A Self-Maintained Memory Module Supporting DMM

Weixing JI, Feng Shi, Baojun QIAO
School of Computer Science and Technology, Beijing Institute of Technology
5 South Zhongguancun Street, Haidian District,
Beijing 100081, P. R. China
+86-010-81614439
pass@bit.edu.cn

ABSTRACT

The memory intensive nature of object-oriented languages such as C++ and Java has created the need of a high-performance dynamic memory management (DMM); however, it is a challenging task to provide efficient reliable system without violating real time performance constraints. Hardware approach emerges as one of the candidate in improving the performance of DMM. This paper presents an efficient design for explicit dynamic memory management which exploits the high speed of a pure hardware implementation. Object allocation and deletion are strictly bounded in time. The whole heap space is divided into two semi-spaces, and a concurrent bidirectional memory compaction algorithm is proposed. So that memory compaction can be done while mutator process is running on the processor. A small built in object-based cache memory is available to avoid indirect object addressing inefficiencies. Experiments show that this hardware scheme can greatly improve the speed and predictability of DMM.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types, classes and objects, dynamic storage management.*

General Terms

Management, Measurement, Performance, Design, Languages.

Keywords

object-oriented programming, dynamic memory management, active memory module, object-based cache.

1. INTRODUCTION

It is a well-known fact that object-oriented programs tend to be dynamic memory intensive. Through the support from the dynamic memory management system, programmers are allowed to be more productive to enhance software reliability and functionality. However, the characteristic of traditional software approaches are also known to be slow and non-deterministic. Studies show that DMM accounts for up to 43% of Java run time [3]. C/C++ applications can spend from 23% to 38% of the execution time performing DMM [15][1]. Embedded systems have real-time constraints; therefore, non-deterministic allocation and collection pause times can severely degrade run-time performances and may even result in system failures. The increasing popularity of object-oriented programming in embedded system environments has created the need of a high-performance DMM for embedded devices. Over the years, there have been several attempts to incorporate hardware support as a way to reduce run-time overhead, increase in system's throughput, and limit the worst case latencies. However, it is hard to build a bounded time scheme without hardware support. With the advancement in VLSI technology, a high density chip with multimillion system gates is currently available and will continue to grow in the future. To fully utilize these hardware resources, it becomes more and more attractive to map basic software algorithms into hardware. DMM is one of the cases and its performance can be improved greatly by shifting high computation extensive software algorithms to hardware domain. In this paper, we propose a self-maintained memory module as a way to provide hardware support for explicit dynamic memory management. The two benefits of the proposed scheme are that the allocation and deletion can be done in constant time. Moreover, the module can perform heap compaction in parallel with mutator process running on processor; thus the memory compaction latency is greatly reduced. This is the major departure from previously counterparts, which are all "stop-the-world" collectors and memory compaction can vary from thousand cycles to several million cycles. Allocation of a new object and deletion of a live object in the proposed scheme is strictly bounded. As a result, the overall speed-up is obvious compared to software-only collection techniques.

The remainder of this paper is organized as follow. Section 2 discusses previous work and section 3 provides a top-level architecture of our self-maintained memory module. Section 4 addresses the architectural support issues for object addressing, while section 5 shows our new bidirectional compaction algorithm. The object-based cache architecture is given in section 6. Section 7 analyzes the experiment results. The last section concludes this paper.

2. Related work

Hardware-assisted DMM is a promising approach to achieve hard real-time performance which no software-only techniques could guarantee. Based on Baker's incremental copying collector, Nilsen and Schmidt proposed a hardware-assisted real-time garbage collector [18][23]. A special memory module, called garbage-collected memory module (GCMM), was designed separately from general CPU so as to share the technology investment cost and make economies of scale feasible. The GCMM contains memory for to-space and from-space, local memory and a microprocessor. The microprocessor communicates with the CPU through a number of memory mapped I/O ports. A design of an application specific instruction extension called Dynamic Memory Management extension (DMMX) was proposed in [14]. The allocation is done through the modified buddy system and a complete binary tree of combinational logic, which allows constant-time object creation. The hardware solution provided utilizes a set of bit-maps and combinational circuit components to perform mark-sweep garbage collection, where the sweeping phase can be accomplished in constant time. This hardware scheme can greatly improve the speed and predictability of DMM [16] [24][17][25]. The proposed DMMX is an add-on approach, which allows easy integration into any CPU, hardware-implemented Java virtual machine, or processor in memory. A replicating garbage collector for a persistent heap is described in [13]. The garbage collector cooperates with a transaction manager to provide safe and efficient transactional storage management. Clients read and write the heap in primary memory and can commit or abort their write operations. Their implementation is the first to provide concurrent and compacting garbage collection of a persistent heap.

Other self-maintained memory modules can also be found in [9] [4][26]. Due to the consideration of hardware complexity and cost, these hardware assisted dynamic memory management systems are all based on simple collecting algorithms, such as copying collector, mark-sweep and mark-compact. Since the entire heap is split into two semi-spaces for copy collectors, only one region can be active during run-time. Copying also rearranges the allocation order. This can be important in some applications where spatial locality should be preserved. Additionally, long-living objects are continuously swapped between the two sections every time the copying collector is running. Though mark-compact and mark-sweep are a bit more complex, they are still "stop-the-world" garbage collection techniques as the same as copying collector; that is, all application threads stop until garbage collection completes. Many types of applications can't tolerate their stop-the-world nature. That is especially true for applications that require near real-time behavior or those that service large numbers of transaction-oriented clients. Hence, in this paper we propose a concurrent bidirectional compaction collector, which is embedded in a self-maintained memory module and can run in parallel with processors.

3. Self-maintained memory module

Figure 1 shows the top-level architecture of our self-maintained memory module. The typical configuration of this memory block consists of the following functional units: interface unit (IU), dynamic memory management unit (DMMU), object-based cache (OBC) and a dual-port RAM block. This self-maintained module supporting explicit dynamic memory management takes advantage of the speed of pure hardware implementation and it is an add-on approach, which allows easy integration into any hardware-implemented Java virtual machine, or object-based architectures. In the view of Java processor, for example, proposed memory module is just like a blank box. All the objects are referred by a location-independent object reference. Processor can send requests to this memory and data will be available on the external bus immediately if return value is necessary.

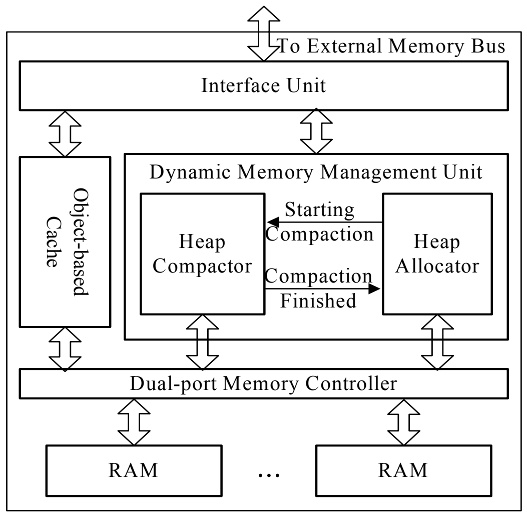

Figure 1. The top-level architecture of active memory module

IU is responsible for communicating between the external memory bus and internal functional units, and it contains a number of memory-mapped I/O ports. Mutator process running on the processor allocates a new object, for example, by writing a request to one of these ports and reading the reference of the allocated object from another. Ports are also used to initialize the DMMU to indicate how large the heap is and read information about the state of memory module itself. All the communicating commands are listed in table 1.

Table 1. Communicating commands and specification

command	parameter	return	Specification
New	size	referenc e	Allocate a new object
Delete	reference	NULL	Delete an old object.
Getfield	reference, offset	field data	Get the field data
Putfield	reference, offset, field data	NULL	Put the filed data

DMMU is composed of two separate parts: object allocator and memory compactor. The memory intensive nature of object-oriented languages such as C++ and Java has created the need of high-performance DMM. In this paper, explicit dynamic memory management is supported. Programmer has to delete the object if it is required to free the occupied memory. Processor allocates a new object by sending an allocation request to IU. And these requests will be passed to DMMU to allocate or delete object internally. Allocator notifies memory compactor that the heap should be compacted whenever needed. A set of registers are embedded in DMMU and maintained for recording the starting address and end address of semi-spaces. Four other registers are also used to facilitate the utilization of object table. Table 2 shows the registers that are required and the usage for the purpose. Their utilization is explained in section 4 and section 5.

Table 2. Registers built in DMMU

Name	Length	Specification
FA	32-bit	Free pointer of semi-space A
FB	32-bit	Free pointer of semi-space B
FM	32-bit	Middle pointer of free space
ULH	32-bit	Used list's head of object table
ULT	32-bit	Used list's tail of object table
FLH	32-bit	Free list's head of object table
FLT	32-bit	Free list's tail of object table

In order to facilitate object relocation and to simplify the process of memory compaction, indirect addressing is supported in our scheme. Although the direct addressing saves a load instruction on field access, this presentation imposes many restrictions on the system. Removing the indirection addressing on every memory access makes it extremely expensive to relocate objects in the memory. The implementation of indirect addressing on our object-based architecture is achieved by maintaining a hardware object table and performing a lookup of the object table on each object access [12][21]. Object table indirection imposes a heavy penalty on memory accesses. In order to avoid such inefficiencies, an object-based built in cache memory is availed. Whilst conventional caches perform an association between an address and a memory location, a virtually addressed object-based cache associates a concatenation of the reference and offset with a memory location. The cache line of object-based cache is tagged directly with (reference, offset), thus contains parts of objects rather than blocks of physical memory. This organization is similar to a virtually-addressed cache, except that the reference and offset bits are concatenated rather than summed. Getfield and Putfield request can be responded within one clock cycle if cache hits.

It is worth noting that both object-based cache and DMMU may access RAM simultaneously. Therefore, a dual-port memory is provided to ensure that both of them can access the RAM concurrently. However, Multi-port memory implementing all ports in the memory cell results in great increase in area occupied. To compromise between bandwidth and area, interleaving technology are exploited and applied in our system based on 1-port memory banks [27][8][10]. The system consists of M memory banks numbered *1,2,3,…,M-1,M*, among which the addresses are distributed cyclically, that is, if i is the address of a memory location, then $j \equiv i(\mathrm{mod}\ M)$ is the address of the bank containing the location. Independent ports are connected to banks dynamically via switching network. The least significant bits of addresses select a bank, while the most significant bits are regarded as bank addresses. On further consideration, the processor is either accessing the object-based cache, or allocating/deallocating an object; therefore, there is no conflict between these two operations. Moreover, with the help of object-based cache, memory access will be greatly reduced and internal memory bus is always idle when cache hits. Thus memory compactor can utilize internal memory idle gap and process its job between cache misses. Once cache misses or object allocation occurs, memory compaction will be stalled. In such a case, 1-port memory block will satisfy system requirements and no dual-port memory is needed. However, the first scheme is exploited in current stage.

4. Object addressing

All the objects are referred by a location-independent object reference, actually an index into an object table. This indirect object presentation makes relocation easier because object's memory address is stored in only one place. Object table entry (OTE) contains object's size, memory base address, and object state bits. The format of OTE is shown in fig. 2. Both object and object table entry are word aligned in memory, thus the two least significant bits of object base address, object size, and next entry address can be used as object state bits. As we propose a bidirectional compaction algorithm, the S bit of OTE is used to indicate in which semi-space the object resides. And the D bit renders deletion of the object if it is asserted. The R bits are not used at present.

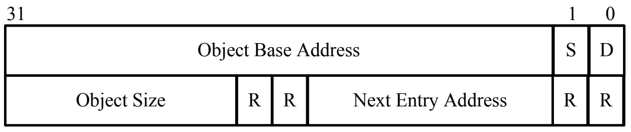

Figure 2. Object table entry

Profiling results for various Java programs indicate that the average object size is around 30 bytes [2]. And C++ programs allocate a significant number of dynamic objects on the average size of 170 bytes [1]. Hence an offset of 12 bits would cover the majority of objects in practice, and larger objects can be broken in to smaller objects by the compiler. In order to keep the original order of objects in memory, next entry address points to next OTE. This can ameliorate locality problems because the allocation ordering is usually more similar to subsequent access orderings than an arbitrary ordering imposed by a garbage collector. Furthermore, the sliding compaction process needs to know where to find the next allocated object in heap semi-space. And for 32-bit memory space, 20-bit is left for next entry.

The object table is organized as two lists: free-list and used-list, as illustrated in fig. 3. Memory compactor scans the used-list and produces free OTE, the free-list's tail and used-list's head are maintained by allocator. Once memory compaction starts operating, Compactor scans the used-list one by one. If a deleted object of specified semi-space is found, the OTE is removed from the used-list and added to the tail of free-list. On the contrary, the head of free-list and the tail of used-list are maintained by allocator, which consumes free OTE and produces used OTE. The reason why we organize the object table in such a way is that: firstly, the original allocation order of objects will not be changed. Secondly, it's convenient to search object table, avoiding scanning the whole table to find out a free OTE. Finally, also the most important one, the allocation and deallocation duration will be bounded. This is really important for systems with real-time constraints.Object table resides in the heap space, which is identified by an object reference and initialized by processor when system booting up.

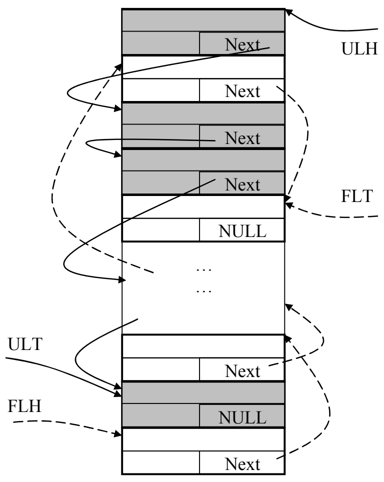

Figure 3. The organization of object table

5. Dynamic memory management

A bidirectional memory compaction algorithm is addressed in this research which describes that processor can freely access the heap space during heap compaction. The basic concept of this algorithm is illustrated in fig. 4. Heap space is divided into two semi-spaces, called semi-space A and semi-space B, as shown in fig. 4 (A). At the beginning of system booting up, objects are allocated in semi-space A until it is exhausted. The live objects are then compacted into one side of this semi-space, while allocation continues concurrently in semi-space B. When B is exhausted, the process is repeated in reverse, as illustrated in fig. 4 (B), (C) and (D) respectively, and the processor can access the heap while memory compaction is going on.

Storage is allocated by advancing a free space pointer over one of the semi spaces. Allocator is the only consumer of heap space. If allocation request is received, a block of memory is allocated for the new object by advancing the free space pointer (FA or FB). Allocator returns the free list head as reference to processor and modifies the free-list and used-list as well. When deallocating object, allocator just sets the deletion bits in the OTE and leaves deallocation to memory compactor. Since all dead objects are explicitly deleted by programmer, no mark phase is needed before memory compaction. The memory compactor module scans the used-list of OTE and slides live objects to one side of the heap, and adds the freed OTE to the tail of free-list. In this case, compaction phase typically performs one pass over semi-space. As a result of compaction, all the live objects in one semi-space are moved into a single contiguous block of semi-space; the memory left unused after compaction is recycled. If semi-space collection completes, the free space pointer of this space is updated. As shown in fig. 4 (D), the free space pointer FA is replaced with FA'. Since live objects slide to the two sides of heap, we call this bidirectional compaction algorithm.

The boundary FM of A and B is not fixed. Once semi-space compaction finished, the boundary will be redefined as the middle point of free space between FA and FB. As it can be seen from fig. 4 (D), the free space middle pointer slides to left a little and points to the middle point of free space. In such a way, the available free space is equally distributed in the two semi-spaces. Accordingly, the collection times may be reduced in some cases. For example, programs always create some longevous objects when system starting up. As a result, most of the space of A is occupied by these long-live objects and has not enough space to allocate new objects. Consequently, a new garbage collection is triggered immediately by allocator, while the compaction of semi-space B has not yet finished at this moment. Fixed boundary will result in frequently processing unit stall and long time system pause.

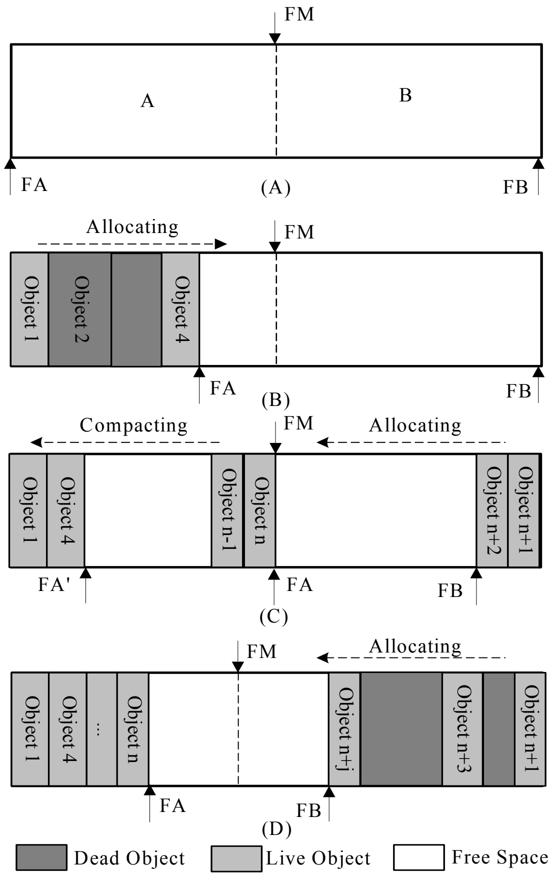

Figure 4. Bidirectional memory compaction algorithm

From fig. 4 we can describe that the long-living objects move towards the two sides of heap space. After several times compaction, they will stay in one place for a long time if no older objects in the same semi-space are deleted. This will greatly reduce the memory traffic during space compaction. Additionally, the whole space memory is utilized to run application, and no twice the amount of memory is required. The most important thing is that heap access and memory compaction can be done concurrently. Of course, all of this is based on the support of the dual-port memory.

6. Object-based cache

Object cache is mostly related to object-oriented architectures. Smalltalk systems such as SOAR, Intel 432 and Mushroom provide architectural support for efficient object addressing and accessing [12][21][5]. Further, various stack based machine provide hardware support for JVM and they are mainly focus on efficient manipulation and relocation of objects. Object-based caches were first proposed by Williams in Mushroom project. But it was never completed and all performance data they obtained was from earlier simulations. [20] and [7] incorporated some of this good idea and object-based cache architecture was implemented in Java processors respectively.

A 32-KByte, two-way associative, write-back cache with an eight-word line size is exploited in our self-maintained memory module. Memory operations that hit the cache complete in one machine cycle. Memory operations that miss the cache require approximately 80 cycles. Our simulation results show that object-based cache is sensitive to cache block size and less insensitive to cache capacities. Since most of objects are small, 32-byte cache line size is recommended. Fig. 5 shows the overall architecture of our new object cache with explanation that we can see that reference and most significant bits of offset are concatenated to form the input of hash function. In this research, we exploit a XOR-based hash function making hash module a complete combinational circuit and index can be calculated in constant time. Another promotion of this new object cache is that the tag is not the concatenation of reference and offset but just contains LSB of reference and partial offset. The 32-bit object references and 16-bit offset pair result in a large overhead for maintaining the tag bits. As discussed previously, object reference is actually the index into the object table, therefore, only 20 least significant bits of reference is necessary.

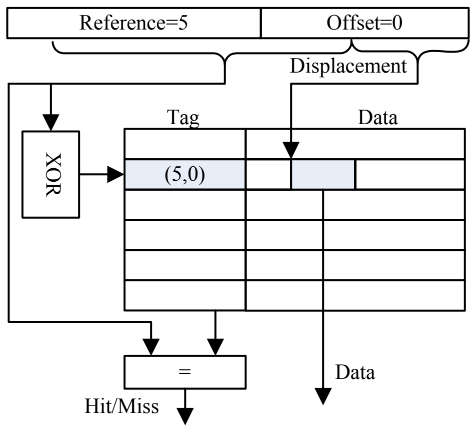

Figure 5. The concept of object-based cache

When the index of tags is calculated, the reference selected from tag is compared with the input reference. If they are equal, then cache hits. Once cache miss occurs, object physical address is obtained from object table according to reference. Missed data are read from memory afterwards.

This object-based cache compromises between direct and indirect addressing and eliminates the base-offset addition overhead; of course the two memory accesses introduced by object table indirection are also replaced by one cache access. Hash function is of main concern while designing object-based caches. Cache interference occurs when more addresses are mapped onto the same cache location. Thus the partitioning of the address to choose the tag bits, block index bits and block offset bits is critical for the performance measure of object cache. Poor choice of the block index bits would result in frequent cache interference. [12] and [20] list four ways in which the block offset and block index bits are generated. However, what constitutes a good hash function depends on the application. In any application, good hash functions have to perform well on two criteria. First, there is no added latency of computing the hash value. Second, the hash function must succeed in spreading the most frequently occurring patterns over all indices. This in turn depends on the executed programs and their input data [11]. Object programs obviously have different memory access patterns from other programs: The first pattern in object-oriented application is consecutive accessing of objects with sequential references. Since lots of object dies when they are young, the objects, which are being accessed, are almost newly created, except some longevous ones, which are created when application starts up. New objects are allocated orderly with continuous references (index into object table) and they are the most likely objects that will be accessed recently. The second pattern is the same object access. Because of the locality of programming execution and memory access, the same object will be accessed again very likely as it is accessed first time. These are the two obviously and important memory access patterns of object-oriented application. Thus if we can construct a hash function to map these two most frequently occurring patterns without conflicts, the performance of object cache can be greatly promoted.

XOR-based hash function computes each set index bit as the Exclusive OR (XOR) of a subset of the bits in the address. The reason of selecting such configuration is that the XOR-based hash functions have the benefit of computing set index with a low latency. Furthermore, they perform well for all applications, ranging from interleaved memories to branch predictors. Vandierendonck et al. constructed XOR-based hash functions based on NULL space theory, and these functions provided conflict-free mapping for a number of patterns and their functions map 2m bits to m(=2k) bits which are conflict free [11]. But they didn't construct hash functions which are conflict free when m is odd. Sung-Jin Cho et al. developed this method and design new hash functions, which compute each set index bit as XOR of a subset of the bits in the address by using the concepts of rank and NULL Space [22].

It's proved that these hash functions map the patterns (rows, columns, (anti)diagonal, and rectangles) without conflicting [22] . Imaging objects make up of a two dimension arrays, each object is a row of the array and object data members are elements in the row. Then the object access patterns mentioned previously are analog to the row pattern and column pattern of arrays. Because modeled XOR-based hash functions map these patterns conflict free, the objects with consecutive reference can be mapped to different cache block, and different blocks of the same object can also be mapped without conflicts. For details of this XOR-based hash function, please refer to [11] and [22].

7. Preliminary Performance

We have developed the self-maintained memory module in VHDL, and integrate this memory module into our object-oriented processor system. We also developed a translator to translate the object manipulate bytecode of Java programs into native assembly language of our processor, so that the performance of this memory module can be obtained before the compiler of our object-oriented processor is available.

Furthermore, some manually generated benchmarks are used to test the performance of our design.

We investigate some of the Java programs by traces obtained from a modified JVM. The modified JVM emits object level events, such as object allocation, object access and object deallocation. Object reference, object size and access offset are all logged in a trace file. The trace is a complete description of the execution of programs in terms of its effects on the objects, but is mostly independent of the actual implementation within JVM. In this experiment, we use Sun's Java2 Platform, Standard Edition, 1.3.1_01 as the experiment platform. By using the instruction tracing ability of JVM and JVMPI, we collected the object creation; object access and object destroy events for each Java programs. The benchmarks and programs are configured to run in interpreter mode in order to gather the original execution behavior without hotspot's distortion [19].

Table 3. Object-based cache hit ratio (%)

Benchmark	**Cache Size (KB)**	**Cache Line Size (Bytes)**		
		16	**32**	**64**
Address	16	89.37	95.97	97.76
	32	89.86	96.34	98.31
	64	90.13	96.54	98.59
	128	90.27	96.67	98.73
Javacc	16	96.11	99.26	99.54
	32	96.32	99.46	99.70
	64	96.39	99.51	99.89
	128	96.40	99.53	99.94
Jedit	16	90.78	96.38	96.86
	32	91.22	96.76	97.76
	64	91.54	97.01	98.28
	128	91.79	97.19	98.60
Jtris	16	93.26	97.10	98.34
	32	93.63	97.52	98.87
	64	93.82	97.73	99.07
	128	93.89	97.93	99.16
Xbrowser	16	89.51	95.02	92.48
	32	90.68	96.15	93.66
	64	90.68	96.15	94.83
	128	91.22	96.64	94.82
Xerlin	16	87.34	94.65	91.51
	32	87.96	95.16	92.21
	64	87.95	95.15	92.83
	128	88.29	95.43	92.82

The programs being analyzed include Xbrowser, Jedit, Javacc, Xerlin, Jlayer, Jtris and Address. Xbrowser is totally free and open-source java application for browsing the web. It will execute on any OS supporting Java 2 platform. It is fully multithreaded and it means that you can open several web pages simultaneously in one session. Jlayer is a library that decodes/plays/converts MPEG 1/2/2.5 Layer 1/2/3(i.e. MP3) in real time for the Java platform. Xerlin is an open source XML modeling application written for the Java 2 platform. It is being written by a team of engineers interested in seeing a nice user interface for working with XML files. Other benchmarks are also open source project on www.souceforge.net.

In table 3, we illustrate the hit ratio of object caches when running benchmarks. Table shows that object cache is highly sensitive to cache line size. Compared to 16-byte cache line, 32-byte cache line has a great upgrade in hit ratio, although the capacity of cache is the same and the number of cache lines is reduced to one half. This is because most of objects are relatively small and studies show that the average size of objects is about 30-byte around. Thus for 32-byte cache line, the entire object can be loaded in for one cache miss while accessing the entire object results in two cache misses for 16-byte cache line. In contrast with cache line size, cache capacity has less effect on cache hit ratio.

With hardware support for explicit memory management, the allocation and deletion of object is strictly bounded. For object allocation, 2 memory reads and 2 memory writes should be done to modify the free-list, used-list and set class function table for new object as well. However, only one memory write is needed for object deletion. That is to set the deleted bit in the object table. In this section, we compare our hardware bidirectional compaction collector with copy collector and compaction collector. The copy collector and compaction collector are manually programmed using assembly language.

Two C++ benchmarks are investigated. The first one is a bank simulation program. This simulation program carries out an event-driven simulation that describes different expected arrival rates for customers and different expected service times for a teller to handle a customer. The second one allocates object of a random size (0-5000 bytes) and creates objects at random position in an array. The old object is explicitly deleted if this position is already occupied. Table 4 lists the execution characteristics of these two benchmarks. In both applications, the number of collection invocations is greatly reduced in the proposed scheme compared to copy collector, this is because live objects are distributed over the two side of heap memory and the whole heap space is available for allocating. It is worth noting that compaction scheme invokes compaction only 50% as often as proposed scheme. Compaction collector is trigged when the entire space is exhausted, while for proposed scheme, compaction will be started once one of the semi-spaces is full. Table 4 also illustrates a comparison in terms of processor cycles. Both bank simulation and random allocation are dynamic memory management intensive. As a result, the proposed scheme improves the overall system performance a lot.

Since bidirectional compaction collector is a concurrent hardware collector, processor stalls only when the accessed object is been writing to or read from the memory by cache and at the same time this object is been sliding to one side of heap memory, however, processor always stalls for software-collector until garbage collection finished. As a result, the maximum stall duration for software-collector is much longer than bidirectional memory compaction collector. As shown in table 4, the max stall duration of software-collector can be as high as million cycles. In a parallel or distributed setting this limitation is severe; delays due to DMM may cause considerable waiting. Developers of real-time systems avoid using dynamic memory because they fear that the worst-case time and space requirements of typical dynamic memory managers are insufficiently bounded. The concurrency compaction scheme of our self-maintained memory module greatly reduces the system delay introduced by memory compaction. Long-living objects are continuously swapped between the two sections every time the copy-collector is running, thus it results in higher memory traffic than bidirectional compaction collector.

Table 4. Execution behavior of benchmarks

	Execution behavior	BDCC	Copy	Compaction
Bank Simulation	Time exhausted ($\times 10^9$cycles)	0.5	2.4	2.34
	Compaction times	18	42	9
	Stall times caused by compaction	0	42	9
	Max. stall duration ($\times 10^6$cycles)	0.00035	0.89	1.21
	Total memory traffic (MB)	0.31	5.36	0.92
Random Allocation	Time exhausted ($\times 10^9$cycles)	0.006	0.16	0.04
	Compaction times	99	256	52
	Stall times caused by compaction	0	256	52
	Max. stall duration ($\times 10^6$cycles)	0.00048	0.58	0.55
	Total memory traffic (MB)	1.26	9.04	1.41

The memory utilization comparison of these two benchmarks is shown in fig. 6 and fig. 7. It can be seen that the bidirectional compaction collector (BDCC) obtains higher memory utilization ratio than copy collector. This is because only half of heap space is available for copy collector. And also this explains why copy collector results in more garbage collections. Although BDCC splits the whole space into two semi-spaces as done by copy collector, the whole space is utilized to allocate objects. BDCC is a concurrent collector, thus memory compaction is going on in one semi-space while new object is allocated in another semi-space. And hardware-assisted memory compaction of first semi-space always completes before the second semi-space is exhausted. As a result of compaction termination, occupancy factor decreases suddenly. This is why proposed scheme has lower memory utilization than compaction collector, but not comparable in theory.

Fig. 8 and fig. 9 show the accumulated memory traffic for these three collectors. The program execution based on hardware bidirectional compaction collector terminates much earlier than copy collector and compaction collector. The bidirectional compaction introduces higher memory traffic in the initial state, since it is a concurrent hardware collector, both processor core and DMMU can do their job simultaneously. In addition, all memory compaction are done before program completion. For the convenient of comparison, we think that the memory traffic dose not change after program termination. As can be seen from the figures, there has been a steady increase in memory traffic for based copy collector. However, it is observed that BDCC and compaction collector have similar memory traffic caused by memory compaction.

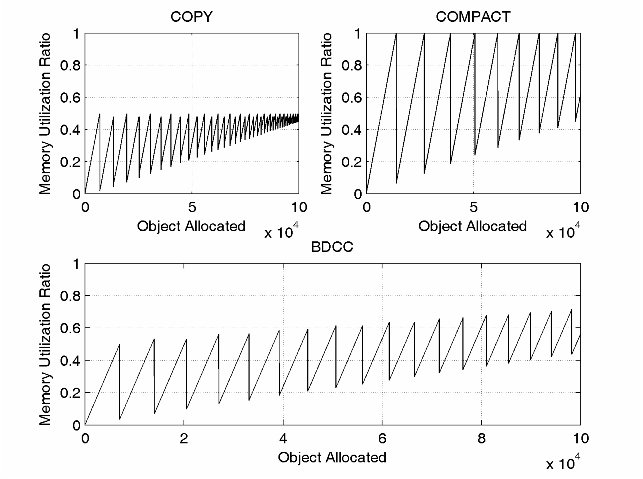

Figure 6. Memory utilization ratio comparison for BS

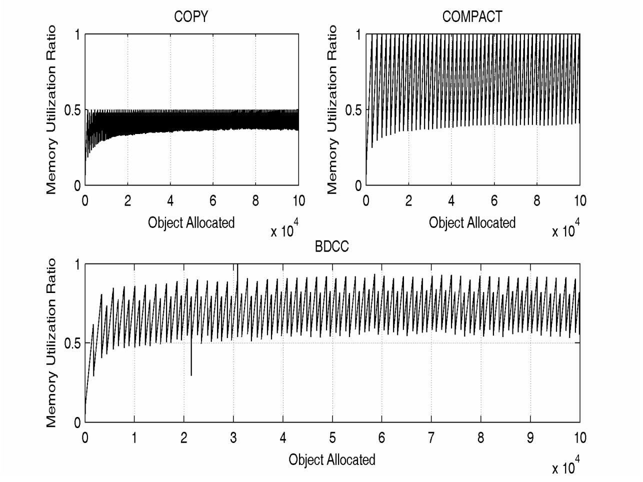

Figure 7. Memory utilization ratio comparison for RA

8. Conclusion

In this paper, we propose hardware assisted memory module, which supports explicit dynamic memory management. The given architecture facilitates support for object addressing, object allocation and heap compaction. A new bidirectional compaction collector is presented which is a concurrent memory compactor working with the help of object cache. Mutator process can access object field while memory compaction under process which ultimately optimizes the total operation time. Experiments results show that this new collector will greatly speed up the execution of object-oriented programs.

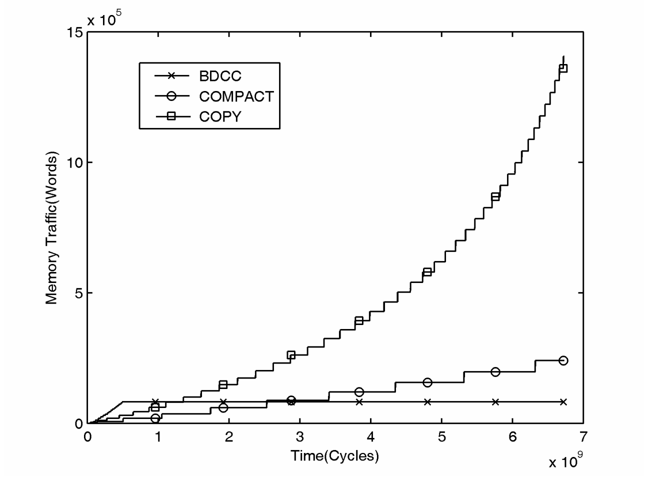

Figure 8. Accumulated memory traffic comparison for BS

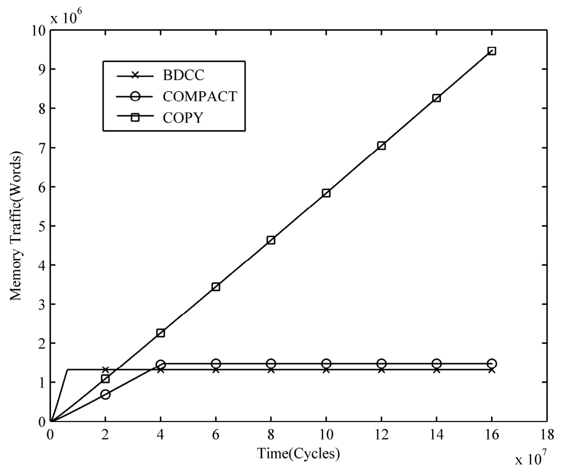

Figure 9. Accumulated memory traffic comparison for RA

9. REFERENCES

[1] Benjamin Zorn, Custo-Malloc, *efficient synthesized memory allocators.* Technical Report CU-CS-602-92, Computer Science Department, University of Colorado, July 1992.

[2] Brad Calder, Dirk Grunwald, Benjamin Zorn, Quantifying Behavioral Difference Between C and C++ programs. *Journal of Programming Languages*, VOL. 2, NO.4, 313-351, 1994.

[3] C. D. Lo, W. Srisa-an and J. M. Chang, A multithreaded concurrent garbage collector which parallelizes the new instruction in Java. In *International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida, April 15-19, 2002, pp. 59-64.

[4] Chia-Tien Dan Lo, The Design of a Self-Maintained Memory Module for Real-Time Systems. *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, 337-342, 2003.

[5] Dieckmann S, Hölzle U, A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. *In 13th European Conference on Object-Oriented Programming*, pp. 92-115, 1999.

[6] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson, Architecture of SOAR: Smalltalk on a RISC. *In Eleventh Annual International Symposium on Computer Architecture*, 1984.

[7] Greg Wright, Matthew L. Seidl, Mario Wolczko, *An object-aware memory architecture*. Sun Technical report, 2005.

[8] H. J. Mattausch, *Hierarchical architecture for area-efficient integrated Nport memories with latency free multi-gigabit per second access bandwidth.* ELECTRONICS LETERS, vol.35, no.17, pp. 1441-1443, September 1999.

[9] Hasan Cam, Mostafa Abd-El-Barr, and Sadiq M. Sait, A High-Performance Hardware-Efficient Memory Allocation Technique and Design. *International Conference on Computer Design，Proceddings of ICCD’99*, 274– 276, 1999.

[10] Hans Jiirgen Mattausch, Hierarchical N-Port Memory Architecture based on 1-Port Memory Cells. *The 23rd European Solid-State Circuits Conference*, Southampton, UK, September 1997, pp..348-351.

[11] Hans Vandierendonck, Koen De Bosschere, XOR-Based Hash Functions. *IEEE Transactions on computers*, vol. 54, 2005.

[12] Ifor W. Williams, *Object-based memory architecture*. PHD thesis, university of Manchester, 1989

[13] James O'Toole , Scott Nettles , David Gifford, Concurrent compacting garbage collection of a persistent heap. *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pp.161-174, December 1993

[14] J. M. Chang, Gehringer E.F., A high-performance memory allocator for object-oriented systems. *IEEE Transactions on Computers*, VOL. 45, NO. 3, 357–366. 1996.

[15] J. M. Chang and W. H. Lee, A Study on Memory Allocations in C++. *Proceedings of 14th International Conference on Advanced Science and Technology*, Naperville, Illinois, Apr. 4-5, 1998. pp. 53-62.

[16] J. Morris Chang, Witawas Srisa-an, Chia-Tien Dan Lo, Edward F. Gehringer, DMMX: Dynamic memory management extensions. *The Journal of Systems andSoftware*, VOL. 63, 187-199, 2002

[17] J. Morris Chang, Witawas Srisa-an and Chia-Tien Dan Lo, Architectural Support for Dynamic Memory Management. *Proceedings of the 2000 IEEE International Conference on Computer Design*, 99-104, 2000.

[18] KD Nilsen and WJ Schmidt, A high-performance hardware-assisted real-time garbage collection system. *Journal of Programming Languages*, VOL. 2, NO. 1, 1-40, 1994.

[19] L indho lm T, Yellin F., *The Java Virtual Machine Specification*, Addison-Wesley, 1996.

[20] N. Vijaykrishnan, N. Ranganathan, R. Gadekarla, *Object-Oriented Architectural Support for a Java Processor*, ECOOP'98, pp.330-355, 1998.

[21] R. Colwell, E. Gehringer, and E. Jensen, Performance effects of architectural complexity in the intel 432. *ACM Transactions on Computer Systems*, vol. 6, pp. 296-339, 1988.

[22] Sung Jin Cho, Un-Sook Choi, Yoon-Hee Hwang, Han-Doo Kim, Modeling Efficient XOR-Based Hash Functions for Cache Memories. *International Conference on Computational Science*, pp.1067-1070, 2006.

[23] William J. Schmidt, Kelvin D. Nilsen, Performance of a Hardware-Assisted Real-Time Garbage Collector. *Proceeding of ASPLOS*, 76-85, 1994.

[24] Witawas Srisa-an, Chia-Tien Dan Lo, and J. Moms Chang, Scalable Hardware-algorithm for Mark-sweep Garbage Collection. *Proceedings of the 26th Euromicro Conference*, vol.1, 274 – 281, 2000

[25] Witawas Srisa-an,Chia-Tien Dan Lo,and Ji-en Morris Chang, Active Memory Processor: A Hardware Garbage Collector for Real-Time Java Embedded Devices, *IEEE TRANSACTIONS ON MOBILE COMPUTING*, vol. 2, no. 2, 89-101,2003

[26] Yau Chi Hang, Tan Yi Yu, Fong Anthony S., and Yu Wing Shing, Hardware Concurrent Garbage Collection for Short-lived Objects in Mobile Java Devices. *Lecture Notes in Computer Science*, vol. 3824, pp 47-56. Dec 2005, Springer

[27] Zhaomin Zhu, Koh Johguchi, A Novel Hierarchical Multi-port Cache. In *the 29th European Solid-State Circuits Conference*, Estoril, Portugal, September 2003, pp. 405 – 408.