Architectural Support for Dynamic Memory Management

J. Morris Chang, Witawas Srisa-an and Chia-Tien Dan Lo
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, 60616-3793, USA
{chang | witty | danlo} @charlie.iit.edu

Abstract

Recent advances in software engineering, such as graphical user interfaces and object-oriented programming, have caused applications to become more memory intensive. These applications tend to allocate dynamic memory prolifically. Moreover, automatic dynamic memory reclamation (garbage collection, GC) has become a popular feature in modern programming languages. As a result, the time consumed by dynamic storage management can be up to one-third of the program execution time. This illustrates the need for a high-performance memory management scheme.

This paper presents a top-level design and evaluation of the proposed instruction extensions to facilitate heap management. These instructions are h_malloc for memory allocation, mark, and sweep for garbage collection. Simulation results show that the hit ratio for 2 Kbits and 8 Kbits buffer range from 84-99% and 95-99%, respectively. The hardware complexity of the proposed scheme is O(n), where n is the size of the bit-vector. For a design with 20K gates and 97% miss rate, the overall speedup can be as high as 1.41.

1. Introduction

By early 2010s, the VLSI technology would allow fabricators to pack 1 billion transistors into a single chip that can run at Giga-Hertz clock speed. Consequently, the challenge is no longer how to make a billion-transistor chip, but instead, what kind of facilities should be incorporated into the design [4]. The current trend in CPU design is to include application specific instruction sets such as MMX and 3D-now as extensions to basic functionalities. The rationales behind such approaches are obvious. First, space and cost limitations are no longer issues. High-density chips can be manufactured cheaply in current semiconductor technology. Second, these application specific instruction sets are included to alleviate performance bottlenecks in the most commonly used applications such as 3-D graphic rendering and multimedia. These rationales closely follow the corollary of Amdahl's law: *Make the common case fast.* Amdahl's Law reminds us that the opportunity for improvement is affected by how much time the event consumes. Thus, making the common case fast will tend to enhance the performance better than optimizing rare cases [6]. Since the biggest merit of hardware is speed, the significant speedup can be gained through hardware implementations of common cases.

As the popularity of object-oriented programming and graphical user interface increases, applications become more and more dynamic memory intensive. It is well-known among experienced programmers that automatic dynamic memory management functions (i.e. allocation and garbage collection) are slow and non-deterministic. Since object-oriented applications prolifically allocate memory in the heap, it is also no coincident that such applications can run up to 20 times slower than the procedural counterparts. A study has also shown that Java applications can spend 20% of the execution time in dealing with dynamic memory management [1]. Unlike stack or queue, heap is not a well-defined data structure. Allocating memory in the heap often requires some form of search routines. In software approaches to heap management, searching is done in sequential fashion (i.e. linked list search). As the number of existing objects grows, the search time would grow linearly longer as well. Studies have shown that applications written in C++ can invoke up to ten times more dynamic memory management calls than comparable C applications [2]. Apparently, dynamic memory management is a common case in object-oriented programming. With Amdahl's corollary in mind, the need of a high-performance dynamic memory manager is obvious.

Deterministic turnaround time is a very desirable trait for real-time applications. Presently, software approaches to automatic dynamic memory management often fail to yield predictable turnaround time. The most often used software approach in maintaining allocation status is sequential fit or segregated fit. These two approaches utilize linked-list to keep the occupied chunks or free chunks. With linked-list, the turnaround time often relates to the length of the list. As the linked-list becomes longer the sequential search time would grow longer as well [7]. Similarly, the software approaches to garbage collection [5] also yield unpredictable turnaround time. Basically two of the most common approaches for garbage collection are mark-sweep and copying collector. In both instances, the turnaround time is not deterministic.

According to Nilsen and Schmidt, one of the ways to achieve hard real-time performance for garbage collection is through the hardware support [5]. In this paper, we introduce

0-7695-0801-4/00 $10.00 © 2000 IEEE

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 13, 2009 at 13:59 from IEEE Xplore. Restrictions apply.

an application specific instruction extension called Dynamic Memory Management eXtension (*DMMX*) that includes *h_malloc*, *mark*, and *sweep* instructions at the user-level. In *h_malloc*, our high-performance allocation scheme allows allocation to be completed in a few instruction cycles. Unlike software approaches, our scheme is fast and deterministic. To perform garbage collection, the *mark* instruction is invoked repeatedly until all the live objects are marked on a bit-map. Once the marking phase is completed, the *sweep* instruction is called. Since we have a dedicated hardware to perform the sweeping, this phase can be completed in a few instruction cycles.

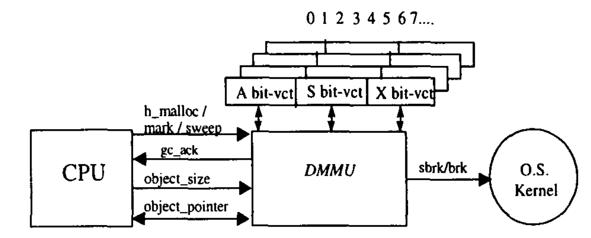
The remainder of this paper is organized as follow. Section 2 provides a top-level architecture of our instruction set. Section 3 describes the internal structure of the Dynamic Memory Management Unit (DMMU). Section 4 addresses the architectural support issues for the DMMU. Section 5 analyzes the simulation results. Section 6 provides analysis on the hardware cost and the potential performance gain. The last section concludes this paper.

2. Overview of the DMMX

In our proposed Dynamic Memory Management eXtenstion (*DMMX*), there are three user-level instructions, *h_malloc*, *mark*, and *sweep*. These three instructions are used as the communication channels between the CPU and the Dynamic Memory Management Unit (*DMMU*). This DMMU can either be packaged inside CPUs or outside. This unit can also be included inside the hardware implemented Java Virtual Machines (i.e. PicoJava II from Sun Microsystems). The main purpose of the *DMMU* is to take responsibility for managing heap space for all processes in the hardware domain. The proposed *DMMU* utilizes the modified buddy system combined with the bit-map approach to perform constant-time allocation [3]. Usually, each process has a heap associated with it. In the proposed scheme, each heap requires three bit-maps, one for allocation status (*A bit-map*), one for object size (*S bit-map*), and one for marking during the garbage collection (*X bit-map*). It is necessary to place these three bit-maps together all the time, since searching and modification to these three bit-maps are required for each garbage collection cycle. Figure 1 demonstrates the top-level integration of the *DMMU* into a computer system.

Figure 1 illustrates the basic functionality of the *DMMU*. First, the *DMMU* provides services to CPU by maintaining the memory allocation status inside the heap region of the running process. Thus, the *DMMU* must be able to access the *A bit-map*, *S bit-map*, and *X bit-map* of the running process. Similar to *TLB*, the *DMMU* is shared among all processes. The parameters that the CPU can pass to the *DMMU* are the *h_malloc, mark, or sweep* signal, the *object_size* (for the allocation request), and the *object_pointer*. The operations of the *DMMU* are very similar to the function calls (i.e. *malloc()*) in C language. Thus, *object_pointer* is either returned from the *DMMU* in allocation or passed on to the *DMMU* during the garbage collection process. The *gc_ack* is also returned at the completion of garbage collection cycle. If the allocation should failed, the *DMMU* would make a request to the operating system for additional memory using system call *sbrk()* or *brk()*.

Figure 1. The top-level description of a *DMMU*



Since the algorithms used in the *DMMU* are implemented through pure combinational logic, the time to perform a memory request or memory sweeping is constant. On the other hand, the time for a software approach in performing an allocation or a sweeping cycle is non-deterministic. As stated earlier, Java applications spend about 20% of the execution time in dealing with automatic dynamic memory management. This extensive execution time can be greatly reduced with the use of the *DMMU*.

3. Internal architecture of the DMMU

Inside the *DMMU*, three bit-vectors are used to keep all of the object relevant information such as allocation status of the heap, the size information of occupied blocks and free blocks, and the live object pointers. The allocation status is kept on the *Allocation bit-vector* (*A bit-vector*). When a *h_malloc* is called, the size information is received by the *Complete Binary Tree* (*CBT*). This dedicated hardware unit is responsible for locating the first free memory chunk that can satisfy the request using the modified buddy system. Besides locating the memory chunk, the *CBT* also has to send out the address of that newly allocated memory and updates the status of that memory block from free to allocated. It is worth noting that while the free block lookup is done using size index of 2^n, the system only allocates the requested size. For example, if 5 blocks of memory is requested, the system will have to find the first free chunk of size 8 (2^3). After a chunk is located, the system only allocates 5 blocks and relinquishes the remaining 3 blocks. Each time an object is created or reclaimed, the *Size bit-vector* (*S bit-vector*) is instantly updated by a dedicated hardware, *S-Unit*. The *auXiliary bit-vector* (*X bit-vector*) is only used during the marking phase of the garbage collection cycle. Once the marking phase is completed, the *sweep*

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 13, 2009 at 13:59 from IEEE Xplore. Restrictions apply.

instruction is invoked. A dedicated hardware, *bit-sweeper*, is used to perform this task in constant time. The internal architecture of the *DMMU* is given in Figure 2.

Figure 2. Internal architecture of the *DMMU*.

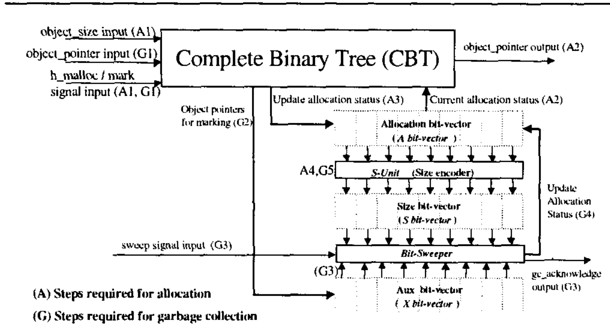


Figure 2 depicts the sequence needed to complete the allocation or garbage collection. For example, if an allocation of size 5 is requested, *A1s* indicate the first step needed to complete the allocation. According the Figure 2, the *h_malloc* and *input signal* would go to logic '1' and the requested size would be given to the *CBT*. Since the *CBT* is a combinational hardware, the free memory chunk lookup, the return address pointer, and the new allocation status signals can be produced at the same time (*A2s*). Next, the new allocation status is latched in the *A bit-vector* (*A3*). Since the *S-Unit* is also a combinational hardware, as soon as the *A bit-vector* is latched, the new size information is available to the *S bit-vector*. Lastly, the new size information is latched in the *S bit-vector* (*A4*) and the allocation is completed. The sequence of garbage collection can also be traced in a similar fashion.

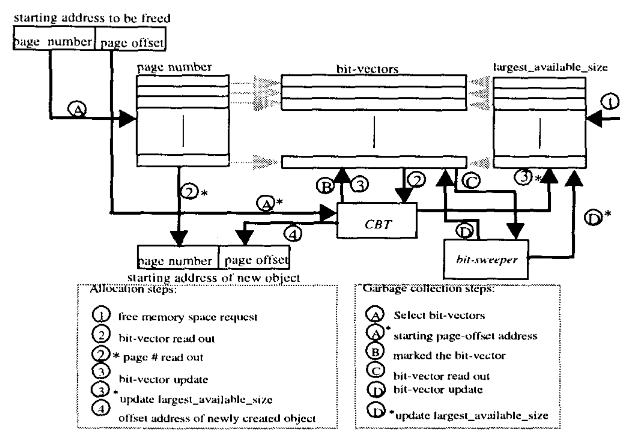
4. Architectural support for *DMMU*

This section summarizes the process of memory allocation and garbage collection in the *DMMU*. Since the bit-maps of a given process may be too large to be handled in the hardware domain, the *bit-vector*, a small segment of the bit-map, is used in the proposed system. This idea is very similar to the idea of using *TLB* (*Translation Look-aside Buffer)* in the virtual memory. Due to the close tie between the *S bit-map, A bit-map,* and *X bit-map*, the term *bit-vector* used in this section represents one *A bit-vector* (of *A bit-map*), one *S bit-vector* (of *S bit-map*), and one *X bit-vector* (of *X bit-map*). Figure 3 presents the operation of the proposed *DMMU*.

When a memory allocation request is received (step 1), the requested size is compared against the *largest_available_size* of each bit-vector in a parallel fashion. This operation is similar to the tag comparison in a fully associated cache. However, it is not an equality comparison. There is a hit in the *DMMU*, if one of the *largest_available_size* is greater or equal to the request size. If there were a hit, the corresponding bit-vector would be read out (step 2) and sent to the *CBT* [3]. The *CBT* is a hardware unit to perform allocation/deallocation on a bit-vector. For the purpose of illustration, we assume that one bit-vector represents one page of the heap.

After the *CBT* identified the free chuck memory from the chosen page, the *CBT* will update the bit-vector (step 3) and the *largest_available_size* field (step 3*). The object pointer (in terms of page offset address) of the newly created object is generated by the *CBT* (step 4). This page offset combines the page number (from step 2*) into the resultant address.

Figure 3.The allocation and garbage collection processes of the *DMMU*



For the garbage collection, when the *DMMU* receives a mark request, the page number of the object pointer (i.e. a virtual address) is used to select a bit-vector (step A). This process is similar to the tag comparison in cache operation. At the same time, the page offset is sent to the *CBT* as the address to be marked (step A*). The process is repeated until all the memory references to live objects are marked. When the marking phase is completed, the sweeping phase (step C) would begin by reading out the bit-vectors and send them to the *bit-sweeper*. The *bit-weeper* would keep all of the objects where the starting addresses were provided by step A* and update the bit-vector (step D) and the largest available size field (step D*). The page number, bit-vectors, and the *largest_available_size* are placed in a buffer, called the *Allocation Look-aside Buffer* (*ALB*).

Since the *DMMU* is shared among all processes, content of the *ALB* will be swapped during the context-switching. This issue also exists in *TLB*. To solve this problem, we can add a *process-id* field in the *ALB*. This will allow bit-vectors of different processes to coexist in the *ALB*. We expect the performance of the *ALB* to be very similar to the much-studied *TLB*. However, further research in the *ALB* organization, hit ratio and miss penalty is required.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 13, 2009 at 13:59 from IEEE Xplore. Restrictions apply.

5. Simulation Results

This section presents detailed simulation results of the proposed *DMMU*. The simulator accepts memory allocation and deallocation traces as inputs and provides hit ratio as the result. The memory allocation/deallocation traces are obtained by instrumenting the *malloc* and *free* functions of the source programs. In the following subsections, the characteristics of the programs we traced and the performance evaluation of the *DMMU* are summarized.

5.1. Application overview:

We evaluate our scheme on several memory allocations and deallocations traces from various C, C++, and Java programs. These programs are drawn from different application areas, including compiler (*gcc*), assembly language simulator (*xspim*), CAD tool (*electric*), robotic simulator (*roboop*), PDF document viewer (*xpdf*), visual calculator (*calcJ*), rich-text editorJ (*txteditJ*), and widgetdemo (*widgetJ*). All eight programs are publicly available software applications. The first three programs are written in C. *Roboop* and *Xpdf* are written in C++, and *calcJ*, *txtEditJ*, and *widgetdemo* are written in Java.

The *gcc* was used to compile the *electric*. The *xspim* was use to run an assembly program of recursive Ackerman's function. We use *electric* to draw a very simple circuit. The *xpdf* was used to open a 25 pages pdf document. *Roboop* was run in the demo mode to generate graphs. Lastly, our three Java programs were used to perform simple tasks in calculator and editor. The numbers of *malloc* and *free* invocation are ranging from about 5000 to nearly a million. The average object size of memory allocation request for each programs is ranging from 30 bytes to 2100 bytes. This shows that our experiments cover a good variety in allocation patterns.

5.2. Investigating block size

Before we can evaluate the system performance, the first parameter to be studied is the block size. Again, in the bit-map, one bit stands for one block worth of memory. The block size affects the bit-map size for a given heap size. The larger block size would yield a smaller bit-map size. A smaller bit-map size means a lower cost for the bit-map. However, the larger block size may lead to higher internal fragmentation during the allocation. The higher internal fragmentation can contribute to a higher watermark (i.e. the highest memory address allocated). Apparently, the higher watermark is considered as the memory overhead in the proposed scheme. Next Table summarizes the memory overhead (through watermark) with block size ranging from 8 bytes to 64 bytes. The smallest block size, 4 bytes for one block, is used as the benchmark.

Table 1 Memory overhead as compared to 4 bytes/block (%)

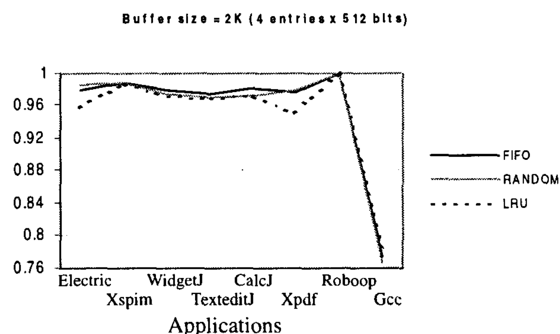
Bytes/Block	GCC	XSPIM	ELECTRIC	XPDF	ROBOOP	TXTEDITJ	CALCJ	WIDGETJ
8	0	0	0.8	0	0	0	0	0
16	0	4.4	4.5	5.1	0	0	0	0
32	0	8.8	21.1	15.3	0	3.0	3.1	3.2
64	1.2	30.6	56.8	38.1	17.6	37.7	38	37.5

From the table above, 16 bytes/block is the most logical block size. When compare block size of 16 and 8, the overhead in block size of 16 is minimal (5.08%). However, the overall size of bit-map would be reduced by 50% compared to block size of 8. Thus, we will use 16 bytes/ block throughout the subsequence simulations.

5.3. Investigating replacement policy

Similar to cache, the replacement policy can determine the performance of the *ALB*. The three most common replacement policies, FIFO, Random, and LRU are investigated in the simulation. The two basic buffer configurations used in the simulation are 4 entries x 512 bits and 4 entries x 1Kbits. Figure 4 demonstrates the performances of the buffers with different replacement policies.

Figure 4. Comparison between different replacement policies.



The performances between different policies do not differ much. A closer look reveals that in most instances, FIFO performs a little better than the other policies. The reason why FIFO performs better has to do with the object life-span. Studies have shown that young objects tend to die young while old objects continue to live. FIFO strategy can guarantee the bit-vector that contains the oldest objects will

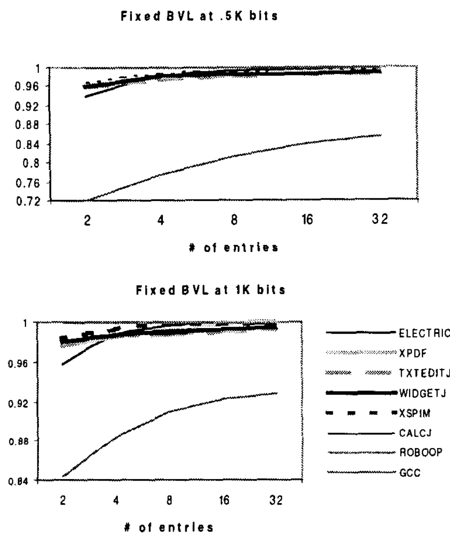
Authorized licensed use limited to: IEEE Xplore. Downloaded on January 13, 2009 at 13:59 from IEEE Xplore. Restrictions apply.

always be replaced. Thus, FIFO will be used as the replacement policy throughout our simulations.

5.4. *ALB* Performance Evaluation

We investigate the performance of the *ALB* through two approaches. First, we fix the size of the Bit-Vector Length (*BVL*) and increase the number of entries (this also increases the buffer size). In doing so, we can find a good saturation point where the hit ratio of all or most of the programs begin to stabilize. The result is illustrated in Figure 5.

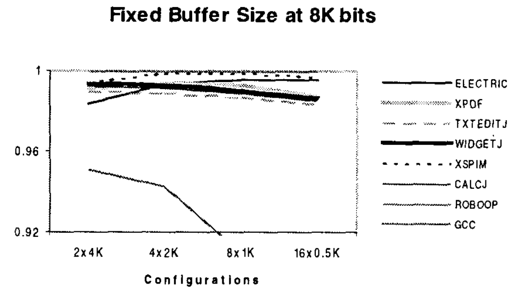
Figure 5. Buffer size VS. Hit ratio



In most applications, the good saturation point for case a is at 16 entries. This translates to the buffer size of 8Kbits. For case b, the saturation point is at 8 entries. This also translates to 8Kbits buffer size. It is worth noting that *gcc* invoked nearly 100,000 objects in the size of 4K bytes (i.e. one page worth memory). This is because the *gcc* maintained its own free list for certain objects. The burdensome overhead of *malloc* and *free* is a well known issue among experienced programmers. The most common way to lower the penalty is to make less frequent calls to *malloc*. Thus, programmers tend to request a large chunk of memory once, then keep track of their own free list. It may need to request another large chunk of memory if the current one has run out the space. This scheme is used in *gcc* and the chunk size is 4K bytes.

In the second approach, we set the buffer size to the value provided by the first approach (in this case 8K bits). Then, we would investigate the effect of buffer configuration (number of entries x *BVL*) on the hit ratio. Since the buffer size is set to 8K blocks, we can have the following configuration, 2x4K, 4x2K, 8x1K, and 16x0.5K. The result is demonstrated in Figure 6.

Figure 6. Buffer size VS. Hit ratio (different configurations).



From the above Figure, hit ratio decreases as the *BVL* decreases. This phenomenon is similar to caches (i.e. larger cache line size may lead to a higher hit ratio). It is worth noting that the hit ratio varies more with *gcc* which has a larger average object size. This is because the higher miss probability occurs in a smaller *BVL* with many objects that are relative large. Obviously, the configuration the allows longer *BVL* with less entries (2x4K) has the best hit ratio. On the other hand, the configuration that allows more entries with shorter *BVL* (16x0.5K) would also have smaller miss penalty (i.e. less data need be moved to buffer for each miss). Similar to the cache design, trade-off between lower miss penalty and higher hit ratio must be made by the system architects.

6. Hardware cost and performance gain

We perform analysis on the hardware cost to construct the *DMMU* with all three instructions included. The cost is expressed as the number of gates. We use N (which represents the bit-vector length in bits) equal to 500 because the simulation result in Figure 5 indicates that the BVL of 500 bits already produces hit-ratio of 97% in most applications. Thus, the BVL length of 500 bits is used to minimize the hardware cost. It is worth noting that *mark* instruction does not require any additional hardware because the bit-flipper is used to mark live object on the *X bit-vector.*

For $N = 500$ bits, the number of gates required is less than 20,000 gates. At the same time, the memory required for each bit vector is only 64 bytes (500 bits / (8 bits/ byte)). Thus, only 192 bytes are required for all three bit-vectors, *A bit-vector, S bit-vector,* and *X bit-vector.*

The potential performance gain analysis of dynamic memory allocation is also performed. To get the actual machine cycles needed to complete memory allocation, a performance profiling tool, Quantify (developed and distributed by Rational Software Corporation) is used to measure the number of machine cycles spent on memory allocation. The *malloc* function used is written by Doug Lea and is distributed as part of the GNU's G++ library. While this malloc is neither the fastest nor the lowest in memory overhead, it represents a good balance between

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 13, 2009 at 13:59 from IEEE Xplore. Restrictions apply.

high-speed and low-memory overhead. Our study indicates that malloc execution time can vary from 51 cycles to 900 machine cycles (mean value = 192 cycles / malloc). From Figure 6, the ALB misses in our proposed scheme is about 3% in most applications. On a typical PC with 100MHz system bus, the data transfer rate is about 800 MBytes/sec (i.e. 64 bit data-bus running at 100 MHz). Assuming that the CPU's clock rate is 400 MHz, on an ALB miss, the miss penalty is about 96 cycles.

$$\text{miss penalty} = \frac{192bytes \times 400MHz}{\frac{800Mbytes}{Sec}} = 96 \text{ cycles}$$

Note: the amount of memory required for three bit-vectors is 192 bytes.

In our proposed scheme, the number of machine cycles required to perform allocation with ALB hit is 2 clock cycles. Thus, $Average\ Malloc_{Hardware}$ is:

$$Average\ Malloc_{Hardware} = (0.03 \times 96) + (0.97 \times 2)$$
$$= 4.82 \text{ cycles.}$$

Since the average $Average\ Malloc_{Software}$ is 192 cycles. The $Speedup_{malloc}$ is:

$$Speedup_{malloc} = \frac{192}{4.82} = 39.83$$

As stated earlier, studies have shown that C++ applications can spend up to 38% of its execution time in dynamic memory management [2]. To calculate overall speedup, Amdahl's law is applied. We use 30% as the value of $fraction_{enhanced}$. Thus, $Overall\ Speedup$ is:

$$\frac{1}{(1 - fraction_{enhanced}) + \frac{fraction_{enhanced}}{Speedup_{malloc}}} = 1.413$$

Therefore, 41.3% of overall speedup can be gained with the proposed memory allocation scheme.

7. Conclusion

The memory intensive nature of object-oriented languages such as C++ and Java has created the need of a high-performance dynamic memory management. As today's VLSI technology advances, it becomes more and more attractive to map software algorithms such as *malloc()* and garbage collection into hardware. To maintain the backward compatibility with existing architectures, these algorithms can be implemented in hardware as application specific instruction extensions. Moreover, this approach allows high level languages to map the most time consuming functions into primitive instructions. For example, the *malloc* function in C, the *new* operator in C++ and the bytecode *new* in JVM can be mapped to the proposed *h_malloc* instruction directly.

The innovative bit-maps approach has following advantages over traditional software approaches. First, it eliminates the cache pollution due to traversing the linked-list during object allocation, object marking and object size look up. This is true since all the object management information is kept separately from the object itself. Second, it requires no splitting and coalescing during the allocation. Third, it permits parallel operations, through hardware, on the bit-maps during object creation and liberation (i.e. garbage sweeping). Software algorithms tend to employ sequential search. Fourth, it uses less memory space to keep object management information as long as object size is less than 24 x *BL* where *BL* is the block size.

The detailed design and evaluation of the proposed scheme are presented in this paper. Simulation results show that the hit ratio for 2 Kbits and 8 Kbits buffer range from 84-99% and 95-99%, respectively. The hardware complexity of the proposed scheme is $O(n)$, where n is the size of the bit-vector. A design with 20K gates and 97% miss rate, the overall speedup can be as much as 41%. The proposed *DMMX* can be included in various architectures such as general purpose CPU, Java chip, and garbage-collected memory module.

8. References

[1] E. Armstrong, "Hotspot, A new breed of virtual machine", *JavaWorld*, March 1998.

[2] B. Calder, D. Grunwald, and B. Zorn, "Quantifying behavioral differences between C and C++ programs", *Journal of Programming Languages*, 2(4):313-351, 1994

[3] M. Chang and E. F. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems," *IEEE Transactions on Computers*. March, 1996. pp. 357-366.

[4] K. Kavi, J.C. Browne, and A. Tripathi, "Computer Systems Research: The pressure is on" *Computer*, January 1999, pp. 30-39.

[5] R. Jones, R. Lins, *Garbage Collection: Algorithms for automatic Dynamic Memory Management,* John Wiley and Sons, 1996, pp.20-28, 87-95, 296

[6] D. Patterson and J. Hennessy, "Computer Architecture, A Quantitative Approach", Morgan Kaufmann Publishers, Inc., Second Edition 1996.

[7] P. Wilson, M. Johnstone, M Neely and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review", Proc. 1995 *Int'l workshop on Memory Management*, Scotland, UK, Sept. 27-29, 1995.

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 13, 2009 at 13:59 from IEEE Xplore. Restrictions apply.