

# Splitting Data Objects to Increase Cache Utilization (Preliminary Version, 9<sup>th</sup> October 1998)

Michael Franz and Thomas Kistler  
*Department of Information and Computer Science*  
*University of California*  
*Irvine, CA 92697-3425*

## **Abstract**

We present a technique to increase data cache utilization of pointer-based programs. These caches are often underutilized when an algorithm accesses certain data-members much more frequently than others. In programs that allocate large numbers of identical objects, this can result in some cache lines becoming very busy, whereas others are left virtually untouched because “cold” program data is mapped onto them.

Our solution consists of grouping all the frequently accessed data members together and physically separating them from the less frequently accessed ones. Each object gets split into a “hot” part and zero or more “cold” parts of equal size, which are evenly spaced from each other in memory, facilitating simple address calculation. The hot parts of different objects can then be allocated contiguously, evenly tiling the data cache and thereby increasing its utilization.

Our technique is fully automatic, based on profiling, and extends earlier work on automated cache-conscious storage layout of dynamically allocated data structures. It is applicable to all type-safe programming languages that completely abstract from physical storage layout; examples of such languages are Java and Oberon.

## **1 Introduction**

Cache behavior is increasingly becoming a limiting factor of overall performance, which is why a significant amount of research has recently been directed at compiler-based code optimizations that make better use of caches. There are two fundamental approaches to improving cache performance: One is to employ explicit *prefetching*, thereby ensuring that data items are loaded early enough ahead of their use so that memory latency can be

masked effectively [MLG92]. The other is to *transform the storage layouts and control structures* of programs in such a manner that the resulting storage access patterns exploit the particular cache characteristics to the fullest [WL91]. Both of these techniques have been studied extensively for array-based programs, with excellent results.

Unfortunately, applying similar techniques to *pointer-based programs* is much more difficult because pointer-based programs access data in a much less predictable manner than array-based ones. For example, it is difficult to enhance a linear-list traversal with sufficient data-prefetching instructions to mask memory latency completely, because one might need to look several elements ahead in the list in order for this optimization to be effective. Similarly, techniques such as cache-blocking, loop-skewing, and loop-tiling are difficult to translate to pointer-based applications because they are based upon modeling data accesses via polynomial address-calculation functions. Pointer-based data structures, on the other hand, allow arbitrary insertion and deletion of elements, so that even an initially regular memory layout can soon degenerate to a chaotic state that defies formal description.

However, even if it is impossible to *precisely* map dynamically allocated data structures onto the cache, one can still do so *statistically* using profiling information as a guide. In a previous paper [KF98], we have shown how cache locality can be increased by re-ordering the constituent data members of dynamically allocated heap objects. In this paper, we go one step further, demonstrating that it may even be advantageous to split individual objects (which on the level of the programming language are considered indivisible entities) into physically disjoint memory regions.

Our optimization applies specifically to programming languages that are fully type-safe, such as Java [AG96] and Oberon [Wir88]. These languages do not permit programmers to make any assumptions about the internal storage layout of the data structures they declare; as a consequence, choosing such an internal layout lies completely in the domain of the compiler.

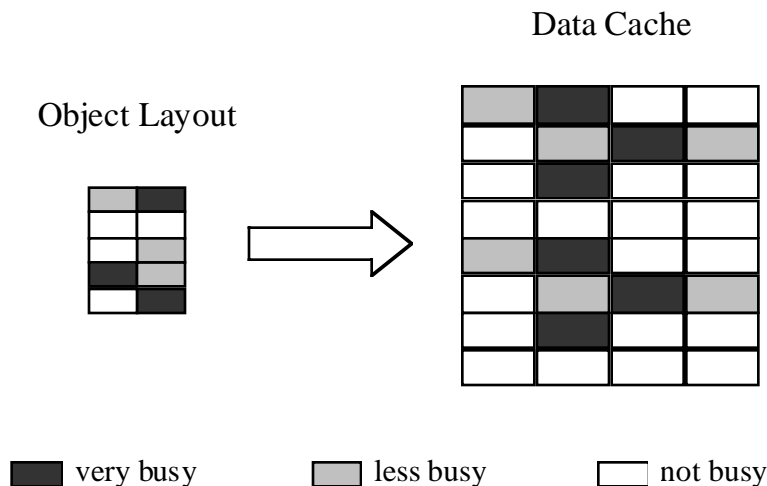
The remainder of this paper is organized as follows: Sections 2 and 3 outline our memory allocation scheme. Section 4 evaluates the performance of our method. Section 5 presents related work and Section 6 concludes our paper.

## 2 Object Layout vs. Cache Performance

Traditionally, compilers have relied on the programmer to specify the internal layout of dynamically created objects. This is perhaps a consequence of the widespread practice of *type-casting*, which implies that programmers have made certain assumptions about the memory representation of their data that the compiler cannot simply override. Further, when designing data structures, programmers are usually more interested in modeling the problem domain than reaching peak performance. Hence, they usually declare data

members in an order that corresponds to their logical relationship at the model level rather than their actual algorithmic interaction at the program level.

This is a waste of cache resources. If data members are grouped in a model-related fashion, it is quite likely that often accessed (“busy”) members will come to lie next to data members that are accessed much less frequently, or not at all. Consequently, a program’s working set of dynamic data is spread over unnecessarily many cache lines (Figure 1).



*Figure 1: Cache Utilization Without Data Layout Optimizations*

The advent of genuinely type-safe programming languages such as Java [AG96] and Oberon [Wir88] has liberated compiler designers from the need to follow programmers’ directions when deciding on an optimal internal layout for dynamic data structures. A much more sensible data layout than the one shown in Figure 1 (for semantically the same data structure) is presented in Figure 2. Here, all the busy data members have been moved so that they are co-located on the same cache line. This has the effect that the first load of *any* of these busy data members will simultaneously also bring *all the others* into the cache, decreasing memory latency on subsequent accesses.

In previous work [KF98], we have shown that this optimization is not only practical but that it can yield substantial execution-time improvements. We have built a system that re-orders individual data members to increase spatial and temporal locality. Our system is able to perform this optimization at run-time on live data structures. By modeling the temporal relationship between individual accesses to data members, we were further able to automatically order the individual members on the same cache line in such a manner that the predominant storage access pattern in the algorithm matches the ordering in which the cache-line buffer is filled from memory. As a consequence, when

cache-line fill-buffer forwarding is available in the hardware, successive accesses to the same cache line produce only a minimal number of pipeline stalls.

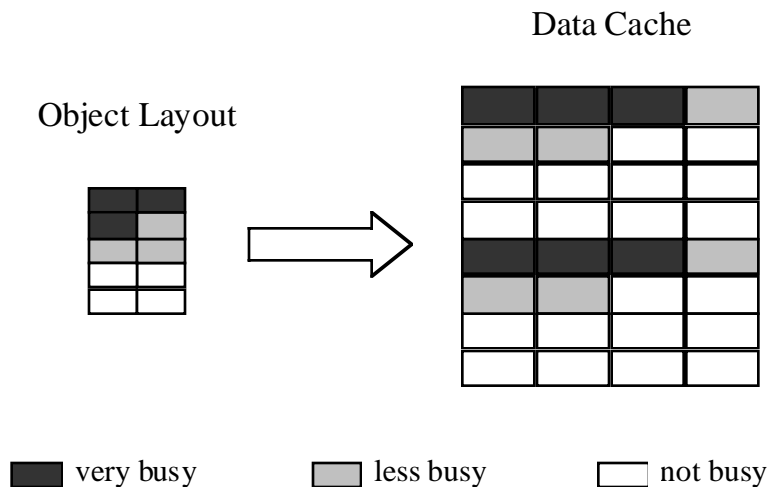


Figure 2: Cache Utilization After Data-Member Re-Ordering

Data-member re-ordering results in a data cache mapping that separates very busy locations from less busy ones. This may still lead to sub-optimal cache usage when an algorithm accesses a large number of objects that all have the same data type (and hence, the same internal storage layout). Depending on the alignment of heap blocks in memory, the situation may occur that the objects used by the algorithm tile the cache evenly, imprinting their local distribution of busy vs. non-busy locations onto the data cache as a whole (Figure 2). This is particularly likely if the cache has a low degree of associativity.

Note that this scenario is still much better than the completely unoptimized case depicted in Figure 1, in which the non-busy cache locations are evenly sprinkled across the whole cache. Data-member re-ordering works well if several different types of data reside in the cache simultaneously, and if care has been taken during the data layout phase to vary the offset of the busy data members relative to the start of each memory block so that busy cache lines for one type coincide with non-busy ones for another type.

However, many important algorithms operate on elements of just a single data type. Hence, the cache soon fills up with data that has a homogeneous distribution of busy and non-busy cache lines. To counter this effect, we have investigated splitting objects into several parts that are allocated at different locations in memory, a “hot” part that holds the very busy data members and zero or more “cold” part that store the remaining ones. This layout makes it possible to group the “hot” data members of all the objects in such a manner that they tile the cache evenly and completely (Figure 3). It also enables the hot

parts of multiple objects to share the same cache line, which is otherwise usually prevented by heap-block alignment.

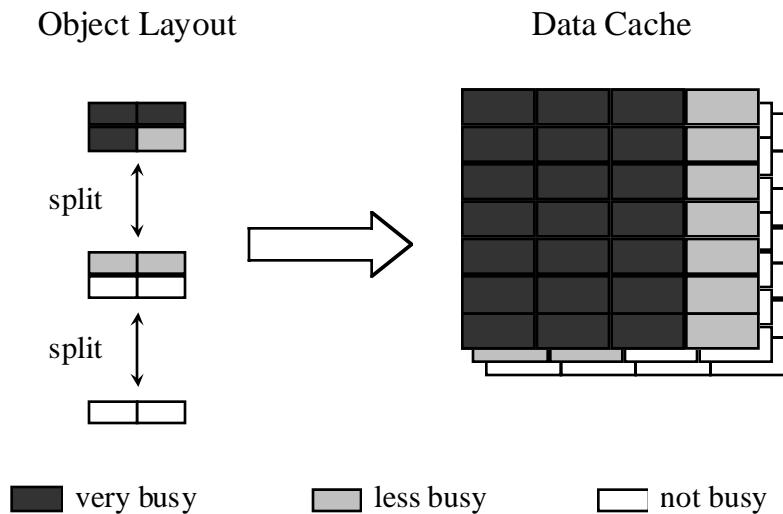


Figure 3: Cache Utilization After Object Splitting

### 3 Automated Object Splitting

Our system collects detailed profiling information about the access frequency of each data member of every dynamically allocated data type. It then classifies the individual data members of each type as “hot” or “cold”. We call the size relationship between the hot and the cold parts the *temperature gradient* of the object. Classifying objects by their temperature gradient simplifies object allocation and garbage collection significantly.

The objective is to tile the data cache evenly with hot data members. The assumption is that cold members are accessed infrequently enough so that their presence doesn’t have much of an impact on the cache. However, the cold data members still need to be allocated somewhere, and for practical reasons, their addresses must be easily computable from the base address of the object (which points directly to the “hot” part). These constraints naturally lead to an allocation of objects by their temperature gradient.

We require the size of each cold part to be exactly the same size as that of the hot part. This may require the addition of “padding” if insufficiently many cold data members are available. Whenever this constraint leads to problems, for example, because the cold part contains a large indivisible entity such as a string, the corresponding object type is simply not split at all.

The allocation of split objects then occurs within larger heap sub-structures that we call “pages” and that are taken from the heap and returned to it only as a whole. Each page has a temperature gradient associated with it and stores only objects with this gradient.

For example, objects with a gradient of 1:1 are allocated on memory pages that are slightly larger than twice the size of the data cache. The hot parts are allocated in the front of the page, while the cold parts are allocated in the back of the page. For each object allocated in such a page, the distance between its constituent parts is equal to the sum of the cache size plus the size of a single cache line. As a consequence, the cold part is mapped to exactly the same cache lines to which it would have been mapped even if the object hadn’t been split at all (Figure 4).

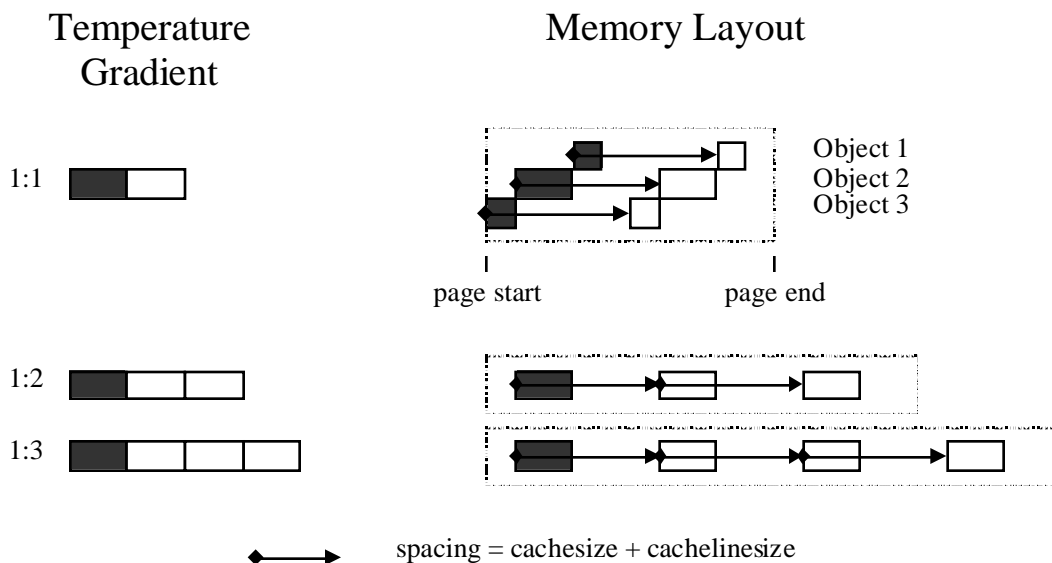


Figure 4: Allocating Objects by Temperature Gradient

Objects with a gradient of 1:2 are allocated on memory pages that are about three times as large as the data cache. The cold part is itself subdivided in two halves, and all three parts are placed equidistantly from each other in memory, with the same offset (cachesize + cachelinesize) as above. Hence again, the cold data members are mapped to exactly the same cache lines to which they would have been mapped if the object hadn’t been split. This is so that cold data members of an object can never evict a hot data member of the same object from the cache, in situations where an algorithm accesses all fields of an object in rapid succession.

The analogous strategy is applied to objects with a temperature gradient of 1:3 and 1:4. Objects that have a larger gradient than 1:4 or a smaller gradient than 1:1 (i.e., that

exhibit a relatively homogeneous access to their data members) are not split but allocated in the general heap in the usual manner. Splitting an object only makes sense if the access frequency of all the cold data members combined is significantly smaller than that of the hot ones.

Splitting objects in this manner comes at very low additional cost. This is because the compiler can statically compute the offsets of the distant cold fields. Further, objects are allocated in pages with a uniform temperature gradient. Hence, the storage allocator need only consider the hot-space when searching for free memory to allocate new objects; each hot-space is statically associated with the appropriate amount of cold-space “further down” on the same page. Exactly the same argument applies to the scan phase of our mark-and-sweep garbage collector, which can safely ignore the existence of cold-space altogether.

The remaining overhead associated with splitting objects largely depends on processor architecture. For example, on the PowerPC that our implementation is based on, the constant displacement for the register indirect addressing mode needs to be specified in 16 bits while the cache size is  $2^{16}$  Bytes. Hence, access to the cold fields requires an additional instruction to compute the actual address. Similarly, initializing objects (to zero) and cloning objects requires setting up multiple loops rather than just a single one. Except for object initialization, these costs apply only to the infrequent case when cold data members are actually accessed.

## 4 Benchmarks

Data layout optimizations are intended to provide performance gains on applications with poor data locality and large working sets. As others have noted [TBS98], benchmarks such as *SpecInt95* do not exhibit this behavior. Consequently, to test our results we have used a nonstandard suite of programs, all of which make extensive use of dynamically allocated data structures.

Our preliminary benchmarks are based on the following programs: *TreeAdd* and *TSP* (travelling salesman) from the Olden benchmark suite [RCR+95], and *Jigsaw* from the WPI benchmark suite [FKL+92]. Each of these benchmarks allocates many megabytes of data, executes billions of instructions, and represents frequently used operations on dynamic data structures.

For our benchmarks, we first instrumented all load/store instructions prior to program execution to obtain fully accurate profiling information and collected profiles for each of the programs in our benchmark suite. Based on these profiles, we then generated optimized versions of the programs. Finally, the optimized versions of the programs were tested using different sets of input data than were used during the profiling phase.

To evaluate the performance of our algorithm in terms of improvement in execution time and cache miss rates, we ran the entire benchmark suite multiple times on a PowerPC 604e (32Kbyte, four-way set-associative first-level data cache, 1Mbyte second-level cache), utilizing the PowerPC’s performance monitor. The performance monitor includes four 32-bit hardware counters that record detailed events during execution, such as instruction dispatches, instruction cycles, misses in the cache, and load/store miss latencies.

The Table below shows our preliminary performance results<sup>1</sup>. *Jigsaw* with a hot part of size 16 Bytes achieves the largest speedup. Its data structure is composed of 11 longwords and its algorithm can benefit ideally from our optimization. Choosing a slice size that is half the length of a cache line enables the hot parts from pairs of objects to reside in the cache simultaneously, which causes the considerable speedup.

Benchmark (slice size)	Cache Misses	Execution Time
Jigsaw (16)	-51 %	- 24 %
Jigsaw (32)	+ 2 %	+ 2 %
TreeAdd (16)	0 %	+2 %
TreeAdd (32)	0 %	0 %
TSP (32)	-4 %	+1 %

*Table 1. Overall Performance*

The other programs in the benchmark suite do not show any significant improvement, or even a performance reduction which seems to be due to object initialization. For a split object, separate initialization loops (that zero memory) need to be set up for each of the slices.

## 5 Related Work

Cache optimizations aim to reduce the gap between memory and processor speeds. For example, data locality can be increased in scientific, array-based programs by applying techniques such as loop-reversal, loop-tiling, loop-skewing, and cache-blocking [WL91]. They change algorithmic behavior by reordering the execution sequence of iterations and by changing the shape of a loop’s iteration space and iteration depth. Rivera and Tseng’s

---

<sup>1</sup> These are early preliminary results. This preliminary report will soon be superseded by one that presents several additional benchmarks and a substantially more thorough analysis.

algorithm [RT98] inserts inter-variable and intra-variable padding to control the placement of arrays in memory and to control the optimal row size of arrays. This technique can be applied orthogonally to control-transforming optimizations. Previous work has also studied the problem of data prefetching in the context of array-based programs [MLG92]. Prefetching reduces memory latency by loading data values ahead of time into the cache. This is particularly beneficial for array-based programs that exhibit highly regular data access patterns.

Because of fundamental differences in program structure, most of these techniques developed for scientific programming cannot be applied directly to pointer-based applications that often exhibit much more complex access patterns. Also, the overall size of dynamically allocated data structures can usually not be determined at compile time. It is therefore not surprising that work on data prefetching [LM96], automatic placement of data in memory [CKJ+98, CLH98], and automatic data transformations in the context of pointer-based applications is happening only recently. With the ever-increasing memory hunger of pointer-based programs, further factors are becoming relevant for data cache performance, such as the effects of memory allocators [GZH93, SZ97, AG98] and garbage collectors [Rei94] in modern operating systems.

Truong et al. [TBS98] have recently proposed a technique they call “object interleaving” that is very similar to our independently developed one. The major difference is that our technique is fully automatic, classifies and allocates objects by their temperature gradient, and supports garbage collection. Truong et al.’s technique, on the other hand, is manual. It requires programmers to insert padding between the individual data members of an object, and forces them to call non-standard allocation and deallocation routines that support interleaved objects. Garbage collection is not supported by Truong’s method.

## 6 Summary and Conclusion

We have implemented a fully automatic storage management system that increases cache utilization by separating busy memory locations from less busy ones, enabling the former to be allocated in a more cache-friendly manner. Our strategy permits the relevant parts of different objects to share the same cache line even when each object by itself is larger than a single cache line. The technique is based on dynamic profiling data and compatible with garbage collection. It has a low overhead that sometimes can be compensated entirely improved cache performance, resulting in impressive net speed gains.

## Acknowledgement

We would like to thank Joachim Büchse and Ziemowit Laski who provided many helpful comments on this research. Part of this work is funded by the National Science Foundation under grant CCR-97014000.

## References

- [AG98] A. Aiken and D. Gay. “Memory Management with Explicit Regions”. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.
- [AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.
- [CKJ+98] B. Calder, C. Krintz, S. John, and T. Austin. “Cache Conscious Data Placement”. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1998.
- [CLH98] Trishul Chilimbi, James Larus, and Mark Hill, *Improving Pointer-Based Codes Through Cache-Conscious Data Placement*. Downloaded via [www.cs.wisc.edu/~larus](http://www.cs.wisc.edu/~larus).
- [FKL+92] D. Finkel, R. E. Kinicki, J. A. Lehmann, and J. CaraDonna; *Comparison of Distributed Operating System Performance Using the WPI Benchmark Suite*; Technical Report WPI-CS-TR-92-2, Worcester Polytechnic Institute; 1992.
- [GZH93] D. Grunwald, B. Zorn, and R. Henderson. “Improving the Cache Locality of Memory Allocation”. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, 177–186, June 1993.
- [KF98] T. Kistler and M. Franz. *Automated Record Layout for Dynamic Data Structures*. Technical Report No. 98-22, Department of Information and Computer Science, University of California at Irvine; May 1998 (revised September 1998).

- [LM96] C.-K. Luk and T. Mowry. "Compiler-Based Prefetching for Recursive Data Structures". In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.
- [MLG92] T. Mowry, M. Lam, and A. Gupta. "Design and Evaluation of a Compiler Algorithm for Prefetching". In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.
- [Mot96] Motorola Inc. *PowerPC 604e Microprocessor Supplement and User's Manual Errata*. Motorola Inc. September 1996.
- [RCR+95] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. "Supporting Dynamic Data Structures on Distributed Memory Machines". In *ACM Transactions on Programming Languages and Systems*, 17(2), March 1995.
- [Rei94] M. Reinhold. "Cache Performance of Garbage-Collected Programs". In *Proceedings of the 21th Annual Symposium on Principles of Programming Languages*, June 1994.
- [RT98] G. Rivera and C.-W. Tseng. "Data Transformations for Eliminating Conflict Misses". In *Proceedings of the 25th Annual Symposium on Principles of Programming Languages*, June 1998.
- [SZ97] M. Seidl and B. Zorn. *Predicting References to Dynamically Allocated Objects*. Technical Report CU-CS-826-97, Department of Computer Science, University of Colorado, Boulder, January 1997.
- [TBS98] D. N. Truong, F. Bodin, and A. Sez nec; "Improving Cache Behavior of Dynamically Allocated Data Structures". In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, Paris, France, October 1998.
- [Wir88] N. Wirth. "The Programming Language Oberon". *Software-Practice and Experience*, 18:7, 671-690; 1988.
- [WL91] M. Wolf and M. Lam. "A Data Locality Optimization Algorithm". In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.