Appears in *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999.

Cache-Conscious Structure Definition

Trishul M. Chilimbi
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
chilimbi@cs.wisc.edu

Bob Davidson
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
bobd@microsoft.com

James R. Larus
Microsoft Research
One Microsoft Way
Redmond, WA 98052
larus@microsoft.com

ABSTRACT

A program's cache performance can be improved by changing the organization and layout of its data—even complex, pointer-based data structures. Previous techniques improved the cache performance of these structures by arranging distinct instances to increase reference locality. These techniques produced significant performance improvements, but worked best for small structures that could be packed into a cache block.

This paper extends that work by concentrating on the internal organization of fields in a data structure. It describes two techniques—*structure splitting* and *field reordering*—that improve the cache behavior of structures larger than a cache block. For structures comparable in size to a cache block, structure splitting can increase the number of hot fields that can be placed in a cache block. In five Java programs, structure splitting reduced cache miss rates 10–27% and improved performance 6–18% beyond the benefits of previously described cache-conscious reorganization techniques.

For large structures, which span many cache blocks, reordering fields, to place those with high temporal affinity in the same cache block can also improve cache utilization. This paper describes `bbcache`, a tool that recommends C structure field reorderings. Preliminary measurements indicate that reordering fields in 5 active structures improves the performance of Microsoft SQL Server 7.0 2–3%.

Keywords

cache-conscious definition, structure splitting, class splitting, field reorganization

1. INTRODUCTION

An effective way to mitigate the continually increasing processor-memory performance gap is to allocate data structures in a manner that increases a program's reference locality and improves its cache performance [1, 5, 6]. Cache-conscious data layout, which clusters temporally related objects into the same cache block or into non-conflicting blocks, has been shown to produce significant performance gains.

This paper continues the study of data placement optimizations along the orthogonal direction of reordering the internal layout of a structure or class's fields. The paper describes two cache-conscious definition techniques—*structure splitting* and *field reordering*—that can improve the cache behavior of programs. In other words, previous techniques focused on the external arrangement of structure instances, while this paper focuses on their internal organization. In particular, previous techniques (with the exception of techniques for reducing cache conflicts) worked best for structures smaller than half of a cache block. The techniques in this paper apply to larger structures as well.

Figure 1 indicates different opportunities for improving cache performance. Caches have finite capacity (<< main memory capacity) and transfer data in units called cache blocks that encompass multiple words. Caches also have finite associativity, and this restricts where a block can be placed in the cache. Placing contemporaneously accessed structure elements in the same cache block improves cache block utilization and provides an implicit prefetch. Moreover, it makes more efficient use of cache space by reducing a structure's cache block footprint. Finally, mapping concurrently accessed structure elements (which will not all fit in a single cache block) to non-conflicting cache blocks reduces cache misses. The techniques in this paper directly improve a structure's cache block utilization and reduce its cache block working set. In addition, since decreasing a structure's cache footprint also reduces the number of blocks that potentially conflict, the techniques may indirectly reduce conflict misses.

Figure 2 illustrates the relationship of cache-conscious definition technique to the size of structure instances. Instances significantly smaller than a cache block (case 1), are unlikely to benefit from additional manipulation at definition time. Previous techniques—such as Chilimbi and Larus' cache-conscious object co-location, which uses a copying garbage collector to place objects referenced together near each other in memory [5]—are effective.

If the structure instance size is comparable to the size of a cache block (case 2), splitting structure elements into a hot and cold portion can produce hot structure pieces smaller than a cache block, which permits application of cache-conscious reorganization techniques to these portions. As this paper will show, many Java objects belong to this category. In addition, since Java is a type-safe language, class splitting can be automated. The first step in this process is to profile a Java program to determine member access frequency. These counts identify class member fields as hot (frequently accessed) or cold (rarely accessed). A compiler extracts cold fields from the class and places them in a new object, which is referenced indirectly from the original object. Accesses to cold fields require an extra indirection to the new class, while hot field accesses remain unchanged. At run time, Chilimbi and Larus' cache-conscious garbage collector co-locates the modified object instances. For five medium-sized Java benchmarks, class splitting combined with Chilimbi and Larus' cache-conscious object co-location reduced L2 cache miss rates by 29–43%, with class splitting accounting for 26–62% of this reduction, and improved performance by 18–28%, with class splitting contributing 22–66% of this improvement.

Finally, when structure elements span multiple cache blocks (case

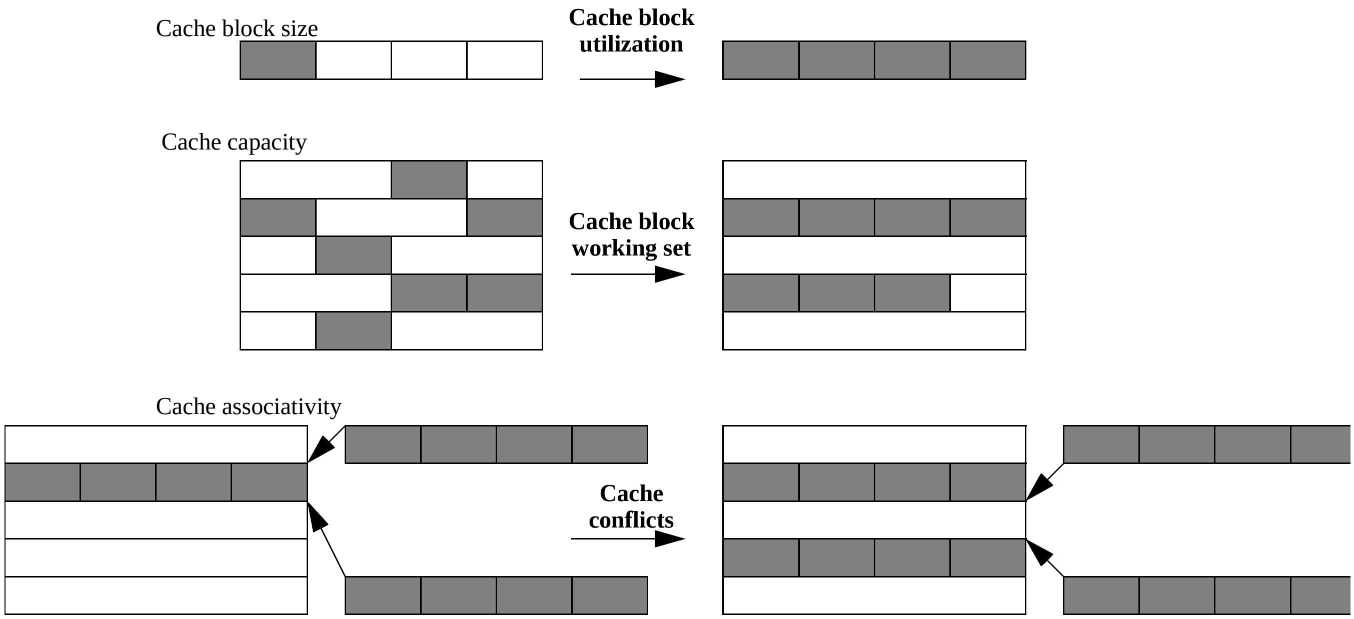


Figure 1. Improving cache performance.

3), reordering structure fields to place those with high temporal affinity in the same cache block can also improve cache block utilization. Typically, fields in large structures are grouped conceptually, which may not correspond to their temporal access pattern. Unfortunately, the best order for a programmer may cause structure references to interact poorly with a program's data access pattern and result in unnecessary cache misses. Compilers for many languages are constrained to follow the programmer-supplied field order and so cannot correct this problem. Given the ever-increasing cache miss penalties, manually reordering structure fields, to place those with high temporal affinity in the same cache block, is a relatively simple and effective way to improve program performance.

Legacy applications were designed when machines lacked multiple levels of cache and memory access times were more uniform. In particular, commercial C applications often manipulate large structures. To explore the benefits of field reordering, this paper describes `bbcache`, a tool that recommends C structure field reorderings. `bbcache` correlates static information about the source location of structure field accesses with dynamic information about the temporal ordering of accesses and their execution frequency. This data is used to construct a field affinity graph for each structure. These graphs are then processed to produce field order recommendations. Preliminary measurements indicate that reordering fields in 5 active structures improves the performance of Microsoft SQL Server 7.0, a large, highly tuned commercial application, by 2–3% on the TPC C benchmark [13].

The rest of the paper is organized as follows. Section 2 describes structure splitting. Section 3 discusses field reordering for C and describes `bbcache`. Section 4 presents our experimental results. Finally, Section 5 briefly discusses related work.

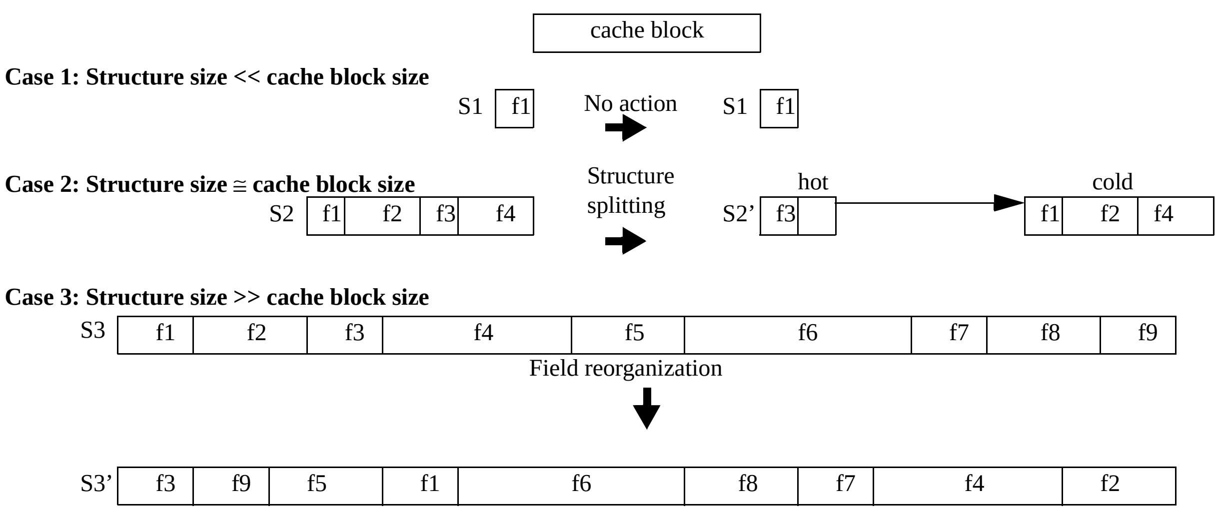


Figure 2. Cache-conscious structure definition.

Table 1: Java benchmark programs.

Program	Lines of Code[a]	Description
cassowary	3,400	Constraint solver
espresso	13,800	Martin Odersky's drop-in replacement for javac
javac	25,400	Sun's Java source to bytecode compiler
javadoc	28,471	Sun's documentation generator for Java source
pizza	27,500	Pizza to Java bytecode compiler

a. Plus, a 13,700 line standard library (JDK 1.0.2).

2. STRUCTURE SPLITTING

Chilimbi and Larus [5] proposed using a generational garbage collector to lay out objects dynamically, so those with temporal affinity are placed near each other and are likely to reside in the same cache block. They demonstrated that the vast majority of live objects in Cecil [2, 3] (an object-oriented language bearing similarities to Java) were smaller than half a cache block (< 32 bytes). This characteristic permitted low overhead, real-time data profiling of Cecil programs. They also described a new copying algorithm that utilized this profile information to produce a cache-conscious object layout.

Our experiments (Section 2.1) found that Java objects are also small, but they are on average approximately 8 bytes larger that Cecil objects. Directly applying Chilimbi and Larus' cache-conscious co-location scheme to Java programs yields smaller performance improvements (10–20%, Section 4.1.2) than those reported for Cecil (14–37% [5]). This difference is attributable to the larger Java objects reducing the number of contemporaneously accessed object instances that could be packed into a cache block.

One way to reduce the effective size of Java objects is to split Java classes into a hot (frequently accessed) and a cold (rarely accessed) portion, based on profiled field access frequencies. Splitting classes allows more hot object instances to be packed into a cache block and kept in the cache at the same time. Structure splitting is a well known optimization, that is often applied manually to improve performance. However, to the best of our knowledge, this is the first completely automatic implementation of the technique.

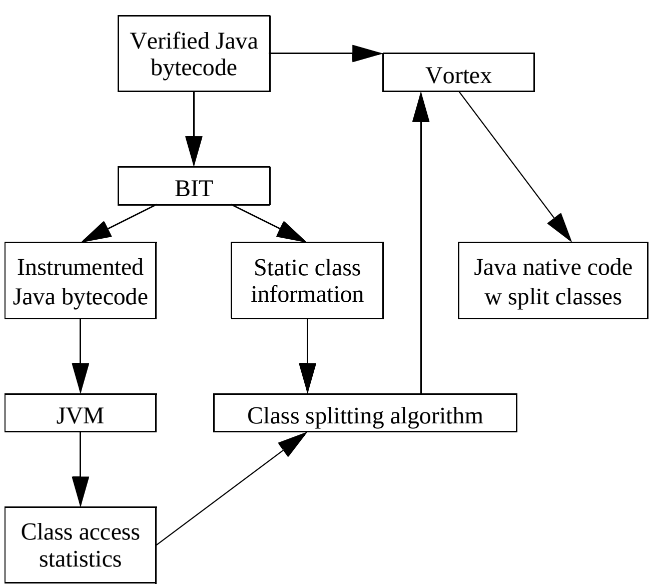


Figure 3. Class splitting overview.

Figure 3 illustrates the class splitting process. First, a Java program, in the form of verified bytecodes, is statically analyzed and instrumented using BIT [10] (the standard library was not instrumented). The static analyses produces a variety of class information, including class and field names, field types, and field sizes. Next, the instrumented Java program is executed and profiled. The profile measures class instantiation counts and instance variable (non-static class fields) access statistics on a per class basis. An algorithm uses the static and dynamic data to determine which classes should be split. Finally, these splitting decisions are communicated to the Vortex compiler [4], which compiles Java bytecode to native code. The compiler splits the specified classes and transforms the program to account for the change. The class splitting algorithm and program transformations are described in more detail in subsequent sections.

Applying Chilimbi and Larus' cache-conscious object co-location scheme to the Java programs with split classes results in performance improvements of 18–28%, with 22–66% of this improvement attributable to class splitting (see Section 4.1.2).

2.1 Average Java Object Size

We ran some experiments to investigate whether Java follows the same size distribution as Cecil objects. Our system uses the Vortex compiler developed at the University of Washington [4]. Vortex is a language-independent optimizing compiler for object-oriented languages, with front ends for Cecil, C++, Java, and Modula-3. Table 1 describes the Java benchmark programs used in the experiments. The programs were compiled at the highest optimization level (o2), which applies techniques such as class analysis, splitting, class hierarchy analysis, class prediction, closure delaying, and inlining, in addition to traditional optimizations [4]. The experiments were run on a single processor of a Sun Ultraserver E5000, which contained 12 167Mhz UltraSPARC processors and 2GB of memory running Solaris 2.5.1.

Table 2 shows the results of our first set of experiments, which tested the hypothesis that most heap allocated Java objects are on average smaller than a cache block (for reasons having to do with garbage collection, objects greater than or equal to 256 bytes are considered large and managed differently).

However, small objects often die fast. Since the cache-conscious layout technique described in the Chilimbi-Larus paper is only effective for longer-lived objects, which survive scavenges, we are more interested in live object statistics. Table 3 shows the results of the next experiment, which measured the number of small objects live after each scavenge, averaged over the entire program execution. Once again, the results support the hypothesis that most Java objects are small, and the average live object size is smaller than a cache block (64 bytes). However comparing the average live small object size for Java programs (23–32 bytes) with that for Cecil programs (15–24 bytes [5]), it appears that Java objects are approximately 8 bytes larger (possibly due to larger object headers). This larger size reduces the effectiveness of packing objects in the same cache block.

2.2 Class Information

BIT is used to gather static class information, including class name, number of non-static fields, and the names, access types, and descriptors for all non-static fields. Non-static fields are tracked since these constitute the instance variables of a class and are allocated on the heap. In addition, BIT instruments the program to generate field access frequencies on a per-class basis. An instrumented program runs an order of magnitude slower than its original.

2.3 Hot/Cold Class Splitting Algorithm

Class splitting involves several trade-offs. Its primary advantage is the ability to pack more (hot) class instances in a cache block. Its disadvantages include the cost of an additional reference from the hot to cold portion, code bloat, more objects in memory, and an extra indirection for cold field accesses. This section describes a class splitting algorithm that considers these issues while selecting classes to split.

The problem of splitting classes into a hot and cold portion based on field access counts has a precise solution only if the program is rerun on the same input data set. However, we are interested in splitting classes so the resulting program performs well for a wide range of inputs. An optimal solution to this problem is unnecessary since field access frequencies for different program inputs are unpredictable. Instead, the class splitting algorithm uses several heuristics. While none of these heuristics may be optimal, measurements in Section 4.1 demonstrate that they work well in practice. In addition, they worked better than several alternatives that were examined. In the ensuing discussion, the term “field” refers to class instance variables (i.e., non-static class variables).

Figure 4 contains the splitting algorithm. The splitting algorithm only considers classes whose total field accesses exceed a specified threshold. This check avoids splitting classes in the absence of sufficient representative access data. While alternative criteria undoubtedly exist, the following formula worked well for determining this threshold. Let *LS* represent the total number of program field accesses, *C* the total number of classes with at least a single field access, F_i the number of fields in class *i*, and A_i the total number of accesses to fields in class *i*, then the splitting algorithm only considers classes where:

$$A_i > LS / (100*C)$$

These classes are called the ‘live’ classes. In addition, the splitting algorithm only considers classes that are larger than eight bytes and contain more than two fields. Splitting smaller classes is unlikely to produce any benefits, given the space penalty incurred by the reference from the hot to the cold portion.

Next, the algorithm labels fields in the selected ‘live’ classes as hot or cold. An aggressive approach that produces a smaller hot partition, and permits more cache-block co-location, also increases the cost of accessing cold fields. These competing effects must be balanced. Initially, the splitting algorithm takes an aggressive approach and marks any field not accessed more than $A_i / (2 * F_i)$ times as cold. If the cold portion of class *i* is sufficiently large to merit splitting (at least 8 bytes to offset the space required for the cold object reference), the following condition is used to counterbalance overaggressive splitting:

$$(max(hot(class_i)) - 2 * \Sigma cold(class_i)) / max(hot(class_i)) > 0.5$$

where the *hot* and *cold* functions return the access counts of a class’ hot and cold fields, respectively. This condition can be informally justified as follows. Consider instances of two different classes, o_1 and o_2, that are both comparable in size to a cache block and that have a high temporal affinity. Let instance o_1 have *n* fields that are accessed $a_1, .., a_n$ times, and o_2 have *m* fields that are accessed $b_1, ..., b_m$ times. It is reasonable to expect the following access costs (# of cache misses) for the class instances o_1 and o_2:

Table 2: Most heap allocated Java objects are small.

Program	# of heap allocated small objects (< 256 bytes)	Bytes allocated (small objects)	Avg. small object size (bytes)	# of heap allocated large objects (>= 256 bytes)	Bytes allocated (large objects)	% small objects
cassowary	958,355	19,016,272	19.8	6,094	2,720,904	99.4
espresso	287,209	8,461,896	29.5	1,583	1,761,104	99.5
javac	489,309	15,284,504	31.2	2,617	1,648,256	99.5
javadoc	359,746	12,598,624	35.0	1,605	1,158,160	99.6
pizza	269,329	7,739,384	28.7	1,605	1,696,936	99.4

Table 3: Most live Java objects are small.

Program	Avg. # of live small objects	Bytes occupied (live small objects)	Avg. live small object size (bytes)	Avg. # of live large objects	Bytes occupied (live large objects)	% live small objects
cassowary	25,648	586,304	22.9	1699	816,592	93.8
espresso	72,316	2,263,763	31.3	563	722,037	99.2
javac	64,898	2,013,496	31.0	194	150,206	99.7
javadoc	62,170	1,894,308	30.5	219	148,648	99.6
pizza	51,121	1,657,847	32.4	287	569,344	99.4

$$max(a_1, ..., a_n) < cost(o_1) < \Sigma(a_1, ..., a_n)$$

$$max(b_1, ..., b_m) < cost(o_2) < \Sigma(b_1, ... b_m)$$

Now if the hot portion of o_1 is co-located with the hot portion of o_2 and these fit in a cache block, then:

$$cost(o_1) + cost(o_2) \cong (max(hot(class_1), hot(class_2)) + \varepsilon) + 2 * (\Sigma cold(class_1) + \Sigma cold(class_2))$$

since cold fields are accessed through a level of indirection. This will definitely be beneficial if the sum of the (best case) costs of accessing original versions of the instances is greater than the access cost after the instances have been split and hot portions co-located:

$$max(a_1, ..., a_n) + max(b_1, ..., b_m) > ((max(hot(class_1), hot(class_2)) + \varepsilon) + 2*(\Sigma cold(class_1) + \Sigma cold(class_2))$$

i.e.:

$$min(max(hot(class_1)), max(hot(class_2))) > 2 * (\Sigma cold(class_1) + \Sigma cold(class_2)) + \varepsilon$$

Since apriori we do not know which class instances will be co-located, the best we can do is to ensure that:

$$TD(class_i) = max(hot(class_i)) - 2 * \Sigma cold(class_i) >> 0$$

This quantity is termed the ‘temperature differential’ for the class. For classes that do not meet this criteria, a more conservative formula is used that labels fields that are accessed less than $A_i / (5*F_i)$ as cold. If this does not produce a sufficiently large cold portion (> 8 bytes), the class is not split.

2.4 Program Transformation

We modified the Vortex compiler to split classes selected by the splitting algorithm and to perform the associated program transformations. Hot fields and their accesses remain unchanged. Cold fields are collected and placed in a new cold counterpart of the split class, which inherits from the primordial Object class and has no methods beyond a constructor. An additional field, which is a reference to the new cold class, is added to the original class, which now contains the hot fields. Cold fields are labelled with the `public` access modifier. This is needed to permit access to `private` and `protected` cold fields through the cold class reference field in the original (hot) class.

```
split_classes()
{
    for each class {
        mark_no_split;
        if((active)&&(suitable_size)){
            mark_flds_aggresive;
            if(suff_cold_fields)
                if(nrmlized_temp_diff > 0.5)
                    mark_split;
                else{
                    re-mark_flds_conservative;
                    if(suff_cold_fields)
                        mark_split;
                }
        }
    }
}
```

Figure 4. Class splitting algorithm.

Finally, the compiler modifies the code to account for split classes. These transformations include replacing accesses to cold fields with an extra level of indirection through the cold class reference field in the hot class. In addition, hot class constructors must first create a new cold class instance and assign it to the cold class reference field. Figure 5 illustrates these transformations for a simple example.

2.5 Discussion

Some programs transfer structures back and forth to persistent storage or external devices. These structures cannot be transparently changed without losing backward compatibility. However, when new optimizations offer significant performance advantages, the cost of such compatibility may become high, and explicit input and output conversion necessary. Translation, of course, is routine in languages, such as Java, in which structure layout is left to the

```
class A {
    protected long a1;
    public int a2;
    static int a3;
    public float a4;
    private int a5;
    A(){
        ...
        a4 = ..;
    }
    ...
}
class B extends A {
    public long b1;
    private short b2;
    public long b3;
    B(){
        b3 = a1 + 7;
        ...
    }
    ...
}
```

```
class A {
    public int a2;
    static int a3;
    public cld_A cld_A_ref;
    A(){
        cld_A_ref = new cld_A();
        ...
        cld_A_ref.a4 = ..;
    }
    ...
}
class B extends A {
    public long b3;
    public cld_B cld_B_ref;
    B(){
        cld_B_ref = new cld_B();
        b3 = cld_A_ref.a1 + 7;
        ...
    }
    ...
}
```

```
class cld_A {
    public long a1;
    public float a4;
    public int a5;
    cld_A(){...}
}

class cld_B {
    public long b1;
    public short b2;
    cld_B(){...}
}
```

Figure 5. Program transformation.

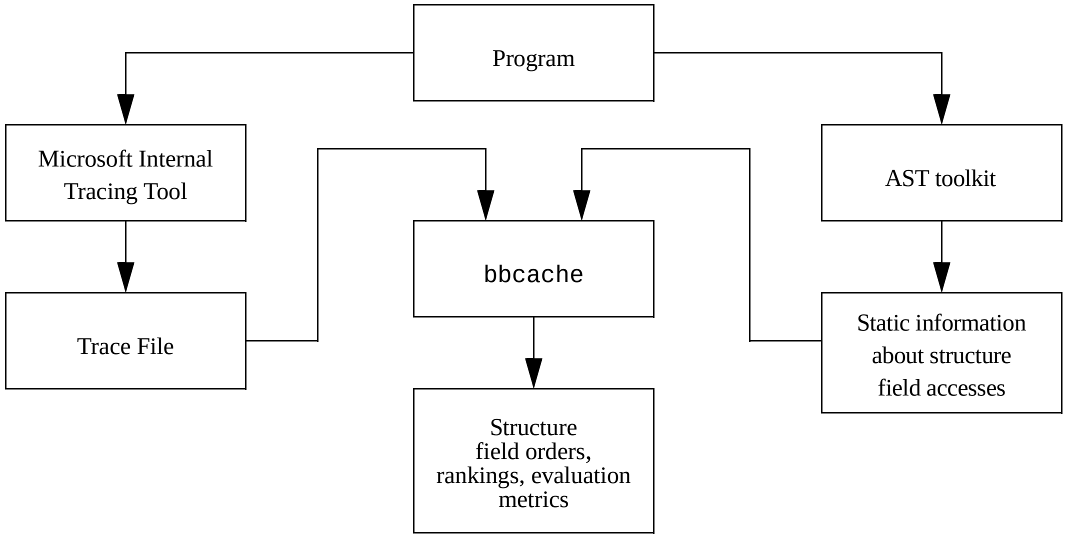


Figure 6. `bbcache` overview.

compiler.

The splitting technique in this paper produces a single split version of each selected class. A more aggressive approach would create multiple variants of a class, and have each direct subclass inherit from the version that is split according to the access statistics of the inherited fields in that subclass. To simplify our initial implementation, we choose not to explore this option, especially since its benefits are unclear. However, future work will investigate more aggressive class splitting.

Since this paper focuses on improving data cache performance, class splitting only considers member fields and not methods. Method splitting could improve instruction cache performance. In addition, it offers additional opportunities for overlapping execution of mobile code with transfer [9].

3. FIELD REORDERING

Commercial applications often manipulate large structures with many fields. Typically, fields in these structures are grouped logically, which may not correspond to their temporal access pattern. The resulting structure layout may interact poorly with a program's data access pattern and cause unnecessary cache misses. This section describes a tool—`bbcache`—that produces structure field reordering recommendations. `bbcache`'s recommendations attempt to increase cache block utilization, and reduce cache pressure, by grouping fields with high temporal affinity in a cache block.

For languages, such as C, that permit almost unrestricted use of pointers, reordering structure fields can affect program correctness—though this is often a consequence of poor programming practice. Moreover, C structures can be constrained by external factors, such as file or protocol formats. For these reasons, `bbcache`'s recommendations must be examined by a programmer before they can be applied to C programs.

3.1 bbcache

Figure 6 illustrates the process of using `bbcache`. A program is first profiled to create a record of its memory accesses. The trace file contains temporal information and execution frequency for structure field accesses. `bbcache` combines the dynamic data with static analysis of the program source to produce the structure field order recommendations.

The algorithm used to recommend structure field orders can be divided into three steps. First, construct a database containing both static (source file, line, etc.) and dynamic (access count, etc.) information about structure field accesses. Next, process this database to construct field affinity graphs for each structure. Finally, produce the structure field order recommendations from these affinity graphs.

`bbcache` also contains an evaluation facility that produces a cost metric, which represents a structure's cache block working set, and a locality metric, which represents a structure's cache block utilization. These metrics help compare the recommended field order against the original layout. They, together with a ranking of active structures based on their temporal activity and access frequency, can be used to identify structures most likely to benefit from field reordering.

3.1.1 Constructing the Structure Access Database

The ASTtoolkit [7], a tool for querying and manipulating a program's abstract syntax tree, is used to analyze the source program. It produces a file containing information on each structure field access, including the source file and line at which the access occurs, whether the access is a 'read', 'write', or 'read-write', the field name, the structure instance, and the structure (type) name. A structure instance is a <function name, structure (type) name> pair, where the function name corresponds to the function in which the instance is allocated. With pointer aliasing, computing structure instances statically in this manner is an approximation. The following example helps illustrate the problem. Consider consecutive accesses to fields a and b in two different structure instances (though indistinguishable with our approximation). This could lead to incorrectly placing fields a and b next to each other. However, this did not appear to be a serious problem for our purposes, since most instances showed similar access characteristics (i.e., consecutive accesses to the same field in different (indistinguishable) instances, rather than different fields). `bbcache` reads this file and builds a structure access database, which it represents as a hash table on structure names (Figure 7). Each hash table entry repre-

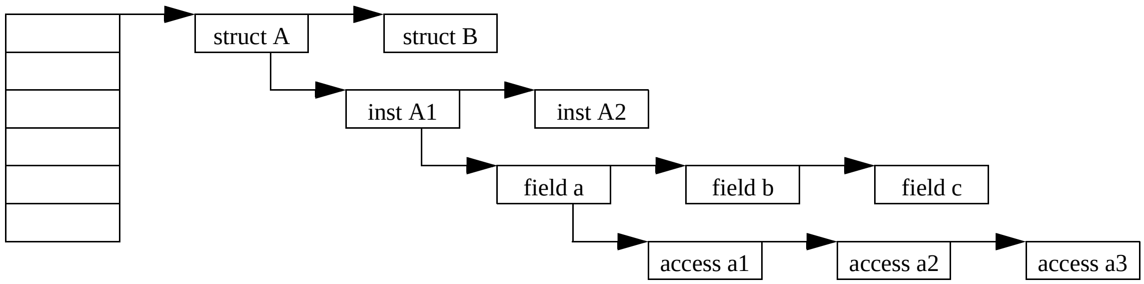


Figure 7. Structure access database.

sents a structure type and points to a list of structure instances. Every structure instance points to a list of fields that were accessed through that instance, and each field in turn points to a list of access sites which record the source location from which the access took place. `bbcache` uses program debug information to associate temporal information and execution frequency, from the program trace, with each field access site.

3.1.2 Processing the Structure Database

The structure database contains information about field accesses for many instances of the same structure type. For each structure instance, `bbcache` constructs a field affinity graph, which is a weighted graph whose nodes represent fields and edges connect fields that are accessed together according to the temporal trace information. Fields accessed within 100 milliseconds of each other in the trace were considered to be accessed contemporaneously. While we experimented with several intervals ranging from 50—1000 ms, most structures did not appear to be very sensitive to the exact interval used to define contemporaneous access, and the results reported in Section 4.2 correspond to a 100ms interval. Edge weights are proportional to the frequency of contemporaneous access. All instance affinity graphs of each structure type are then combined to produce a single affinity graph for each structure (Figure 8).

3.1.3 Producing Structure Field Orders

Since structure alignment with respect to cache block boundaries can only be determined at run time (unless the *malloc* pointer is suitably manipulated), our approach is to be satisfied with increasing inherent locality by placing fields with high temporal affinity near each other—so they are likely to reside in the same cache block—rather than try to pack fields exactly into cache blocks. If alignment (natural boundary) constraints would force a gap in the layout that alternative high temporal affinity fields are unable to occupy, we attempt to fill these with structure fields that were not accessed in the profiling scenario.

We introduce the notion of configuration locality to explain `bbcache`'s algorithm. Configuration locality attempts to capture a layout's inherent locality. The first step is to compute a layout affinity for each field, which is the sum of its weighted affinities with neighboring fields in the layout up to a predefined horizon (presumably equivalent to the cache block size) on either side. If field f_i is surrounded by fields $f_1, ..., f_n$, in the layout, then its layout affinity is:

$$\textit{Field layout affinity}(f_i) = wt(f_1, f_i)*aff(f_1,f_i) + \ldots + wt(f_n, f_i)*aff(f_n, f_i)$$

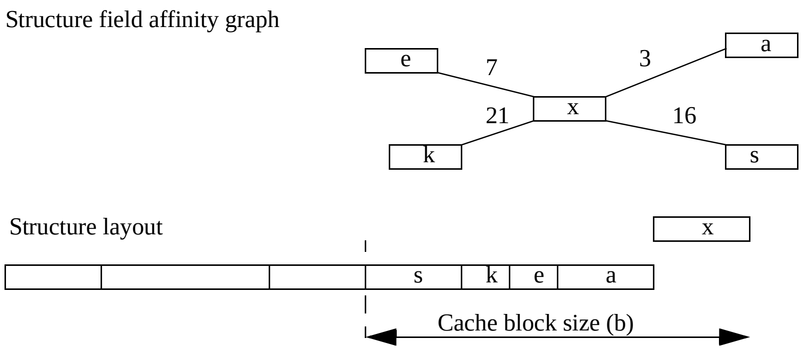
The weights correspond to the distance between the fields—the number of bytes separating the start of the fields—and are a mea-

```
for each structure type
{
   for each instance of this type
   {
      combine field access information for multiple
      occurrences of the same field;

      // Build a field affinity graph for this instance
      for each pair of instance fields
      {
         compute field affinity edge weight;
      }
   }

   //Combine instance field affinity graphs to create a structure
   // field affinity graph
   for each pair of structure fields
   {
      find all structure instances for which this pair of fields
         has an affinity edge and compute a weighted affinity;
   }
}
```

Figure 8. Processing the structure access database.



$$\Delta(configuration - locality) = affinity(x, a) \times \frac{b-4}{b} + affinity(x, e) \times \frac{b-6}{b}$$
$$+ affinity(x, k) \times \frac{b-8}{b} + affinity(x, s) \times \frac{b-12}{b}$$

Figure 9. Producing field orders from the structure field affinity graph.

sure of the probability that the fields will end up in the same cache block. The weighting factor used is:

$$wt(f_i, f_j) = ((cache\_block\_size - dist(f_i, f_j)) / cache\_block\_size)$$

A structure's configuration locality is the sum of its field layout affinities. Figure 9 illustrates the process of computing the increase in configuration locality from adding field *x* to an existing layout.

`bbcache` uses a greedy algorithm to produce structure field order recommendations from a structure field affinity graph. It starts by adding the pair of fields, connected by the maximum affinity edge in the structure field affinity graph, to the layout. Then at each step, a single field is appended to the existing layout. The field selected is the one that increases configuration locality by the largest amount at that point in the computation. This process is repeated until all structure field are laid out.

3.1.4 Evaluating Structure Field Orders

While the best way to evaluate a structure field ordering is to measure its impact on performance, this entails a tedious cycle of editing, recompiling, and rerunning the application. A quality metric for structure field orderings can help compare a recommended layout against the original layout and help evaluate alternative layouts, without rerunning the application. This is especially useful when field layout constraints prevent directly following the field ordering recommendations.

`bbcache` provides two metrics to evaluate structure field orders, as well as a query facility to compare alternative layouts. The first is a metric of the average number of structure cache blocks active during an application's execution (i.e., a measure of a structure's cache block working set or cache pressure). This metric is computed by combining temporal information for field accesses with a structure's field order to determine active cache blocks. A program's execution is divided into temporal intervals of 100ms each. This metric assumes that structures start on cache block boundaries, and uses the field order (and field sizes) to assign fields to cache blocks. If any of the fields in a cache block are accessed during an execution interval, that block is considered to be active in that interval. Let *n* represent the total number of program execution intervals, and $b_1, ..., b_n$ the number of active structure cache blocks in each of these intervals.Then a structure's cache block pressure is:

$$Cache\ block\ pressure = \Sigma(b_1, ..., b_n) / n$$

The second metric is a locality metric that measures a structure's average cache block utilization. Let f_{ij} represent the fraction of cache block *j* accessed (determined by accessed field sizes relative to the cache block size) in program execution interval *i*, then:

$$Cache\ block\ utilization = \Sigma(f_{11}, ..., f_{nbn}) / \Sigma(b_1, ..., b_n)$$

4. EXPERIMENTAL EVALUATION

This section contains experimental evaluation of class splitting and field reordering.

4.1 Class Splitting

This section describes our experimental methodology and presents experiments that measure the effectiveness of the splitting algorithm and its impact on the performance of Java programs.

4.1.1 Experimental Methodology

As described earlier, we used the University of Washington Vortex compiler infrastructure with aggressive optimization (o2). Table 1 describes the benchmarks. The compiled programs ran on a single processor of a Sun Ultraserver E5000, which contained 12 167Mhz UltraSPARC processors and 2GB of memory running Solaris 2.5.1. The large amount of system memory ensures that locality benefits are due to improved cache performance, not reduced paging activity. This processor has two levels of data cache. The level 1 cache is 16 KB direct-mapped with 16 byte cache blocks. The level 2 cache is a unified (instruction and data) 1 MB direct-mapped cache with 64 byte cache blocks. The system has a 64 entry iTLB and a 64 entry dTLB, both of which are fully associative. A level 1 data cache hit requires one processor cycle. A level 1 cache miss, followed by a level 2 cache hit, costs 6 additional cycles. A level 2 cache miss typically results in an additional 64 cycle delay.

4.1.2 Experimental Results

The first set of experiments were designed to investigate the potential for class splitting in the Java benchmarks, study the behavior of

Table 4: Class splitting potential.

Benchmark	# of classes (static)	# of accessed classes	# of 'live' classes	# of candidate classes (live & suitably sized)	# of split classes	Splitting success ratio (#split/ #candidates)
cassowary	27	12	6	2	2	100.0%
espresso (input1)	104	72	57	33	11 (8)	33.3%
espresso (input2)	104	69	54	30	9 (8)	30.0%
javac (input1)	169	92	72	25	13 (11)	52.0%
javac (input2)	169	86	68	23	11 (11)	47.8%
javadoc (input1)	173	67	38	13	9 (7)	69.2%
javadoc (input2)	173	62	30	11	7 (7)	63.6%
pizza (input1)	207	100	72	39	10 (9)	25.6%
pizza (input2)	207	95	69	36	10 (9)	27.8%

Table 5: Split class characteristics

Benchmarks	Split class access / total prog. accesses	Avg. pre-split class size (static)	Avg. pre-split class size (dyn)	Avg. post-split (hot) class size (static)	Avg. post-split (hot) class size (dyn)	Avg. reduction in (hot) class size (static)	Avg. reduction in (hot) class size (dyn)	Avg. normalized temperature differential	Additional space allocated for cold class field ref (bytes)
cassowary	45.8%	48.0	76.0	18.0	24.0	62.5%	68.4%	98.6%	56
espresso (input1)	55.3%	41.4	44.8	28.3	34.7	31.6%	22.5%	79.2%	74,464
espresso (input2)	59.4%	42.1	36.2	25.7	30.1	39.0%	16.9%	79.5%	58,160
javac (input1)	45.4%	45.6	26.3	27.2	21.6	40.4%	17.9%	75.1%	50,372
javac (input2)	47.1%	49.2	27.2	28.6	22.4	41.9%	17.6%	79.8%	36,604
javadoc (input1)	56.6%	55.0	48.4	29.3	38.1	46.7%	21.3%	85.7%	20,880
javadoc (input2)	57.7%	59.4	55.1	33.6	44.0	43.4%	20.1%	85.2%	12,740
pizza (input1)	58.9%	37.8	34.4	22.9	27.3	39.4%	20.6%	79.4%	55,652
pizza (input2)	64.0%	39.4	30.9	23.7	24.4	39.9%	21.0%	82.1%	38,004

our splitting algorithm, and examine the sensitivity of splitting decisions to program inputs.

Table 4 shows that the five Java benchmarks for two different sets of inputs have a significant number of classes (17–46% of all accessed classes), that are candidates for splitting (i.e., live and sufficiently large). Even more promising, 26–100% of these candidate classes have field access profiles that justify splitting the class. The cold fields include variables that handle error conditions, store limit values, and reference auxiliary objects that are not on the critical data structure traversal path. The splitting algorithm is fairly insensitive to the input data used for profiling field accesses. For all benchmarks, regardless of input data set, 73–100% of the classes selected for splitting were identical (the second number enclosed in brackets indicates the number of common classes split with different inputs), with the same fields labeled hot or cold barring a few exceptions. Closer examination of the classes split with one input set and not the other revealed these to be classes with the smallest normalized temperature differentials (though greater than 0.5).

Table 5 analyses the characteristics of the split classes in more detail. Accesses to fields in split classes account for 45–64% of the total number of program field accesses. The average dynamic split class sizes were computed by weighting each split class with the number of its split instances. The splitting algorithm reduces dynamic class sizes by 17–23% (cassowary shows a 68% reduction), and with the exception of javadoc, permits two or more hot instances to fit in a cache block. The normalized temperature differentials are high (77–99%), indicating significant disparity between hot and cold field accesses. Finally, the additional space costs for the reference from the hot to cold portion are modest—on the order of 13–74KB.

Next, the UltraSPARC's [12] hardware counters were used to measure the effect of our cache-conscious object layouts on cache miss rates. Each experiment was repeated five times and the average value reported (in all cases the variation between the smallest and largest values was less than 3%). With the exception of cassowary, the test input data set differed from the input data used to generate field access statistics for class splitting. First, we measured the impact of Chilimbi and Larus' cache-conscious object co-location scheme on the original versions of the five Java benchmarks. Next, we measured its impact on the hot/cold split classes versions of the benchmark. The results are shown in Table 6 (we do not report L1 miss rates since L1 cache blocks are only 16 bytes and miss rates were marginally affected, if at all). **CL** represents direct application of Chilimbi and Larus' cache-conscious object co-location scheme, and **CL + CS** represents this scheme combined with hot/cold class splitting. The results indicate that Chilimbi and Larus' cache-conscious object co-location scheme reduces L2 miss rates by 16–29% and our hot/cold class splitting increases the effectiveness of this scheme, reducing L2 miss rates by an further 10–27%

Finally, we measured the impact of our techniques on execution time. The results shown in Table 7 indicate that hot/cold class split-

Table 6: Impact of hot/cold object partitioning on L2 miss rate.

Program	L2 cache miss rate (base)	L2 cache miss rate (CL)	L2 cache miss rate (CL + CS)	% reduction in L2 miss rate (CL)	% reduction in L2 miss rate (CL + CS)
cassowary	8.6%	6.1%	5.2%	29.1%	39.5%
espresso	9.8%	8.2%	5.6%	16.3%	42.9%
javac	9.6%	7.7%	6.7%	19.8%	30.2%
javadoc	6.5%	5.3%	4.6%	18.5%	29.2%
pizza	9.0%	7.5%	5.4%	16.7%	40.0%

Table 7: Impact of hot/cold object partitioning on execution time.

Program	Execution time in secs (base)	Execution time in secs (CL)	Execution time in secs (CL + CS)	% reduction in execution time (CL)	% reduction in execution time (CL + CS)
cassowary	34.46	27.67	25.73	19.7	25.3
espresso	44.94	40.67	32.46	9.5	27.8
javac	59.89	53.18	49.14	11.2	17.9
javadoc	44.42	39.26	36.15	11.6	18.6
pizza	28.59	25.78	21.09	9.8	26.2

Table 8: `bbcache` evaluation metrics for 5 active SQL Server structures.

Structure	Cache block utilization (original order)	Cache block utilization (recommended order)	Cache pressure (original order)	Cache pressure (recommended order)
ExecCxt	0.607	0.711	4.216	3.173
SargMgr	0.714	0.992	1.753	0.876
Pss	0.589	0.643	8.611	5.312
Xdes	0.615	0.738	2.734	1.553
Buf	0.698	0.730	2.165	1.670

ting also affects execution time, producing improvements of 6–18% over and above the 10–20% gains from Chilimbi and Larus' co-location scheme.

4.2 Structure Field Reordering for C

We used a 4 processor 400MHz Pentium II Xeon system with a 1MB L2 cache per processor. The system had 4GB memory with 200 disks, each a 7200 rpm Clarion fiber channel drive. The system was running Microsoft SQL Server 7.0 on top of Windows NT 4.0. We ran the TPC-C [13] benchmark on this system. Microsoft SQL Server was first instrumented to collect a trace of structure field accesses while running TPC-C. `bbcache` used this trace to produce structure field order recommendations

Out of the almost 2,000 structures defined in the SQL Server source, `bbcache` indicated that 163 accounted for over 98% of structure accesses for the TPC-C workload. In addition, the top 25 of these 163 active structures account for over 85% of structure accesses. For this reason, we focused on these 25 active structures

SQL Server uses a number of persistent, on-disk structures that cannot have their fields reordered without affecting compatibility (Section 2.5). In addition, there are dependencies, such as casting, between structures that prevent reordering the fields of one, without also reordering the other. Finally, SQL server is a highly tuned commercial application, and many of the 25 active structures previously had their fields reordered by hand. We used `bbcache` to select 5 structures that had no constraints on reordering and which showed the largest potential benefits according to the cost and locality metrics provided (Table 8). We reordered these 5 structures according to `bbcache`'s recommendations and ran the TPC-C benchmark on this modified SQL Server several times. The performance of the modified SQL Server was consistently better by 2–3%.

5. RELATED WORK

Recent research has focused on reorganizing the data layout of pointer-based codes to improve memory system performance [1, 6, 5, 14, 8, 11]. Calder et al. [1] apply a compiler-directed approach that uses profile information to place global data, constants, stack variables, and heap objects. Their techniques produced significant improvements for globals and stack data, but only modest gains for heap objects. Their approach differs from ours in two respects. First, they adjusted the placement of entire objects, while we reorganized the internal field of objects. Second, we focus on heap object.

Chilimbi et al. [6] describe two tools—a data reorganizer for tree-like structures and a cache-conscious heap allocator—for improving the cache performance of C programs. The tools require few source code modifications and produce significant performance improvements. Both tools reorganize the memory arrangement of entire objects. This work complements their work, since the combination of the two techniques yields larger benefits than either alone.

Chilimbi and Larus [5] showed how to use generational garbage collection to reorganize data structures so that objects with high temporal affinity are placed near each other, so they are likely to reside in the same cache block. We extend their technique to Java, and increase its effectiveness by partitioning classes into a hot and cold portion.

Truong et al. [14] also suggest field reorganization for C structures. They develop a memory allocation library to support interleaving identical fields of different instances of a structure that are referenced together and demonstrate significant reductions in cache miss rates and execution times. Our work complements theirs since they perform field reorganization manually using profiling data, whereas we describe a tool—bbcache—that automates part of this process. Moreover, we showed how to fully automate cache-conscious layout for Java-like languages.

Concurrently, Kistler and Franz [8] describe a technique that uses temporal profiling data to reorder structure fields. Their work differs from ours in four ways. First, they use path profiling data to capture temporal relationships. Second, they optimize their layouts for cache-line fill buffer forwarding, a hardware feature supported on the PowerPC, whereas we optimize layouts for inherent locality. Third, their algorithm divides the affinity graph into cache-line sized cliques. A consequence of this technique is that there may be no affinity between fields placed in consecutive cache lines. Without cache-line alignment at allocation time (i.e., by suitably manipulating the *malloc* pointer), the resultant layout may not perform well. Finally, we provide structure activity rankings and two metrics for evaluating structure field orders that permit an informed selection of suitable candidates for structure field reordering.

Seidl and Zorn [11] combine profiling with a variety of different information sources present at run time to predict an object's reference frequency and lifetime. They show that program references to heap objects are highly predictable and that their prediction techniques are effective. They use these predictions to generate customized allocators that decrease a program's page fault rate. Our techniques on the other hand aim at reducing a program's cache miss rate.

6. CONCLUSIONS

This paper describes two techniques—*structure splitting* and *field reordering*—that improve cache performance by changing the internal organization of fields in a data structure. While previous techniques, which concentrate on arranging distinct structure instances, worked best for structures smaller than half a cache block, the techniques in this paper improve the cache behavior of larger structures.

Measurements indicate that Java programs have a significant number of classes with field access profiles that permit a simple, bimodal division into hot (frequently accessed) and cold (rarely accessed) fields. In addition, these classes account for a significant fraction of all field accesses. The structure splitting algorithm described in this paper is effective at dividing these classes into hot and cold portions. Perhaps more importantly, the splitting decisions are robust, being fairly insensitive to input data used for profiling class field accesses. This structure splitting algorithm reduced the cache miss rates of five Java programs by 10–27%, and improved their performance by 6–18% beyond the improvement from previously described cache-conscious reorganization techniques. These promising results encourage further experimentation with a larger variety of benchmarks.

For large structures, which span multiple cache blocks, reordering fields, to place those with high temporal affinity in the same cache block also improves cache utilization. This paper describes a tool that recommends C structure field reorderings. Preliminary measurements indicate that reordering fields in 5 active structures improves the performance of Microsoft SQL Server 7.0 by 2–3%. Unfortunately, constraints due to persistent data formats, as well as code that relied on particular field orders, prevented reordering several other promising structures.

These results suggest that structure layouts are better left to the compiler or runtime system, rather than being specified by programmers. Modern languages, such as Java, provide opportunities to exploit this flexibility to improve programs' cache performance.

ACKNOWLEDGEMENTS

The authors would like to thank Ronnie Chaiken, Roger Crew, Richard Shupak, and Daniel Weise for helpful discussions. Bruce Kuramoto, and Hoi huu Vo provided assistance with the Microsoft tracing tool. Sameet Agarwal, Maurice Franklin, Badriddin Khessib, and Rick Vicik helped with SQL Server. The authors are indebted to Craig Chambers for writing the Java SPARC assembly code generator, and providing the Vortex compiler infrastructure. Dave Grove assisted with Vortex. We are grateful to Han Lee and Ben Zorn for providing us with BIT, the Java bytecode instrumentation tool. Finally, the anonymous referees offered several useful comments. This research is supported by NSF NYI Award CCR-9357779, with support from Sun Microsystems, and NSF Grant MIP-9625558. The field reordering work was performed while the first author was an intern at Microsoft Research.

REFERENCES

[1] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. "Cache-conscious data placement." In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 139-149, Oct. 1998.

[2] Craig Chambers. "Object-oriented multi-methods in Cecil." In *Proceedings ECOOP'92, LNCS 615, Springer-Verlag*, pages 33–56, June 1992.

[3] Craig Chambers. "The Cecil language: Specification and rationale." *University of Washington Seattle, Technical Report TR-93-03-05*, Mar. 1993.

[4] Craig Chambers, Jeffrey Dean, and David Grove. "Whole-program optimization of object-oriented languages." *University of Washington Seattle, Technical Report 96-06-02*, June 1996.

[5] Trishul M. Chilimbi, and James R. Larus. "Using generational garbage collection to implement cache-conscious data placement." In *Proceedings of the 1998 International Symposium on Memory Management*, Oct. 1998.

[6] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. "Cache-conscious structure layout." In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation,* May 1999.

[7] R. F. Crew. "ASTLOG: A language for examining abstract syntax trees." In *Proceedings of the USENIX Conference on Domain-Specific Languages,* Oct. 1997.

[8] T. Kistler, and M. Franz. "Automated record layout for dynamic data structures." *Department of Information and Computer Science, University of California at Irvine, Technical Report 98-22*, May 1998.

[9] C. Krintz, B. Calder, H. B. Lee, and B. G. Zorn "Overlapping execution with transfer using non-strict execution for mobile programs." In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 159-169, Oct.

1998.

[10] H. B. Lee, and B. G. Zorn. "BIT: A Tool for Instrumenting Java Bytecodes." In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS'97)*, pages 73-83, Dec. 1997.

[11] M. L. Seidl, and B. G. Zorn. "Segregating heap objects by reference behavior and lifetime." In *Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 12-23, Oct. 1998.

[12] Sun Microelectronics. *UltraSPARC User's Manual*, 1996.

[13] Transaction Processing Council. *TPC Benchmark C, Standard Specification, Rev. 3.6.2*, Jun. 1997.

[14] Dan N. Truong, Francois Bodin, and Andre Seznec. "Improving cache behavior of dynamically allocated data structures." In *International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1998.