Cache-Conscious Memory Allocation

by

Christopher Cyll

A thesis submitted in partial fulfilment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

WILLIAMS COLLEGE
Williamstown, Massachusetts

May 24, 2004

Abstract

Today, almost all software uses dynamic memory allocation. In some programming languages, it is impossible to even write "Hello World!" without implicitly calling the allocator. However, the costs of dynamic memory allocation on cache performance are relatively unstudied. This thesis explores the effects of dynamic memory allocation on cache performance, as well as methods to make memory allocators cache-conscious.

We use profiling techniques to build custom memory allocators. These allocators contain statically generated memory layouts that arrange memory in a cache-conscious manner for previously analyzed workloads. Using a variety of techniques to build memory layouts, we demonstrate that such layouts can dramatically affect program speed and cache miss rate. We find that some layout strategies speed up certain programs by as much as 28% while others cause certain programs to run up to 2.6 times slower. Additionally, we provide an open source foundation which is fully compatible with all standard C allocation functions in order to aid future research

Contents

Chapter 1

Introduction

Cache memory is vital to achieving high performance in modern architectures. Memory accesses are frequent and miss penalties are high. Using the twin concepts of spacial and temporal locality, caches keep ever faster CPU's supplied with data and instructions. However, the past decade of programming language evolution has drastically changed the layout of program data in memory. Gone are the days when predictable array accesses were typical. Ever growing data structure complexity and increased reliance on dynamically allocated memory have conspired to scatter related data throughout the address space. Linked data structures now rule the bus.

A walk through those pointer connected structures meanders through the address space, trashing a cache. Worse, if mapped to addresses that conflict in the cache, contemporaneously accessed structures can force each other out of the cache.

However, the data access patterns that were obscured with the dominance of linked data structures still exist. The patterns are extremely difficult for a compiler to deduce statically on its own; however, researchers are exploring a variety of techniques for using other available information to predict and optimize for contemporaneous accesses.

Information can be gathered from the programmer, from the source code,

and from the running program itself. Each method comes with its own costs and benefits. Extracting information from the programmer often takes precious time. Extracting information from the source or profiling can be done automatically; however, this adds complexity to the build process. Programmers may have knowledge about algorithms that are not discernible from the source or the execution. On the other hand, a computer is tireless and precise. Additionally, profiling is the only method that has full knowledge about machine level execution. The information gained from automatic strategies can, in turn, be leveraged to improve performance in several ways.

One approach, prefetching, works around the unpredictable layout of linked structures. It uses knowledge to anticipate accesses and preload memory. This can be done at both the hardware and software level and can be transparent to the programmer.

Another approach attempts to lay out memory in a cache-friendly fashion. That is, it tries to arrange memory so that despite the linked nature of the structures, the accesses ultimately follow patterns similar to those for arrays. This approach can be broken down further into techniques that order data within memory blocks and techniques that arrange the memory blocks in the address space. The success of each approach varies depending on programming language features.

Our research investigates linked structure layout within memory. Linked structures are typically built using a dynamic memory allocator. In C the standard dynamic allocator is `malloc()`, a function that programmers must explicitly invoke. In other languages the allocation can happen implicitly, using C's `malloc()` or their own custom allocator.

In this thesis, we study methods of gathering memory access information and use it to build dynamic allocators that lay out memory in a cache-optimal fashion. We demonstrate that memory layouts can effect program speed and cache miss rates. In addition, we investigate a wide variety of simple layout strategies. Each strategy has different effects on the cache performance

of the program. We document features that make for a successful and an unsuccessful layout, and we attempt to draw conclusions about the effects of our placement strategies.

Chapter 2

Background

The study of dynamic memory allocation on cache performance is relatively new. However, the fields of cache performance and memory allocation have each been thoroughly explored.

2.1 Cache Use

Cache performance is a major concern in modern architectures. Jeffery Gee and Mark Hill[10] suggest that benchmark systems like SPEC'92 often severely underestimate the importance of cache to improving performance. They found that misses for a single data cache were significantly lower in SPEC'92 than in real world work loads, suggesting that good caching strategies are essential to achieving better performance for real world programs.

Additionally, evidence shows that programs written in an object-oriented paradigm have worse performance than programs written in traditional procedural styles. For instance, Calder, Grunwald, and Zorn[6] document a number of differences between C and C++ programs. In their study of data cache miss rates, they found a 2.43% mean miss rate for C programs and a 2.48% mean miss rate for C++ programs. However, the mean miss rate for C++ programs written in an object-oriented fashion was 5.08%, over twice

that of the non-object-oriented programs.

Rule of Thumb 1:
Object-oriented programming degrades cache performance.

Combined, these two observations present a problem. Modern programming languages and styles are generating more cache misses and preventing systems from achieving their full performance.

2.2 Prefetching

Prefetching is one technique used to reduce data cache misses. Its fundamental goal is to anticipate memory accesses and preload them into cache. This is very easy for traditional linear access patterns. However, in order for this strategy to be successful for linked structures, the hardware and software need to understand the relationships between linked structures. Dependence based prefetching can have dramatic effects on cache performance, as demonstrated by Roth, Moshovos, and Sohi[16]. Hardware engines can be built that observe access patterns and predicts pointer loads are starting to become available. These engines then prefetch the pointer destinations that follow the traversal pattern. This helps prevent cache misses for linked structures. They found that when using one of these engines they could load structures well ahead of time for better performance using jump pointers. Jump pointers link structures that are several traversal iterations apart ensuring that the hardware prefetcher has enough information to load structures far enough in advance. Their dependence-based prefetching system produced a 10% speedup using a 1 kilobyte prefetch buffer. Prefetching can also be done in software, although it is more complex and typically requires the compiler to understand program behavior in order to insert prefetching instructions into the compiled code.

Dependence-based prefetching provides a significant speed increase. However, it accepts the notion that linked data structures must be scattered throughout memory. To solve this problem engineers add additional hardware or software complexity. Could similar gains be observed by changing the way linked data structures are organized to better suit conventional caching strategies? We believe so.

2.3 Structure Layout

One approach to organizing memory in a cache-conscious fashion is to rearrange fields within data structures. Programmers often describe the structures with little thought to internal layout. They certainly do not arrange fields in a cache-conscious fashion. Some researchers have studied speedups obtained by internally rearranging structures using information gathered about access patterns.

Panda, Semeria, and di Micheli[15] studied this strategy, applying it to data structures that contain both arrays and field structures. Their ability to rearrange the data stems from the fact that a structure of arrays or an array of structures are fundamentally equivalent; however, they are laid out very differently. In large structures composed of a variety of both arrays and field structures, there exists a multitude of possible arrangements. For example, one might have a single structure containing multiple arrays of values or a single array containing structures with multiple values. Based on the program's access patterns one of these layouts might have fewer cache misses. As the number of possibilities increases with nested data structure complexity, so does the possibility of one layout performing better than another. Their algorithm seeks to find a cache-optimal layout and change the program's internal representation accordingly. It does this by analyzing accesses within the inner most loops of a programs. Since these accesses are relatively sequential, arranging them contiguously increases locality. This technique is

especially useful for embedded systems that are compiled together in their entirety, giving these tools maximum freedom to implement the appropriate changes.

Chilimbi, Hill, and Larus[8] use clustering, coloring, pointer elimination, and hot/cold splitting to help avoid cache misses when traversing linked data structures. *Clustering* moves fields accessed contemporaneously near each other, hopefully placing them in the same cache block. *Coloring* lays out data so that the most important fields in each structure do not conflict with each other, and so that the less important data will not force the most important elements out of the cache. *Pointer elimination* changes code to use precalculated offsets instead of pointers. *Hot/cold splitting* breaks structures into a small block which is accessed frequently and a larger block that is accessed less frequently. The hot blocks can then be arranged for best cache performance. Like Panda, Chilimbi provide automated tools that manipulate structure definitions.

The success of rearranging structures depends heavily upon the programming language. In C for instance, linking issues make reordering very difficult. Libraries are built with one structure definition. Changing that structure's layout later requires recompilation of all programs built upon that library. In dynamic languages, there is a great potential to perform such optimizations automatically. However, both static and dynamic languages could benefit from a system that placed the dynamic memory allocations in a cache-conscious manner. Since dynamic allocation happens *a priori* at runtime, such as system is very flexible.

2.4 Dynamic Memory Allocation

The techniques used to allocate memory vary greatly. The simplest allocator issues an `sbrk()` system call. The `sbrk()` function increases the size of a process's data segment and returns a pointer for the new memory. A simple

allocator can ignore calls to `free()` and can always allocate new memory. Such a system would allocate very quickly. But it would waste large amounts of memory.

In reality, allocation schemes are more complex due to the necessity of reusing freed memory. A variety of algorithms are used, including First-Fit, BestFit, QuickFit, and segregated storage[12]. Many real allocators are hybrids, following different strategies depending on the requested object's size. Real world allocators include the BSD Kingsley allocator[12] or GNU's modified version of Doug Lea's[14] allocator. Each of these implementations makes different tradeoffs regarding allocation speed and memory usage. The traditional BSD allocator is known to be fast and memory inefficient, while the GNU allocator is considered slightly slower, but reduces memory usage dramatically.

Additionally, Grunwald, Zorn, and Henderson[12] have studied methods for minimizing the cache impact of allocation. They found that certain allocation schemes traverse large quantities of memory, knocking the program's data out of the cache. QuickFit seems to have the least impact on the cache because it uses size hashed bins to avoid traversal.

2.5 Custom Memory Allocation

Not all programmers are satisfied with the system provided allocator. They write custom allocators to leverage knowledge about their application's memory allocation patterns for better allocator performance. However, studies have shown this is often unproductive[4].

There are certain situations where custom allocators can provide a significant performance boost. The greatest gains are observed when allocation sizes are highly regular or memory is allocated and freed in groups called *regions*.

There are disadvantages to custom allocators as well. Such custom al-

locators are not only error prone, but also require that a memory allocation pattern exist within the program that can be exploited for performance gains[4]. Additionally, the programmer must understand these patterns in order to code the custom allocator.

One solution was proposed by Berger, Zorn, and McKinley[3]. Their system, Heap Layers, provides a library that allows programmers to easily create a variety of allocators and compose them into a single high performing allocator. Heap Layers uses inheritance to make optimal use of programmer domain knowledge while requiring minimal programmer effort.

Other research explores automatically generating custom allocators. Grunwald and Zorn's[11] CustoMalloc profiles an application, then synthesizes an allocator using information gathered about allocation patterns. CustoMalloc builds fast paths for the most frequent allocation patterns. They demonstrated that their custom-built allocators were faster than existing general-purpose allocators. It is unclear if this is still the case as general-purpose allocators may have gained speed in recent years. Additionally, CustoMalloc only speeds up the allocation process. It cannot improve the performance of code using dynamic memory. Allocation only happens once per structure, but structures may be accessed many times.

Barret and Zorn[2] investigated another trace-based approach. Instead of optimizing the allocator for the frequency of size requests, they created a tool to build allocators that optimize based on object lifetime. It profiles a single run of an application and builds a database that reflects correlations between object lifetime and a combination of object size and call stack depth at allocation time. Their specialized allocator then uses this database to allocate short and long lived objects from different areas. The result is that that short-lived objects can be recycled with little effort, improving performance.

Custom allocation can be a useful tool when the speed of allocation is a concern. However, as illustrated by these systems, building a custom allocation requires additional program information. It is still not clear what

means for gathering this information is optimal. The best answer is a mechanism which requires the least programmer effort while still gathering the most useful information. Automatic tools are a possible solution, but only if they can be made transparent to the programmer.

2.6 Garbage Collection

Like object orientation, garbage collection is becoming more and more common in programming languages. There are a variety of techniques that garbage collectors can use to account for and collect unused memory. Collectors seldom rely on one technique and tend to use hybrid approaches. Hybrid collectors allow the best features of different methods to be combined into one system. Modern systems typically utilize copying garbage collectors with generational techniques and special extensions to handle large object areas[17].

Since copying collectors necessarily rearrange memory, there is an opportunity for collectors to arrange memory so that contemporaneously accessed objects are located near each other. This takes advantage of cache prefetching and helps avoid cache conflicts. In addition, when garbage collection is part of a virtual machine, there is the ability to gather runtime statistics which can be used along side a layout engine for maximum cache efficiency. Hardware assisted garbage collection techniques could make this even easier with the proper support. This field is very promising as it comes with little additional cost over standard garbage collection and can be done transparently to the user.

2.7 Cache-Conscious Memory Allocation

Optimal caching seems to be crucial for performance in modern systems. Additionally, memory reordering at the structure level has demonstrated

increased cache hit rates[15]. Finally, memory allocators have the unique ability to arrange structure blocks within the address space. Could a cache-conscious dynamic memory allocator provide a performance increase?

Chilimbi, Larus, and Hall[7] set out to answer this question. They created two cache-conscious memory placement functions. The first is an allocator called `ccmalloc()`. It requires that the programmer supply information about access patterns. Each call to `ccmalloc()` takes not only the size of the request, but also the address of an existing data structure that will be accessed at similar times to the new structure. The other function, called `ccmorph()`, actually restructures trees in memory for optimal access speed. Programmers pass `ccmorph()` a tree root, a traversal function, and information about the cache. Using this information `ccmorph()` attempts to arrange the tree contiguously in memory. Because `ccmorph()` changes pointers, its safety cannot be guaranteed like `ccmalloc()`; however, as long as the programmer follows the interface procedures, the information that `ccmorph()` can gather during run time can result in significant speed ups. The `ccmorph()` functions is well-suited for languages that do not allow arbitrary memory access. The `ccmalloc()` function, however, is a relatively unintrusive option that can be used in almost all situations. It requires only a moment for a programmer to consider which other structures will be accessed concurrently when coding a new allocation. Unfortunately, there are no guarantees that a programmer understands the machine-level implications of the code. An ideal solution would still provide the 28-138% performance increase over hardware prefetching that `ccmalloc()` provides[7], while automatically gathering access information using tracing techniques similar to those used to build custom allocators.

Calder et al.[5] began to investigate this possibility using profiling to understand object relationships and memory accesses. They use stack information to uniquely label objects. Their dynamic allocator then uses these labels to map the most heavily used objects into bins with contemporane-

ously accessed objects. This technique combined with their other tools has reduced cache data misses by 24%. However, because their allocator it is part of a much larger system, it is not useful as a stand alone tool.

2.8 Synthesized Allocators and Cache Performance

Custom-built allocators and cache-conscious allocation are both successful techniques for increasing performance. However, the combination may yield additional benefits. Cache-conscious allocation offers substantial speedups for minimal effort. Profile-based allocator synthesis provides complete execution information and requires even less programmer effort to gather. The goal of this thesis is to explore the potential for increased cache performance using trace data to automatically optimize memory allocations.

Our technique logs information about program execution in order to automatically discover contemporaneous accesses to dynamically-allocated memory. During a trace run, allocations, deallocations, and accesses are recorded. This information is then used to build a cache-conscious allocation layout prior to subsequent runs. This allocation layout is built with complete knowledge of all memory requests and their uses. In theory, this gives our system information with which to build a cache-conscious allocator. Of course, the drawback is that subsequent runs of the program must follow the same allocation pattern in order to benefit from the new allocator. When request patterns differ from the prior run, the cache-conscious allocator falls back to a simple allocator.

Our technique works best for long running programs that allocate early and then spend the majority of their execution time manipulating those allocations. For a given data set, programs that fit this pattern can be run briefly with logging, aborted, and then re-run using a cache-conscious allocator layout. General-purpose programs with unpredictable allocation patterns will

be better served by Chilimbi's[8] approach. However, the potential exists for greater performance increases using our technique because of the additional information gathered in the profiling phase.

2.9 Summary

Cache-conscious memory allocation is a promising area of research that attempts to place dynamic memory requests in memory so as to decrease the number of cache misses and speed up programs. In this thesis we explore how profiling information can be used to pre-build cache-conscious memory placement strategies for the dynamic allocator.

Chapter 3

A Library for Static Allocation

In order to explore the potential of a profile based cache-conscious allocator, we wrote a C library called `libstaticmalloc`. Existing C programs require minimal modification to utilize `libstaticmalloc`. The library gathers execution information about the program during a logging run. A variety of strategies can then be used to build a cache-conscious memory layout from this information. The goal of this library is to explore the cache performance differences between placement strategies.

3.1 Static Allocation

The `libstaticmalloc` library has two modes. When no allocation profile is available, the library operates in logging mode. It uses a minimal `malloc()` implementation that places requests sequentially with no memory reuse. Additionally, the library records a variety of information about the program execution to a log file. Once execution has ceased or the program has been terminated, the log can be converted into a static allocation profile using the `sm-place` tool. A variety of algorithms can be used to synthesize the profile. After an allocation profile is available for a program, the library functions in static allocation mode. This mode compares each request to its expected

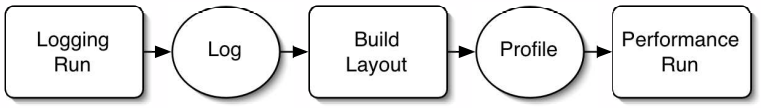

Figure 3.1: Building a cache-conscious memory layout for faster execution.

requests list and places it at a predetermined location. If the sequence of requests ever deviates from the profile, the allocator falls back on the allocator used in logging mode with logging turned off. Figure 3.1 demonstrates the process visually.

3.2 Interface

The library exposes the standard C allocation functions, including `malloc()`, `free()`, `calloc()`, and `realloc()`. These functions are interchangeable with their C standard library equivalents.

> **Rule of Thumb 2:**
> A good tool has a low barrier to entry.

Additionally, `libstaticmalloc` provides a variety of other functions used to communicate information to the logging system about runtime behavior. The `ping()` function allows programmers to describe memory accesses. It takes a pointer to a memory block as well as a weight representing the number of accesses and records them in the log file. The `ccmalloc()` function is compatible with Chilimbi's `ccmalloc()` function. It takes not only an allocation size, but also a pointer to another allocation that will probably be accessed contemporaneously. These functions and others allow the programmer to provide the profile generator with increasing amounts of information. A complete listing of the API is available below.

```
/* Must be called before any other calls to libstaticmalloc */
void malloc_init (char* name);

/* Can be called at program end to print statistics */
void malloc_status ();

/* If called during a logging run, libstaticmalloc halts
   execution. If called during a static run, libstaticmalloc
   stops allocating from the profile. */
void malloc_done_logging ();

/* Standard C calls. */
void* malloc (unsigned int size);
void* calloc (unsigned int nmemb, unsigned int size);
void* realloc(void *ptr, unsigned int size);
void free (void* ptr);

/* Informs libstaticmalloc that ptr is being accessed.
   Weight represents how heavily the data is being used. */
void ping (void* ptr, unsigned int weight);

/* Identical to malloc except that friend points to another
   dynamic allocation that may be accessed contemporaneously
   with this request. Friend may be NULL. */
void* ccmalloc (unsigned int size, void* friend);

/* Identical to calloc except accepts a friends pointer like
   ccmalloc. */
void* cccalloc (unsigned int nmemb, unsigned int size, void* friend);
```

3.3 Logging

The log file is a binary file located in a user's `.malloc/` directory. The format of a log entry is a 4 byte logging code followed by a 4 byte parameter. This pattern is repeated as many times as there are log entries. Tools exist to convert log files into plain text and back again. If there is no profile available when a program compiled with `libstaticmalloc` is run, logging mode is automatically enabled. However, once a profile is available, the allocator runs in static allocation mode and all logging is disabled.

3.4 Static Allocation Profiles

An allocation profile is a binary file located in a user's `.malloc/` directory. It consists of several headers followed by a set of allocation requests. Each request is encoded as a 4 byte integer size and a 4 byte address offset.

When in static allocation mode, the library maps the profile into memory. This is doubly advantageous. First, it vastly simplifies profile access. The standard Unix `mmap()` call returns a pointer to the file data, which then can be treated as an array of memory request structures. This avoids complex I/O and storage allocation for the profile. Additionally, `mmap()` typically uses a special interface to the kernel. Since `mmap()` attempts to return pages that cannot become part of the data or stack segments, the profile is kept separate from the addresses made available via `sbrk()` (the allocation function used by `libstaticmalloc` to allocate user requests).

The total size of the requests is stored in the profile header, and enough space to satisfy all of them is preallocated on application start. An index is kept into the memory-mapped request array. Each request is compared against the size of the next expected request. If at any point the size differs, indicating a deviation from the allocation record, `libstaticmalloc` reverts to simple allocation without logging mode. On the other hand, if the sizes match, `libstaticmalloc` returns the predetermined address.

3.5 Usage

In order to take advantage of `libstaticmalloc` the programmer needs to add a call to `malloc_init()` at the beginning of the `main()` function. In addition, the programmer needs to compile the target program, linking to `libstaticmalloc`.

1. Include the header file:
   `#include <staticmalloc.h>`
2. Call the library initialization function at the start of your program:
   `malloc_init("program name");`
3. Compile linking to the library:
   `gcc -lstaticmalloc ...`

> **Rule of Thumb 3:**
> More information should correspond to better performance.

The programmer may also choose to provide the system with additional information about his program's memory usage. The most effective addition is to replace calls to `malloc()` and `calloc()` with calls to `ccmalloc()` and `cccalloc()`. The extra information these functions provide is potentially valuable to the profile generator. In addition, calls to `ping()` provide information about memory request lifetime as well as helping the generator evaluate the importance of each memory block. The more frequently an object is accessed, the greater the potential for cache conflicts.

Calls to these functions might look like this (examples taken from cover):

- `c = (cell*)cccalloc(1,sizeof(cell),lst);`
- `malloc_init(argv[0]);`

- `ping (ent, 1);`

It is difficult to anticipate the effects of an incomplete annotation on the allocator synthesis process. For this reason if a function is used to provide the compiler with information, it should be used in applicable situations.

3.6 Summary

The `libstaticmalloc` library provides all the standard C dynamic allocation functions making porting easy. In addition, the process of adding extra function calls that provide `libstaticmalloc` with more information is simple and can be accomplished with minimal effort.

Chapter 4

Memory Layout Algorithms

The offline allocator synthesizer can use a number of possible placement algorithms. Each strategy results in different performance for different programs. In addition, the best strategy may be dependent upon the thoroughness of the programmer's annotations. For example, programs with minimal usage information will be better served by techniques that make use of friendship information. It is worth noting that none of the strategies below use knowledge about freed memory in order to recycle memory locations, although this possible.

4.1 Simple

The simple allocator generates a statically allocated profile that is identical to the logging run. Since the logging run places requests sequentially and reuses no space, a profile generated with this strategy will have one request after another. This is the simplest of all possible allocation strategies. It is a useful reference because it resembles the layout strategies of typical allocators.

4.2 Random

The random strategy creates a new profile by shuffling the requests randomly and then placing them sequentially. This strategy is not designed to demonstrate a performance increase. Instead it serves as an example of a system using completely uninformed placement.

4.3 Size

The size heuristic takes all logged allocations and sorts them in order of size. The allocations are then placed, smallest first, in memory. This naïve algorithm hypothesizes that allocations of the same size may be similar types of objects and therefore might be accessed contemporaneously. Placing contemporaneously accessed requests near each other should minimize collisions and maximize the effect of line loading.

4.4 Friends: Depth-First

The friends strategy depends on information gleaned from the Chilimbi[7] `ccmalloc()` style interface. This information is used to build a graph of the memory requests where nodes are connected by unidirectional edges. The connections point from the friend supplied to `ccmalloc()` to the returned request. This graph can then be traversed for a variety of purposes. This strategy performs a depth-first search, assigning sequential address to nodes as they are traversed. See Figures 4.1 and 4.2 for an example. It starts traversal at the largest request. When the depth-first traversal of friends completes, it continues from the next largest unplaced request.

This strategy helps prevent conflicts between requests that the programmer thinks are likely to be accessed contemporaneously. Of course, the success of the this strategy depends on the programmer's ability to chose accu-

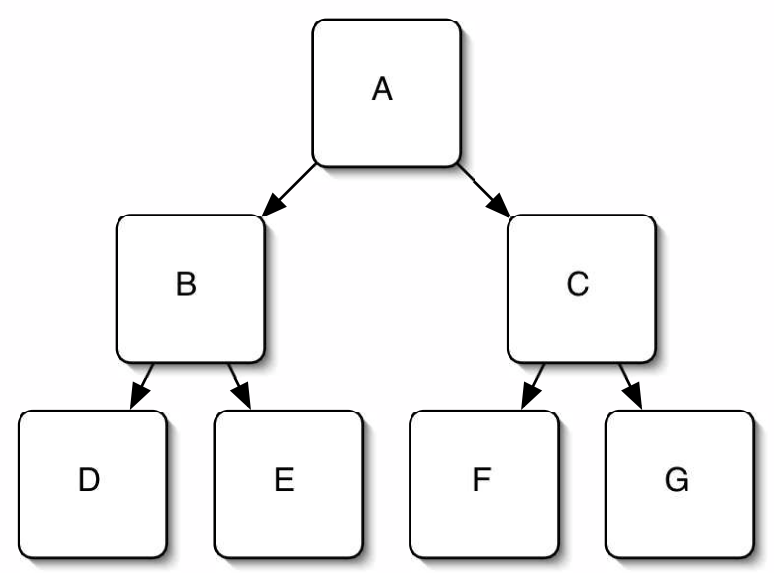

Figure 4.1: A sample linked structure where all node connections are also Friend connections or Usage connections.

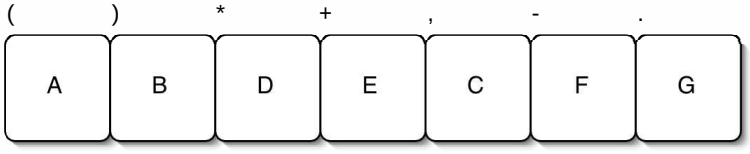

Figure 4.2: A sample linked structure laid out with Friends Depth-First or Usage Depth-First.

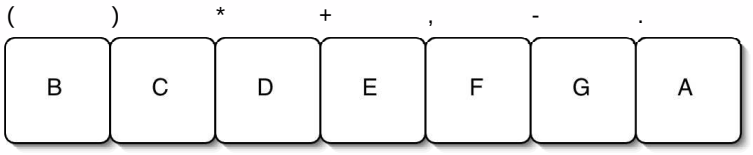

Figure 4.3: A sample linked structure laid out with Friends Grouped or Usage Grouped.

rate friends for his calls to `ccmalloc()`. It should be particularly effective in situations where the data is accessed via depth-first traversals. In this case, this layout may maximize prefetching.

4.5 Friends: Grouped

The grouped friends strategy is similar to the friends depth-first strategy; however, instead of placing requests as it follows the friends chain, the grouped strategy moves through the request list and searches for requests that have friends. When it encounters one, it places all the friends sequentially, without placing the request that they are all friends with. This strategy supposes that requests which all share a common root friend may actually be accessed contemporaneously. This resembles breadth-first search, but it is slightly different. Groups of friends are placed in the sequential order of their hub instead of using the queue system that breadth first search relies on. After all friend groups have been placed, the list is searched again and all unplaced requests are sequentially laid out. The success of this strategy greatly depends on the application's memory access patterns. See Figures 4.1 and 4.3 for an example.

4.6 Usage: Depth First

The memory access information generated during a logging run via the `ping()` function is the single most valuable form of information the profile synthesizer has at its disposal. It represents actual program behavior. This strategy builds a graph of request objects connected unidirectionally by actual knowledge of their contemporaneous access. The request that is accessed first gets a weighted edge pointing to the request that is accessed second. In cases where the same access patterns occur multiple times, the edge weights between requests are increased. The usage depth first strategy follows the same algorithm as the friends depth first strategy, except it uses this usage data instead of the friends information. See Figures 4.1 and 4.2 for an example.

4.7 Usage: Grouped

The grouped version of usage uses the same strategy as friends grouped. It searches for requests that have links to other requests used after them. It then places those requests as a group in the hope that they might be accessed in the same general part of the program, despite not being immediately contemporaneous. Placing these requests together helps to keep them from conflicting. See Figures 4.1 and 4.3 for an example.

4.8 One Per Line

This strategy is like Simple in that it places requests in the order of their occurrence. However, this strategy aligns each new request at the start of the cache line. This means that there are empty spaces in memory between requests. This effectively shortens line size. It eliminates prefetching, because there cannot be multiple objects per line. In addition to cache performance,

this also potentially increases the TLB miss rate because of address space dilution.

4.9 Smart

The Smart allocation strategy tries to predict (based on cache size, line size, and way) which addresses will conflict in memory. Given information about friendships between requests and probabilistic information after request lifetime, the Smart strategy tries to place each request so that it conflicts with the minimum possible number of friends. When faced with a potential conflict, this strategy will make sure the conflict occurs with a friend request least like to be used soon.

Unfortunately, due to its the excessive computational requirements, we were unable to benchmark the Smart algorithm; however, basic testing indicated that it performed too poorly to be considered.

HERE

4.10 Creating New Algorithms

Creation of an a new algorithm is a simple process; one merely registers a new function in the Python library of placement strategies. The function must accept a hash table of memory requests and assign locations to each request object in the table. The memory requests objects have a number of fields containing information about their size, friends, access weight, known lifetime, etc. Information is also available about the cache size, cache line size, and cache way. A programmer is free to use this information in any way seen fit in order assign addresses to the requests. This easy extensibility ensures that `libstaticmalloc` can be adapted to meet the needs of programs with unusual allocation patterns. The code for the Simple placement strategy is included below as a demonstration.

```
# The procedure accepts a hashtable of requests.
def place_simple (reqs):
    # Build a list of all the requests from the hashtable.
    req_list = reqs.values()
    # Sort the list by request time.
    req_list.sort (lambda x,y: x.index - y.index)

    # Location is an address pointing just beyond
    # the last allocated memory.
    location = 0

    # For each request in our sorted request list...
    for req in req_list:
        # we assign it an address...
        req.location = location
        # and increase our high water mark.
        location = location + req.size

    # Return the amount memory we used for the layout.
    return location
```

4.11 Summary

There are a variety of strategies that can be used with `libstaticmalloc`. These strategies are easy to describe and implement. Given the variety of potential layouts it seems possible that some might result in fewer cache misses than the standard sequential allocation strategy used in most system allocators.

Chapter 5

Results

In order to compare the effectiveness of `libstaticmalloc`, we put together a suite of test programs. These programs were then annotated to use the more sophisticated `ccmalloc()` API. This API conversion allowed us to benchmark the applications using all of our placement strategies. The original versions as well as these modified versions, were benchmarked on several applications. The versions that used `libstaticmalloc` were tested using a variety of memory layout algorithms.

Results showed that memory layout can dramatically effect program speed and cache miss rate. Certain strategies performed better than sequential layout while others performed worse. We saw performance increases of up to 28% for contrived test cases and up to 5% for certain real programs. We also witnessed programs that took 2.6 times as long to complete when using poor memory layouts. Building memory layouts ahead of time for the dynamic allocator using profiling information appears to be a successful strategy for achieving better cache performance.

5.1 Testcases and Data Sets

The first program, **cover**, was written in C to solve the cover problem using Don Knuth's[13] famous dancing links method. Problems are represented internally by a two-dimensional array that is built from circularly linked lists. Cover consisted of 479 lines of code and was annotated for `libstaticmalloc` in under 20 minutes.

Three randomly generated data sets were used to benchmark this program. Neither data set has a cover solution, but this is unimportant for benchmarking purposes because the program must still exhaust the search space. One data set had 100 rows and 100 columns. Another had 300 rows and 300 columns. The last had 500 rows and 500 columns. In all cases, the probability of a position being occupied was 1 in 5; however, this only affects the likelihood of a solution, not the run time of the program.

The second program, **hp**, was written by Chris Douglas to generate and make use of a game state space for a small chess-like game called Hexapawn devised by Martin Gardner[9]. It expands all possible board states for a given board in order to find the optimal moves for a player. The hp program was benchmarked on a board with 4 rows and 3 columns. The columns represent the number of pawns each player controls. Hexapawn consists of 322 lines of code and was annotated in around 20 minutes. Though the code for Hexapawn is relatively dense, annotating it was not difficult.

The last program, **traverse** was specially written as a testcase. It allocates a binary tree of the specified depth containing a size 20 integer array at each node. It then traverses the tree a specified number of times adding 1 to every integer in every array. The most important feature about its behavior is that it builds the tree depth-first in the left-most direction. However, when it traverses the tree, it does so in a depth-first, right-most fashion. Though obviously a contrived example, traverse illustrates the poor performance cause by an allocator/traversal mismatch. This program was primarily benchmarked on a 17 layer deep tree performing 10 traversals per

run. The traverse program was only 55 lines of code and was written to use the `ping()` and `ccmalloc()` functions.

5.2 Comparison of Layout Strategies

Each of the placement strategies was compared on identical test programs with identical test cases. The `libstaticmalloc` library automatically starts timing the application as soon as user code begins execution and prints the time spent as soon as user code finishes. This is accomplished by two functions that are called at the beginning and end of every program that uses `libstaticmalloc`. The times are reported by the Unix `gettimeofday()` function, which has microsecond accuracy. In situations where system allocator performance was measured, we used a new library called `libtiming` that contained only the timing parts of those two functions and no allocation code or functions. Each data set was run a fixed number of times. All tests were run at least 1,000 times, although the quicker ones were run more times for increased accuracy. The time for the fastest execution was kept in order to identify the best possible hardware performance. Modern computer systems are very complex and any number of situations can temporarily slow down program execution. In these situations, the times do not accurately represent the amount of CPU time required by the program. By taking the best time, we ensure that operating system behavior and transient affects do not unduly affect the results. The best times for each strategy are listed in Table 5.1 and the percent speedup compared to Simple for each strategy is available in Table 5.2. This data is a represented graphically in Figures 5.1, 5.2 and 5.3.

It is worth noting that we were unable to benchmark the Smart algorithm due to its the excessive computational requirements. However, basic testing indicated that it performed too poorly to be considered.

Algorithm	cover 100x100	cover 300x300	cover 500x500	hp 4x3	traverse 16, 10
Simple	0.002352	0.080595	0.926647	0.007496	0.111173
Random	0.002395	0.211720	1.949964	0.010427	0.213957
Size	0.002346	0.080677	0.926026	0.007491	0.110867
Friends DF	*0.002340*	0.080653	0.925796	0.007841	0.079430
Friends Grouped	0.002353	0.083250	0.933972	*0.007370*	0.114451
Usage DF	0.002361	*0.076603*	*0.923315*	0.007476	*0.079179*
Usage Grouped	0.002373	0.090737	1.047584	0.007497	0.111261
Oneperline	0.002442	0.151122	1.245283	0.009993	0.144253

Table 5.1: Best times in seconds for various test cases and data sets. (In all tables, overall best performance appears in italics.)

Rule of Thumb 4:

Do not try to be too clever.

All speed tests were performed on an 2.4 GHz Pentium 4 Xeon with 8KB of L1 cache, 512KB of L2 cache, and 2MB of L3 cache. This system ran FreeBSD 4.8 and had 1 GB of RAM.

The single most important fact about the data is that performance numbers varied. All test cases ran identical code. The only difference between the test cases was the memory layout used for the dynamic allocation requests. The variation in execution time suggests that memory layouts can actually affect data caching.

This hypothesis was investigated further using Valgrind's Cachegrind tool[1]. Cachegrind simulates program execution on an Intel Architecture 32 bit processor and uses a cache model to gather cache usage statistics. All Cachegrind simulations were performed with a simulated 16KB, 8-way, 32B per line, L1 instruction cache; a 8KB, 4-way, 64B per line, L1 data cache; and a 512KB, 8-way, 64B per line, combined L2 cache. This is a common cache

Alogrithm	cover 100x100	cover 300x300	cover 500x500	hp 4x3	traverse 16,10
Simple	1.0000	1.0000	1.0000	1.0000	1.0000
Random	1.0182	2.6270	2.1043	1.3910	1.9245
Size	0.9974	1.0010	0.9993	0.9993	0.9972
Friends DF	*0.9949*	1.0007	0.9991	1.0460	0.7145
Friends Grouped	1.0004	1.0329	1.0079	*0.9832*	1.0295
Usage DF	1.0038	*0.9505*	*0.9964*	0.9973	*0.7122*
Usage Grouped	1.0089	1.1258	1.1305	0.9973	1.0008
Oneperline	1.0383	1.8751	1.3439	1.3331	1.2976

Table 5.2: Time ratio compared to Simple for test cases and data sets.

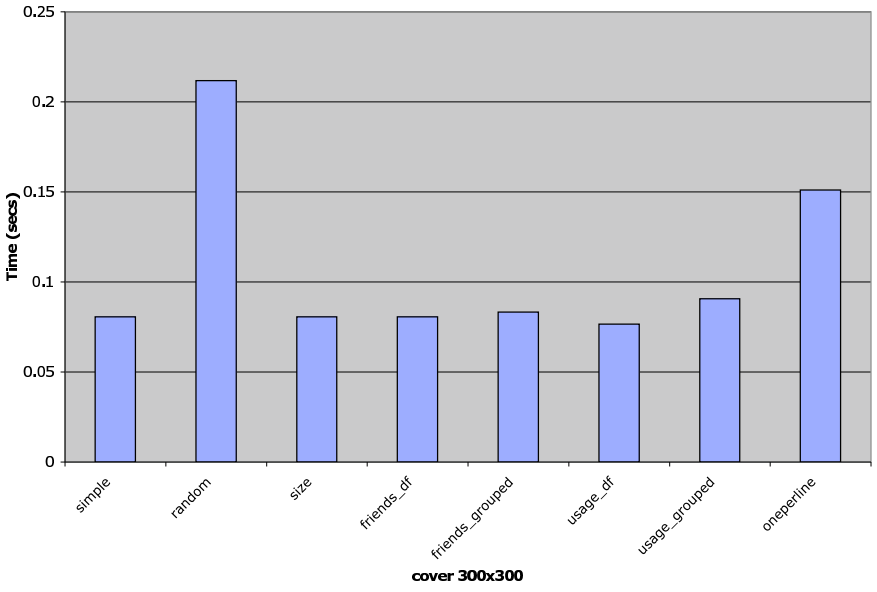

Figure 5.1: Best cover times per strategy on a 300x300 data set.

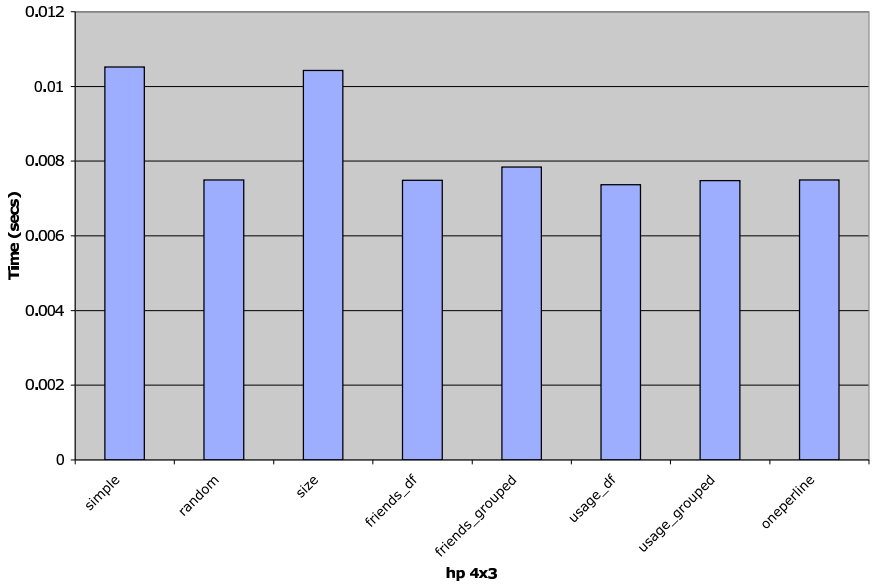

Figure 5.2: Best hp times per strategy on a 4x3 board.

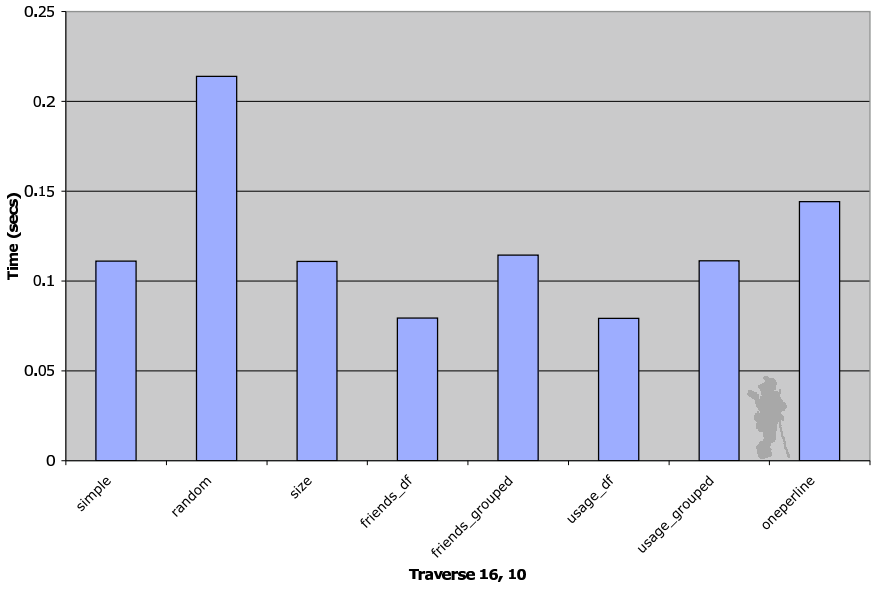

Figure 5.3: Best traverse times per strategy for a tree of depth 16 with 10 traversals.

Algorithm	Data Refs	L1 Data Misses	L1 Data Miss Rate	L2 Data Misses	L2 Data Miss Rate
Simple	28,937,189	5,814,499	20.00%	556,132	1.90%
Random	28,937,189	6,815,227	23.50%	1,125,087	3.80%
Size	28,937,187	5,811,861	20.00%	551,960	1.90%
Friends DF	28,937,187	*5,809,314*	20.00%	556,153	1.90%
Friends Grouped	28,937,187	5,827,200	20.10%	583,593	2.00%
Usage DF	28,937,187	5,809,386	20.00%	*535,461*	*1.80%*
Usage Grouped	28,937,187	6,083,590	21.00%	604,834	2.00%
Oneperline	28,937,189	6,470,792	22.30%	1,520,016	5.20%

Table 5.3: cover running on a 300x300 data set.

configuration for many Pentium IV machines. A real Pentium IV would have a 12K micro-op trace instruction cache. However, since `libstaticmalloc` is only concerned with data cache performance, this does not affect results. It is also worth noting that the number of total data references varies by one or two because of timing-dependent code. These numbers are not large enough to affect the results in any significant manner.

The cache performance figures in Tables 5.3, 5.5, and 5.7 corroborate the execution times referenced above. Relative comparisions to Simple are also available in Tables 5.4, 5.6, and 5.8. As we can see in Figures 5.4, 5.5, and 5.6 the number of L1 and L2 data cache misses vary depending on the placement strategy used to build the memory layout.

The differences in execution time combined with Valgrind's confirmation of fewer cache misses for certain strategies are strong evidence that it is possible to optimize programs through careful dynamic memory layout. Given this understanding, a variety of interesting features present themselves in the data.

The first feature is the difference in execution times and cache miss rate between the Simple and Random strategies. In many ways, Simple represents the standard approach to memory layout. The vast majority of allocations

Algorithm	L1 Data Miss Ratio	L2 Data Miss Ratio
Simple	1.0000	1.0000
Random	1.1721	2.0231
Size	0.9995	0.9925
Friends DF	0.9991	1.0000
Friends Grouped	1.0022	1.0494
Usage DF	0.9991	*0.9628*
Usage Grouped	1.0463	1.0876
Oneperline	1.1129	2.7332

Table 5.4: Cache miss ratio of Simple for cover running on 300x300 data set.

Algorithm	Data Refs	L1 Data Misses	L1 Data Miss Rate	L2 Data Misses	L2 Data Miss Rate
Simple	5,508,372	*24,330*	0.40%	23,383	0.40%
Random	5,508,372	55,144	1.00%	35,450	0.60%
Size	5,508,370	24,570	0.40%	23,376	0.40%
Friends DF	5,508,372	24,485	0.40%	23,380	0.40%
Friends Grouped	5,508,372	24,390	0.40%	*23,369*	0.40%
Usage DF	5,508,372	24,444	0.40%	23,377	0.40%
Usage Grouped	5,508,372	24,333	0.40%	23,384	0.40%
Oneperline	5,508,372	51,013	0.90%	49,086	0.80%

Table 5.5: hp running on 4x3 board.

Algorithm	L1 Data Miss Ratio	L2 Data Miss Ratio
Simple	1.0000	1.0000
Random	2.2665	1.5161
Size	1.0099	0.9997
Friends DF	1.0064	0.9999
Friends Grouped	1.0025	*0.9994*
Usage DF	1.0047	0.9997
Usage Grouped	1.0001	1.0000
Oneperline	2.0967	2.0992

Table 5.6: Cache miss ratio of Simple for hp running on 4x3 board.

Algorithm	Data Refs	L1 Data Misses	L1 Data Miss Rate	L2 Data Misses	L2 Data Miss Rate
Simple	28,455,701	994,636	3.40%	970,229	3.40%
Random	28,455,636	1,571,797	5.50%	1,480,751	5.20%
Size	28,455,703	994,636	3.40%	970,229	3.40%
Friends DF	28,455,638	987,289	3.40%	*970,224*	3.40%
Friends Grouped	28,455,703	1,002,154	3.50%	970,322	3.40%
Usage DF	28,455,703	*987,094*	3.40%	970,237	3.40%
Usage Grouped	28,455,638	997,315	3.50%	970,260	3.40%
Oneperline	28,455,703	1,408,294	4.90%	1,381,616	4.80%

Table 5.7: traverse running with a depth of 16 and performing 10 traversals.

Algorithm	L1 Data Miss Ratio	L2 Data Miss Ratio
Simple	1.0000	1.0000
Random	1.5803	1.5262
Size	1.0000	1.0000
Friends DF	0.9926	0.9999
Friends Grouped	1.0076	1.0001
Usage DF	*0.9924*	1.0000
Usage Grouped	1.0027	1.0000
Oneperline	1.4159	1.4240

Table 5.8: Cache miss ratio of Simple for traverse with a depth of 16 and performing 10 traversals.

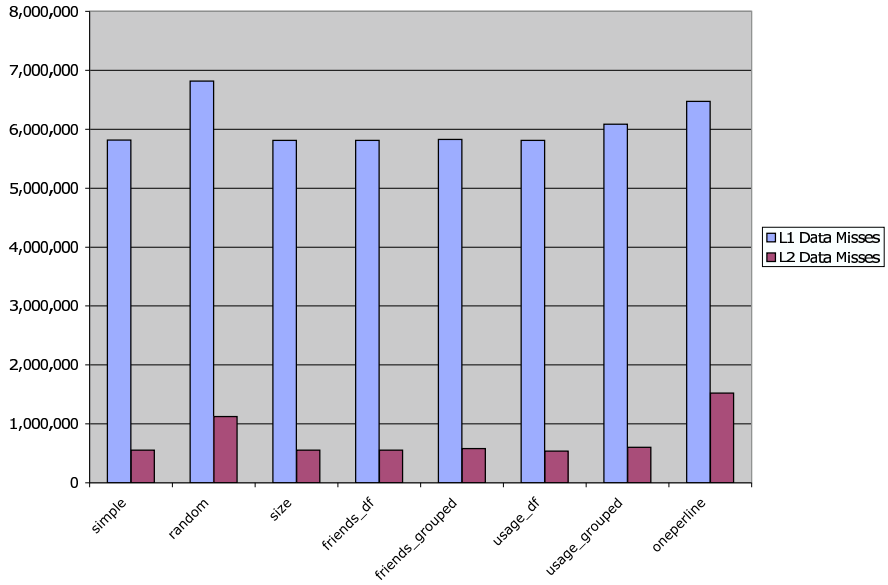

Figure 5.4: Cache misses for cover running on a 300x300 data set.

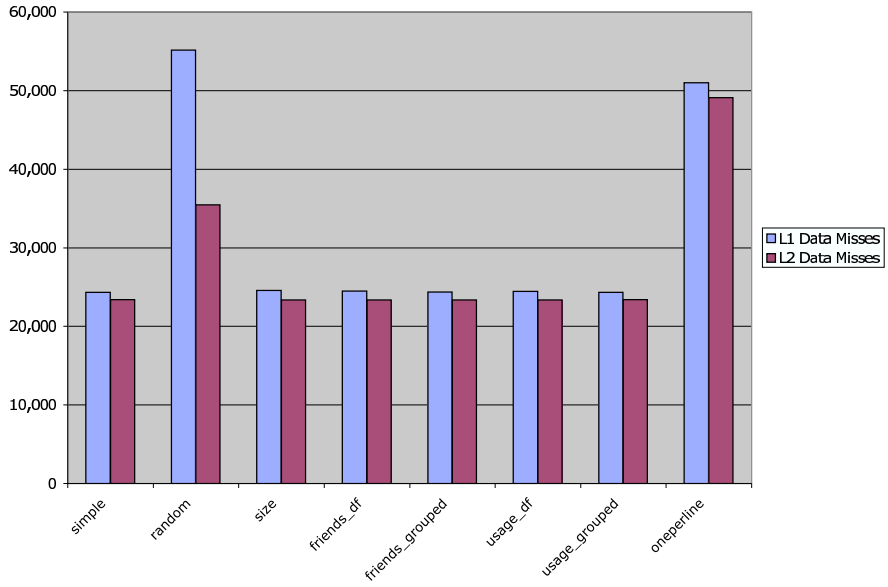

Figure 5.5: Cache misses for hp running on a 4x3 board.

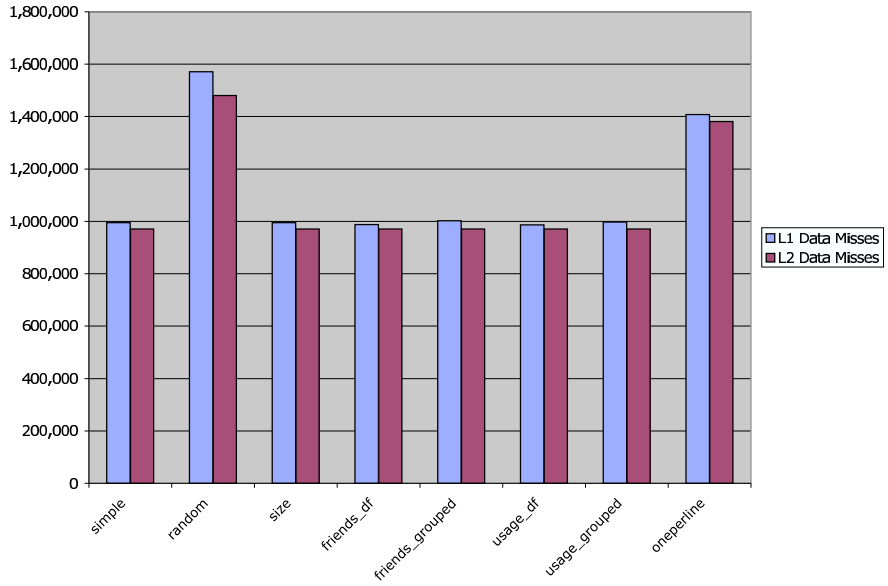

Figure 5.6: Cache misses for traverse running with a 16 depth and 10 traversals.

under traditional allocators are linear and sequential. The time of allocation is usually proportional to the memory address of the allocated memory. Reuse of freed memory complicates the scenario, but only slightly. On the other hand, Random represents a completely uninformed layout. Any information about usage encoded in the temporal sequence of the requests is destroyed.

The data show a striking difference between Simple and Random performance. Simple was 2.63 times faster than Random and suffered 15% fewer L1 data misses and 50.6% fewer L2 data misses for the Cover program and a 300 by 300 data set. Since all memory is preallocated by `libstaticmalloc` the only difference between these two executions was the memory layout used. This shows that the sequential layouts that most system allocators produce actually do a reasonable job preventing collisions and taking advantage of cache prefetching. This should not be entirely surprising. If caching and linear layouts combined poorly the question of allocation layout would have been more thoroughly explored previously. Since sequential layouts perform better than random layouts, it may be possible to find strategies that performs better than sequential.

Rule of Thumb 5:
Sequential layouts are not unreasonable for many applications.

Indeed, it seems likely that the best layout strategy will vary by program, and may even vary by data set. We explore this hypothesis below. However, first consider the execution and cache statistics for the Cover problem run over a 300 by 300 board in Tables 5.1 and 5.3.

In order from best performance to worst, the strategies were: Usage DF, Simple, Friends DF, Size, Friends Grouped, Usage Grouped, Oneperline, and Random. Of the strategies, only Usage DF was faster than Simple. However, it proved 5.2% faster with 3.7% fewer L2 data cache misses than Simple.

This corresponded to 5113 fewer L1 data misses and 20671 fewer L2 data misses than Simple. This shows that it is possible to create cache-conscious layouts that perform better than a sequential layout. The connection between allocation order and eventual use typically exists; however, profiling should always be able discover other, stronger connections by observing execution.

Rule of Thumb 6:
Cache-conscious allocation can speed up programs.

Both Random and Oneperline both performed significantly worse than Simple. Of the remaining strategies, we can see that Size and Friends DF came very close to Simple, trailing by less than 0.2% execution time. Remembering that sequential layouts do significantly better than random placement, the fact that these two strategies come close to the performance of sequential layouts suggests that the techniques they use are an effective means of optimizing for cache performance. That they fail to surpass Simple only means that the techniques are perhaps not sufficient on their own.

5.3 Libstaticmalloc vs. the System Allocator

Obviously, the choice of allocation strategy can have a significant impact on a program's speed. How then, does `libstaticmalloc` compare to the system allocator?

In the above data, `libstaticmalloc` performs extremely well. The `libstaticmalloc` Usage DF strategy is 95% faster than the FreeBSD 4.8 system allocator. However, this is a misleading way to look at the data. It is vital to note that the system allocator's strategy is very similar to `libstaticmalloc`'s Simple strategy. Any differences in execution times between these two is strictly the result of `libstaticmalloc`'s slimmer functions and reduced number of system calls because of preallocation. The data shows that there is indeed a major difference between the two. Just switching to `libstaticmalloc`

Allocator	Prealloc	Strategy	Time (secs)	Ratio to System
System	No	N/A	0.149683	1.0000
Libstaticmalloc	No	Simple	0.094855	0.6337
Libstaticmalloc	Yes	Simple	0.080595	0.5384
Libstaticmalloc	Yes	Usage DF	*0.076603*	*0.5118*

Table 5.9: Cover running on a 300 by 300 data set comparing libstaticmalloc with and without preallocation to the FreeBSD system allocator.

Allocator	Strategy	Data Refs	L1 Data Misses	L1 Data Miss Rate	L2 Data Misses	L2 Data Miss Rate
System	N/A	16,947,726	5,942,015	35.00%	739,109	4.30%
Libstaticmalloc	Simple	1,284,175	164,030	12.70%	2,767	0.20%
Libstaticmalloc	Usage DF	*1,284,177*	*163,406*	12.70%	2,767	0.20%

Table 5.10: Cover running on a 300 by 300 data set comparing libstaticmalloc to the GNU LibC system allocator.

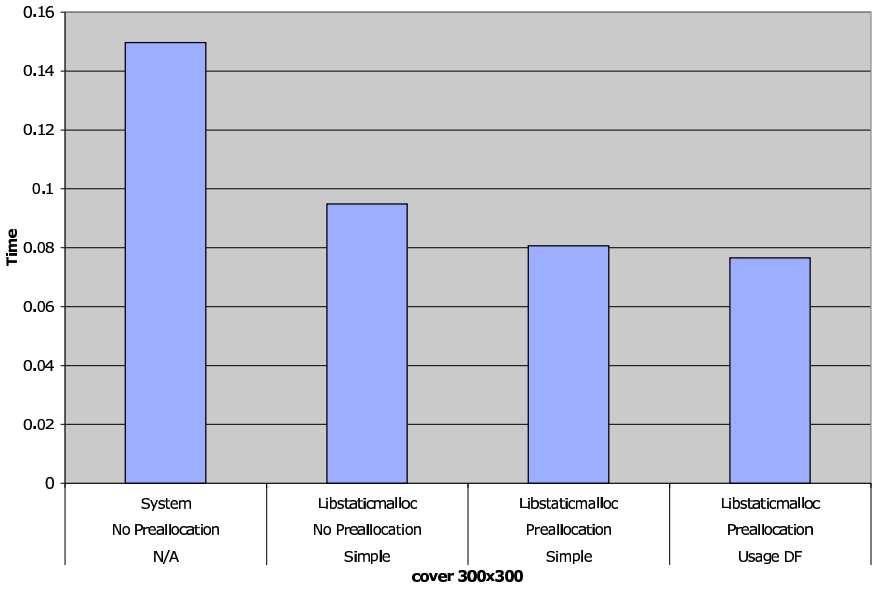

Figure 5.7: Cover running on a 300 by 300 data comparing the effects the System allocator, libstaticmalloc, and preallocation.

shaves 47% (0.069088 seconds) off cover's execution time on the 300 by 300 data set.

Rule of Thumb 7:
There are many good reasons to use the system allocator, but using it comes with a performance cost.

The majority of the speedup comes from the simpler execution process. Table 5.9 and Figure 5.7 shows that there is a 36.6% speedup just from switching to a non-preallocated version of `libstaticmalloc`. This version of `libstaticmalloc` only calls `sbrk()` to enlarge the data segment when a static address is returned that is past the current segment boundary. For a sequential allocation scheme like Simple, this results in an `sbrk()` call for every single allocation. This is the worst possible ratio of system calls

to memory requests. Most system allocators operate more intelligently and request memory in chunks, which they then subdivide to handle multiple allocation requests. However, the difference between the system allocator and `libstaticmalloc` with no preallocation represents the minimum speedup gained only from `libstaticmalloc`'s simplified internals.

There may also be an associated decrease in cache miss rate. The `libstaticmalloc` library does not need to store request size tags near the allocations like many system allocators do. This may eliminate cache pollution. However, it is difficult to measure this effect independently, so we will consider it together with the effects of `libstaticmalloc`'s simplified internals.

Additionally, the performance difference between `libstaticmalloc` with and without preallocation represents the speedup gained by preallocation. For the cover 300 by 300 data set, there is a .014260 second speedup which amounts to a 9.5% gain over the system allocator from just preallocation. Like all system calls, `brk()` and `sbrk()` come at a cost. The fewer times they are called, the faster an allocator will perform.

Rule of Thumb 8:
Avoiding unnecessary work is always advantageous.

These results are worth noting despite their peripheral relationship to cache-conscious allocation. Applications that have allocations requests routinely made at the beginning of execution may benefit from a dynamic allocation interface that can make use of this information. Using simplified allocation pathways and preallocation may speedup these allocations accordingly.

Since `libstaticmalloc` must take the time to map a memory profile into memory before each execution, it is important to consider how short-lived programs perform using `libstaticmalloc`. The chief worry is that the initial overhead might overwhelm any benefits accrued.

Analysis of Tables 5.11 and 5.12 shows that smaller problems do benefit less from `libstaticmalloc` because of amortization issues. However, both cover running on a 100 by 100 data set or hp running on a 4 by 3 board perform better than the system allocator. Cover running on a 100 by 100 board using Simple performs 7.1% percent faster than the system allocator. A slightly larger problem, hp using a 4 by 3 board and Simple performs 28.7% better than the system allocator. Cover on a 300 by 300 board benefits even more, speeding up by 46.1%. Clearly, `libstaticmalloc`'s initial overhead is amortized over total run time; however, even for very short running programs, `libstaticmalloc` runs faster.

Obviously, the system allocator is likely the preferred allocation library for most programs. Because `libstaticmalloc` does not reuse freed memory, it can be memory inefficient. A default system allocator needs to be able to reuse memory. In fact, memory reuse is the main source of complexity within modern allocation libraries. Additionally, the problem of generating profiles is unnecessarily complex for most applications. However, `libstaticmalloc` does serve computationally intensive programs with specific allocation patterns rather well. Of course, memory reuse would not be difficult to add to `libstaticmalloc`. Any allocation strategy is currently free to reuse memory, though none yet do. Combined with calls to the system allocator during the logging run, `libstaticmalloc` could avoid memory inefficiency.

Additionally, my experiments with `libstaticmalloc`'s performance suggest that perhaps additional interfaces should be added to C allocation libraries to optimize requests allocated through the dynamic memory system that are actually static.

Rule of Thumb 9:
Systems should provide alternative allocation functions.

Allocator	Strategy	cover 100x100	hp 4x3
System	N/A	0.002533	0.010524
Libstaticmalloc	Simple	0.002352	0.007496
Libstaticmalloc	Friends DF	*0.002340*	*0.007370*

Table 5.11: libstaticmalloc and the FreeBSD system allocator times in seconds for cover and hp on small data sets.

Allocator	Strategy	cover 100x100	hp 4x3
System	N/A	1.0000	1.0000
Libstaticmalloc	Simple	0.9285	0.7123
Libstaticmalloc	Friends DF	*0.9238*	*0.7003*

Table 5.12: Time ratios for libstaticmalloc compared to the FreeBSD system allocator for cover and hp for small data sets.

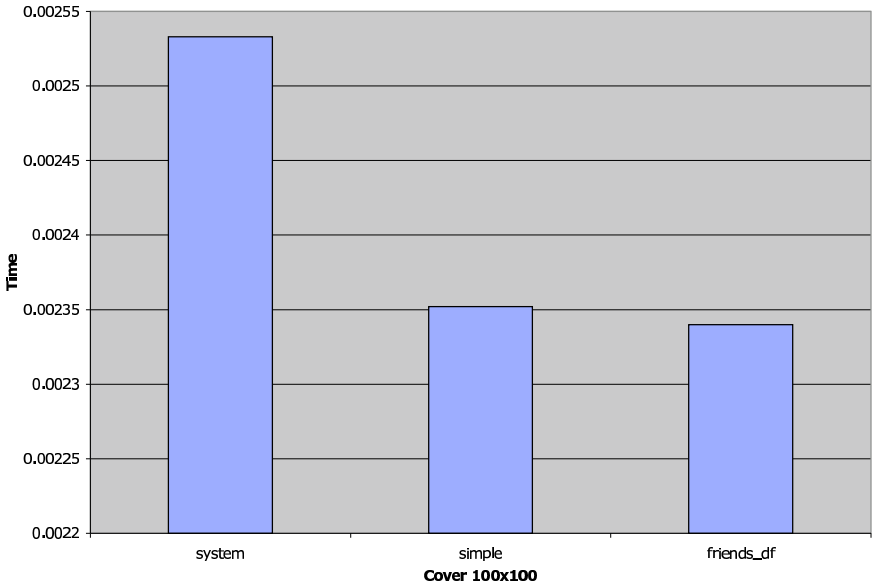

Figure 5.8: Cover running on a 100 by 100 data set comparing libstaticmalloc to the FreeBSD system allocator.

5.4 Effectiveness of Strategies Varies

The impact that the data set had in relation to `libstaticmalloc`'s performance versus the system allocator suggests that the relationship between the strategy and the data set deserves further attention. A brief review of strategy performance in 5.1 shows no single winning strategy.

For cover with a smaller 100 by 100 data set, the Friends DF strategy wins by a very small margin. The two larger cover data sets both perform best using Usage DF. Lastly, hp performs best using Friends Grouped.

The differences between Grouped and Depth-First should not be surprising. Different traversal patterns will have correspondingly different optimal layouts. In fact, this is the main reason that the effectiveness of different strategies depends on the program. There is interesting future work for programmers to code static allocator strategies that are custom-tailored to their applications.

However, the differences between Usage and Friends should be surprising. In general Usage layouts should perform better than Friends layouts. Friends represents the programmer's best understanding of the object relationships, while Usage represents the actual relationships (provided the annotation is correct). The short execution time of cover on a 100 by 100 data set probably explains this behavior. It seems possible that the limited linkage provided by friends information actually was easier to place than the more complicated usage information that did not contain any strong edges because of the short execution time.

In addition, it is theoretically possible for the data set to help determine the optimal placement strategy if the data set effects the memory relationships. Though this likely did not manifest in any of our benchmarks, it is an interesting avenue for future exploration.

All the strategies discussed in the this thesis operate by putting related objects near each other in memory. They do this to avoid conflicts and take advantage of prefetching. Using only layouts of this type, there does not

exist a single, optimal, layout strategy for all programs and data sets. The best strategy for any given program and data set will depend on the access pattern during execution. However, if a layout strategy fully understood the cache implications of every placement and access, it might be possible to find an algorithm for the absolute cache-optimal layout. However, given that even system architects have a difficult time predicting exact cache behavior, creating such a perfect strategy is likely impossible.

5.5 Slimming Down Libstaticmalloc

In order to support both logging mode and static mode from the same library, `libstaticmalloc` is sprinkled with conditionals that test the current state of the library. This is unnecessary overhead during a static allocation run. It is possible that using two separate libraries for logging and static allocation could dramatically accelerate `libstaticmalloc`. We built a second library, `libstaticmalloc-fast`, to investigate whether these conditionals were affecting the overall speed of the system. `libstaticmalloc-fast` was created by removing all logging or debugging features from `libstaticmalloc`. The result was a trimmer library that would run identically to `libstaticmalloc`, provided a profile had already been generated. The performance results for `libstaticmalloc` and `libstaticmalloc-fast` are available in Table 5.13 and Table 5.14.

The performance increase is noticeable, but minimal for cover on a 300 by 300 data set. It is quite significant for runs of hp on a 4 by 3 board, which is a shorter problem than cover. However, the actual time wasted by using `libstaticmalloc` for hp is even less. This suggests that the added complexity of supporting an additional library and forcing programs to link against a separate library after the logging run is not justified. However, were a library like `libstaticmalloc` ever to become a core part of a system, the dynamic linker could intelligently choose which library to use, thus eliminating the

Strategy	Libstaticmalloc	Libstaticmalloc-fast
Simple	0.080595	0.080332
Random	0.211720	0.210171
Size	0.080677	0.079961
Friends DF	0.080653	0.080265
Friends Grouped	0.083250	0.082766
Usage DF	0.076603	0.075938
Usage Grouped	0.090737	0.090269
Oneperline	0.151122	0.150881

Table 5.13: cover on 300x300 data set - times for libstaticmalloc and libstaticmalloc-fast in seconds.

Strategy	Libstaticmalloc	Libstaticmalloc-fast
Simple	0.007496	0.004799
Random	0.010427	0.008199
Size	0.007491	0.004875
Friends DF	0.007841	0.005279
Friends Grouped	0.007370	0.004826
Usage DF	0.007476	0.004694
Usage Grouped	0.007497	0.004698
Oneperline	0.009993	0.007800

Table 5.14: hp on a 4x3 board - times for libstaticmalloc and libstaticmalloc-fast in seconds.

recompile complexity while gleaning a small additional performance boost.

5.6 The Importance of Packing

Another important layout strategy is memory packing. Memory holes can slow programs significantly. This phenomenon manifested when benchmarking the Oneperline placement strategy. In order to ensure that each request begins on a new cache line, this strategy pads between allocations. This blank space comes with a price. Compare Oneperline's performance to Simple in Table 5.1. With padding as the only difference between Simple and Oneperline, Oneperline was 1.88 times slower. Additionally, Table 5.3 shows that Oneperline had 28383 more L1 Data Cache Misses, raising the program's total miss rate from 12.7% to 14.9%. In order to be successful, it seems that a memory layout strategy should pack memory.

Rule of Thumb 10:
Low object density leads to poor performance.

Chapter 6

Conclusions

In this thesis, we investigated the potential benefits of cache-conscious dynamic memory allocation. In the process, we made a number of important observations. However, there is room for further exploration.

6.1 Benefits of Profile-Based Allocation

Profile-based allocation as a general technique has advantages over traditional allocation in certain situations. Its slight start up overhead is easily paid back by the extremely slim allocation code path and the preallocation speedup. This technique is certainly not new. It is common for applications that perform the same actions on start up to cache the result. It might be advantageous for allocation libraries to allow programmers to inform the library about dynamic requests that are actually static. This not only allows the allocator to optimize its own performance for the those requests, but it also allows the far more exciting possibility of static allocation reordering.

6.2 Dynamic Memory and Cache

Certain programs exhibit allocation behavior such that it is possible to fully understand their memory usage patterns by profiling only the allocation phase of their life and a small portion of their computationally expensive main phase. Any program that builds a complex data structure and then repeatedly traverses, reads, and modifies the structure fits this pattern. For these programs, the static allocation techniques discussed above are actually feasible for the majority or entirety of the program's memory requests. In this situation, there exists the potential to arrange memory layouts ahead of time for optimal cache performance.

We have shown that memory layouts can have a real impact on total program performance. A variety of strategies exist to build these layouts, and the ones that used the most actual profile information (like Usage DF) seemed to perform the best. Friends DF, which uses programmer knowledge instead of runtime knowledge, performed reasonably well, but in most cases worse than Usage DF. This suggests that information available only at runtime may be extremely valuable to cache-conscious memory allocation systems. This is an advantage for profile based system.

Of course, it is worth noting that sequential allocation strategies already take some advantage of cache. Sequential layouts perform significantly better than random layouts. However, the added performance boost of static allocation, combined with the potential for cache-optimal layouts, may tempt programmers of certain classes of programs.

Additionally, it is worth comparing these results to Chilimbi's work[7]. Chilimbi et al. ultimately showed a much larger cache related performance boost, despite our system's extra profile knowledge. This could be attributed to a variety of factors, including more time spent on his system, better algorithms, better test cases, or better data sets. Additionally, it is unclear that the entirity of Chilimbi's performance gains are the result of increased cache performance.

Regardless, memory layouts can have a significant effect on overall program performance, and static memory allocation combined with profiling is an elegant way take advantage of this opportunity.

6.3 Opportunities for Further Work

Future work should explore the effects of new placement strategies. The success of the various layout strategies in this thesis is extremely promising. All the strategies tested were relatively simple. It seems likely that more advanced strategies might be able to achieve even greater speedups. Additionally, tailoring placement strategies to programs might allow even greater speeds. The ease with which strategies can be written opens the door for programmer exploration.

Another interesting possibility is connecting `libstaticmalloc` to the default system allocator. For example, we could use the system allocator as a fall-back allocator when requests diverge from the profiled order. Currently, `libstaticmalloc` falls back upon the strictly sequential logging allocator (with logging turned off for speed reasons). This is a simple solution, but once the application has diverged from its profile, there is no reason not to use the highly tuned system allocator instead. This would require linking into the system allocator's internal hooks.

One could also attempt to use the system allocator as the logging allocator. A simple wrapper around the internal functions of the system allocator could provide logging while still allowing reuse of freed memory and other features expected of modern allocators, even during the logging run. These two uses of the system allocator might come at minimal cost while tightly integrating `libstaticmalloc` with the operating system it is running on.

Another area for future work is to make profile generation automatic. A tool that could select an optimal profile simply by analyzing the log would very helpful for programmers. It would eliminate the need for programmers

to experiment with alternate placement algorithms as well as opening the path for a fully automatic profile generation system that automatically builds a profile on termination of a logging run. This could make `libstaticmalloc` even easier to use.

Chilimbi depends on user information and works at run time. We use profile information, but work offline. Perhaps an even more effective strategy could use profile information to synthesize allocators while simultaneously allowing for adjustment at run time. At the very least, it would be advantageous to have an allocator that did not revert to completely naïve behavior on departure from the profiled sequence of requests.

A significantly more advanced version of a profile-based allocator synthesizer might generate trend-based allocators. These allocators could respond to more than just a sequence of requests and would handle deviations from request order while still understanding enough about the program's runtime behavior to optimize cache performance.

Indeed, the ultimate static allocator would gather information over many consecutive runs. During periods of low system load, the analyzer/synthesizer would run and use techniques to tune a flexible allocator profile.

Although `libstaticmalloc` was written for C, the technique it uses is applicable to languages with implicit allocation. Indeed, many interpreted languages use custom-built allocators already. These interpreters could be easily modified to use cache-conscious static allocation techniques. In addition, languages that lack explicit pointers are not constrained to use the one placement per execution model. This allows them to take advantage of situations where ideal memory layout varies over program time.

The potential for cache-conscious memory layouts in managed runtimes is almost limitless. Because of garbage collection, these systems already have the capability to rearrange objects in memory. Since collection is typically performed by a copying garbage collector, the runtime has the potential to rearrange memory with every collection. Building a cache-conscious memory

layout at this point is identical to how we build layouts profiles offline in our system. Combined with the potential to gather memory usage information from within the managed runtime, these environments are a good match for the techniques described in this thesis.

Such systems are still far away. However, in the meantime, the knowledge that there is performance to be gained using cache-conscious allocation techniques is enough.

Appendix A

Rules of Thumb

1. Object-oriented programming degrades cache performance.
2. A good tool has a low barrier to entry.
3. More information should correspond to better performance.
4. Do not try to be too clever.
5. Sequential layouts are not unreasonable for many applications.
6. Cache-conscious allocation can speed up programs.
7. There are many good reasons to use the system allocator, but using it comes with a performance cost.
8. Avoiding unnecessary work is always advantageous.
9. Systems should provide alternative allocation functions.
10. Low object density leads to poor performance.

Appendix B

The Price of Tea

To help place this work in time and space, we provide the following economic, technical, and physical indicators.

My CPU Speed:	1.5GHz
My Weight:	156 lbs.
Linux Kernel Version:	2.6.6
Number of States in the Union:	50
American National Debt:	$7,193,909,860,082.37
Price of Gasoline:	$1.95 per Gallon
25 Earl Grey Tea teabags:	$3.30
Williams Tuition, Room, and Board (one year):	$38,100

Bibliography

[1] Valgrind website. http://valgrind.kde.org/.

[2] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 187–196, 1993.

[3] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 114–124, 2001.

[4] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002*, Seattle, Washington, November 2002.

[5] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.

[6] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, 1994.

[7] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[8] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *IEEE Computer*, pages 67–74, 2000.

[9] Martin Gardner. Mathematical games. *Scientific American*, (227):176–182, 1972.

[10] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith. Cache performance of the SPEC benchmark suite. Technical Report CS-TR-1991-1049, 1991.

[11] Dirk Grunwald and Benjamin G. Zorn. Customalloc: Efficient synthesized memory allocators. *Software - Practice and Experience*, 23(8):851–869, 1993.

[12] Dirk Grunwald, Benjamin G. Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–186, 1993.

[13] Donald E. Knuth. Dancing links. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millenial Perspectives in Computer Science*, pages 187–214. Palgrave, Houndmills, Basingstoke, Hampshire, 2000.

[14] Doug Lea. Lea allocator, 2002. ftp://g.oswego.edu/pub/misc/malloc.c.

[15] Preeti Ranjan Panda, Luc Sria, and Giovanni De Micheli. Cache-efficient memory layout of aggregate data structures. In *ISSS*, pages 101–106, 2001.

[16] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. *ACM SIGPLAN Notices*, 33(11):115–126, 1998.

[17] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.