Available online at www.sciencedirect.com

SCIENCE @ DIRECT®

Journal of Systems Architecture 52 (2006) 41–55

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

Intelligent memory manager: Reducing cache pollution due to memory management functions

Mehran Rezaei *, Krishna M. Kavi

University of North Texas, Department of Computer Science and Engineering, P.O. Box 311366, Denton, TX 76203, USA

Received 29 May 2003; received in revised form 5 February 2004; accepted 22 February 2005
Available online 23 May 2005

Abstract

In this work, we show that data-intensive and frequently-used service functions such as memory allocation and de-allocation entangle with application's working set and become a major cause for cache misses. We present our technique that transfers the allocation and de-allocation functions' executions from main CPU to a separate processor residing on chip with DRAM (Intelligent Memory Manager). The results manifested in the paper state that, 60% of the cache misses caused by the service functions are eliminated when using our technique. We believe that cache performance of applications in computer system is poor due to their indulgence for the service functions.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Memory management; Memory latency; Intelligent memories; Address ordered binary tree; Segregated free lists

1. Introduction

The speed gap between CPU and memory continues to widen and memory latency continues to grow due to the fact that CPU speed grows with much faster rate than memory speed grows with [10]. Recent studies by Stephen Chou from Princeton University show that we might be able to beat Moore's law [21], in which case memory speed will lag farther behind and consequently memory latency will become even more pronounced. He, Stephen Chou, claims that his new invention, LADI (laser assisted direct input)—a new technology that may replace photolithography[1] increases chip's density by a factor of 100.

Standard techniques such as deeper memory hierarchies and larger on chip caches fail to tolerate memory latency, mainly because application's data sizes are growing and programming styles

* Corresponding author. Tel.: +1 940 390 9327; fax: +1 940 565 2799.
E-mail addresses: mehran@acm.org (M. Rezaei), kavi@cs.unt.edu (K.M. Kavi).

[1] The process of transferring geometric shapes onto a mask on the surface of a silicon wafer.

doi:10.1016/j.sysarc.2005.02.004

are changing. Nowadays, programmers practice the use of linked data structures, which requires dynamic memory allocation. The proximity of storage layout of such applications does not imply the same degree of spatial locality that array based applications' does.

More recent approaches such as Multithreading [2,23,31], Memory Forwarding [20], Prefetching [6,19,22], and Jump Pointers [29] have been explored to address memory latency in pointer based applications. Multithreading tends to combat latency by passing the control of execution to other threads when a long latency operation is encountered. Prefetching tries to predict the access patterns of data ahead of time and bring what is needed into the cache. Jump Pointers provide direct access to non-adjacent nodes in linked data structures to alleviate the shortcomings of prefetching when insufficient work is found to cover the prefetching latency. Memory Forwarding relocates non-adjacent nodes to adjacent memory spaces. Multithreading requires parallelism in the applications and extra hardware that manages threads and switching among threads. Software-Controlled Multithreading relaxes the hardware complexity by shifting the responsibility of switching time decision from hardware to software and enhances the performance when application lacks parallelism. Prefetching, Jump Pointers, and Memory Forwarding do not require parallelism in an application, but hardware or software overhead is added to the system. Hardware overhead decelerates the clock ("simpler is faster"), whereas software overhead uses CPU cycles and pollutes the cache.

Our approach of tolerating memory latency is originally motivated by a different trend in the design of systems, i.e., Intelligent Memory Devices such as Berkeley IRAM [26] and Active Pages [25]. Similar to Active Pages, within the DRAM chip, in our research we use a small processor as an aid to the main CPU, particularly for dealing with memory related operations. These operations include memory management, prefetching of data, management of Jump Pointers, and relocation of data (i.e., Memory Forwarding) to improve locality. Thus we address memory latency using memory hierarchies by better utilizing hierarchies to improve the performance. Using a separate processor integrated on chip with DRAM for these operations reduces (or eliminates) the possibility of cache pollution inherent in current approaches.

In order to evaluate the efficacy of our approach before actually developing an Intelligent Memory unit, in this work we investigate the cache misses caused by a subset of memory management functions (allocation and de-allocation). In Object Oriented and Linked Data Structured applications allocation and de-allocation functions are invoked very frequently. These memory management functions are also very data intensive, but require only integer arithmetic. Therefore, a simple integer CPU embedded inside a DRAM chip offers a viable option for migrating allocation/de-allocation functions from main CPU to DRAM, and eliminates the cache pollution caused by those functions. In this paper we show that moving memory management functions from main CPU to Intelligent DRAM eliminates, on average, 60% of cache misses (as compared to the conventional method where allocation/de-allocation functions are performed by the main CPU). Furthermore, we compare the performance of a variety of well known general purpose allocators. We also show that our own variations to the Binary Tree allocator results in better cache localities when compared to other allocators [14,27,28].

In this paper, first we provide a brief background of related research. Then, we introduce our Intelligent Memory Manager's architecture. Section 4 represents the framework used to empirically validate the claim of this work. Final sections illustrate the results and draw the conclusions based upon the results and thoughts of the authors.

2. Research background

Memory often is the limiting factor in achieving high performance in modern computer systems. A variety of techniques are proposed to hide memory latency and improve application's performance. Of these attempts, the approach of merging logic with memory inspired our work. The objective of this work is to benefit from embedded logic in DRAM for migrating memory management operations currently performed by the main CPU and accordingly eliminate the CPU cache pollution caused by these operations. To provide a context and background for our work, in the following subsections we briefly describe the research in these two areas that underlie our research.

2.1. Intelligent memory devices

Utilizing more metal layers with faster transistors in DRAM chips, it is possible to provide a reasonably fast CPU in the heart of memory. By co-locating processor and memory, memory latencies can be drastically reduced. DRAM can provide much larger streams of data at a much faster rate to a processor if both processor and DRAM are in the same chip. This idea attracted researchers, leading to the development of Intelligent Memory Devices. Some researchers have utilized Intelligent Memories for vector processing as a stand-alone processor called Intelligent RAM (IRAM) [26]. The Berkeley IRAM project demonstrates that even when operating at a moderate clock rate of 500 MHz, Vector-IRAMs can achieve 4 Gflops, while Cray machines achieve only 1.5 Gflops [5]. IRAMs are particularly suitable for DSP and multimedia applications which contain significant amounts of vector level parallelism. More recently, researchers are exploring the use of IRAM devices to build energy efficient portable multimedia devices.

Embedded DRAM (eRAM) is another family of Intelligent Memory Devices [24]. M32R, the main core of eRAM which contains a CPU and DRAM, has been used in a variety of applications. M32R/D with an off-chip I/O ASIC is used for multimedia applications, for example, JPEG compression and decompression. Open Core M32R which integrates CPU, SRAM, DRAM, and versatile peripherals into a single chip can be used for portable multimedia devices. M32R media Turbo, which includes a vector processor and a super-audio processor along with a CPU and DRAM, is promoted for applications that demand high performance when dealing with large streams of data (such as speech recognition and image processing). M32R core is based on an extendable dual-issue, VLIW instruction set.

Active Pages is yet another approach that takes advantage of logic and memory integration [25]. Active Pages consists of a page of memory and a set of associated functions to improve processing power. RADram[2] is the basis of Active Pages that gives flexibility in customizing functionality for each application. RADram can potentially replace DRAM in conventional architectures when some additional control logic and control lines are added.

The literature contains other variations to the idea of embedding logic within DRAM devices for improving memory latencies or improving energy efficiencies.

2.2. Memory management techniques

Memory is the most critical resource in computer systems both in terms of speed and capacity. The efficiency of memory management algorithms, particularly in object oriented environments, has captured the attention of researchers. For fully comprehending and appreciating the memory management systems, it is necessary to realize its roles in a typical computer system. As shown in Fig. 1, the OS Memory Manager allocates large chunks of memory to user level runtime systems. These runtime systems (user level processes) are responsible for allocating small amounts of memory when new program variables are created [7]. The separation of memory management is needed to eliminate too frequent kernel calls. Although in principle both the Operating System and Process Memory Managers can be excised away from main CPU and be placed under the responsibility of Intelligent Memory Manager, our work is limited to the Process Memory Manager for several reasons. One reason is that the OS Memory Manager's task is very simple.[3] On the other hand, the Process Memory Manager is required to allocate and de-allocate several tens of thousands of

[2] Reconfigurable Architecture DRAM.

[3] All chunks are in the form of fixed size pages, and about only several hundred pages are needed for the total execution period of each running process.

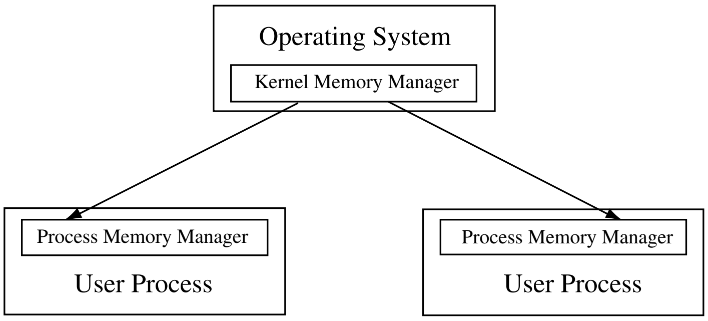

Fig. 1. Memory management hierarchy.

objects of different sizes for each process [27]. Objects can be as small as a byte or as big as several pages. The following subsection describes the most commonly used Process Memory Management techniques.[4]

2.2.1. Allocation techniques

Dynamic memory management is an important problem studied by researchers for the past four decades. Every so often the need for a 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 manager 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. The primary goals of any efficient memory manager are high storage utilization and execution performance [32]. However, current implementations have failed to achieve both aims at the same time. For example, Sequential Fit algorithms show high storage utilization but poor execution performance [13,27]. While Segregated Free lists cause higher fragmentations, yet their performance is the best among allocators. Well-known placement policies such as Best Fit and First Fit have been explored with both Sequential Fit and Segregated Free lists.

[4] They are also known as allocation techniques.

Currently used memory allocation schemes can be classified into *Sequential Fit* algorithms, *Buddy Systems*, *Segregated Free Lists*, and *Binary Tree* techniques.

The Sequential Fit approach (including First Fit and Best Fit) keeps track of available chunks of memory in a linked list. Known Sequential Fit techniques differ in how they track the memory blocks, how they allocate memory requests from the free blocks, and how they place newly freed objects back into the free list. When a process releases memory, these chunks are added to the free list, either in front or in place, if the list is sorted by addresses (Address Order [32]). 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). The Best Fit method tries to find the smallest chunk that is at least as large as the request, whereas the First Fit method finds the first chunk that is at least as large as the request [17]. The Best Fit method may involve delays in allocation while the First Fit method leads to more external fragmentation [13]. If the free list is in Address Order, newly freed chunks may be combined with their surrounding blocks. Such practice, referred to as coalescing, is made possible by employing boundary tags in the doubly linked list of address ordered free chunks [17].

In Buddy Systems the size of any memory chunk (live, free, or garbage) is 2^k for some k [16,17]. 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 free chunks, the two buddies can be combined into a larger chunk of size 2^{k+1}. During allocation, larger blocks 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 reported that as much as 25% of memory is wasted due to fragmentation in buddy systems [13]. 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 [13,33].

Segregated Free List approaches maintain multiple linked lists, one for each different sized chunks of available memory. Allocation and deallocation requests are directed to their associated lists based upon the size of the requests. Segregated Free Lists are further classified into two categories: *Simple Segregated Storage* and *Segregated Fit* [32]. 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 acquired using *sbrk* or *mmap* system calls. In contrast, the Segregated Fit allocator attempts to satisfy the request from a list containing larger sized chunks—a larger chunk is split into several smaller chunks if required. Coalescing is also employed in Segregated Fit allocators to improve storage utilization. Simple Segregated Storage allocators are best known for their high execution performance while Segregated Fit allocators' advantage is their higher storage utilization.

In Binary Tree allocators free chunks of memory are kept in a binary search tree whose search key is the address of the free chunks of memory. Known Binary Tree allocators are *Cartesian Tree*, *Address Ordered Binary Tree*, and *Segregated Binary Tree*. The Cartesian Tree, which was proposed almost two decades ago, is an address ordered binary search tree, which also forces its tree of free chunks to form a heap tree in terms of chunk size [30]. In other words, Cartesian Tree allocators maintain a binary tree whose nodes are the free chunks of memory with the following conditions:

(a) address of descendents on left (if any) $\leqslant$ address of parent $\leqslant$ address of descendents on right (if any),
(b) size of descendents on left (if any) $\leqslant$ size of parent $\geqslant$ size of descendents on right (if any).

The latter that mandates Cartesian Tree to have its largest node at the root of the tree, causes the tree to usually become unbalanced and possibly degrade into a linked list. In Address Ordered Binary Tree, the free chunks of memory are also maintained in a binary search tree similar to Cartesian Tree [14,27]. However, to overcome the inefficiency forced by the size restriction (condition b) of Cartesian Tree allocator, not only is this restriction entirely removed in Address Ordered Binary Tree, but also it is replaced with a new strategy that enhances the allocation speed of this technique. In this specific implementation of Binary Tree algorithms, each node of the tree contains the sizes of the largest memory chunks available in its left and right subtrees. This information can be utilized to improve the response time of allocation requests and implement Better Fit policies [27,32]. Binary Tree algorithms in general are ideally suited for coalescing the free chunks of memory, since the tree is address ordered. This leads to better storage utilization.

In a manner similar to Segregated Fit technique, Segregated Binary Tree keeps several Address Ordered Binary Trees, one for each class size [28]. Each tree is typically small, thus reducing the search time while retaining the memory utilization advantage of Address Ordered Binary Tree.

3. Intelligent Memory Manager's architecture

It is our goal to make the Intelligent Memory Manager (IMM) not limited to a specific design. Nonetheless, this section illustrates the possibilities one could consider for the architecture of IMM.

Generally speaking any piece of logic other than main CPU is a potential host for performing memory management service functions. For instance, we can build a coprocessor with its designated cache, to execute memory management service functions, along with the main CPU on the same chip. This scenario simplifies the design, where the memory bus interface needs no change.

The characteristics of memory management algorithms is that they are very data intensive as they maintain the free chunks of memory in linked lists. Their main functionality also requires very frequent visits to the nodes of the free chunk lists. This property suggests that memory management service functions be executed by a processor close to or on chip with DRAM. Therefore, we propose two streams of design for IMM; an extension to the centralized controller used in DRAM bus configurations[5] [8], or Embedded DRAM [4,25,26]. The latter limits the amount of memory on DRAM chip. With current Gb DRAM technology, we can mount no more than 256 MB DRAM and reasonable logic (powerful enough to perform simple memory management functions) on a single chip. On the other hand, centralized controller design suffers from poor execution speed since it needs to communicate with DRAM chips via the common bus. In both designs the conventional memory bus interface needs to change. We propose the addition of two functions to the standard memory interfaces.

- allocation and de-allocation interfaces (additional interfaces)
  - *allocate(size)*
  - *de-allocate(virtual-address)*

- standard conventional interfaces
  - *read(virtual-address)*
  - *write(virtual-address,data)*

Figs. 2 and 3 depict the high level design of these two configurations.

It is reported that 5 Gb DRAM technology can accommodate up to 32 Kbits internal bus width with 1.7 GHz speed for Embedded DRAM, and 0.8 GHz 128 bit wide external bus in the case of standard DRAM [12]. Using Embedded DRAM or centralized controller, we feel that IMM is feasible with current technologies. The functionality required to implement memory management can be achieved by either using ASIC or more traditional pipelined execution engines. In this paper, we will not deal with detailed design of IMM. The goal of this paper is to illuminate on our hypothesis that migration of memory management functions to a separate processor will eliminate significant amount of cache pollution.

4. Experimental framework

To confirm the claims we have made, and also to compare the cache performance of different memory managers (allocators), we have conducted two stems of experiments on an Alpha 21264 running the Tru64 operating system [15]. First, a single process is used to execute both application and the memory management functions. This scenario simulates conventional systems using a single CPU for both application and memory manager. Next, a pair of processes are used to execute application and its service functions separately. This simulates the use of a separate processor for memory management functions, which can potentially be embedded in a DRAM chip. The latter experiment exploits a shared memory segment for interprocess communication. These processes are instrumented using ATOM instrumentation and analysis routines [9]. Instrumentation routines detect the memory references and call analysis routines, which simulate different cache organizations. The use of shared memory interprocess communication adds a considerable amount of system overhead and consequently blurs the aim of this work. To avoid such artifact, using instrumentation routines, we have discarded the references made by interprocess communication system calls. We have also separated the application heap and analysis routines' heap so that ATOM activities do not impact the locality behavior of the applications. Fig. 4 depicts IMM's framework.

To illustrate the wide applicability of our claim we have employed two sets of benchmarks, a subset of SPEC CINT2000 [11] and a subset of

[5] Non-interleaved SDRAMs or SLDRAMs in Rambus configuration.

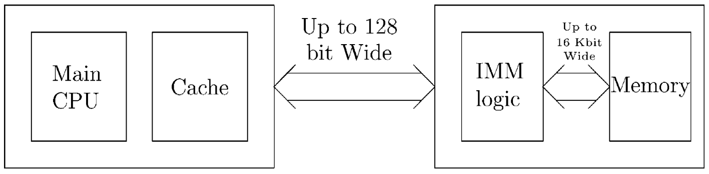

Fig. 2. Intelligent Memory Manager: embedded DRAM configuration.

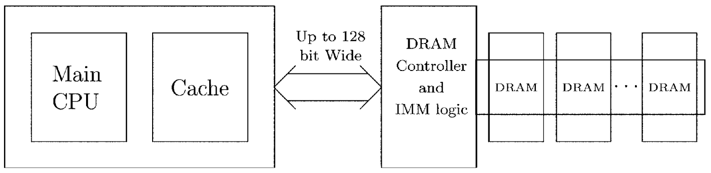

Fig. 3. Intelligent Memory Manager: extended centralized controller.

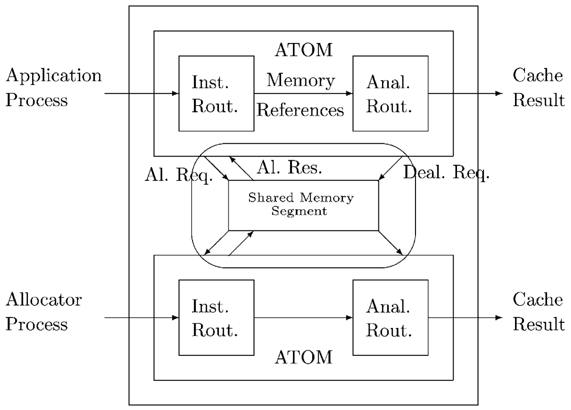

Fig. 4. IMM framework: the use of two kernel processes for simulating IMM configuration.

Table 1
Benchmark description

Benchmark	Description	Input
SPEC2000int		
gzip	gnu zip data compressor	test/input.compressed 2
parser	English parser	test.in
twolf	CAD placement and routing	test.in
vpr	FPGA circuit placement and routing	test
Allocation intensive benchmarks		
boxed-sim	Balls and box simulator	-n 10 -s 1
cfrac	It factors numbers	a 36 digit number
ptc	Pascal to C convertor	mf.p
espresso	PLA optimizer	largest.espresso

benchmarks widely used to evaluate memory allocators. They are briefly explained in Table 1.

In this work four general purpose allocators are studied for their locality behaviors based on different allocation strategies. The allocators studied in this work are:

- *BSD allocator.*[6] This allocator is an example of a Simple Segregated Storage technique [34]. It is among the fastest allocators but it reports high memory fragmentation. In the figures this allocator is referred to as "bsd".

[6] Known as Chris Kingsley's allocator-4.2 BSD Unix.

- *Doug Lea's allocator.* Perhaps the most widely used allocator is Doug Lea's. We have used version 2.7.0, an efficient allocator that has benefited from a decade of optimizations [18]. For request sizes greater than 512 Bytes it uses a LIFO Best Fit method. For requests less than 64 Bytes it uses pools of recycled chunks. For sizes in between 64 and 512 Bytes it explores a self adjusting strategy to meet the two main objectives of any allocator: speed and high storage utilization. For very large size requests (greater than 128 KBytes), it directly issues *mmap* system call. "lea" is used to reference data for this allocator in our figures.

- *Segregated Binary Tree.* SBT contains 8 binary trees each for a different class size. Memory chunks less than 64 bytes and greater than 512 Bytes are kept in the first and the last binary tree respectively. Each binary tree is responsible for keeping chunks of a unique size range, and sizes range in 64 byte intervals. For example, the second binary tree's range is [64,128) (viz., if a chunk's size is x then $64 \leqslant x < 128$). In the figures "sbt" is used to refer to the data set belonging to this allocator.
- *Segregated Fit.* We have written our own version of Segregated Fit algorithm referred to as "sgf". In the structure, this allocator is similar to our SBT but the memory chunks are kept in segregated doubly linked lists instead of binary trees. LIFO Best Fit is chosen for placement policy of each list.

5. Empirical results

As mentioned in the last section, we have carried out our experiments under two scenarios:

- *Conv-Conf*: Conventional Configuration, in which case both application and its allocator are running on the main CPU using a single cache.
- *IMM*: Intelligent Memory Manager, where a separate processor executes memory management functions. IMM consists of two parts; IMM-Application that is the application code running on the main CPU, and IMM-Allocator that is the memory management operations running on the processor embedded in DRAM chip (with a separate cache).

Tables 2–4 show the total number of references for Conv-Conf, application part of IMM (total number of loads and stores issued by main CPU when running the applications), and allocator part of IMM (total number of references issued by DRAM logic when running the allocator portion of the applications) respectively.

Several observations must be made with the data shown in these tables. From Table 3, it should be noted that the number of references made by the application (IMM-Application) are

Table 2
Total number of references for Conv-Conf, Conventional Configuration

Bench/alloc.	bsd	lea	sbt	sgf
boxed-sim	2.65e+09	2.62e+09	3.08e+09	2.8e+09
cfrac	3.38e+09	3.23e+09	5.81e+09	4.26e+09
espresso	6.84e+08	7.95e+08	2.05e+10	8.86e+08
gzip	1.016e+10	1.02e+10	1.02e+10	1.02e+10
parser	1.071e+09	1.16e+09	3.16e+09	2.49e+10
ptc	8.87e+07	9.2e+07	8.81e+07	8.81e+07
twolf	2.92e+08	2.93e+08	2.97e+08	2.93e+08
vpr	1.81e+10	1.81e+10	1.82e+10	1.81e+10

Table 3
Total number of references for IMM-Application

Bench/alloc.	bsd	lea	sbt	sgf
boxed-sim	2.54e+09	2.54e+09	2.54e+09	2.54e+09
cfrac	2.78e+09	2.79e+09	2.79e+09	2.79e+09
espresso	1.41e+08	1.41e+08	1.41e+08	1.41e+08
gzip	1.01e+10	1.02e+10	1.02e+10	1.02e+10
parser	9.28e+08	9.29e+08	9.29e+08	9.29e+08
ptc	9.2e+07	9.2e+07	9.2e+07	9.2e+07
twolf	2.91e+08	2.92e+08	2.92e+08	2.92e+08
vpr	1.8e+10	1.8e+10	1.8e+10	1.8e+10

Table 4
Total number of references for IMM-Allocator

Bench/alloc.	bsd	lea	sbt	sgf
boxed-sim	1.36e+08	1.11e+08	1.01e+08	1.02e+08
cfrac	318459	268149	248351	249147
espresso	1.3e+08	1.86e+08	1.01e+08	1.01e+08
gzip	1.4e+06	2.9e+06	1.2e+06	1.2e+06
parser	1.53e+08	1.97e+08	1.21e+08	1.23e+08
ptc	5.9e+06	9.9e+06	6.2e+06	6.2e+06
twolf	676653	675783	542219	548174
vpr	580947	675783	542219	512907

approximately the same for all allocators, across all benchmarks. The exception is for "cfrac" and "parser". The "bsd" allocator seems to have fewer references, since these applications fit better with the allocation strategies used by "bsd". In almost all benchmarks with different allocators, the represented data demonstrates that the number of memory references of Conv-Conf is equal to the sum of memory references of IMM-Application and IMM-Allocator. For "sbt" and "sgf" allocators we observe a decrease in the number of references for IMM. We suspect the reason for such a behavior due to our unoptimized algorithms. When compared with more established allocators that have benefited from years of fine-tuning, our allocators provide great opportunities for hardware based optimizations such as out of order execution and branch predictions. We have conducted our experiments on Alpha 21264, an out-of-order microprocessor that is able to fetch four instructions per cycle [15]. It employs a sophisticated branch prediction and speculative instruction fetch/execute. While such hardware optimizations are also available for other allocators, since the implementations are already higher optimized, separating the allocator functions have not shown significant improvements, unlike our allocators. Among benchmarks used throughout this paper, "ptc" disagrees with others in showing fewer memory references when application and its allocator functions are separately executed by two processes (IMM). This happens because "ptc" contains only allocation requests and no de-allocation. Although we have partially removed the references associated with interprocess communication overhead due to our framework, this overhead for "ptc" tends to dominate the impact of separating the execution of application and its service functions in terms of number of references. Nonetheless, we feel that the cache miss reduction achieved by extracting the allocator functions from the application is still verified by our data.

5.1. Comparison of cache performance

It is our aim to show the improvement in cache performance obtained using the Intelligent Memory Manager. This improvement holds valid for all cache levels. However, first level cache activity attracts more interest because of its influence on CPU execution time. Hence, in this subsection we include data for first level cache only. We have chosen the cache sizes and block sizes based on modern systems. Figs. 5 and 6 show the total number of cache misses for Conv-Conf and IMM-Application with 32 KBytes cache and 32 Bytes blocks. In almost all benchmarks, it is very clear that IMM configuration has removed the cache pollution caused by memory management service functions. This is better shown by Fig. 7, which reports the percentage of IMM-Application cache miss improvement. The data shows 60% reduction in the number of cache misses on average.

As mentioned before "ptc" tends to behave somewhat differently than other applications in our benchmark suite. This is partially because "ptc" contains only allocations. In both "twolf" and "boxed-sim", the computation core of the application dominates execution time requiring fewer memory management calls. Thus, these applications show insignificant improvements on cache performance. It should be noted that negative

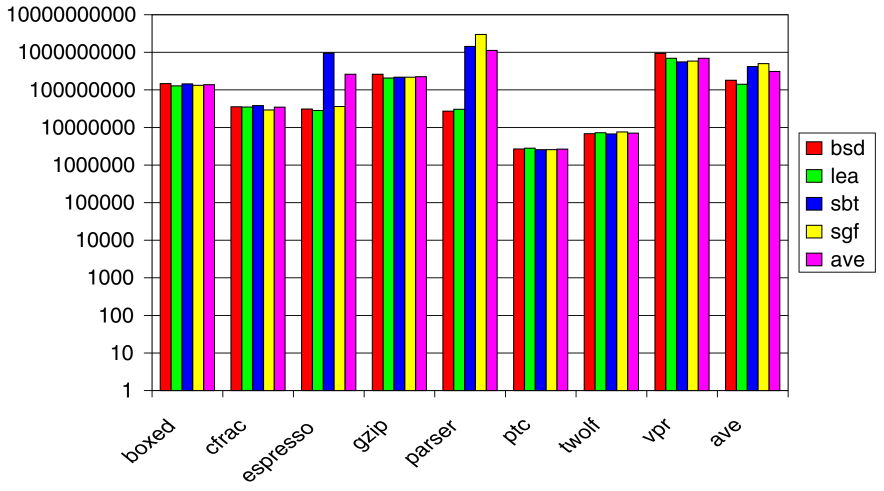

Fig. 5. Conv-Conf cache misses, cache size = 32 KBytes, cache block size = 32 Bytes.

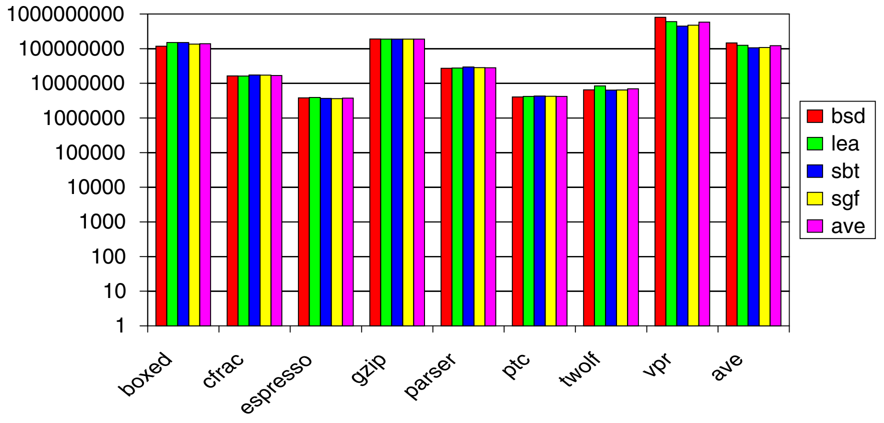

Fig. 6. IMM-Application cache misses, cache size = 32 KBytes, cache block size = 32 Bytes.

impact (i.e., increase in cache misses) is primarily due to the artifact of our experiments involving the shared memory interprocess communication. Although we have done our best to eliminate such memory references, it is impossible to be certain that all references caused by such communication have been removed entirely.

5.2. Impact of cache parameters

In the next experiments we doubled the cache line size to view the impact of changing this cache parameter. Figs. 8 and 9 depict the results for Conv-Conf and IMM-Application. Fig. 8 reports fewer misses for Conv-Conf when cache block size is increased. Fig. 9 shows that on average number of misses increased in the case of IMM, when the cache block is enlarged, albeit slightly.

When application with its memory management functions is running on the main CPU (Conv-Conf), it certainly possesses higher spatial locality as compared with the case of IMM. Each chunk of memory, free or live, contains its size information (normally the first four or eight bytes of the chunk). When the chunk becomes free, it will also contain pointers used by the allocator to track the

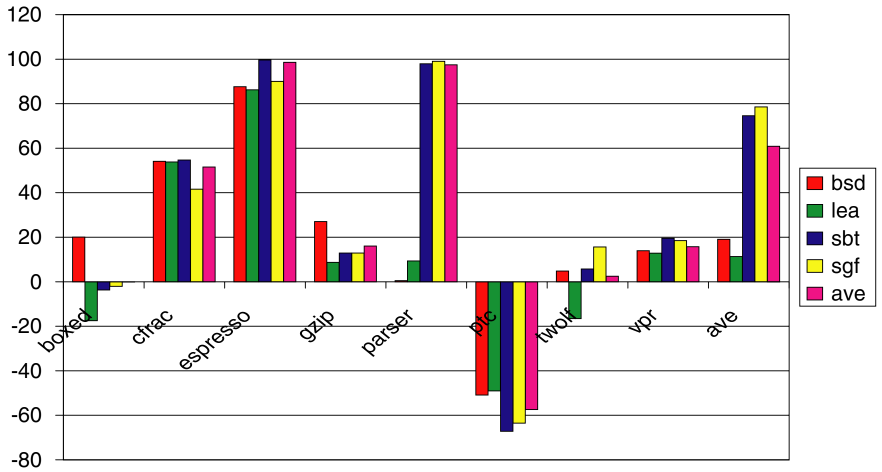

Fig. 7. Percentage of IMM-Application cache miss improvement as to compare with Conv-Conf, cache size = 32 KBytes, cache block size = 32 Bytes.

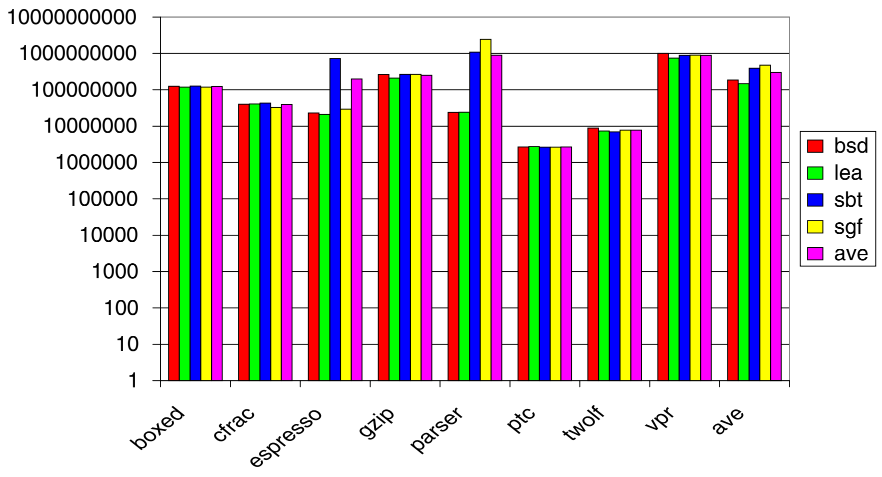

Fig. 8. Conv-Conf cache misses, cache size 32 KBytes, cache block size 64 Bytes.

list of the free chunks. These items are kept in the header of every memory chunk. The role of allocator obliges it to visit free chunks of memory and hence either reading or modifying the information kept in each chunk. When the execution of memory management functions (allocation and de-allocation) entangled with application, its behavior elevates spatial locality and lessens temporal locality of the application. Thus, separating memory management functions from application improves temporal locality dramatically (on average 60% as shown previously), and decreases spatial locality of the application very slightly. Increasing the cache block size for Conv-Conf results in fewer cache misses (comparison of Figs. 5 and 8), because spatial locality of the applications is more utilized. On the other hand, it results in more misses for IMM-Application since it circumvents

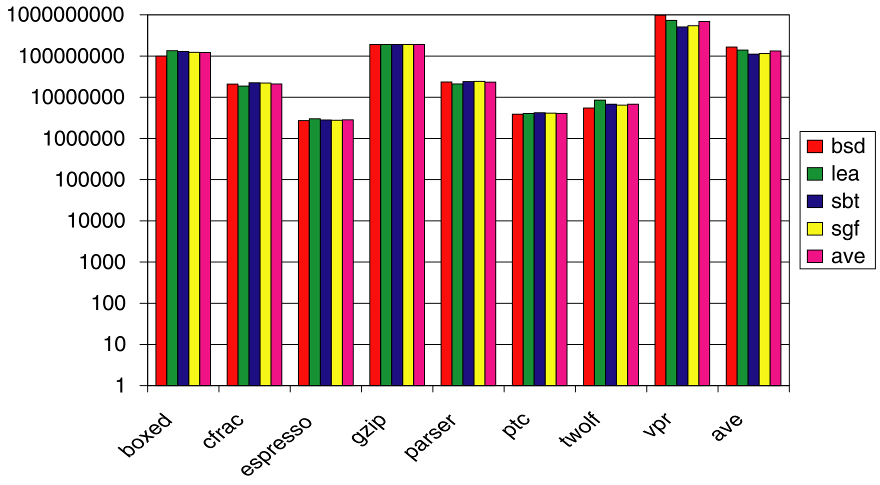

Fig. 9. IMM-Application cache misses, cache size 32 KBytes, cache block size 64 Bytes.

the temporal locality of the application[7] (comparison of Figs. 6 and 9).

5.3. Comparing cache behavior of allocators

Storage Utilization analysis as well as execution performance of different allocators have been adequately studied by others [3,13,32]. Surprisingly, locality behavior of allocators has not been reported as widely. This subsection is an effort, although for only a subset of allocators, that represents the cache behavior of such useful service functions of computer system.

Cache data shown here belongs to the allocator portion of IMM configuration that runs on a separate logic integrated with DRAM chip. A small cache has been considered to serve the IMM-Allocator processor, because the chip area and number of transistors on chip are limited. Fig. 10 illustrates cache performance of different allocators for 512 Bytes direct mapped cache with 32 Bytes block size. "lea" allocator shows the worst performance due to its complexity and hybrid nature. Mixing *sbrk* and *mmap* system calls, which is practiced by "lea" allocator for different object class sizes, may be a cause for such behavior. "lea" allocator is followed with "bsd", which also benefited from strong segregation for speed. This segregation causes "bsd" to reveal poor locality behavior.

Both "sbt" and "sgf" perform almost the same as they benefit strongly from memory chunk reusability. Their implementation leads to the reuse of recently freed objects. Reusability seeds temporal locality, which is the main advantage of these two allocators.

It is quite obvious that cache performance of an allocator is directly associated with its storage utilization. Allocators with higher storage utilization report better cache performance.

6. Conclusions

As the performance gap between processors and memory units continues to grow, memory accesses continue to inhibit performance on modern processors. While memory hierarchy and cache memories can alleviate the performance gap to some extent, cache performance is often adversely affected by service functions such as dynamic memory allocations and de-allocations. Modern applications rely heavily on linked lists and object-oriented programming. This requires sophisticated dynamic memory management, including allocation, de-allocation, garbage collection, data

[7] A larger cache block size with fix cache size means fewer blocks. This is in the favor of spatial locality. Certainly, it is not favorable for temporal locality since fewer blocks of memory can be mapped to the cache at the same time.

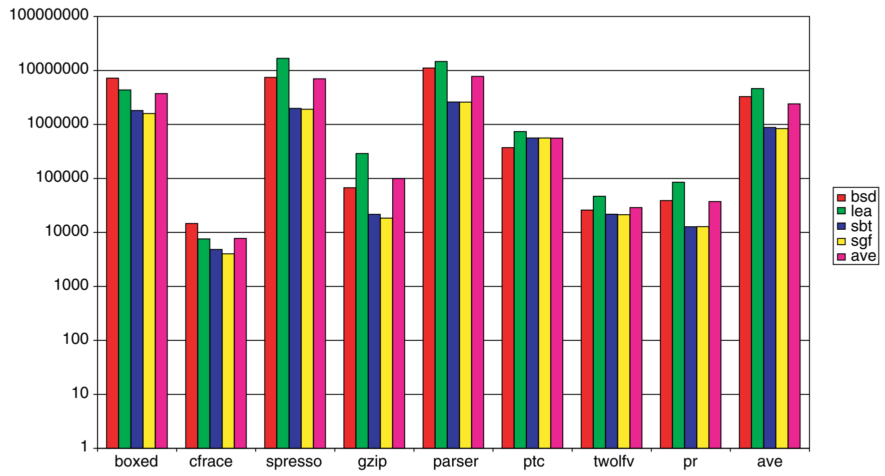

Fig. 10. IMM-Allocator cache misses, 512 Bytes direct mapped cache, 32 Bytes block size.

prefetching, Jump Pointers, and object relocation (Memory Forwarding). 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.

The motivation of our work is partly based on the observations that frequently used service functions when mixed with the execution of application code become the major cause of cache pollutions. The cache pollution can be removed by separating the execution of these functions from application code and migrate them to a different processor. Service functions are also very data intensive, the feature that made them suitable for execution in a processor integrated with DRAM in a single chip. This is yet another observation that motivated Intelligent Memory Management research and directed us towards Intelligent Memory Devices (viz., eRAM, Active Pages, and IRAM). In this paper, we presented the cache data collected from experiments on two schemes. First we conducted our experiment when both application and its memory management functions are executed on the main CPU (Conv-Conf). We have carried out our work by separating the execution of memory management functions and application (IMM). The cache data resulted from the latter shows 60% improvement on average. In the case of IMM, our experimental framework caused some additional overhead due to the interprocess communication, which we tried to remove by discarding the references caused at the time of communication-albeit not completely. We believe that if the interprocess communication overhead is thoroughly removed, one will achieve even more cache miss reduction with IMM configuration.

We also studied the amounts of cache pollution caused by different memory allocation techniques. Some techniques have resulted in more pollution while maintaining their goal for high execution performance. For instance, Simple Segregated Storage techniques are the best in terms of speed, but as we have shown in this work, they illustrate poor cache performance and high cache pollution. Since employing a separate hardware processor eliminates the cache pollution caused by an allocator, we can consider the use 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. These functions can also be off-loaded to the allocator processor of Intelligent Memory Manager in order to benefit from their performance advantages, while maintaining low cache miss rate.

No matter how substantial is the contribution of a research, there is always place for further improvement. The main issue of concern in this work is the synchronization scheme between IMM-Application processor and IMM-Allocator processor. This will become evident as the actual design of the IMM chip is pursued. It is also in our interest to investigate more about the speed of IMM chip and how fast it should be to reveal improved execution performance. We intend to address all of these issues when our cycle time execution driven simulator is completed.

References

[1] S.E. Abdullahi, G.A. Ringwood, Garbage collection the Internet: a survey of distributed garbage collection, ACM Computing Surveys 30 (3) (1998) 330–373.

[2] A. Agarwal, B.-H. Lim, D. Kranz, J. Kubiatosicz, APRIL: A Processor architecture for multiprocessing, in: Proceeding of the 17th ISCA, May 1990, pp. 104–114.

[3] E.D. Berger, B.G. Zorn, K.S. McKinley, Comparing high performance memory allocators, in: Proceedings of the PLDI'01, June 2001, pp. 114–124.

[4] B. Lu, Embedded DRAM: how can we do it right? A presentation given on 15th december 2000 in Dallas, Chapter of IEEE Solid State Circuit Society, Available from: <http://www.infineon.com/edram>.

[5] R. Boyed-Merrit, What will be the legacy of RISC? An Interview with D.A. Patterson, EETIMES, Issue 953, 12 May 1997.

[6] T.-F. Chen, J.-L. Baer, Effective hardware-based data prefetching for high-performance processors, IEEE Transactions on Computers 44 (5) (1995) 609–623.

[7] C. Crowley, Operating System: a Design-oriented Approach, first ed., IRWIN Publisher, 1997.

[8] V. Cuppu, et al., High-performance DRAMs in workstation environements, IEEE Transactions on Computer 50 (11) (2001) 1133–1153.

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

[10] J.L. Hennessy, D.A. Patterson, Computer Architecture: a Quantitative Approach, third ed., Morgan Kaufmann Publishers, UK, 2002.

[11] J.L. Henning, SPEC CPU2000: Measuring CPU performance in the new millennium, IEEE Computer 33 (7) (2000) 28–35.

[12] K. Itoh, et al., Limitation and challenges of multigiga bit DRAM chip design, IEEE Journal of Solid-State Circuits 32 (5) (1997) 624–633.

[13] M.S. Jonhnstome, P.R. Wilson, The memory fragmentation problem: solved? in: Proceedings of ISMM 1998, October 1998, pp. 26–36.

[14] K.M. Kavi, M. Rezaei, R. Cytron, An efficient memory management technique that improves localities, in: Proceedings of 8th ADCOM, December 2000, pp. 87–94.

[15] R.E. Kessler, The alpha 21264 microprocessor, IEEE Micro 19 (2) (1999) 24–36.

[16] K.C. Knowlton, A fast storage allocator, Communications of the ACM (1965) 623–625.

[17] D.E. Knuth, The Art of Computer Programming, third ed.Fundamental algorithm, vol. 1, Addison-Wesley, 1997.

[18] D. Lea, A memory allocator, Available from: <http://g.oswego.edu/dl/html/malloc.html>.

[19] C.-K. Luk, T.C. Mowry, Compiler-based prefetching for recursive data structures, in: Proceedings of 7th ASPLOS, October 1996, pp. 222–233.

[20] C.-K. Luk, T.C. Mowry, Memory forwarding: enabling aggressive layout optimizations by guaranteeing the safety of data relocation, in: Proceedings of 26th ISCA, May 1999, pp. 89–99.

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

[22] T.C. Mowry, Tolerating latency through software-controlled data prefetching, PhD thesis, Stanford University, March 1994.

[23] T.C. Mowry, S.R. Ramkissoon, Software-controlled multithreading using informing memory operations, in: Proceedings of 6th HPCA, January 2002.

[24] Y. Nunomure, T. Shimizu, O. Tomisawa, M32R/D—Integrating DRAM and microprocessor, IEEE Micro 17 (6) (1997) 40–48.

[25] M. Oskin, F.T. Chong, T. Sherwood, Active Pages: a computation model for intelligent memory, in: Proceedings of 25th ISCA, April 1998, pp. 192–203.

[26] D.A. Patterson, et al., A case for intelligent RAM, IEEE Micro 17 (2) (1997) 34–44.

[27] M. Rezaei, K.M. Kavi, A new implementation for memory management, in: Proceedings of IEEE SoutheastCon'00, April 2000.

[28] M. Rezaei, R.K. Cytron, Segregated binary tree: decoupling memory manager, in: Proceedings of MEDEA'00-TCCA Newsletter January 2001, October 2000.

[29] A. Roth, G.S. Sohi, Effective jump-pointer prefetching for linked data structures, in: Proceedings of 26th ISCA, May 1999, pp. 111–121.

[30] C.J. Stephenson, Fast fit: new methods for dynamic storage allocation, in: Proceedings of 9th SOSP, October 1983, pp. 30–32.

[31] D.M. Tullsen et al., Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading process, in: Proceedings of 23rd ISCA, May 1996, pp. 191–202.

[32] P.R. Wilson et al., Dynamic storage allocation: a survey and critical review, in: Proceedings of 1995 International Workshop on Memory Management, Kinross, Scotland, Springer-Verlag LNCS 986, pp. 1–116.

[33] D.S. Wise, The double buddy-system, Technical Report 79, Computer Science Department, Indian University, December 1979.

[34] B. Zorn, Available from: <www.cs.colorado.edu/~zorn/Malloc.html#bsd>.

Mehran Rezaei has received BS and MS degrees in Electrical Engineering, and Ph.D. degree in Computer Science from Insfahan University of Technology, University of Alabama in Huntsville, and University of North Texas in 1993, 1999, and 2005, respectively. He is currently an adjunct faculty at Computer Science and Engineering Department of University of Texas at Arlington. His research interests cover the areas of computer architecture, process memory management, and high performance memory systems.

Krishna M. Kavi is currently a professor and the Chair of Computer Science and Engineering Department at the University of North Texas. Previously he was an Eminent Scholar Chair Professor of Computer Engineering at the University of Alabama in Huntsville from 1997 to 2001. He worked as a faculty at the University of Texas at Arlington from 1982 to1997. He was a program manager at the US National Science Foundation from 1993 to 1995. He has extensive research record covering intelligent memory systems, multithreaded and decoupled architectures, dataflow model of computation, scheduling and load-balancing. Previously, he had developed formalisms based on dataflow graphs and Petri nets for the specification and verification of concurrent processing systems, and for the evaluation of stochastic properties of systems specified using CSP. He has more than 130 technical publications in these and related research areas.