3 Compiler Generated Procedures

In order to perform the transformations, the compiler relies on `next_node`, an automatically generated procedure. Given a pointer tuple, This procedure computes the index to the next node to retrieve from the hyperobject node, and returns a `struct PointerTuple` that represents the pointer to that node. More formally, let c be the branching factor of the tree structure. In the transformed code, if we are at node $\langle ptr.p, ptr.i \rangle$ within the hyperobject and we want to access its j -th child, then:

$$next_node_id = ptr.i + \underbrace{\left(\sum_{k=0}^{L_i} c^k - ptr.i - 1 \right)}_{E_1} + \underbrace{\left(ptr.i - \sum_{k=0}^{L_i-1} c^k \right) * c}_{E_2} + \underbrace{(j-1)}_{E_3} + 1$$

where L_i is the subtree level of node i . The expression E_1 represents the number of nodes that need to be skipped at level L_i . The expression E_2 represents the number of tree nodes that need to be skipped at the next level of the subtree (L_i+1). Finally, the expression E_3 determines the number of siblings of the next node that need to be bypassed.

```

struct PointerTuple
next_node(struct PointerTuple ptr, int j)
{
    struct PointerTuple new_ptr;
    int new_id = next_node_id;
    if (new_id < N) {
        new_ptr.p = ptr.p;
        new_ptr.i = new_id;
    }
    else {
        new_ptr.p = ptr.p->next[i-N]
        new_ptr.i = 0;
    }
    return new_ptr;
};

```

Figure 6: The compiler generated `next_node` procedure.

2.4 Type and Field Access Transformations

In a separate pass, the compiler identifies all variable symbols, parameter arguments, and expressions that are of type `struct treenode*`, and replaces their type with `struct PointerTuple`.

We are only required to modify source code fragments that access a field of the original structure. So, an access to the `key` field in the original code is of the form:

```
ptr->key
```

which after the structure transformations and the type changes is transformed into:

```
ptr.p->key[ptr.i]
```

Despite its simplicity, the pointer representation transformation is very powerful. It allows the user to implement arbitrary algorithms that access the tree, utilize pointers to arbitrary nodes in the middle of the structure, and to dereference them to read or modify field values. The only exception to the above is the `next` pointer. It can be read, but it can not be directly modified by an assignment of the form:

```
ptr->next = new_ptr;
```

Statements like the above are only allowed inside the procedures that perform insertion and deletion. The transformations of those procedures guarantee correctness for assignments to the `next` field. Reading the `next` field is trivially supported. Read accesses of the form:

```
p = ptr->next[j];
```

are transformed into function calls with destination a temporary variable, which replaces the original access in the original expression:

```
tmp = next_node(ptr, j);
p = tmp;
```

Standard copy propagation can clean up trivial cases like the one above. Note that all these transformations should be generic enough to handle any arbitrary combination of the above. For example, a statement of the form:

```
a_var = (ptr->next[j])->key;
```

will be transformed into the two statements below, illustrating the need for the temporary variable:

```
tmp = next_node(ptr, j);
a_var = tmp.p->next[tmp.i][j]->key;
```

After the structure type modifications and pointer type transformations are complete, the compiler identifies static initialization assignments and transforms them appropriately. For example, a static initialization block of the form:

```
struct treenode *ptr = NULL;
```

will be transformed into a static initialization assignment of a multi value block:

```
struct PointerTuple ptr = { NULL, 0 };
```

Similarly, binary expressions where a pointer is checked against `NULL` are transformed into checks against the hyperobject pointer of the pointer tuple.

2.5 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.

Deletion within the hyperobject is trivial. This is done simply by setting the corresponding valid bit in the valid bit mask of the hyperobject to zero. When all of the bits mask is equivalent to zero, the entire hyperobject node can be deleted.

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. Programmer Interface

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. This can be done in numerous ways, and we decided that it was not important which method was used since it is an issue orthogonal to our optimization. The interface between the programmer and the compiler can be modified in the future if necessary.

As was the case, we decided to choose the most straightforward method of communication between the programmer and SUIF2. We require that the user define their tree nodes with specific names in the fields. The structure must have a key field, a next field, and a data field. This simplifies the work that the compiler has to do in order to determine which field is the key, which field contains the next pointers, and which field contains the node data.

In addition to this, the user is required to define an allocate function with two arguments: the key and the data of the node that

they want to create and insert into the hyperobject structure. We currently only implement a version of append for insertion. Finally, the programmer should have a global variable with a well known name to denote the branching factor of the tree, and the cache line size.

4. Assumptions and Limitations

In order to make the problem more tractable, we rely on few assumptions. We assume that the data structures to be fused are tree-like structures. Pointers to arbitrary nodes in the middle of the structure can exist and be utilized as iterators to modify nodes. However, we do not guarantee their safety across tree insertion or deletion points: iterators are invalidated after node insertion or deletion. Any field of the original structure can be accessed and modified through pointer dereferences. The only exception is the next node pointer field which can be read anywhere in the code, but assigned only inside the procedures implementing tree modifications.

We assume all insertions and deletions are performed through corresponding procedures in the source code. The parameters of the insert procedure should provide complete information regarding the new node (i.e. key and data), the insertion point (i.e. the parent node and child pointer to attach the new node, and the child pointer to attach the previous node).

Since our optimization violates the locational transparency of pointer-based data structures, it is the user's responsibility to guarantee the safety of the optimization. However, we believe the restrictions imposed are minimal and do not impair in any significant way the power of the programming language, and have very small programming overhead.

5. Implementation Details in SUIF2

Since object fusion is a high-level optimization, we chose to implement it in SUIF2, which gives us the ability to manipulate high-level constructs like loops and structure fields.

We use collection of selective walkers to traverse the compiler's high level IR and perform the transformations. First we calculate the number of objects that will be fused into a hyperobject. The first walker traverses the variable symbols and creates the hyperobject structure, the pointer tuple structure, and the appropriate pointers and qualified types. It is followed by a walk that modifies the types of variable symbols, globals appropriately, and procedure arguments. The next walk is performed in expressions, and the result types are modified in a similar manner. The following walker corrects binary expressions on the original pointers. This walk is followed by a walk on call statements to modify the parameters of calls to the insert procedure. The body of the insert procedure is completely rewritten by the compiler, and, finally, the `next_node` function is created.

For the type calculations, we consult the type builder to qualify the types, and use methods in the `SuifObjectFactory`, and `BasicObjectFactory` classes. We utilize the `CInformationBlock` and `TargetInformationBlock` to retrieve information on standard C types, and alignment requirements of structure, pointer, array, and procedure types. Every type generated is inserted into the appropriate symbol table

and lookup table. In the creation of the field structures by the compiler, additional care is taken to account for correct alignments of types and holes in the memory utilization of the structure.

In order to detect bugs early in the process, at the end of the object fusion pass all types are checked, the structure of the IR representation is validated, and all invariants are enforced (in separate passes).

6. Problems and Lessons Learned

The biggest problem that we encountered during the progress of the project was the fact that we grossly underestimated the possible extent of this project. The amount of work and time that it would take to complete the initial goals that we set for ourselves in the proposal would be tremendous. Although both of us were somewhat familiar with MachSUIF from the class assignments, MachSUIF was an inappropriate choice for this project. Since we were dealing with data structures, we needed a tool that would allow for analysis at a higher level. Unfortunately, SUIF2 was a completely differently structured tool than MachSUIF. In addition, since the documentation for SUIF2 was not as extensive as the documentation that existed for MachSUIF, the learning curve for SUIF2 was much steeper than we had initially anticipated. Thus, much time was spent figuring out the structure of SUIF2 and what classes were useful for creating compiler passes (such as the Walkers). Looking back, this project was very ambitious for the six weeks that we had to complete it.

Near the beginning of the project's development process, we had thought that it would be possible to transform a general traversal function into one that traverses the hyperobject data structure in the same fashion. The problem that we had not thought about was that the traversal function could be implemented in an infinite number of ways. It would almost be impossible to try to figure out how to transform this behavior into the hyperobject structure. Not only this, but even the node structure that is being traversed can have many different formats. Some nodes have multiple keys (for example, if the node is a member of multiple tree structures) and what would be the best way to identify which field is used as the key? The only solution that we were able to come up with for dealing with this situation in the six weeks that we had to complete the project was to restrict the programmer immensely in the structure of the nodes and the format of the function definitions. Restricting the user also narrowed the scope of the project, making it more feasible in the allotted amount of time.

Unfortunately, mid-way through the project and after the milestone, we decided to change the design of our project. We added the pointer tuple data structure, which contains two fields: a pointer to the hyperobject containing the node, and an index indicating where the node's fields are located. This design was much more reasonable because it allowed the user to have access to the middle of the hyperobject structures through the use of these pointer structures, giving the user much more freedom to implement arbitrary traversal algorithms to retrieve field values. The only restriction that we have with this new design is that once the user inserts a node into the data structure, all iterators are invalidated. It could very well be that if an insertion caused the nodes to shift positions, the hyperobject-index pairs in the pointer tuple structures would potentially be invalid.

However, even with all these restrictions, the main problem that called for redesigning the project was the difficulty and inflexibility of the translating the behavior of originally defined traversal function into one that mirrored the behavior with the hyperobject data structure. We found that the work involved in doing this transformation was much more extensive than we had originally anticipated. For this reason, we found that if we redesigned the project, not only would this release some of the restrictions on the user, but it would also make the transformations of the traversal function much simpler.

Granted, the work and time involved in completing this project was worsened by the fact that we had to redesign two weeks before the deadline. However, we both agree that the new design is much better than the one previously proposed. There was not much that could be done to avoid a situation such as this. We had previously met to sketch down in great detail the implementation details involved and the restrictions that we needed to place on the user in order to make this project feasible. We just did not think of the improved design until very recently.

7. Current State of the Project

The project is completely implemented and is in the final testing phase of the development process. All passes have been implemented, and the final SUIF2 tree is checked successfully for validity of the IR representation and type checked. The SUIF2 output compiles successfully with gcc. Unfortunately a bug in an address calculation currently hampers our ability to evaluate the optimization.

As promised, we do support the insertion of new nodes into the hyperobject structure, although in a more limited fashion. The user is required to define an `allocate()` function, which will be modified by the pass to support the appending of nodes to the data structure. The data structure also can support insertions into the middle of the structure, but due to time constraints this was not implemented.

Since deletions are trivial in this data structure, we decided to concentrate our efforts on the other functions of the hyperobject structure. In order to delete a node from the hyperobject, the compiler merely has to traverse the hyperobject data structure. When it finds the matching key, it just has to set the corresponding valid bit to zero.

Unfortunately, due to time constraints, we were unable to run the optimization on any benchmarks to evaluate the effect of object fusion on the performance of the application.

8. REFERENCES

- [1] J. Rao, K. A. Ross. "Cache Conscious Indexing for Decision-Support in Main Memory." In *Proceedings of the 25th 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. Truong, 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 8th 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 7th 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 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.