Cache Index-Aware Memory Allocation

Yehuda Afek
School of Computer Science, Tel Aviv University
afek@post.tau.ac.il

Dave Dice
SunLabs at Oracle
dave.dice@oracle.com

Adam Morrison
School of Computer Science, Tel Aviv University
adamx@post.tau.ac.il

Abstract

Poor placement of data blocks in memory may negatively impact application performance because of an increase in the cache *conflict miss* rate [18]. For dynamically allocated structures this placement is typically determined by the memory allocator. Cache *index-oblivious* allocators may inadvertently place blocks on a restricted fraction of the available cache indexes, artificially and needlessly increasing the conflict miss rate. While some allocators are less vulnerable to this phenomena, no general-purpose `malloc` allocator is index-aware and methodologically addresses this concern. We demonstrate that many existing state-of-the-art allocators are index-oblivious, admitting performance pathologies for certain block sizes. We show that a simple adjustment within the allocator to control the spacing of blocks can provide better index coverage, which in turn reduces the superfluous conflict miss rate in various applications, improving performance with no observed negative consequences. The result is an *index-aware* allocator. Our technique is general and can easily be applied to most memory allocators and to various processor architectures.

Furthermore, we can reduce inter-thread and inter-process conflict misses for processors where threads concurrently share the level-1 cache such as the Sun UltraSPARC-T2™and Intel "Nehalem" by coloring the placement of blocks so that allocations for different threads and processes start on different cache indexes.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Allocation/Deallocation Strategies

General Terms Performance,experiments,algorithms

Keywords Dynamic storage allocators, memory allocation, malloc, caches, shared caches, conflict misses, placement policies

1. Introduction

Modern `malloc` memory allocator designs tend to focus first on the performance of the allocator itself, often ignoring the performance of the application code that accesses blocks returned by the allocator. The design and policies of the allocator can, for instance, have a significant influence on the data TLB (translation look-aside buffer - a cache of virtual to physical page translations) and data cache miss rates of applications accessing blocks returned from that allocator. We identify key aspects of allocator performance as follows:

- **Latency** and **Scalability** of the allocator itself
- **Memory footprint** – space-efficiency
  - **Peak memory usage** - capacity and consumption of system resources
  - **Data Cache and data TLB locality and span** - reflecting the density of the set of allocated blocks as measured by the number of pages and cache lines underlying those blocks. This measure also includes wastage, fragmentation and the overheads imposed by the allocator such as block headers (metadata - if present), `malloc` size quantization and heap layout.
- Inter-block **false sharing** in concurrent environments
- **Cache line relative block address alignment** – the placement of blocks by the allocator with respect to cache line boundaries
- **Cache index placement**

Our paper focuses on the final aspect – cache index placement. In particular our concern is how blocks returned by `malloc` are distributed over the set of possible cache indices. If the distribution is imbalanced or non-uniform then repetitive access to those blocks by the application might incur excessive conflict misses, which in turn may degrade overall system performance.

As a concrete illustration of the problem consider a `malloc` allocator that maintains arrays of 128-byte blocks – inclusive of both header data (allocator metadata, if any) and the data area – that may be used to satisfy `malloc` requests of sizes suitably close to 128 bytes. These arrays are private to the allocator implementation and opaque to applications. As blocks are packed densely in the array we find blocks starting every 128 bytes. If the base of the array is B then the block addresses will be B, $B+128$, $B+256$, and so on. Consider a data cache with 16-byte blocks and 128 indices. Crucially, the set of level-1 data cache (L1D) indices associated with the base addresses of the blocks within such arrays is restricted to just 16 instead of the full complement of 128. That is, given the way virtual addresses map to cache indices and because of the regular consecutive spacing of the blocks in the array, blocks of that particular *size-class* (128 bytes) can start on only 16 of the 128 possible indices. If an application allocates a set of such blocks and then repeatedly accesses just a few fields in a group of blocks then it may suffer excessive conflict misses as some cache indices are "hot" and others underutilized and "cold". Conflict misses, which arise from lack of cache associativity, cause the application accessing blocks returned by interfaces such as `malloc` to underutilize the data cache, robbing the application of potential performance.

One aspect of our solution is to insert small *spacer* regions into the array of blocks to better distribute the block indices, forming a *punctuated array* and disrupting the regular ordering of block

ISMM'11, June 4–5, 2011, San Jose, California, USA.

addresses. We show that this approach is both effective and simple to implement.

While we describe our techniques in terms of the implementation of a specific `malloc` allocator, it is general and can easily be applied in other environments such as pool allocators [24] or to the object allocators found in managed run-time environments with automatic garbage collection. Furthermore, while we explain our technique in terms of the Sun UltraSPARC-T2™processor, it carries to other architectures as well.

This paper starts with a discussion of modern `malloc` allocator design and show how such allocators can easily cause cache index imbalance – a poor distribution of blocks over the set of possible cache indexes. We then proceed to describe a simple solution that involves inserting *spacers* into the block arrays to provide a better distribution, yielding an index-aware allocator. We provide experimental data to support our claim. Next, we describe other varieties of index conflicts and provide insight on how they can also be easily avoided, followed by a survey of related work and conclude with a discussion of future research directions related to our topic.

2. Modern malloc allocator design

The default Solaris™`libc` allocator uses a single global heap protected by one mutex. Memory is allocated from the operating system by means of the `sbrk` system call. The global free list is organized as a splay tree [33] ordered by size and allocation requests are serviced via a best-fit policy. The heap is augmented by a small set of segregated free lists of bounded capacity, allowing many common requests to operate in constant-time. This results in an allocator with excellent heap density, reasonable single-threaded latency, but poor scalability. Furthermore, applications using the `libc` allocator may be subject to excessive allocator-induced false sharing, where blocks allocated to different threads happen to abut in the midst of a cache line.

Modern state-of-the-art allocators include `Hoard` [8], `CLFMalloc` [27], `LFMalloc` [14], `libumem` [9], `jemalloc` [16] and `tcmalloc` [6]. They are broadly categorized as *segregated free-list* [19] allocators as they maintain distinct free lists based on block size. Such allocators round requested allocation sizes up to the nearest *size-class* where a size-class is simply an interval of block sizes and without ambiguity we can refer to a size-class by its upper bound. The set of size-classes forms a partition on the set of possible allocation sizes. The choice of size-classes is largely arbitrary and defined at the whim of the implementor, although a step size of 1.2x between adjacent size-classes is common [8] as the worst-case internal fragmentation is constrained to 20%.

We will use `Hoard` as a representative example of modern allocator design. `Hoard` uses multiple heaps to reduce contention. Specifically, `Hoard` attempts to diffuse contention and improve scalability by satisfying potentially concurrent `malloc` requests from multiple local heaps – this strategy also mitigates the allocated-induced false sharing problem. Each heap consists of an array of references to *superblocks*, with one slot for each possible size-class. A superblock is simply an array of blocks of a certain size class. Superblocks are all the same size, a multiple of the system page size, and are allocated from the system via the `mmap` interface which allocates virtual address pages and associates physical pages to those addresses. `Mmap` is used instead of the more traditional `sbrk` operator as pages allocated through `mmap` may later be returned to the system, if desired, through `munmap`. The superblock is the fundamental unit of allocator for `Hoard`. Each superblock has a local singly-linked free list threaded through the free blocks and maintained in LIFO order to promote TLB and data cache locality. A small superblock header at the base of the array contains the head of the superblock-local free list. Superblocks and heaps are opaque to the application that uses the allocator. The `Hoard` implementation places superblocks on highly aligned addresses. The `free` operator then uses address arithmetic – simple masking – on the block address to locate the header of the enclosing superblock, which in turn allows the operator to quickly push the block onto the superblock's free list. As such, in-use blocks do not require a header field. If a superblock becomes depleted it can be detached from a heap and moved to a global heap. The local heap can be reprovisioned from either the global heap, assuming a superblock with sufficient free space is available, or by allocating a new superblock from the system. Superblocks can circulate between various local heaps and the global heap, but will be associated with at most one local heap at any one time. Allocator metadata is minimal, consisting of the heap structures and superblock headers. The implementation associates a `malloc` request with a heap by hashing the identity of the current thread. To reduce collisions `Hoard` overprovisions the number of heaps to be twice the number of processors. Concurrency control is provided by per-heap locks.

`Hoard's malloc` operator first quantizes the requested size to an appropriate size-class, identifies a heap, locks the heap, locates a superblock of the appropriate size-class in that heap, unlinks a block from that superblock's free list, unlocks the heap, and finally returns the address of the block's data area. As `Hoard` employs segregated free lists (segregated by size), in the common case finding a free block of a given size is a simple constant-time operation. Given this allocation policy the returned addresses for a given size-class may be regular in a manner that results in *inter-block cache index conflicts* and excessive conflict misses if a group of blocks of a size-class are accessed frequently by the application. More generally, array-based superblock allocators coupled with inopportune index-oblivious block sizes can easily result in patterns of block addresses that map to only a few cache indices.

Superblock-based allocators of this design allow for good scaling although their footprint is often somewhat larger than that of `libc` as they attempt to diffuse contention by distributing requests over multiple heaps. Latency varies but usually reflects path length through `malloc` and `free` and metadata access costs, which are properties of the implementation and not fundamental to the category of segregated free list allocators.

`CLFMalloc` is structurally similar to `Hoard`, differing mostly in the policy by which it associates `malloc` requests with heap instances and in that `CLFMalloc` is lock-free.

`Libumem` and `tcmalloc` use a central heap but diffuse contention via multiple local free lists. In the case of `tcmalloc` the central heap uses segregated free lists which are populated by allocating runs of pages and then splitting those pages into contiguous arrays of the desired size-class.

3. CIF : Improving index distribution – Punctuated arrays

We introduce a new index-aware segregated-free list allocator, `CIF` (Cache-Index Friendly), which was derived from `LFMalloc`. `CIF` and `LFMalloc` are structurally similar to `Hoard`. `LFMalloc` used hardware transactional memory [15] or restartable critical sections for concurrency control but for the sake of portability `CIF` uses simple mutual exclusion locks. `CIF` is easily portable and currently runs on Solaris SPARC and Linux x86 (32-bit and 64-bit).

In `CIF` each processor is associated with a processor-private heap. A superblock consists of a coloring region (described below), a header containing metadata and an array of blocks of the given size-class. As in `Hoard` the superblock header contains a pointer to the head of a LIFO free list of available blocks within that superblock. All blocks in a superblock are of the same length. Superblocks are 64KB in length.

CIF does not explicitly request *large pages* for superblocks. Large pages, if supported by the processor and operating system, can improve performance by decreasing TLB miss rates. Solaris attempts to provision mappings with large pages as a best-effort optimization. On SPARC large pages must be physically contiguous and both physically and virtually aligned to the large page size. The UltraSPARC-T2 supports 8KB, 64KB, 4MB and 256MB pages.

Concurrency control in CIF is implemented by heap-specific locks. Contention is rare and arises only by way of preemption. The impact of contention can be reduced by using techniques such as the Solaris schedctl mechanism [1] to advise the scheduler to defer involuntary preemption by time slicing for threads holding the heap lock.

Threads use the schedctl facility to efficiently identify the processor on which the thread is running, thus enabling the use of processor-specific heaps. On Linux/x86 CIF can be configured to use the CPUID or RDTSCP instructions to select a heap.

In CIF threads instantiate superblocks via mmap. On CC-NUMA systems that use a "first touch" page placement policy this means that the pages in a superblock will tend to be local to the node where the thread is running, improving performance.

All the allocators except Hoard, tcmalloc and jemalloc require at least a word-size metadata header field for in-use blocks. In CIF, for instance, an in-use block consists of a header word – a pointer to the enclosing superblock – followed by the data area. Malloc returns the address of the data area, which by convention must aligned on at least an 8-byte address boundary. The free operator consults this header to locate the free list in the superblock's header. CIF places the header word on the last word of the cache line preceding the address returned by malloc so the address returned by malloc is always aligned on 16-byte boundaries.

In CIF the size-classes inclusive of the header are simple powers-of-two starting at 16 bytes. We intentionally selected powers-of-two for the purposes of comparison against other allocators, whereas a production-quality allocator would use finer-grained size-classes.

To avoid undesirable index distributions and reduce the rate of inter-block cache conflicts the CIF allocator inserts a cache line-sized and aligned *spacer* into the superblock array when indices start to repeat, yielding a *punctuated array*. This allows the allocator to retain its existing size-classes. Say we have a superblock with 768-byte blocks and a sequence of blocks within that superblock that fall on addresses B, $B + 768$, $B + 1536$, $B + 2304$, $B + 3072$, $B + 3840$, $B + 4608$, $B + 5376$, $B + 6144$, etc. The UltraSPARC-T2 has 16-byte lines, 128 possible indices, and a 2048-byte cache page size. (Refer to Appendix A for details on the UltraSPARC-T2 cache organization). Our blocks would fall on indices I, $I + 48$, $I + 96$, $I + 16$, $I + 64$, $I + 112$, $I + 32$, $I + 80$ and I, respectively, where I is the cache index associated with block address B. If block address B falls on index I then the N-th block beyond B falls on address $B + (768 * N)$ having index $I + (((768 * N)/16) mod 128)$. In our example the indices repeat after just 8 blocks or 6144 bytes as the least common multiple of 2048 (the cache page size) and 768 (the block size) is 6144 bytes. If an implementation inserts a spacer after every 8 blocks, however, then a punctuated array of such blocks will land on the full set of 128 indices. A more naive implementation could simply insert a spacer after each contiguous run of blocks totaling at least 2048 bytes. The implementation in CIF uses this latter policy. In the worst case punctuated arrays require just one cache line of spacer per cache page within the superblock, putting a tight bound on wastage. Furthermore, the spacer lines are never accessed, so while they might increase TLB pressure and physical RAM usage, they do not influence L1D pressure. Finally, we note that we only need to employ spacers in superblocks that have index-unfriendly size-classes, where a simple unpunctuated array of blocks would otherwise land on only a subset of the possible indices.

As an alternative to the punctuated array, changing the set of size-classes to be index-aware can also provide relief by ensuring that the block addresses within a superblock array fall on the full complement of indices. We discuss this approach in more detail in Appendix B.

To derive benefit from an index-aware allocator we presume an access model where multiple instances of a structure type are accessed repetitively and frequently, some fields in the type are "hot" (accessed frequently relative to other fields) and those hot fields tend to be clustered. Furthermore each instance is allocated separately. That is, we assume temporal locality for blocks and temporal and spatial locality within individual blocks. Such an access pattern is not atypical. Bonwick et al. [9] calls out the kernel inode construct as an example. The pattern is common in object graphs with intrusive linkage where the linkage fields reside in a "header" that precedes the body of the object.

CIF can also be configured by means of an environment variable to use a simple "flat" array of blocks with no spacers. We refer to this form as CIU – Cache-Index Unfriendly. This form yields extremely poor cache index distribution similar to that which would be achieved with a binary buddy allocator [23]. It serves as a useful measure of cache index sensitivity.

4. Index placement survey

Using a simple program we show that a number of popular allocators are index-oblivious and that index-oblivious block placement can result in performance pathologies.

In Figure 1 each point in the graph represents a distinct run of a simple single-threaded benchmark program **mcache** that mallocs 256 blocks of size B byte. The program then reports the cache index of the base address for each of the blocks. The index can be computed with simple address arithmetic. On the X-axis we vary the block size B with a step of 16 bytes. The Y-axis values are the number of distinct UltraSPARC-T2 L1D indices on which those blocks were placed, reflecting cache index distribution. A value of 128 – the number of L1D indices – is ideal. (See Appendix A). Each UltraSPARC-T2 core has a 128-way fully associative data TLB and thus more than sufficient capacity to cover 1024 blocks of 256 bytes for a reasonable heap layout without incurring TLB misses. Other more descriptive statistics might better reflect index distribution, such as a histogram, standard deviation or spread between maximum and minimum of the index population, but a simple count of the number of distinct indices serves to illustrate our assertion that many allocators have non-uniform index distribution.

The various allocators were configured by way of the LD_PRELOAD dynamic linking facility. Data was collected under actual execution, not simulation. Libumem and libc are provided with Solaris. Hoard version 3.8 was obtained from [2] and CLFMalloc version 0.5.3 was obtained from [3]. We used jemalloc version 2.0.1 and tcmalloc version 1.6 in the tcmalloc-minimal configuration without call-site profiling. Where SPARC executables were not available, source code was compiled with gcc version 4.4.1 at optimization level -O3. Unless otherwise noted all data in this paper was collected with 32-bit programs under the Solaris 10 operating system on a UltraSPARC-T2 processor model T5120 which has 8 cores and which exposes 64 logical processors.

As can be seen in Figure 1, all of the allocators except CIF have one or more size values where blocks fall on only a fraction of the 128 possible indices, potentially limiting the performance of an application that repeatedly accesses a few "hot" fields (fields exhibiting strong temporal locality) in a set of such blocks. CIF gives an ideal uniform index distribution over all sizes.

The same experiment on a Linux/x64 Nehalem system revealed index imbalance under the default `libc` allocator, itself based on Lea's `dlmalloc` [5], although the situation was not as dire for mid-sized blocks as the cache has higher associativity. (The Nehalem processor has an L1D with 64 indices, 8 ways, and 64-byte lines. The cache page is 4KB so page coloring is not possible).

In Figure 2 we configure `mcache` so that the 256 blocks are configured in a ring by ascending virtual address. The first field in the block contains a pointer to the next block in the ring. The remainder of the block is not accessed during the run. Our program runs for 10 seconds, traversing the ring and then reports the number of steps per millisecond on the Y-axis. Again, we vary the block size on the X-axis. The only activity during the measurement interval is "pointer chasing" over the ring of allocated nodes. As can be seen, block placement greatly impacts performance. Note that we selected 256 blocks intentionally, as the L1D can contain 512 distinct lines and, ideally, with uniform index distribution, could accommodate all 256 blocks in cache without incurring any cache misses. As expected, when we collect CPU performance counter data when running `mcache` under the various allocators we see that the L1D miss rate correlates strongly with the performance reported by the application, supporting our claim that the slow-down, when present, arises from cache misses.

5. Conflict varieties and remediation

This section provides a partial taxonomy of index conflict varieties and enumerates various ways to lessen the rate of such conflicts.

Simple **inter-block** index conflicts, described above, may be inter-superblock or intra-superblock. We can address and often reduce the degree of intra-superblock conflicts by choosing index-aware size-classes or insertion of spacers but note that such approaches also provide benefit against inter-superblock conflicts simply by making index access more uniform and diluting hot spots.

Inter-thread conflicts arise with the advent of shared level-1 caches. Assume for instance that threads *T1* and *T2* run concurrently on the same *core* and share the L1D. Both threads `malloc(100)` immediately after they start. Each thread will typically access distinct CPU-private heaps and within those heaps, superblock instances of the size-class appropriate for 100 bytes. The superblocks will be instantiated via `mmap` which returns addresses that will be at least page-aligned and in practice often have much higher alignment. Thus, if the allocator creates the superblock at the address returned from `mmap` it is very likely that the blocks returned from the `malloc` requests by *T1* and *T2* will collide at the same cache indices. We have *intra-core*, inter-thread, inter-superblock, inter-heap index conflicts. One way to reduce the odds of such inter-thread conflicts is to insert a randomly sized variable length *coloring* area at the start of each superblock. We initially placed the superblock header on the address returned by `mmap` and then inserted the coloring region after the superblock header and before the array of blocks, but noticed that the superblock header itself was vulnerable to index conflicts. We ultimately placed the coloring area before the header, providing better index distribution for the cache lines underlying the superblock headers.

With shared level-1 caches, applications can also encounter **inter-process** index conflicts where different processes have threads running concurrently on the same core. One way to mitigate such conflicts is to seed the pseudo-random number generator – used to generate superblock colorings – differently for each process, perhaps based on the time-of-day, process-ID, or a system random number generator. Absent per-process seeding of the random number generator used for superblock coloring, allocations in similar but distinct processes may fall on precisely the same virtual addresses, increasing the likelihood of inter-process conflicts. This effect can be easily demonstrated by spawning a number of concurrently executing single-threaded processes, each of which iterates over a small ring of `malloc-ed` blocks. Without seeding we can find destructive interference in the L1D and degraded performance. All of the allocators except `CIF` exhibited this problem.

We note that Solaris randomly colors the offset of the stack for a process's primordial thread, in part to lessen the odds of inter-process conflict between stacks. Similarly, the HotSpot Java™ Virtual Machine explicitly colors the stacks for threads created by the JVM.

The `CIF` and `CIU` allocators employ random superblock coloring – 16 possible colors in the interval [0,15] with the length of the coloring region taken as the color times the L1D line length - – and process-specific seeding of the random number generator. Ideally an implementation would provide one color for each of the 128 possible indices. Recall, however, that the coloring region is never allocated from and never accessed. It exists solely to control the offset of the array of blocks. As a practical concern to bound wastage from the coloring area we restrict ourselves to just 16 colors.

6. Experimental results

We first establish the existence of index-sensitive applications, and then show the efficacy of index-aware allocation on that set of benchmarks. Next, we show the benefits of coloring on system where multiple threads concurrently share caches. Finally, we report on the scalability of various allocators.

While not shown for lack of space, we have tested various allocators on a wide set of pointer-intensive benchmarks and found the index-aware size-classes or punctuated arrays do no harm and that no particular trade offs are required. `CIF` is competitive with the current best-of-breed allocators.

We ran each benchmark 5 times and took the median result, observing extremely low variation between runs.

6.1 Index-sensitive applications

In Figure 3 we report results from a set of single-threaded cache index-sensitive applications. Formally, cache index sensitivity is an aspect of application performance determined both by application structure and allocator design choices. We define an application as index sensitive *under an allocator* if it suffers excessive conflict misses because of poor index distribution. Index distribution, in turn, is largely determined by the allocator policies and design choices. These applications are sensitive because of the block sizes requested and access patterns to those blocks. Excluding `CIF`, no one allocator is best over all the applications as each exhibits different pathological index-unfriendly sizes as was previously seen in Figure 1.

We intentionally selected the applications below as (a) they were insensitive to `malloc-free` performance with such operations typically confined to a brief initialization phase; (b) they were cache index-sensitive; (c) they were insensitive to cache line relative block alignment, and (d) they were sufficiently simple so as to be amenable to direct analysis, allowing us to establish that the benefit arose solely from index-aware allocation.

Regarding (c), above, during our investigation we discovered that some applications were extremely sensitive to how allocators placed blocks with respect to cache line boundaries - whether, for instance, the blocks were always aligned, never aligned, or sometimes aligned for a given size. Our set of allocators used various policies. By default `CIF` always returns blocks aligned on cache line boundaries, but, as a test of sensitivity can be configured otherwise. We only reported on applications that were not sensitive to cache line relative block alignment.

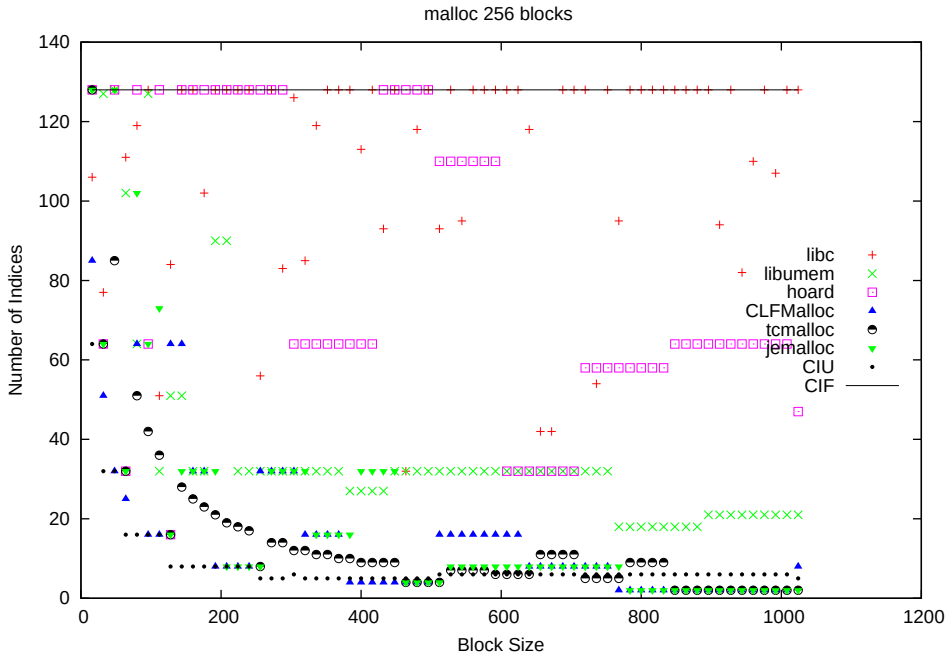

Figure 1. Index distribution over 256 allocated blocks

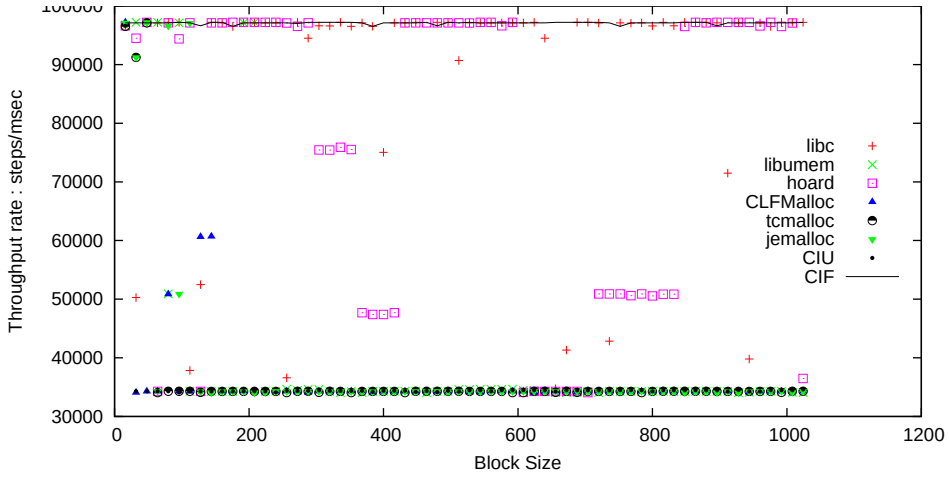

Figure 2. Pointer chasing performance : traversal rate over a list of 256 blocks

- **llubenchmark** [36] allocates groups of nodes and then iterates over those nodes during the benchmark interval. We used the version of `llubenchmark` found in the LLVM test suite version 2.7, and augmented it to report cache index distribution and allow for variable-length node sizes. The original form from Zilles allowed the node size to be specified on the command line but used a custom allocator while the form in the LLVM test suite used `malloc` but with a fixed node size. Our form `malloc`s each node and allows the node size to be specified on the command line. We used a command line of "-i 2000000 -n 1 -l 341 -g 0.0 -s 250" which specifies one list of 341 nodes of 250 bytes and 2000000 iterations over the list. As configured by the command line, all allocation is performed at startup time, so differences in reported performance reflect the rate at which the thread iterates over the list. We collect the elapsed time by running the program under the `time` command. We note that `llubenchmark` behaves similarly to our own `mcache`.
- **egrep** is GNU grep version 2.6.3, a regular expression search utility based on deterministic finite automata. We timed the search of a 500Mb text file containing nucleotide sequences. The key block size `malloc`-ed by the application is 1024 bytes, which represents arrays of 256 `int`s which serve as transition tables for the state machine. Only a few indices are actually accessed, however. All significant allocation occurs at startup time and the run is dominated by pointer chasing operations over the DFA graph structures. The performance differences in the figure are almost entirely attributable to `malloc` placement

policies in the different allocators. All the other benchmarks report elapsed time, but for `egrep` we report user-mode CPU time to factor out the IO-time required to read the file. (For reference, the run under `libc` took 11.2 seconds elapsed time with 2.01 seconds IO time. An IO time of 2.01 seconds is constant over the various allocators). Similar results were seen with the Google `RE2` regular expression package which is also index-sensitive.

- **dnapenny**[7] is benchmark in the "phylip" Phylogenic Inference Package component of the BioPerf bioinformatics benchmark suite. It uses a branch-and-bound algorithm to compute parsimonious trees. The source code was obtained from [4]. When starting, the application allocates 16 "tip" nodes. Each contains 4 buffers of size 6872 which are allocated separately. In the main loop there is an iteration that accesses a single buffer in each tip node. The program is moderately long-running, requiring more than 8 minutes under `libc`. Other phylip components such as `promlk` are similarly index-sensitive.
- **stdmap** creates a `std::map<int,int>` standard template library collection at startup. The benchmark then times the collection's iterator. The key-space is [0,499] and is approximately half populated. `std::map<>` is implemented as a red-black tree. The tree nodes are 40 bytes in length which some allocators round up to 64 bytes – a size that is cache index-unfriendly. The benchmark is written in C++ and produces a 64-bit executable. By default the C++ `new` and `delete` operators map directly to `malloc` and `free`. The tree implementation and nodes are opaque to the application so we were unable to directly report node addresses and indices, but instead used the Solaris `dtrace` and `truss -fl -t\!all -u ::malloc` commands to observe the allocation patterns.
- **Xml** is a 64-bit microbenchmark written in C that constructs an in-memory XML document tree via the Solaris `libxml2` package and then repeatedly iterates over the tree, reporting iteration times. The `xmlNode` instances are individually allocated by `malloc` and 120 bytes in length although only a few fields are accessed by the iterator. The `libxml2` library package parses the XML document and directly allocates the nodes which form the internal representation of the document.
- **Gauss** performs Gaussian elimination on $200x200$ matrices of 64-bit floating point numbers using the partial-pivot method. Each row is individually allocated.
- **DotProduct** computes the dot-product of a 200 vectors each of 200 elements.

  Both Gauss and DotProduct have array accesses of the form $a[I][J]$ where an inner loop advances I and an outer loop advances J and each row is individually allocated. (That is, the outer loop varies column and the inner loop varies row). There is no temporal locality as each element is accessed just once, although spatial locality is potentially available between iterations. While processing index I the inner loop may access a cache line underlying a row at address A only to find that same line subsequently evicted later in the inner loop because of conflict displacement. On the next iteration of the outer loop the code will access index $I + 1$ adjacent to I in that same line underlying A, incurring a conflict miss. In this case our code is iterating over multiple arrays simultaneously, in lock-step, and there are no hot fields. Because of avoidable index conflicts, the application may fail to leverage potential spatial locality. Cache index-aware allocation can often avoid this problem.

  This is a fundamentally different mode of benefit than is seen in the other applications, where index aware allocation leverages temporal locality in a small number of "hot" fields.

6.2 Superblock coloring

In Figure 4 we use `mcache` to demonstrate the efficacy of superblock coloring to reduce inter-thread index conflicts. Each thread `malloc`s two blocks of 100 bytes at startup and configures them as a ring via intrusive "next" pointers. (The choice of size is largely irrelevant in this benchmark). All the threads are completely independent. During the 10 second measurement interval each thread iterates over its private ring, visiting the two nodes in turn. When finished, the program reports the aggregate throughput rate of the threads. Figure 4 reports that throughput rate on the Y-axis in steps per millisecond while varying the number of threads on the X-axis. `CIF-NoColor` represents `CIF` configured with superblock coloring disabled. In an ideal system we would see perfect linear scaling but our real system has shared resources such as the pipeline (2 per core), caches, memory channel, etc. [34]. Beyond 8 threads, assuming ideal dispersion of those threads by the scheduler [13], threads start sharing the L1D. At 32 threads we have 4 threads per core. Recall that the L1D is 4-way set associative, so above 32 threads index collisions start to manifest as misses and impede scaling, even to the extent of actually reducing performance in some cases. As we can see the application scales reasonably up to 32 threads under all the allocators. Beyond 32 threads we see that performance bifurcates: we still find reasonable scaling under `libc`, `tcmalloc`, `CIF` and `libumem`, while under `Hoard`, `CLFMalloc`, `jemalloc` and `CIF-NoColor` we find that scaling fades. `libc`, `tcmalloc` and `libumem` are not vulnerable as the blocks distributed to the various threads come from a centralized heap instead of per-thread `mmap-ed` heaps.

We encountered an interesting performance phenomenon where access performance dropped precipitously under `jemalloc` at thread counts above 32. The problem manifested both under `mmicro` and `mcache`. We observed that only a fraction of the currently executing threads were afflicted, and those threads suffered extremely high level-2 cache (L2) miss rates. Investigation revealed that `jemalloc` requests memory in units of 4MB chunks via `mmap` – each thread that invokes `malloc` will have at least one such thread-private 4MB region. 4MB happens to precisely coincide with a large page size on our platform. Indeed, the `pmap -s` command confirmed that Solaris was placing those 4MB regions on 4MB pages. `jemalloc` does not provide any type of superblock coloring, so when a homogeneous set of threads invoke `malloc` they will obtain addresses that are the same offset from the base of their 4MB block. 4MB pages must start on 4MB physical address boundaries. Thus the physical addresses underlying the 4MB blocks are extremely regular, differing in only a small number of bits between threads. The set of addresses returned by `malloc` to the threads thus tend to conflict as they select only a small set of the possible L2 banks and L2 indices, resulting in conflict misses in the L2. The UltraSPARC-T2 applies an XOR-based hash to physical addresses to avoid such behavior, but in our case the physical addresses were so regular that the hash did not avoid the problem. We confirmed our suspicion by using an unsupported Solaris API to translate virtual addresses to physical addresses within our benchmark program, allowing us to analyze the distribution of physical addresses underlying the blocks allocated within the 4MB regions. Once the problem was understood we could avoid the issue by setting the `MALLOC_CONF` environment variable to "lg_chunk:20" which directs `jemalloc` to use 1MB regions instead of its default 4MB regions. All `jemalloc` data in this paper was collected in this mode (1MB). We could also induce the same performance problem under `CIF` by forcing the superblock size to 4MB and disabling superblock coloring, further illustrating the benefits of coloring.

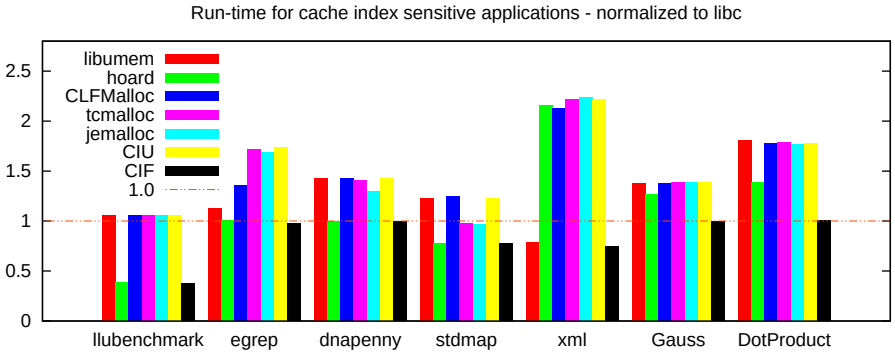

Figure 3. Cache index-sensitive applications

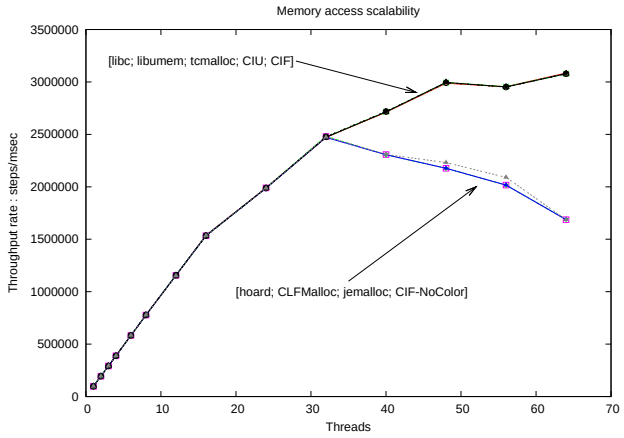

Figure 4. Impact of superblock coloring on memory access scalability

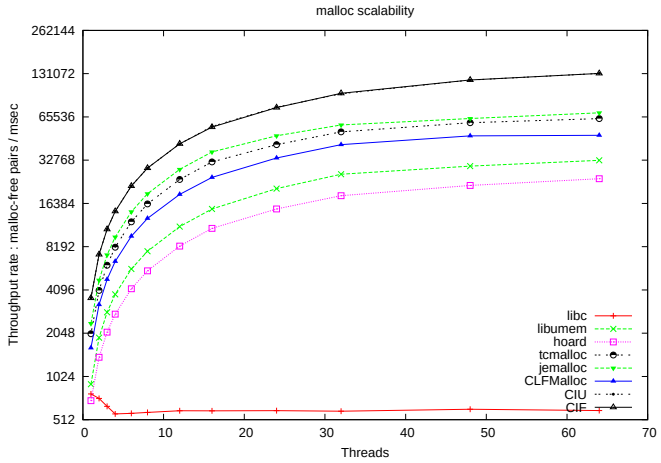

Figure 5. Malloc scalability varying thread count and allocators

6.3 Allocator Scalability

Here, we show that making an allocator index-aware does not affect its performance or scalability. In Figure 5 we use the `mmicro` benchmark from [15] and [14] which runs concurrent threads within a single process, each of which invokes `malloc` and `free` repetitively over a 50 second measurement interval, reporting the aggregate throughput rate of the threads in `malloc-free` pairs per millisecond. The threads are completely independent and do not communicate or write to any shared data. The UltraSPARC-T2 has only two pipelines per core, so scaling above 16 threads is modest and arises largely from memory-level parallelism [12]. As we can see from the graph all the allocators scale well except `libc`, which uses a single heap with a centralized lock. Broadly, the ratio of performance between the allocators observed at 1 thread holds as the number of threads increases, suggesting that path length through the `malloc` and `free` dominates multithreaded performance, and that the allocators have no substantial scaling impediments. As is made obvious in the graph, the application is insensitive to cache index placement as `CIU` effectively yields the same results as `CIF`.

7. Related work

The literature is rich with studies that show how layout and placement can influence cache behavior and impact performance for pointer-based programs [25][37]. Petrank [29] shows that cache-conscious data placement is, in the general case, NP-hard. By exploiting common access patterns and behavior found in applications we can still, however, provide benefit in many circumstances. Broadly, the optimization techniques involve changing the access pattern; intra-object field layout changes; and inter-object placement policies. Calder et al. [10] introduce *cache conscious data placement* which reduces cache misses by profiling an application, building a temporal relationship graph of data accessed, and finally using the temporal access patterns discovered in the profiling stage to refine data placement. Kistler et al. [22] develop an algorithm that clusters data members to promote and enhance spatial locality. Chilimbi et al. [11] show the benefit of cache-conscious data layout and field placement. Their allocator interface is non-standard, however, and does not allow drop-in binary replacement under the standard `malloc-free` interface.

Lvin et al. [26] use object-per-page allocation in the `archipelago` allocator to probabilistically detect errors in the heap arising from software flaws. Naively, if all objects were to start on page-aligned

virtual address boundaries then applications accessing such objects could suffer from excessive conflict misses. To reduce the conflict miss rate their allocator randomly colors the offset of the object with the page. Coloring was used to salve the impact of page alignment, and not applied in a general and principled fashion that minimizes wastage. Furthermore, an object-per-page allocator may impose high TLB pressure.

Bonwick et al. [9] (section 4.1-4.3) also suggested superblock coloring, but only as remedy for *inter-superblock* intra-thread cache index conflicts and to relieve bus and bank imbalance for systems with multiple memory channels. Their paper predates commodity CMT (chip multithreading) systems with shared caches. We believe that index-aware size-classes or punctuated block arrays largely obviate and supersede the use of superblock coloring for the purpose of addressing inter-superblock intra-thread conflicts. That is, index-aware size-classes or punctuated arrays reduce both intra- and inter-superblock conflicts for a given thread accessing a set of superblocks. Superblock coloring remains useful, however, as it provides a new mode of benefit for intra-core inter-thread inter-superblock conflicts on modern shared cache CMT platforms. Bonwick also noted that binary buddy allocators are pessimal with respect to cache index distribution. We concur and generalize to sizes other than simple powers-of-two. We also note that facilities such as `memalign` should be used judiciously as excessive unneeded alignment can induce conflict misses.

Page coloring [30] operates at the level of the operating system or virtual machine monitor by influencing the choice of physical pages to assign to virtual addresses. The color of a physical page is just the value in the intersection of the physical page number field and the cache index field of the page address. Page coloring attempts to provide a uniform distribution of page colors for the physical pages assigned to a set of virtual pages, which in turn promotes balanced utilization of the set of available cache indices. Say that in the physical address layout we find that the page number field overlaps the cache index field by 2 bits, giving 4 possible page colors. If the kernel does not provide ideal page coloring and inadvertently mapped virtual pages V0, V1, V2 and V3 to physical pages P0, P1, P2 and P3, respectively, and those physical pages happened to be of the same page color, then cache lines underlying V0, V1, V2 and V3 would be able to reside in just one quarter of the available cache indices, possibly underutilizing parts of the cache and creating a "hot spot" in other sectors. Page coloring attempts to avoid such unfavorable assignments of physical pages to virtual addresses.

Hardware-based means of reducing the rate of conflict misses were suggested Seznec [32] (skew-associative caches) and later by by Gonzales [17] and Wang [35]. All entail changes to the hash function that maps addresses to cache indices and none is currently available in commodity processors. Min and Hu [28] suggest completely decoupling memory addresses from cache addresses in order to reduce conflict misses while Sanchez and Kozyrakis [31] subsequently suggest decoupling ways and associativity.

Our approach is most similar to that of page coloring except that it is implemented entirely in user-space within the virtual address `malloc` allocator and operates only on low-order bits of the cache index field of addresses that are not part of the physical page number field. Page coloring and cache index-aware allocation are complementary optimizations. Like page coloring our approach is non-intrusive in that it operates without any need to profile the application or modify the application's source code. If the `malloc` library is implemented as a separately deliverable dynamically loadable module, as is the case on most platforms, then our approach can be used by simply substituting a new `malloc` library, eliminating the need to recompile and providing benefit to legacy binary applications. In addition, our technique is orthogonal to but benefits from complementary mechanisms that change field layout to promote spatial locality [22]. Instead of specifically increasing locality it simply leverages ambient locality already present in the application.

8. Conclusion

Optimal index placement - like optimal field placement – is NP-hard. Techniques such index-aware allocation can, however, still benefit index-sensitive applications, avoiding a performance pitfall. Not all applications will benefit from an index-aware allocator but our approach is benign and has no observed negative impact. We make no general claims or guarantees about performance but note that all other factors being equal, balanced index distribution, like balanced page coloring, is preferable, given that it is relatively easy to avoid the vagaries of index-oblivious allocators. Finally, in most allocators, application of our technique requires extremely simple modifications, often changing just a few lines of code.

The phenomena of index sensitivity has been noticed before, and solutions along similar lines have been proposed but here we approach the issue methodologically, providing guidelines to future designers and developers. We clarify, explain and analyze the behavior and provide general solutions.

8.1 Contributions of this paper

- We identify the problem of *inter-block cache index conflicts* arising from excessive regularity in addresses returned by memory allocators. We show that the placement policies of `malloc` and related allocators, by virtue of conflict miss rates, can have a significant impact on application performance.
- We provide a simple solution to inter-block conflicts through index-aware size-classes or punctuated superblocks arrays with interspersed spacers. While not all applications are cache index-sensitive and thus show no benefit from an index-aware allocator, we claim our solution has no observed negative consequences reflecting the principle of "first, no harm" and argue that it should be used in new and existing allocators.
- We note that superblock coloring provides new benefits for CMT systems with shared caches, reducing both intra-thread and inter-thread inter-superblock index conflicts.
- We propose process-specific color seeding to avoid inter-process cache index conflicts that can come to be on CMT systems where concurrently executing threads share a data cache.
- We provide a partial taxonomy for allocator-based index conflicts.
- Taken collectively index-aware size-classes, punctuated block arrays, superblock coloring, and process-specific color randomization provide index-aware block placement and allow the construction of index-aware allocators.

We note that cache line relative block alignment has an impact on performance but is largely neglected in the literature. We found many applications to be sensitive to whether blocks were returned on addresses that coincide with cache line boundaries. The least significant nibble of addresses returned by `malloc` must be either 0 or 8. Some allocators always return addresses of the latter form for certain size-classes. We recommend that allocators should, to the extent possible and reasonable, return addresses aligned on cache line boundaries – this policy minimizes the number of cache lines underlying an object and decreases cache pressure, as well as reducing the odds of allocator-induced false sharing.

8.2 Future work

We plan to further explore using our techniques in other `malloc` allocators as well as in a Java Virtual Machine, where we can enforce index-aware size-classes in the object layout manager and provide random coloring either at the start of or within thread local allocation buffers (TLABs), which are contiguous thread-local object allocation regions managed by a simple *bump pointer*. Initial testing with `mcache` transliterated to Java has shown that the HotSpot JVM exhibits index unfriendly placement for certain object sizes with reduced performance and increased L1D miss rates.

We believe that cache index-aware allocation should be particularly helpful for hardware transactional memory implementations where the address-sets are tracked in the L1D and where conflict misses cause transaction aborts.

When multiple processors share a cache it may be useful to modify the hardware to mark cache lines with the identity of the processor that inserted a given line into the cache. It is possible that identity could be inferred from the tag value, depending on the address space layout. New performance counters and performance sampling facilities could be implemented that could differentiate *intrinsic* (intra-processor) and *extrinsic* (inter-processor) cache line displacement. If processor 1 displaced a line that was installed by processor 2 then we have extrinsic eviction, for instance. That information, in turn, could be useful to the developer or perhaps to the operating system scheduler in order to better place threads within the system topology in order to reduce miss rates. If the index and CPU ID (or tag) were made visible to a sampling facility then software could also measure rates of intra-core inter-thread conflict misses.

We hope to extend our analysis to the level-2 cache and also to determine if our approach might yield better DRAM bank and channel balance, admitting more parallelism in the memory subsystem for accesses that miss in caches.

Instead of using random number generators to assign color, it may be profitable to track the population of superblock colors and assign the least used color when creating a superblock.

A source of surprise was that, holding all other parameters fixed, using widely-spaced size-classes (2x) often yielded better performance than the more traditional 1.2x stepping recommended in the literature, which bounds fragmentation at 20%. For a given set of `malloc` requests, a coarse 2x set of size-classes may result in fewer underlying pages but more intra-block fragmentation. This suggests that TLB span might be more important in some cases than the wastage and increased data cache span arising from using coarse-grained size-classes. We hope to investigate this effect -- the tensions between wastage and data cache-span versus TLB span - in the future, possibly implementing size-class schemes that adaptively refine the set of size-classes and superblock sizes at runtime.

8.3 Observations

We note that address-space randomization (ASR), while often used for security purposes to make programs less vulnerable to exploits such as buffer overrun attacks, may have a beneficial effect as it provides implicit coloring.

Caches with much higher levels of associativity largely obviate our approach of index-aware size-classes, but the trend of platform design is not toward such complex implementations. Similarly, *victim caches* [20] would provide relief, but these are not found in commodity systems.

The UltraSPARC-T2 has 8 replicated *L2 Banks*. Each L2 bank has 512 indices with 64-byte lines and is 16-way set associative. The L2 cache line is the unit of coherence. L2 banks are physically-indexed and physically tagged. The L2 is unified, caching both code and data. Pairs of L2 banks share a DRAM channel. A central cross-bar resides between the cores and the L2 banks and routes physical addresses from cores to the appropriate L2 bank based on the value of physical address bits [8:6]. To help improve L2 index distribution the cross-bar applies a hash to the physical address by XORing high-order physical address bits into bits [17:11], which overlap the L2 index field. Physical address bits [17:9] constitute the L2 index field. Virtual address bits [10:0] pass through verbatim into the physical address. As such, the L2 bank select field of the physical address is a sub-field of the L1 index field, and the low 2 bits of the L2 cache index field overlap the 2 highest of the L1 index field. Thus, better and more uniform L1 index distribution yields better L2 bank distribution (inter-bank benefit) and better L2 index distribution within a bank (intra-bank benefit). Better L2 bank balance may admit more memory-level parallelism by reducing contention on the channel or path between the cores and L2 banks. Better L2 index distribution within a bank may lower the L2 miss rate. Cache index-aware allocation reduces L1 conflict misses, but accesses that miss in the L1 may also enjoy benefits from index-aware allocation.

References

[1] *The Solaris Schedctl Facility*. US Patent #5937187.

[2] *hoard.org*, 2010 (accessed 2010-1-14). `hoard.org`.

[3] *Amino-CBBS*, 2010 (accessed 2010-1-5). `amino-cbbs.sourceforge.net`.

[4] *BioPerf*, 2010 (accessed 2010-5-19). `http://www.bioperf.org/`.

[5] *dlmalloc*, 2010 (accessed 2010-7-2). `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[6] *tcmalloc : version 1.6*, 2010 (accessed November 9, 2010). `http://code.google.com/p/google-perftools/`.

[7] D. Bader, Y. Li, T. Li, and V. Sachdeva. Bioperf: a benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. *IEEE Workload Characterization Symposium*, 0:163–173, 2005.

[8] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proc. ninth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, New York, NY, USA, 2000. ACM.

[9] J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[10] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. *SIGPLAN Not.*, 33(11):139–149, 1998.

[11] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, 2000.

[12] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. *SIGARCH Comput. Archit. News*, 32(2):76, 2004.

[13] D. Dice. *Dave Dice's blog*, 2010 (accessed 2010-7-21). `http://blogs.sun.com/dave/entry/solaris_scheduling_and_cpuids`.

[14] D. Dice and A. Garthwaite. Mostly lock-free malloc. In *Proc. 3rd International Symposium on Memory Management*, pages 163–174, New York, NY, USA, 2002. ACM.

[15] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Oleszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 325–334, New York, NY, USA, 2010. ACM.

[16] J. Evans. A scalable concurrent malloc(3) implementation for freebsd, 2006.

[17] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating cache conflict misses through xor-based placement functions. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 76–83, New York, NY, USA, 1997. ACM.

[18] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[19] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *ISMM*, pages 26–36, 1998.

[20] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Comput. Archit. News*, 18(3a):364–373, 1990.

[21] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, 1992.

[22] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Trans. Program. Lang. Syst.*, 22(3):490–505, 2000.

[23] K. C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965.

[24] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, New York, NY, USA, 2005. ACM.

[25] A. R. Lebeck and D. A. Wood. Cache profiling and the spec benchmarks: A case study. *Computer*, 27(10):15–26, 1994.

[26] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. *SIGOPS Oper. Syst. Rev.*, 42(2):115–124, 2008.

[27] M. M. Michael. Scalable Lock-free Dynamic Memory Allocation. In *Proc. ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 35–46, 2004.

[28] R. Min and Y. Hu. Improving performance of large physically indexed caches by decoupling memory addresses from cache addresses. *IEEE Trans. Comput.*, 50(11):1191–1201, 2001.

[29] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 101–112, New York, NY, USA, 2002. ACM.

[30] T. Romer, D. Lee, B. N. Bershad, and J. B. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *In 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–266, 1994.

[31] D. Sanchez and K. Christos. The zcache: Decoupling ways and associativity. In *MICRO 43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010.

[32] A. Seznec. A case for two-way skewed-associative caches. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 169–178, New York, NY, USA, 1993. ACM.

[33] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[34] V. Čakarević, P. Radojković, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Characterizing the resource-sharing levels in the ultrasparc t2 processor. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 481–492, New York, NY, USA, 2009. ACM.

[35] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 83–93, Washington, DC, USA, 2008. IEEE Computer Society.

[36] C. B. Zilles. Benchmark health considered harmful. *SIGARCH Comput. Archit. News*, 29(3):4–5, 2001.

[37] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *DaMoN '06: Proceedings of the 2nd international workshop on Data management on new hardware*, page 6, New York, NY, USA, 2006. ACM.

A. UltraSPARC-T2 "Niagara" Level-1 Data Cache Organization

The UltraSPARC-T2 level-1 data cache is organized as follows:

- 16-byte lines
- 4-way set associative
- 128 sets per cache – 128 possible indices
- 8KB cache with 512 lines
- Shared over 8 logical processors in a core [34]
- Physically-indexed and physically-tagged : PIPT
- Physical addresses map to cache indices by way of a hash function that shifts the address 4 bits to the right and then masks off the low 7 bits to form the index. Physical addresses presented to the cache by the processor have the following format: bits [3:0] form the cache line offset; bits [10:4] form the Level-1 cache index, and the remainder form the tag.
- Addresses A and B that refer to distinct cache lines map to the same index in the L1D if and only if $(A/16) mod 128 = (B/16) mod 128$. We say A and B conflict in the L1D. If more than 4 addresses map to the same index then we have *index contention* and repetitive accesses to those addresses can result in conflict misses.
- 2KB *cache page* size : 128 indices * 16 bytes per line. A cache page is the set of addresses that share a common value in the tag field. Address A and $A + 2KB$ map to the same index in the L1D.
- L1D-based page coloring [21] is not applicable as there is no overlap between the physical page number and index fields. That is, the cache page size is less than the system base page size of 8KB.

B. Cache index-aware size-classes

To avoid the undesirable behavior exhibited above by index-oblivious allocators we can simply choose, as an alternative to punctuated arrays, a slightly different set of size-classes that is less prone to inter-block conflicts. That is, we simply avoid block sizes that underutilize the available indices. We can apply the adjustments, below, either statically at compile-time or at run-time, to transform a set of size-classes to be index-aware, producing an index-aware allocator.

Simplifying the problem slightly for the purposes of exposition, we will assume the effective block size S inclusive of any header is always an integer multiple of 16 bytes (the cache line size). We define the cache index of a block as the index of that block's base address. Within a given superblock the constituent blocks will have addresses of the form $(S * n) + B$ where B is the base of the block array in the superblock and $n \in \mathbb{N}$ up to the number of blocks in the superblock. Given S, the number of distinct indices for blocks in a sufficiently long superblock of size-class S is $2048/GCD(2048, S)$ where 2048 is the cache page size. Equivalently, we can state the number of usable indices for S as $128/GCD(128, S/16)$. Notice that we have a cyclic subgroup $\mathbb{Z}/(128)$ where 128 is the number of indices in the cache and the cycle length of the size corresponds to the number of indices on which blocks of that size can fall. As such, to provide an ideal tessellation we want to ensure $GCD(128, S/16) = 1$. That is, $S/16$ should be coprime with 128 and thus a generator of $\mathbb{Z}/(128)$, in which case blocks of size S will land uniformly on all possible indices. The following simple transformation will adjust any size S to be index-aware :

```
if GCD(2048,S) > 16 then S += 16
```

`CIF` can be configured to use index-aware size-classes instead of the punctuated array. And in fact the form with index-aware size-classes yields the same performance as the form that uses the punctuated array. In this mode `CIF` uses size-classes of the form $(2^N + 1) * 16$ for N=0,1,2,3 etc., yielding a favorable index distribution.

In the case of `CLFMalloc` only 4 lines of codes – an array of `int`s that defines the size-classes – needed change to render `CLFMalloc` cache index-aware. While not reported for lack of space, we constructed both an intentionally index-unfriendly form of `CLFMalloc` with power-of-two size-classes and an cache index-friendly form using the transformation described above. With respect to index sensitivity, the performance of these two forms parallels that of `CIU` and `CIF` but we opted to report data from `CIU` and `CIF` as those allocators show better latency and scalability than the `CLFMalloc`-based forms and because `CLFMalloc` does not expose the `memalign` interface, which is required by some applications.

We note that even a small change in size-classes can have a profound impact on footprint and greatly perturb the heap layout, possibly resulting in large changes in the conflict miss rate and confounding causal analysis. One set of size-classes might simply be a better fit for the choice of sizes used by the application, resulting in less wastage. Paradoxically, adjusting each size-class upward to create an index-friendly allocator might decrease footprint.