Utilization of Separate Caches to Eliminate Cache Pollution Caused by Memory Management Functions

Mehran Rezaei and **Krishna M. Kavi**
The University of North Texas
{mehran, kavi}@cs.unt.edu

Abstract

Data intensive service functions such as memory allocation/de-allocation, data prefetching, and data relocation can pollute processor cache in conventional systems since the same CPU (using the same cache) executes both application code and system services. In this paper we show the improvements in cache performance that can result from the elimination of the cache pollution using separate caches for memory management functions. For the purpose of our study we simulate the existence of separate hardware units for the application and the memory management services using two Unix processes. One process executes application code (simulating main CPU) while the other executes memory management code. We collected address traces for the two processes and used Dinero IV cache simulator to evaluate the expected cache behaviors. A second goal of this paper is to examine the cache performance of different memory allocators. In this paper we compare two allocators: a very popular segregated list based allocator (originally due to Doug Lea) and our own binary-tree based allocator (called Address-ordered Binary Tree).

Keywords: Memory Management, Memory Latency, Binary Tree

1. Introduction

The research presented in this paper is a by-product of our ongoing work in overcoming the ever-increasing speed gap between CPUs and memories. Besides exploring multithreading [2, 7-9, 21], we have been investigating the use of intelligent memories (i.e., IRAMs[13, 17, 18]) to tolerate the CPU – Memory speed deference. In our research, we are investigating the use of a small processor in memory like IRAMs, but we employ the processor to aid the main CPU in executing memory management operations. Data references made by applications conflict with the data references made by memory management operations and cause cache misses. The cache misses that are not due to the locality behavior of applications are the main source of cache pollution. Use of a separate processor in DRAM for memory service operations can reduce (or eliminates) such cache pollutions..

In order to evaluate the effectiveness of our approach before actually developing an intelligent RAM unit, in this paper we investigate the cache misses caused by memory management operation. Object-oriented and linked-data structure applications invoke allocation/de-allocation functions frequently. These memory management functions are themselves very data intensive in terms of managing available and live memory objects. However, the data manipulations involve only addresses (integer arithmetic). Therefore, a simple integer CPU embedded inside a RAM offers a viable option for migrating the allocation/de-allocation functions from the main CPU to IRAM, and for improving application performance by eliminating cache pollution. In this paper we show that moving allocation/de-allocation functions from the main CPU to IRAM eliminates, on average, 26% of actual cache misses (as compared to the current situation where allocation/de-allocation functions are performed by the main CPU). A second result of this paper is to compare the performance of different general-purpose memory allocators in terms of their cache behaviors. Here we compare two allocators – a very popular allocator used by Unix systems (originated by Doug Lea [14]) and our own allocator that uses binary trees [10, 19].

2. Research Background

Although our overall goal is to develop a separate processor inside a memory chip (i.e., IRAM like), in this paper we are only investigating the amount of cache pollution that can be eliminated by considering a separate cache for memory management (regardless if the actual memory management functions are delegated to an IRAM processor or not). As such in this section we will provide a brief introduction to memory management techniques.

2.1. Allocation Techniques

Dynamic memory management is an important problem studied by researchers for the past several decades. Every so-often the need for more efficient implementation of memory allocation, both in terms of memory usage and execution performance becomes acute leading to newer techniques. The need for more efficient memory management is currently being driven by the popularity of object-oriented languages in general, and Java in particular [1, 3].

An allocator's task is to organize and track the free chunks of memory as well as memory currently being used by the running process (i.e., live objects). The two primary aims of any efficient memory manager are high storage utilization and fast execution performance [22]. Well-known placement policies such as best-fit and first-fit have been utilized within most modern memory allocators. However, current implementations have failed to achieve both aims at the same time. For example, sequential-fit algorithms show high storage utilization [6], but poor execution performance [19]. On the other hand, segregated free lists cause higher fragmentations, but their performance is the best among allocators [22].

Currently used memory allocation schemes can be classified into sequential fit algorithms, buddy systems, segregated free lists, and binary tree algorithms. ***Sequential fit*** approach (including first-fit and best-fit) keeps track of available chunks of memory on a doubly linked list. When a process releases memory, these chunks are added to the free list, either at the head or in place, if the list is sorted by addresses (Address Order [22]); freed chunk may be **coalesced** with adjoining chunks to form larger chunks of free memory. When an allocation request arrives, the free list is searched until an appropriate chunk is found. The memory is allocated either by granting the entire chunk or by **splitting** the chunk (if the chunk is larger than the requested size). Best-fit methods [12] try to find the smallest chunk that is at least as large as the request. First-fit [12] methods find the first chunk that is at least as large as the request. Best-fit method may involve delays in allocation while first-fit method may lead to more external fragmentation [6].

In ***buddy systems*** [11, 12], the size of any memory chunk (live, free or garbage) is 2^k for some k. Two chunks of the same size that are next to each other in terms of their memory addresses are known as buddies. If a newly freed chunk finds its buddy among the free chunks, the two buddies can be combined into a larger chunk of size 2^{k+1}. During allocation, larger chunks are split into equal sized buddies, until a small chunk that is at least as large as the request is created. Large internal fragmentation is the main disadvantage of this technique. It has been shown that as much as 25% of memory is wasted due to fragmentation in buddy systems [6]. An alternate implementation, *double buddy*, which creates buddies of equal size, but does not require the sizes to be 2^k, is shown to reduce the fragmentation by half [6, 23].

Segregated free list approaches maintain multiple linked lists, one for each different sized chunks of available memory. Returning a free chunk from one of the lists satisfies allocation requests (by selecting a list containing chunks, which are at least as large as the request). Freeing memory, likewise, will simply add the chunk to the appropriate list. Segregated free lists are further classified as: ***simple segregated storage*** and ***segregated fit*** [22]. No coalescing or splitting is performed in simple segregated storage and the size of chunks remains unaltered. If a request cannot be satisfied from its associated sized list, additional memory from operating system is requested via **sbrk** or **mmap** system calls. Segregated fit allocator, instead, attempts to satisfy the request from a list containing larger sized chunks – a larger chunk is split into several smaller chunks. As can be seen, simple segregated storage is faster than segregated fit. However, the splitting and coalescing of chunks employed by segregated fit algorithms may lead to better storage utilization and result in fewer **sbrk** or **mmap** system calls.

Cartesian tree is an allocation technique proposed almost two decades ago [20]. In this method, free chunks of memory are kept on a binary tree instead of a linearly linked list. Cartesian tree uses both the starting address of the chunks (i.e., Address Ordered Tree) and the size of the chunks to construct the binary tree (it is called Cartesian tree because it uses both size and address to form the binary tree).

A Cartesian tree must satisfy the following conditions.

a. address of descendents on left (if any) ≤ address of parent ≤ address of descendents on right (if any)
b. size of descendents on left (if any) ≤ size of parent ≥ size of descendents on right (if any)

The second condition requires the largest free chunk to be at the root of the tree. In our research, we have developed a variation to the Cartesian tree and we will describe it in next subsection.

2.2. Binary Tree Memory Manager

The size condition of the Cartesian Tree allocator that mandates the tree to have its largest node at the root of the tree causes the tree to usually become unbalanced, and possibly degrading into a linked list. In our approach to the memory management, the free chunks of memory are also maintained in binary tree similar to a Cartesian tree [10, 19]. However, we remove the size condition (condition b). In our implementation the tree is ordered by addresses and each node of the tree contains the size of the largest chunk of the memory available in its left and right sub-tree. This information can improve the response time for memory allocation requests. This information can also be used for implementing better-fit policies, which can improve memory utilization [6]. Since Address Ordered Binary tree is ideally suited for coalescing of free chunks, memory utilization is further enhanced.

3. Experimental Framework

To investigate the cache pollution caused by CPU-resident memory management functions including allocation/de-allocation; and to compare the cache performance of different allocators, we have conducted our experiments on Dec Alpha 21164 processors, running Digital Unix. First we have used a single process (simulating conventional systems using a single CPU) to

run both the application code and memory allocator code. The benchmarks are instrumented using ATOM [5] to collect load and store traces. In the next step, we have used two separate processes -- one process executes the application code while the other executes allocator code (simulating the use of a separate processor for memory management, which can potentially be embedded in a DRAM chip). We have used ATOM to instrument allocator and application processes to collect the load/store traces separately for each process. We have then used Dinero IV cache simulator [4] to evaluate the cache memory performance in the two cases of our experiment.

For our studies we have used 4 benchmarks that are popular for evaluating memory allocation techniques. These benchmarks are all written in C and briefly explained in Table 1.

Benchmarks	Description	Inputs
boxed-sim	balls in box simulator	-n 10 –s 1
Cfrac	it factors numbers.	a 36-digit number
Ptc	Pascal to C converter	mf.p
espresso	PLA optimizer	largest.espresso

Table 1. Benchmark descriptions and their inputs

In our experiments we have used two general-purpose allocators.

Doug Lea's allocator *– perhaps the most widely used allocator. We used version 2.7.0 of this allocator, which benefited from several years of optimizations. It is an efficient hybrid allocator with respect to the size of objects [14]. For sizes less than 512 bytes it uses simple segregated storage technique. The different sizes differ in increments of 8 bytes. For sizes greater than 512 bytes and less than 128 Kbytes it uses segregated fit allocator. And it keeps the rests in a sequential free list (this algorithm is labeled as lea in our figures).*

ABT*: Address-ordered Binary Tree, which is described in the previous section (this algorithm is labeled as abt in our figures).*

4. Empirical Results

We have collected traces for two scenarios:

Con-Conf*: Conventional Configuration, in which case both application and its allocator are running on the main CPU (in our experiment this is simulated using a single process for running both the application code and allocator code).*

IMM*: Intelligent Memory–Manager, where a separate process executes memory management functions. This situation is simulated in our experiment utilizing two separate processes.*

As mentioned in section 3, we have emulated the use of a separate processor embedded in a DRAM chip for memory allocation functions using two separate (Unix) processes. The processes have used shared segments to communicate memory allocation and de-allocation requests (and responses). This inter-process communication and synchronization has added overhead in terms of accesses to shared segments. Each time the allocator process runs out of memory it issues an **mmap** system call (with shared flag) to acquire more memory form operating system. It then sends the mapped address and its size to application process. Application process finally calls **mmap** to map the same address to its address space, further adding to memory references. The number of **mmap** system calls is a function of allocator behavior. An allocator with better memory utilization requires fewer **mmap** calls.

4.1. Comparison of Cache Localities

The major contribution of this paper is to affirm that removing the dynamic memory allocator can lead to improved processor cache performance (and eliminate processor cache pollution by the allocator functions). For the data shown in this section we have used Dinero IV cache simulator. Figure 1 depicts the actual number of cache misses using 8 Kbytes cache with 32-byte blocks and Figure 2 shows the miss rates.

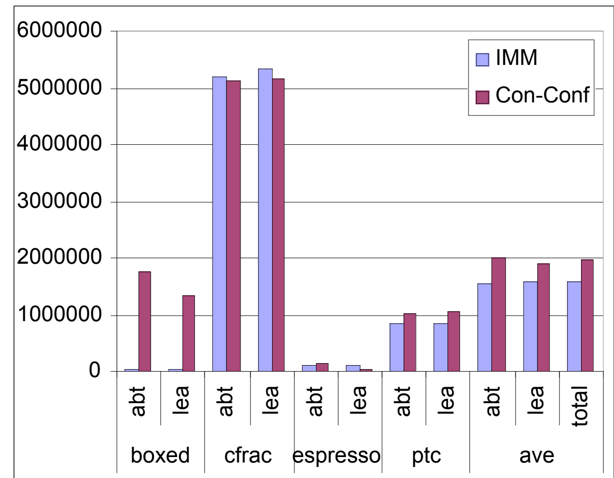

Figure 1. total number of cache (8 Kbytes direct map cache and 32 bytes line size)

Figure 1 shows that on average about 26% of all cache misses are eliminated. Some computationally intensive applications such as cfrac show less drastic reduction in cache misses, while applications that are intensive in terms of allocation/de-allocation requests like boxed-sim show significant reduction in cache misses (since the allocator code is removed from the application code). As stated previously, our experiment has added overhead to the total number of memory references caused by the application since our experiment used shared memory segment for simulating communication between the allocator and the application. When a real hardware implementation is used (removing the overhead

memory accesses), we claim that the number of cache misses caused by an application will show even more dramatic reductions.

Figure 2 depicts the cache **miss rates** for applications using a single process (Con-Conf) and two processes (IMM). The figure shows that the IMM configuration reports dramatically reduced cache miss rates for all applications using either allocator technique. On average the data shows a reduction of 70% in cache miss-rates across all applications. Once again, our experimental artifact causes some of the cache misses. When these misses are eliminated (using hardware for allocator), we feel that the cache miss rates can show even greater reductions.

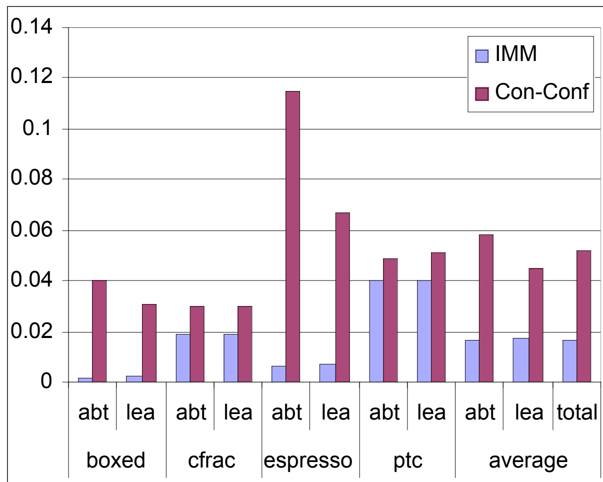

Figure 2. Cache miss rate (8 Kbytes direct map cache and 32 Bytes line size)

As can be seen from Figure 2, ABT allocator behaves better than Doug Lea's allocator. The amount of cache pollution caused by an allocator depends on how data intensive the allocator is. The headers used with memory objects for the purpose of maintaining free chunks by the allocator are the main causes of cache pollution. Thus the cache pollution is related to the header size used by a specific allocator. Our allocator ABT uses more header information than other allocators, causing greater cache pollution. ABT can potentially traverse several levels of the binary tree to find an appropriate chunk of memory for allocation. This in turn can lead to more cache accesses by the allocator, hence greater cache pollution. Doug Lea's allocator is highly optimized and efficient – hence it causes less pollution than other allocators. However when we utilize a separate processor for memory management, the memory accesses of the allocator are separated from the processor cache – obviating the advantages of Doug Lea's allocator (at least in terms of cache behavior).

4.2. Impact of Cache Parameters.

Figure 3 and 4 are used to show the impact of cache line size and cache capacity. As usual, increasing cache line (or block) size increases locality and thus reduces miss rate. This improvement is uniform across all benchmarks and allocators. But the figure shows more dramatic improvements for IMM when using a separate hardware allocator and a separate cache.

The effect of keeping the line size at 32 Bytes and increasing the cache size is shown in Figure 4. Since ABT is more data intensive (and use more header information), increased cache size is less beneficial for these allocators than increased line size.

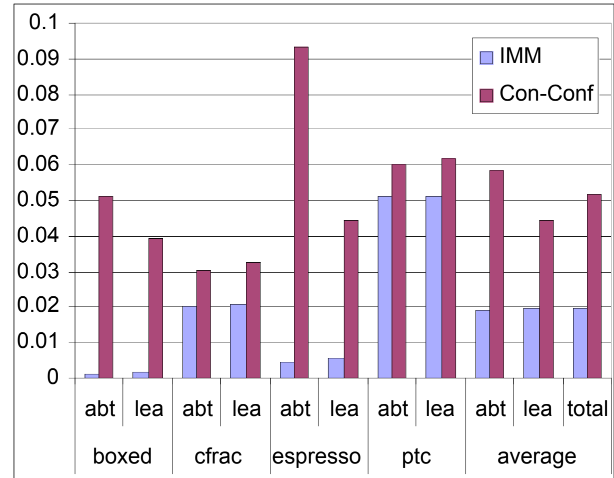

Figure 3. Cache Miss Rate (8 Kbytes direct map cache with 64 Bytes line size)

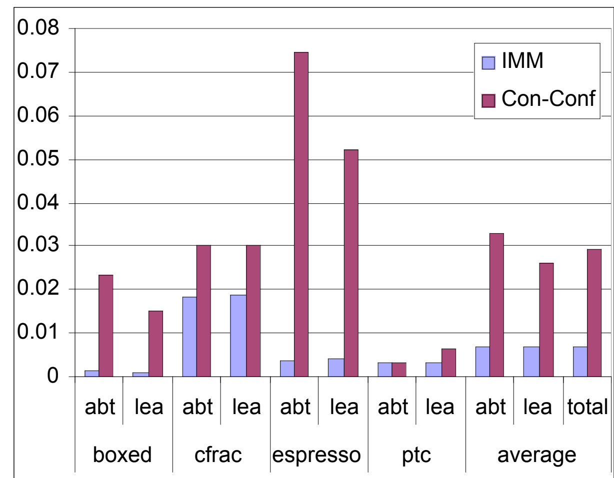

Figure 4. Cache Miss Rate (16 Kbytes direct map cache with 32 Bytes line size)

4.3. Comparing Cache Behaviors of Allocators.

We now directly compare the cache behaviors of the allocators. This is achieved by collecting address traces for the (separate) allocator process for the two allocators in our experimental framework. Figure 5 shows the cache performance for the two allocators chosen for this paper. Even with a very small cache in Intelligent Memory (1

Kbytes, for use only by the memory management functions), very few misses are reported for either allocator. Allocator's task is to keep track of free chunks of memory in a list. Either allocation or free involves only address or size modifications. If an allocator reuses recently freed objects, the cache performance of the allocator is improved since the headers of the recently freed objects may still be in the allocator cache. This is the case with simple segregated storage allocators (such as Doug Lea's allocator). Using coalescing of freed objects can also lead to better localities since we will be using recently freed objects to satisfy requests (indirect effect). This is the case with ABT.

The data shown in Figure 5. includes overhead memory references because of our experimental framework (caused by inter-process communication between the application and allocator processes via shared segments). As noted before when these overhead references are removed using separate hardware processors for allocator and main CPU (and use memory bus for communication), we feel that the allocator cache need only be very small (say a few kilo bytes) to achieve very few cache misses (and low miss rates).

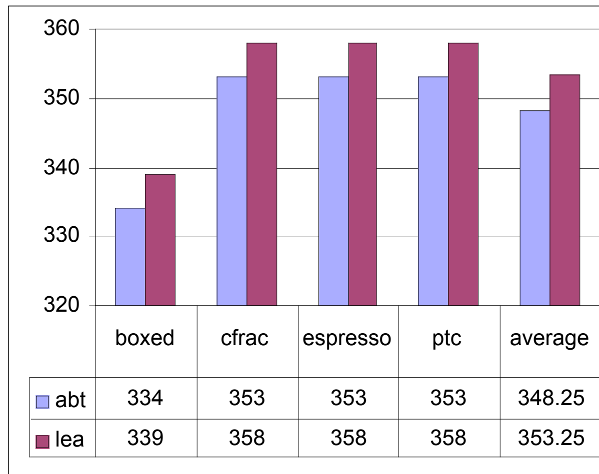

Figure 5. IMM-allocator cache misses (1 Kbyte Direct map cache with 32bytes line size)

5. **Conclusions**

As the performance gap between processors and memory units continues to grow, memory accesses continue to inhibit high performance on modern processors. While memory hierarchies utilizing cache memories can alleviate the performance gap somewhat, cache performance is often adversely affected by service functions such as dynamic memory management. Modern applications rely heavily on linked lists and object-oriented programming and this requires sophisticated dynamic memory management services, including allocation, de-allocation, garbage collection, data prefetching and object relocatio. Using a single CPU (with its cache) for executing both service related functions and application code often leads to poor cache performance. Sophisticated service functions need to traverse user data objects—and this requires the objects to reside in cache even when the application is not accessing them.

Our research is motivated by these observations. We feel that, with a simple integer processor and a small cache included on DRAM chips (similar to IRAM devices), we can offload memory management functions from main CPU, thus eliminating the pollution of processor caches. In this paper we have demonstrated this contention by utilizing a simulated environment. We have investigated the use of different memory allocators and their cache behaviors. Our research shows that on the average the use of a separate processor and separate cache for memory allocation and de-allocation functions can eliminate 26% of actual reference misses and lead to 70% reduction in miss rates. Our experimental framework caused some additional memory references by both the application and the allocator, since inter-process communication between the allocator process and application process has used shared memory segments. We feel that even more dramatic cache performance improvements will result when real hardware is used for the two processors.

We also have explored the amounts of cache pollution caused by different memory allocation techniques. Some techniques have resulted in more pollution, but these allocators have shown other benefits in terms of allocation speed or memory utilization. Since the use of a separate hardware processor can eliminate the cache pollution caused by an allocator, we can benefit from the advantages of more sophisticated memory managers. Other dynamic service functions such as Jump Pointers to prefetch linked data structures and relocation of closely related objects to improve localities can also cause cache pollution if a single CPU is used – such service functions drag the objects through the processor cache. Again these functions can be safely offloaded to the Intelligent Memory Unit in order to benefit from their performance advantages without suffering from degraded cache performance. We will extend our experiments to investigate the increase in cache performance for these management functions.

6. References

[1] S. E. Abdullahi and G. A. Ringwood. "Garbage Collection the Internet: A survey of distributed garbage collection", *ACM Computing surveys*, Sept. 1998, pp 330-373.

[2] A. Agrawal, B. - H. Lim, D. Kranz, and J. Kubiatowicz. "April: A Processor Architecture for

Multiprocessing", In *Proc. 17th ISCA*, pp 104-114, May 1990.

[3] E. D. Berger, B. G. Zorn, and K. S. Mckinley. "Comparing High Performance Memory Allocators", In *Proc. PLDI'0*, pp 114-124, June 2001.

[4] J. Elder and M. Hill. "Dinero IV Trace driven uniprocessor cache simulator", University of Wisconsin (http://www.cs.wisc.edu/~markhill).

[5] A. Eustance and A. Srivastava. "ATOM: A flexible interface for building high performance program analysis tools" *DEC Western Research Laboratory, TN-44*, 1994.

[6] M.S. Johnstone and P.R. Wilson. "The memory fragmentation problem: Solved?" *Proc. Of the International Symposium on Memory Management*, Vancouver, British Columbia, Canada, October 1998, pp 26-36.

[7] K.M. Kavi, R. Giorgi and J. Arul. "Scheduled Dataflow: Execution paradigm, architecture and performance evaluation", *IEEE Transactions on Computer*, Vol. 50, No. 8, pp 834-846, Aug. 2001.

[8] K.M. Kavi, J. Arul and R. Giorgi. "Performance Evaluation of a Non-Blocking Multithreaded Architecture for Embedded, Real-Time and DSP Applications", *Proceedings of the ISCA PDCS-2001*, Dallas Texas, August 8-11, 2001, pp 365-371.

[9] K.M. Kavi, J. Arul and R. Giorgi. "Execution and cache performance of the Scheduled Dataflow Architecture", *Journal of Universal Computer Science*, Special Issue on Multithreaded and Chip Multiprocessors, Oct. 2000, pp 948-967, Vol. 6, No. 10.

[10] K.M. Kavi, M. Rezaei and R. Cytron. "An efficient memory management technique that improves localities", *Proc. 8th International Conference on Advanced Computing and Communications (ADCOM 2000),* Cochin, India, Dec. 14-16, 2000, pp 87-94.

[11] K.C. Knowlton. "A fast storage allocator", *Communications of the ACM*, Oct. 1965, pp 623-625.

[12] D.E. Knuth. *The Art Of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, 1973.

[13] C. E. Kozyrakis and D. A. Patterson. " A New Direction for Computer Architecture Research", *IEEE Computer*, pp 24-32, November 1998.

[14] Doug Lea. "A Memory Allocator", (http://g.oswego.edu/dl/html/malloc.html).

[15] T. McDonald. "Researchers claim new chip technology beats Moore's law", *NEWSFACTOR Network*, June 28th 2002.

[16] R. Boyed – Merrit. "What will be the legacy of RISC?" – An Interview with David A. Patterson – *EETIMES*, May 12th 1997, Issue 953.

[17] Y. Nunomure, T. Shimizu, and O. Tomisawa, "M32R/D – Integrating DRAM and Microprocessor", *IEEE Micro*, pp 40-48, November/December 1997.

[18] M. Oskin, F. T. Chong, and T. Sherwood. "Active Pages: A Computation Model for Intelligent Memory", In *proc. 25th ISCA*, pp 192-203, April 1998.

[19] M. Rezaei and K.M. Kavi. "A new implementation for memory management", *Proceedings of the IEEE Southeastcon 2000 Conference*, April 7-9, 2000, Nashville, TN.

[20] C. J. Stephenson. "Fast Fit: New methods for dynamic storage allocation", In *proc. 9th SOSP*, pp 30-32, October 1983.

[21] D. M. Tullsen et al. "Exploiting Choices: Instruction fetch and issue in an implementable simultaneous multithreading processor", In *proc. 23th ISCA*, pp 191-202, May 1996.

[22] P.R. Wilson, et. Al. "Dynamic storage allocation: A survey and critical review" *Proc. Of 1995 International Workshop on Memory Management*, Kinross, Scotland, Springer-Verlag LNCS 986, pp1-116.

[23] D.S. Wise. "The double buddy-system". Technical Report 79, Computer Science Department, Indiana University, Bloomington, IN, Dec. 1979.

[24] B.Zorn www.cs.colorado.edu/~zorn/Malloc.html #bsd.

Acknowledgement. This research is supported in part by a NSF grant, ITR 0081214 (subcontracted through Washington University in St. Louis).