Improving the Cache Locality of Memory Allocation

Dirk Grunwald Benjamin Zorn Robert Henderson
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

Abstract

The allocation and disposal of memory is a ubiquitous operation in most programs. Rarely do programmers concern themselves with details of memory allocators; most assume that memory allocators provided by the system perform well. This paper presents a performance evaluation of the reference locality of dynamic storage allocation algorithms based on trace-driven simulation of five large allocation-intensive C programs. In this paper, we show how the design of a memory allocator can significantly affect the reference locality for various applications. Our measurements show that poor locality in sequential-fit allocation algorithms reduces program performance, both by increasing paging and cache miss rates. While increased paging can be debilitating on any architecture, cache misses rates are also important for modern computer architectures. We show that algorithms attempting to be space-efficient by coalescing adjacent free objects show poor reference locality, possibly negating the benefits of space efficiency. At the other extreme, algorithms can expend considerable effort to increase reference locality yet gain little in total execution performance. Our measurements suggest an allocator design that is both very fast and has good locality of reference.

1 Introduction

.

The allocation and disposal of memory is a ubiquitous operation in most programs, yet one largely ignored by most programmers. Some programmers use domain-specific knowledge to improve the speed or memory utilization of memory allocators; however, the majority of programmers use the memory allocator provided in a given programming environment, believing it to be efficient in time and/or space. In virtual memory systems, space efficiency is usually a secondary concern, although important in some applications. Rarely are programmers concerned with "secondary" effects, such as cache locality.

In this paper, we show that the structure of dynamic storage allocation (DSA) implementations contribute to the page and cache miss rate. Our measurements show that increased cache misses can increase program execution time on conventional architectures by up to 25%.

In a previous paper [8], we measured and compared the time and space efficiency of standard memory allocators. Based on empirical observations[29], we designed a program (CUSTOMALLOC) that synthesizes a memory allocator customized for a specific application. Empirical measurements show that the synthesized allocators are uniformly faster than the allocators distributed with widely-used operating systems (e.g., BSD) and are also more space efficient. In that study, we measured the number of *machine instructions* used to allocate and free memory, and did not account for the effects of the memory hierarchy, which is the focus of this paper.

1.1 The Importance of Reference Locality

In recent years, faster processor speeds have shifted interest in memory system performance from slower main memories to high-speed cache memories. Cache memories must be able to provide instructions and data to processors at increasingly faster rates. As processors and cache memories increase in speed, the cost of missing in the cache and retrieving data from main memory increases. Jouppi estimates the cost of a cache miss will increase to over 100 cycles if current trends continue [11]. Mogul and Borg evaluate the effect of context switches on a hypothetical two-level cache that requires 200 cycles to service a second-level cache miss [19]. Although the performance of caches is improving, new processors commonly use a smaller on-chip primary cache, with a larger secondary cache.

Increased cache misses are difficult to detect. Some recent tools [7, 17] indicate what regions of a program incur excessive cache misses. However, they do not indicate the *reason* for those misses. Although the cache misses may be seen in one region of the program, the cause may arise elsewhere. More insidiously, the increased cache misses may be spread over all program sections that reference heap allocated objects, belying the true influence of the DSA algorithm. We believe that some algorithmic and implementation decisions in DSA design influence the *overall* reference locality of a program. Furthermore, unlike the related problem of virtual memory locality, it is difficult to devote more resources to reduce cache misses. Although main memory can be expanded to reduce page faults in an existing system, it is often impossible to expand the size of a primary or secondary cache. Thus, for existing computers, improved management of the cache resource is the only alternative.

In this paper, we investigate the effects of DSA algorithms and implementation choices on cache locality and total performance in a number of significant allocation-intensive programs. We show that these choices affect the overall performance of allocation-intensive programs by up to 25%. Misses are increased directly by DSA im-

[0]This paper will appear in the SIGPLAN'93 Conference on Programming Language Design and Implementation, June 1993, Albuquerque, New Mexico

plementations with poor reference locality and indirectly by implementations that fill valuable cache space with storage management overhead. Combining the results of this paper with our previous work on synthesized DSA implementations, we expect it is possible to design fast, space-conservative DSA implementations that have very good cache locality. Reference locality and allocator speed are two aspects of memory allocation that must be balanced to achieve high performance.

1.2 Structure of the Paper

In the next section, we describe the DSA implementations considered in this paper and review the previous work in measuring the locality of DSA implementations. In §3, we describe our experimental methodology including the suite of programs used to compare the different DSA implementations. Section 4 presents the results of our simulation study, both for page locality and cache locality. Finally, in §5 we summarize the salient features of memory allocators that are both fast and have high reference locality.

2 Previous Work

2.1 Memory Allocation Algorithms

General purpose algorithms for dynamic storage allocation have been proposed, analyzed and tuned for many years. Knuth [13], Bozman *et al* [3], and Korn and Vo [14] all carefully describe and evaluate various allocation strategies and implementations. Traditionally, these algorithms have sought to minimize CPU overhead and reduce total memory usage.

Standish [22] divides algorithms for dynamic storage allocation into three broad categories: sequential-fit algorithms (e.g., first-fit and best-fit), buddy-system methods (e.g., binary-buddy and Fibonacci), and segregated-storage algorithms, which is a catch-all category involving a number of different approaches. Our previous research on DSA customization [8] has given us first-hand experience with several of the most widely-used implementations.

In this paper, we compare the reference locality of five algorithm implementations that are described below. These particular algorithms were chosen because they are well-known, considered to be efficient (either in time, space, or both), and widely-used.

FIRSTFIT: This algorithm is an implementation of a first-fit strategy with optimizations suggested by Knuth [13] and implemented by Mark Moraes. In this algorithm, free blocks are connected together in a doubly-linked freelist that is scanned during allocation for the first free block that is sufficiently large. This block is split into two blocks, one of the appropriate size that is returned, and the other that is placed back in the freelist. As an optimization, if the extra piece is too small (in this case less than 24 bytes), the block is not split. The freelist pointer is implemented as a "roving" pointer, which eliminates the aggregation of small blocks at the front of the freelist.

Allocated blocks in this algorithm require two extra words of overhead (boundary tags), one at each end of the block, which contain the size of the block and its current status (allocated or free). Boundary tags allow objects to be freed and coalesced with adjacent free storage in constant time.

GNU G++: This algorithm, implemented by Doug Lea [16], enhances the standard first-fit algorithm by using an array of freelists segregated by object size. In each freelist, free blocks are connected together in a doubly-linked list. An appropriate freelist is selected based on the logarithm of the allocation request; this is done to increase the probability of a better fit. In other respects, this algorithm is similar to FIRSTFIT.

BSD: Chris Kingsley implemented a fast segregated-storage algorithm that was distributed with the 4.2 BSD Unix release [12]. Kingsley's algorithm rounds object size requests to powers of two minus a constant and a freelist of objects of each size class is maintained. If no objects of a particular size class are available, more storage is allocated. No attempt is made to coalesce objects. Because this algorithm is so simple, its implementation is very fast. On the other hand, it also wastes considerable space, especially if the size requests are often slightly larger than the size classes provided.

GNU LOCAL: Mike Haertel implemented a hybrid of the first-fit and segregated-storage algorithms that is available to the public as the Free Software Foundation implementation of malloc/free [9]. In Haertel's algorithm, requests larger than a specific size (e.g., 4096 bytes) are managed using a first-fit strategy, while objects smaller than this are allocated in specific size classes (powers of two), like the BSD algorithm.

Haertel's approach actively seeks to improve the locality of reference during allocation by dividing allocated storage up into page-sized chunks and storing information about these chunks in small, highly-localized chunk headers. Instead of traversing the entire heap attempting to find a fit, only the information in the chunk headers must be traversed. This algorithm also reduces the per-object space overhead required by other algorithms (such as the boundary-tags in the first-fit algorithms) in the following way. Chunks are allocated so that all objects in a chunk are the same size. The address of any object can be used to locate the chunk header, which contains information about the object size associated with the chunk. The chunk header also contains a count of the number of free blocks within a chunk and deallocates entire chunks when all the objects in the chunk have been freed.

QUICKFIT: Weinstock and Wulf describe a fast segregated-storage algorithm based on an array of freelists[24, 22]. Like the GNU LOCAL algorithm, QUICKFIT is a hybrid algorithm that allocates small and large objects in different ways. Large objects are handled by a general algorithm, while objects smaller than a certain threshold are allocated and deallocated very quickly. The algorithm is very fast because the object request size is used as an index into the freelist array, returning the appropriate freelist in a small number of instructions. Deallocation requires identifying the size of the object being freed (using a *boundary tag* in our implementation), indexing the freelist array to get the appropriate freelist, and placing the free object on that list. Both the BSD and QUICKFIT algorithms make no attempt to coalesce free objects. Unlike the BSD algorithm, which rounds sizes to powers of two, QUICKFIT rounds to multiples of word sizes (e.g, 4, 8, or 16 bytes), reducing internal fragmentation. The configuration of QUICKFIT that we measured handles allocation requests of 4–32 bytes (rounded to word size) with the fast array and larger requests using a general purpose first-fit algorithm (GNU G++, in our case).

2.2 Measurements of Locality

Most studies of reference locality in DSA implementations concern *garbage collection* (GC) algorithms [15, 18, 20, 23, 25, 26, 27, 28]. The main reason for this emphasis is that references made during garbage collection have poor locality compared to program references. This poor locality can result in thrashing of the virtual

memory paging system when a program's address space is larger than the physical memory. Experiences with thrashing led to the development of generational GC algorithms, which focus the efforts of garbage collection on a relatively small part of the total address space [18]. A number of reports, including those of Moon [20] and Ungar [23], indicate that generational GC successfully reduces the page-fault rates in garbage-collected language systems such as Lisp and SmallTalk. Other attempts to improve the reference locality of garbage collection have focused on improving the traversal order of objects in memory during a collection. Techniques such as breadth-first, depth-first, approximate depth-first, hierarchical decomposition, and type-directed traversals have been implemented and measured [20, 15, 26].

More recently, there has been greater emphasis on the cache locality of these algorithms. Zorn measured the cache locality of generational copying and non-copying garbage collection [27, 28]. Wilson also reports on issues related to the cache locality of copying generational collection, including the effect of cache associativity, cache size, and algorithm design on miss rate [25].

Unfortunately, many of the locality issues related to GC algorithms are not relevant for explicit dynamic storage allocation because there is no need for explicit algorithms to traverse the set of living objects. Furthermore, garbage collection techniques that copy data are also not applicable in languages such as C, where dynamicly allocated objects can not be relocated. One possible reason for the dearth of prior work on the reference locality of explicit DSA algorithms is the perception that the reference locality of existing algorithms is good enough. We see in §4 that reference locality can have a dramatic effect on program performance, even with explicit DSA algorithms.

3 Experimental Design

In this section, we describe our measurement methods. The results in §4 are based on measurements of actual allocation-intensive programs. We believe that dynamic memory allocation will play an increasingly important role in existing and future systems. We only consider allocation-intensive programs to better illustrate the problems encounted. Here, we provide information about the programs measured and the tools used to measure them.

3.1 Application Programs

Our goal is to investigate the effect of DSA algorithms on cache locality in programs that are of interest to a wide class of users. We collected five allocation intensive programs, described in Table 1 and Table 2, representing a variety of allocation-intensive application domains including language interpreters (GAWK and GS), a language translator (PTC), a program dependence analysis tool (MAKE), and a PLA logic optimizer (ESPRESSO). As Table 2 shows, the input sets used exercise the storage allocation system significantly, resulting in hundreds of thousands of allocations while executing more than a billion instructions.

In this study, we did not modify the applications; any change in cache misses using different DSA algorithms arises either from references in the DSA subroutines or because the representation of allocated objects has some effect on the locality of the application. In general, it is difficult to measure the exact contribution of these direct and indirect effects on the overall cache miss rate. As with any programmer, we are concerned with the influence of DSA implementation on total execution time.

Figure 1 shows the fraction of execution time the application programs spend doing dynamic storage allocation (both allocation and deallocation), as measured by counting instructions and assuming no cache miss penalty; in §4.2, we measure the influence of cache miss penalties. As the figure shows, the choice of allocator dramatically affects the fraction of time spent doing allocation. For the programs considered, the time spent doing allocation ranges from a few percent to $\approx$ 30%, depending on the allocator and application.

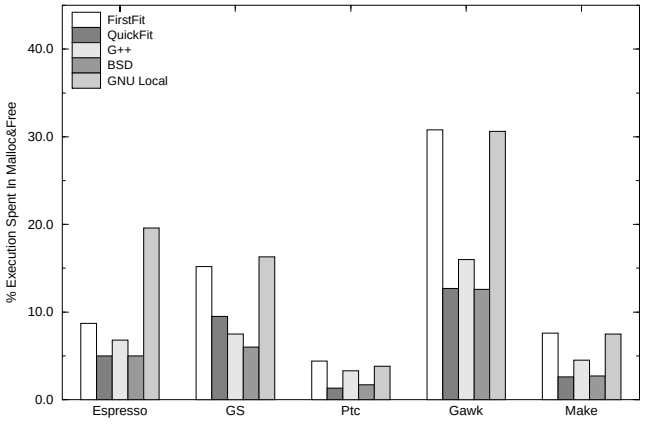

Figure 1: Percent of Time in Malloc and Free (as % of Execution Time)

3.2 Measurement Tools

We measured execution time in terms of machine instructions using Larus and Ball's QP utility [1]. This tool provides a dynamic execution count for each subroutine in terms of instructions. Using this tool removes any variability in counting program instructions, greatly simplifying the experimental design.

To measure cache locality, we instrumented the programs using PIXIE [6]. Although PIXIE can also provide execution information similar to that of QP, we found the output format of QP more useful in our study. Using PIXIE, the instrumented programs emit coded information allowing us to reconstruct the data references. We modified a version of the TYCHO [10] cache simulator to implement execution-driven cache simulation. Since the cache simulator directly consumed the trace from the instrumented application, we were able to quickly trace a large number of references without storing large trace files. We used traces varying from 17 million references to almost 600 million references (see Table 2 for the data references generated by the FIRSTFIT allocator; other allocators have a similar number of references.)

We chose to simulate a direct-mapped cache with a 32-byte block size. All cache miss rates presented are the miss rates for the data cache; in all cases, we assume the instruction cache miss rate is 0%, making our predictions of the cache effect on execution time conservative. Likewise, our miss rate results are conservative because we intentionally avoid introducing the effects of intermittent cache flushes on the miss rate. Because the tools we use generate deterministic results, our experiments did not require statistically averaging multiple runs. We used VMSIM, a fast implementation of a stack simulation algorithm [27], to measure page fault rates; a page size of 4 kilobytes was used for all page fault measurements.

ESPRESSO	Espresso, version 2.3, is a logic optimization program. The input file is an example provided with the release code.
GS	GhostScript, version 2.1, is a publicly available interpreter for the PostScript page-description language. The input files were a variety of small and large files, including a 126-page user manual. This execution of GhostScript did not run as an interactive application as it is often used, but instead was executed with the NODISPLAY option that simply forces the interpretation of the Postscript(without displaying the results).
PTC	PTC, version 2.3, is a Pascal to C translator. The input file was a 19,500 line Pascal program (mf2psv.p) that is part of the TEX release.
GAWK	Gnu Awk, version 2.11, is a publicly available interpreter for the AWK report and extraction language. The input script formatted words in a dictionary.
MAKE	Gnu-make, version 3.62 is a version of the common 'make' utility used on UNIX. The input set was the makefile of another large application.

Table 1: General Information about the Test Programs

Program	Exec. Time (sec.)	Total Instr. ($\times 10^6$)	Data Refs. ($\times 10^6$)	Max. Heap Size (Kbytes)	Objects Alloc'ed (1000s)	Objects Freed (1000s)
ESPRESSO	155.1	2506	595	396	1673	1666
GS	131.3	1344	421	4129	924	898
PTC	25.1	367	125	3146	103	0
GAWK	76.7	1215	374	60	1704	1702
MAKE	4.0	56	17	380	24	13

Table 2: Test Program Performance Information. Execution times were measured on a DECstation 5000/120 workstation (MIPS architecture) with 24 megabytes of memory. All values provided are for the FIRSTFIT allocator, which is used as a baseline in later figures.

4 Comparison

We traced the execution of the five applications using the five different DSA implementations described. We used these traces to measure the *page fault rate* and *cache miss rate*.

4.1 Page Reference Locality

We measured the page fault rate of five applications using the different DSA implementations. Figures 2 and 3 show the page fault rates expressed in faults per memory reference for two applications, GS and PTC. In both figures, the symbols on the horizontal axis indicate the total amount of memory requested by the program using the different DSA implementations. Although each application has different behavior, these datasets effectively capture the range of effects of DSA implementation on the page fault rate.

Figures 2 and 3 indicate two interesting metrics for each DSA implementation: the maximum required space and the slope of the paging rate for each DSA implementation. Large page fault rates are so debilitating that few systems can tolerate even the small amount of paging implied by the subtle differences between the efficient allocators shown in Figures 2 and 3 for a fixed memory size. It is generally true that reducing the amount of memory needed by an application has the greatest influence on the paging rate. Thus, the most effective measure to reduce paging is to select a very space efficient allocator. The slope of the paging rate for each DSA implementation in Figures 2 and 3 indicates the "resiliency" of the allocators to restricted resources. For example, the performance of the FIRSTFIT algorithm in the GhostScript application would rapidly degrade if memory was unavailable, while the performance of QUICKFIT would degrade less rapidly.

Two of the DSA implementations, FIRSTFIT and GNU G++, are variants of the classic first-fit algorithm. They find free regions of memory by traversing a linked-list data structure and coalesce contiguous free regions. These algorithms tend to have poor page locality because the allocator examines several objects in the linked list and those objects may be scattered throughout the address space. Furthermore, when searching for a large object, *all* entries in the linked list may be visited. Recall that the FIRSTFIT implementation uses a single doubly-linked list structure to hold all objects, while GNU G++ uses multiple doubly-linked lists, segregated by the size of the free objects. On average, the GNU G++ allocator examines fewer objects than FIRSTFIT when allocating memory. The effect of this simple algorithmic change is dramatic. By searching less objects in the freelist, the GNU G++ algorithm is more "resilient," and would fault less frequently. As shown in Figures 1 and 2, the conventional FIRSTFIT algorithm increases both the raw execution time *and* the page fault rate.

Another factor that affects both FIRSTFIT and GNU G++ is the decision to coalesce contiguous objects on the freelist. This is usually done either by maintaining a sorted freelist or by using a doubly-linked freelist and boundary tags. Both decisions have drawbacks. Maintaining a sorted list takes considerable CPU time and many pages will be visited when objects are inserted in order. Although the overhead of doubly-linked freelists is smaller, they require that *three* objects be modified to insert an item in the list (the newly freed object, its predecessor and successor), and these references may be to different pages. In our previous studies [29, 8], we found that most allocation requests were for one of a few different object sizes. Consequently, if an object has already been allocated, coalescing an object when it is deallocated may have little benefit because it will be re-used immediately. This design decision influences both the basic execution time of the algorithms and their reference locality.

To one extent or another, this observation of frequent re-use is exploited by the remaining allocators, which form a distinctly sep-

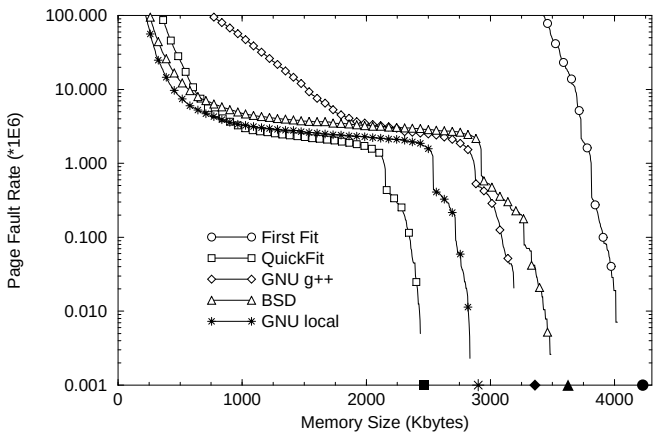

Figure 2: Page Fault Rate for GhostScript (GS) as a function of physical memory size

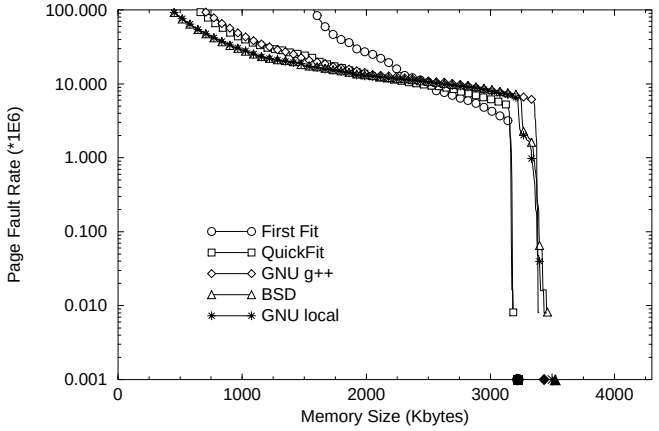

Figure 3: Page Fault Rate for Pascal To C (PTC) as a function of physical memory size

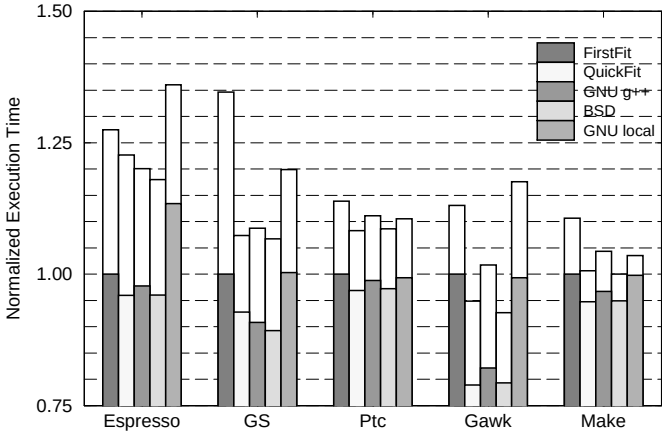

Figure 4: Normalized program execution time with 16K direct-mapped cache, 25-cycle cache miss penalty, overlaid on normalized execution time when we ignore the memory hierarchy.

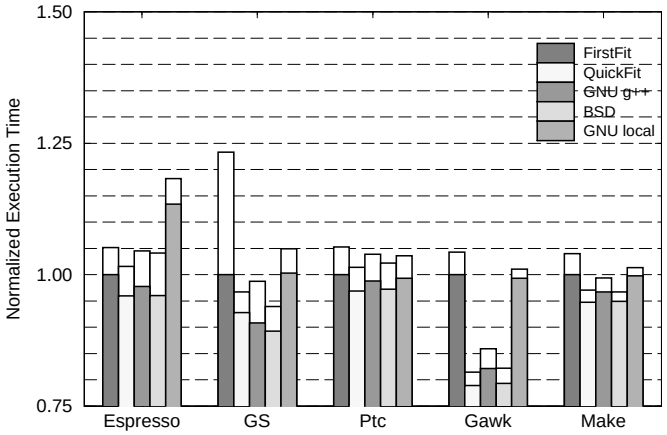

Figure 5: Similar normalized program execution time, assuming a 64K direct-mapped cache with 25-cycle cache miss penalty.

arate class with greater resiliency. However, even in this group, the page fault rate evinces algorithmic differences. The BSD implementation spends relatively little time actually allocating memory, but the BSD implementation suffers from severe internal fragmentation because it allocates $\approx 2^{\lceil \lg N \rceil}$ bytes for an N byte object; much of the allocated space may be wasted. This increases the page fault rate for BSD, because more pages must be resident to access the same set of allocated objects. On the other hand, allocating exactly N bytes implies that the number of *size classes*, or distinct groups of allocation request sizes, will increase. This can increase the CPU time to allocate objects and decrease object re-use.

For some applications, such as GS, increased paging caused by the space wasted by BSD may offset the advantages of the faster algorithm. For other applications, such as PTC, there is little effective difference between the space needed by the different DSA implementations. For example, the design of the GNU LOCAL allocator attempts to explicitly increase reference locality, at considerable expense in execution performance (see Figure 1), but appears to gain little by this careful design.

The impact of DSA design on the page fault rate is dramatic. Some of the DSA design principles that reduce cache misses will also reduce page faults, but the most important DSA design goal should be to reduce space requirements without sacrificing execution speed. We now consider how to improve cache miss rates because it is difficult to expand the size of an existing cache and it is more difficult for programmers to detect the influence of DSA design on cache locality.

4.2 Cache Reference Locality

In a uniprocessor, cache misses occur either because of a "cold start" (when an an item is brought into cache for the first time) or cache overflow (where a referenced item was forced out of the cache to make room for a new item). Because we have simulated reasonably long traces and we are considering moderate cache sizes, we believe that "cold start" misses play a minor role in our study.

Cache misses can be reduced by *prefetching* data. Prefetching may be directed by software [4], but usually arises when cache lines contain multiple words [21] — referencing one word automatically brings other words into the cache. In this paper, we consider only the effect of hardware prefetching.

We first measured a base execution time, ignoring effects of the memory hierarchy. For each application, the execution time for the different DSA implementations was normalized relative to the FIRSTFIT DSA implementation. The shaded values in Figure 4 and 5 show the normalized execution time for each application and allo-

Program	Exec. Time (sec.)	Total Instr. ($\times 10^6$)	Data Refs. ($\times 10^6$)	Max. Heap Size (Kbytes)	Objects Alloc'ed (1000s)	Objects Freed (1000s)
GS-Small	17.0	195	66	1092	109	102
GS-Medium	51.3	539	172	2721	567	551
GS-Large	131.3	1344	421	4129	924	898

Table 3: Characteristics of Different Input Sets for GhostScript

cator. These values are overlaid with the normalized execution time when we include the additional delays introduced by the memory hierarchy. Although some applications execute significantly longer than others, the execution time for different DSA implementations within a single application usually differs by less than 10%-20%, as indicated by Figure 1. The clear blocks overlaying the normalized execution time in Figures 4 and 5 show the normalized execution time for each application when we consider cache misses in small (16K) and medium (64K) direct-mapped caches using a modest cache miss penalty (25 cycles). These figures include only the effects of *data cache misses*. If an application executed I instructions with D data references, a data cache miss rate of M and a miss penalty of P, we estimated the total execution time to be $I + (M \times P)D$. We assume all instructions, including loads and stores, complete in a single machine cycle, and ignore the effects of page faults, instruction cache misses and other cache design issues. Page faults are not considered for reasons described in §4.1, and other effects (such as instruction cache misses) are relatively consistent across the different DSA implementations and less relevant to our study.

Figures 4 and 5 show that the reference locality of DSA implementations can significantly influence the overall execution time of a program. However, there is no single DSA implementation that increases reference locality in any significant way across all applications. Figures 4 and 5 show the effect of cache misses for only a small range of cache sizes and for a single input set for each application. When considering all applications, allocators and cache sizes, the quantity of data precludes an exhaustive presentation, and we will use one application (GhostScript) to illustrate several points in the remainder of this section because Figures 4 and 5 show the most variance for the GhostScript application. We ran the GhostScript application using three different input sets. Table 3 shows that each input set evoked significantly different execution characteristics. Figures 6, 7 and 8 show the cache miss rates for these different input sets while we vary the size of the direct-mapped cache from 16K to 256K.

These figures illustrate several important points for this application. First, there are significant differences in the cache miss rate for different DSA implementations across all the input sets and cache sizes. Quite naturally, these differences are muted for the smaller input set that ostensibly allocates and references a smaller amount of data. For each input set, we see that the DSA implementation with the largest cache miss ratio is FIRSTFIT. Recall that this implementation also suffered from extremely poor page reference locality. The remainder of the DSA implementations have markedly better cache locality, although the other first-fit implementation (GNU G++) has the second highest miss rate. In the other applications, no other DSA implementation has such exceptionally poor behavior.

From this data, we can conclude that searching a freelist, as done by FIRSTFIT and GNU G++, is disastrous for page reference *and* cache locality. As noted, our earlier study [8] indicated that this searching bestowed little advantage on these algorithms; they tend to use slightly less space than other DSA implementations, but at a sizable execution time penalty. Although these allocators have attributes that are theoretically appealing, they are not suitable DSA implementations for modern architectures and programming styles. This is not caused by a naïve implementation; both FIRSTFIT and GNU G++ are well crafted and exploit many of the improvements that have been suggested for first-fit allocators.

The differences between the remaining DSA implementations we considered is inconclusive. Indeed, Figures 6, 7 and 8 show that each implementation has the smallest miss rate in one of the different data sets. This is surprising, because one implementation, GNU LOCAL, expends considerable effort to garner reference locality. Tables 4 and 5 show that, for 16K and especially for 64K direct-mapped caches, this is largely successful. Each table shows the total estimated execution time for each allocator and application and the portion of that execution time attributable to cache misses.[1] The GNU LOCAL allocator *does* reduce delays from cache misses, but it doesn't take advantage of the rapid allocation and deallocation techniques used in the QUICKFIT or BSD allocators and thus the overall execution time for GNU LOCAL is larger than QUICKFIT or BSD. However, for caches with very high miss penalties, the reduced miss rate may have a more significant effect. The algorithmic changes to enhance locality are more effective for moderate size caches. Small caches can not contain enough of the working set and suffer from high cache misses. Large caches contain enough of the working set that *all* algorithms begin to perform well, discounting the additional effort expended by GNU LOCAL. Clearly, "large," "moderate" and "small" are not absolute measure – the important cache sizes depend on the application behavior.

4.3 Design Decisions for Improving Reference Locality

The focus of this paper is to highlight the design principles for developing a DSA implementation that achieves high reference locality. Reference locality, both at the level of pages and caches, is important because it can increase the actual program execution time, as shown by Figures 4 and 5. Efficient storage management for DSA implementations can affect the page reference locality, as shown in Figure 2. Many DSA algorithms, such as FIRSTFIT, GNU G++ and GNU LOCAL attempt to reduce the amount of memory requested from the operating system. In computers lacking virtual memory or having very limited physical memory, this is an appropriate consideration. However, on most modern architectures, space considerations should, within reason, be secondary to overall performance. If an allocator requests memory pages, but they are not actively referenced, then they have little effect on the program performance.

[1] These are *estimated* execution times; however, the execution time for the configuration most similar to our test vehicle (a DECstation-5000/120 with 64K direct-mapped data cache), closely match the actual execution times, some of which are listed in Table 2.

Allocator	ESPRESSO	GS	PTC	GAWK	MAKE
	Total time (sec)/ Miss time (sec)	Total time (sec)/ Miss time (sec)	Total time (sec)/ Miss time (sec)	Total time (sec)/ Miss time (sec)	Total time (sec)/ Miss time (sec)
FIRSTFIT	199.67/43.01	113.13/29.11	26.14/3.19	85.85/9.94	3.93/0.38
QUICKFIT	192.16/41.85	90.18/12.22	24.84/2.62	72.02/12.12	3.57/0.21
GNU G++	188.14/34.94	91.38/15.09	25.50/2.82	77.25/14.87	3.70/0.27
BSD	184.80/34.39	89.65/14.65	24.93/2.62	70.35/10.14	3.55/0.18
GNU LOCAL	213.07/35.40	100.74/16.44	25.36/2.57	89.25/13.84	3.67/0.13

Table 4: Total estimated execution time and time waiting for a 16-kilobyte direct-mapped cache miss in five allocation-intensive programs.

Allocator	ESPRESSO	GS	PTC	GAWK	MAKE
	Total time (sec)/ Miss time (sec)	Total time (sec)/ Miss time (sec)	Total time (sec)/ Miss time (sec)	Total time (sec)/ Miss time (sec)	Total time (sec)/ Miss time (sec)
FIRSTFIT	164.74/8.08	103.60/19.59	24.16/1.21	79.18/3.27	3.69/0.14
QUICKFIT	159.16/8.85	81.29/3.32	23.27/1.04	61.83/1.92	3.45/0.08
GNU G++	163.74/10.55	82.96/6.67	23.83/1.16	65.20/2.82	3.53/0.09
BSD	163.14/12.72	78.95/3.95	23.45/1.15	62.40/2.19	3.43/0.06
GNU LOCAL	185.33/7.67	88.15/3.85	23.77/0.98	76.70/1.29	3.60/0.05

Table 5: Total estimated execution time and time waiting for a 64-kilobyte direct-mapped cache miss in five allocation-intensive programs.

Metric	ESPRESSO	GS	PTC	GAWK	MAKE
GNU LOCAL (w/tags) Miss rate	0.880	0.580	0.600	0.250	0.240
GNU LOCAL (w/tags) Miss penalty (% of total (no tags) exec. time)	5.27	4.51	4.91	1.99	1.78
GNU LOCAL (no tags) Miss rate (%)	0.680	0.560	0.500	0.210	0.200
GNU LOCAL (no tags) Miss penalty (% of total (no tags) exec. time)	4.14	4.37	4.13	1.68	1.49
Penalty due to boundary tags (% of total (no tags) exec. time)	1.13	0.14	0.78	0.31	0.29

Table 6: The effect of boundary tags on execution time in the GNU LOCAL allocator with a 64-kilobye direct-mapped cache. The rows show the miss penalty and execution time degradation for the GNU LOCAL allocator both with and without boundary tags. The final row shows the percentage increase in execution time due to cache misses caused by boundary tags.

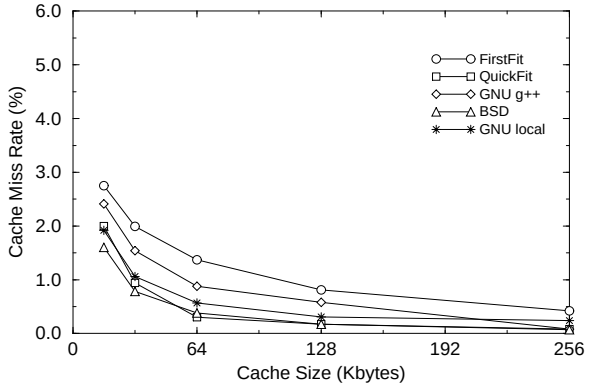

Figure 6: Data cache miss rate for GhostScript (GS-Small)

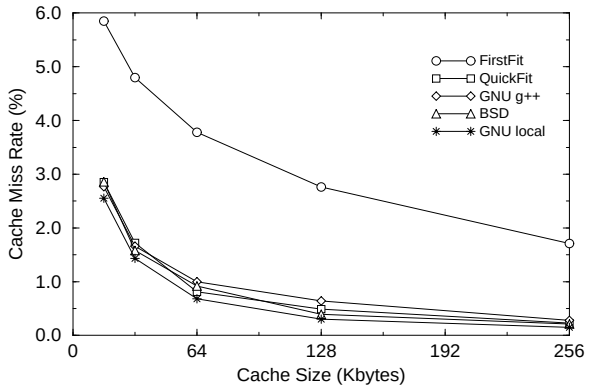

Figure 7: Data cache miss rate for GhostScript (GS-Medium)

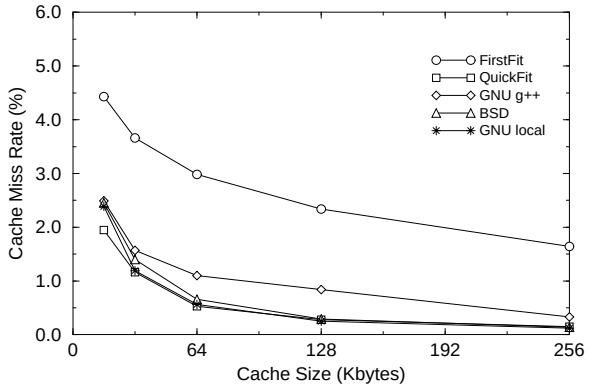

Figure 8: Data cache miss rate for GhostScript (GS-Large)

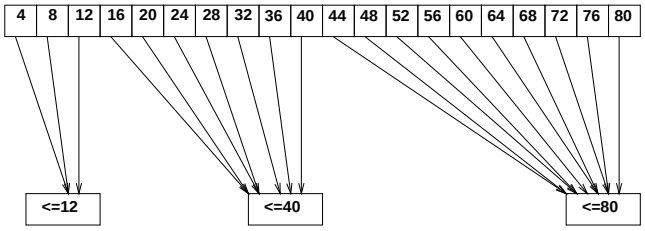

Figure 9: Mapping Allocation Requests

As mentioned earlier, the BSD allocator can waste considerable memory, enough so to warrant the "within reason" caveat. This wasted space occurs because BSD was not designed with sufficient knowledge of the way programs actually behave [29]. Similarly, QUICKFIT is efficient for a pre-determined range of "size classes," determined by experience and anecdotal evidence; various techniques can make QUICKFIT suitable for most programs [8].

We believe the structure of the QUICKFIT allocator should be the foundation for high-performance DSA implementations. Are there ways to improve on QUICKFIT? We believe some techniques used in the GNU LOCAL allocator are applicable, and will further improve the reference locality of the QUICKFIT-style allocators.

Recall that the QUICKFIT allocator manages a subset of the possible allocation request sizes; the remaining requests must be managed by another, more general allocator. When deallocating an object, we must determine which allocator is responsible for that object. The implementation of QUICKFIT we considered uses *boundary tags* to indicate the allocator responsible for the object. Boundary tags "pollute" the cache; the information in the boundary tag is only useful to the allocator, and Figure 1 indicates that a properly designed allocator should contribute only a small fraction to the total execution time. With boundary tags, the memory before and after each allocated object records the size of the object and indicates whether that object is currently allocated or free. This information will typically be brought into the cache when the object is referenced. This wasted space increases the cache miss rate by reducing the effectiveness of prefetching; boundary tags, rather than useful data, may be prefetched. Our measurements in related work [30] confirm previous observations that programs tend to allocate many small objects; we found that 24 bytes was a very common allocation request size. Common boundary tag implementations add eight bytes of overhead information to each allocated object; thus, $\approx$25% of the cache may hold information useful only to the memory allocation subroutines.

As mentioned in §2.1, the GNU LOCAL allocator does not use boundary tags. Not only does this reduce the total memory needed because less memory is devoted to boundary tags, it also slightly improves the reference locality. Table 6 shows the cache miss rates for a modified version of the GNU LOCAL allocator that allocates an additional eight bytes of data for each object. This extra space emulates the effect of cache pollution by the boundary tags without otherwise influencing the DSA implementation. Boundary tags increase *total* execution time by 0.1%-1.1%, and the contribution would increase as cache miss penalties increase.

4.4 An Architecture for Efficient Memory Allocation

We can use the results of our experimentation to guide the design of efficient DSA implementations by discarding inefficient designs and identifying the most important characteristics present that provide efficiency. The inefficient design elements are:

Search & Coalescing: Algorithms that search for free space, such as FIRSTFIT and GNU G++ are generally slower than other algorithms and have poor reference locality. Fortunately, many programs do not need these general allocators for *every* allocation; most programs have patterns of behavior that can be handled in more efficient ways. However, these algorithms *are* needed to allocate infrequently allocated objects or objects that deviate from the "normal" program behavior.

Explicit Cache Management: Our measurements show that the GNU LOCAL allocator, which was carefully crafted to provide good cache locality, did not have miss rates significantly lower that the BSD or QUICKFIT algorithms. Even though the miss rates of the GNU LOCAL allocator were sometimes marginally lower that the BSD or QUICKFIT allocators, the added CPU overhead in the GNU LOCAL allocator resulted in longer total execution times based on cache miss penalties associated with existing computer architectures. In the future, if cache miss penalties increase dramatically, the added CPU overhead required to obtain the marginal increase in locality may then be warranted.

Our measurements have also shown that techniques intended primarily to decrease execution time also result in increased reference locality. In particular, segregated-storage allocators such as BSD and QUICKFIT solve two problems simultaneously: they allow very rapid allocation and deallocation and at the same time they promote rapid object re-use leading to higher reference locality.

A central aspect of the design of segregated-storage algorithms is the choice of object sizes handled by the fast freelists. Algorithms can choose to merge many object sizes together (e.g., rounding up to a power of two in the BSD allocator) handling the entire class of sizes with a single freelist, or algorithms can handle each distinct size request with a different freelist. Merging sizes enhances rapid object re-use but wastes storage space due to internal fragmentation. The BSD algorithm is an example of an allocator that shows good re-use but excessive fragmentation. The alternative, using many distinct size freelists, reduces object re-use but eliminates internal fragmentation problems.

The best allocator strikes a balance between too few and too many size classes. The choice of these size classes can be based on several approaches. First, anecdotal evidence about the best choice of size classes can be used and that kind of evidence was the basis for the QUICKFIT implementation that we measured. Second, size classes can be chosen in such a way that the amount of internal fragmentation is bounded (e.g., if 25% or less internal fragmentation is tolerated, then objects of size 12–16 bytes are rounded to 16 bytes) [5]. Finally, we advocate basing the choice of size classes on empirical measurements of a particular program's behavior. In previous work [8], we have shown that allocator "customization" results in very fast allocators. Such customized allocators could also be designed to promote the most effective object re-use, leading to enhanced cache locality.

Implementing non-uniform size-merging operations requires allocators to implement an arbitrary mapping between the object request size and the associated size class size. One reason for the crude powers-of-two mapping used in the BSD algorithm is that it is easy to compute. However, arbitrary mappings can be implemented efficiently using a size-mapping array, as illustrated in Figure 9. With such an array, size requests can be rounded-up to arbitrary sizes.

One way to reduce the space overhead and increase the reference locality of programs is to eliminate the per-object boundary tag information for objects needed in the BSD and QUICKFIT implementations. While the GNU LOCAL implementation does eliminate per-object tags, measurements in Table 6 show that the cache performance improvement associated with eliminating even 8-byte boundary tags is quite small. We conclude that boundary-tag elimination has mixed performance advantages on current architectures and is not warranted if the elimination increases the cost of allocation and deallocation significantly.

5 Conclusions

In this paper, we investigated the effect of dynamic storage allocation on program reference locality. We conclusively showed that the choice of DSA algorithm has a strong impact on program reference locality. This impact can significantly reduce program performance in modern computer architectures.

Using trace-driven simulation, we measured the cache miss rates and page fault rates for a broad range of cache and memory sizes in five allocation-intensive programs. We conclude that allocators based on sequential-fit methods, such as first-fit, best-fit, etc, have poor reference locality. Even though these algorithms reduce the total memory requirements of programs, this reduction does not result in increased reference locality.

We conclude that efforts to reduce total memory utilization in DSA implementations, such as coalescing adjacent free blocks, will in most cases both increase total execution time and reduce program reference locality. Our measurements show that the most CPU efficient allocators, such as BSD or QUICKFIT, also provide the best locality of reference. Locality in these algorithms is enhanced because they are designed on the principle that programs allocate objects with a small number of distinct sizes, and the allocators rapidly recycle free objects of those sizes.

5.1 Future Work

We are extending our previous work in synthesized allocators using the information we gained in this study. Ideally, our general-purpose allocator will work well for many programs and can be improved further using customization. We also hope to include other work in program behavior prediction based on *call site* information [2] in the synthesized allocators.

6 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grants No. CCR-9010624, CCR-9121269 and CDA-8922510. We would like to thank James Larus for development of the QP utility, which greatly simplified our experimentation, and Doug Lea, Mike Haertel and Mark Moraes for the use of and information about their allocators. We also thank the reviewers for their comments.

References

[1] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[2] David Barrett and Benjamin Zorn. Using lifetime predictors to improve memory allocation performance. In *SIGPLAN'93 Conference on Programming Language Design and Implementation*, Albuquerque, June 1993.

[3] Gerald Bozman. The software lookasize buffer reduces search overhead with linked lists. *Communications of the ACM*, 27(3):222–227, March 1984.

[4] David Callahan, Ken Kennefy, and Allan Porterfield. Software prefetching. In *Fourth Intl. Conf. on Arch. Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[5] John DeTreville. Heap usage in the Topaz environment. Technical Report 63, Digital Equipment Corporation System Research Center, Palo Alto, CA, August 1990.

[6] Digital Equipment Corporation. *Unix Manual Page for PIXIE*, ULTRIX V4.2 (rev 96) edition, September 1991.

[7] A. J. Goldberg and J. Hennessy. Performance debugging shared memory multiprocessor programs with MTOOL. In *Proceedings Supercomputing '91*, pages 481–491, 1991.

[8] Dirk Grunwald and Benjamin Zorn. CUSTOMALLOC:Efficient Synthesized Memory Allocators. Technical Report CS-CS-602-92, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, July 1992.

[9] Mike Haertel. Description of GNU malloc implementation. Personal communication, August 1991.

[10] Mark D. Hill. *TYCHO*. University of Wisconsin, Madison, WI. Unix manual page.

[11] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990.

[12] Chris Kingsley. Description of a very fast storage allocator. Documentation of 4.2 BSD Unix malloc implementation, February 1982.

[13] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, chapter 2, pages 435–451. Addison Wesley, Reading, MA, 2nd edition, 1973.

[14] David G. Korn and Kiem-Phong Vo. In search of a better malloc. In *Proceedings of the Summer 1985 USENIX Conference*, pages 489–506, 1985.

[15] M.S. Lam, P. W. Wilson, and T. G.Moher. Object type directed garbage collection to improve locality. In *Proceedings of the International Workshop on Memory Management*, St. Malo, FRANCE, September 1992. Springer Verlag.

[16] Doug Lea. An efficient first-fit memory allocator. (From comments in source and personal communication).

[17] Alvin R. Lebeck and David A. Wood. CPROF: A cache performance profiler. Technical report, Computer Sciences Dept., Univ. of Wisconsin—Madison, July 1992.

[18] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[19] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 75–84, Santa Clara, CA, April 1991.

[20] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[21] A.J. Smith. Line (block) size choice for CPU cache memories. *IEEE Transactions on Computers*, 36(9):1063–1075, September 1987.

[22] Thomas Standish. *Data Structures Techniques*. Addison-Wesley Publishing Company, 1980.

[23] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[24] Charles B. Weinstock and William A. Wulf. Quickfit: An efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–144, October 1988.

[25] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generation garbage collection. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 32–42, San Francisco, CA, June 1992. ACM.

[26] Paul R. Wilson, M.S. Lam, and T.G. Moher. Effective 'static graph' reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 177–191, Toronto, Ontario, Canada, June 1991.

[27] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, Berkeley, CA, November 1989. Also appears as tech report UCB/CSD 89/544.

[28] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, May 1991.

[29] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Notices*, 27(12):71–80, December 1992.

[30] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. Technical Report CU-CS-603-92, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, July 1992.