An Efficient Memory System for Java

Li, Richard C. L. and Fong, Anthony S. S.,

Department of Electronic Engineering
City University of Hong Kong
83 Tat Chee Avenue, Kowloon, Hong Kong

Summary
One of the significant issues that hinder the performance of Java program execution is dynamic memory usage. Some researchers stated that in executing Java programs, 15.58% of the CPU time is used in handling memory allocation requests and 28.08% of the CPU time in garbage collection. A statistical study on the dynamic memory usage behavior of both desktops and servers, Java applications show similar locality, and that memory allocation requests are concentrated on small block sizes (< 1K bytes), and blocks allocated usually have short life times. Based on these findings, we proposed a hardware/software approach in handling memory allocation and deallocation requests that gives a 17% overall performance gain in Java program execution.

Key words:
Java, dynamic memory management, memory allocation, memory deallocation.

Introduction

In the technology world, there are motivations and reasons for designs and implementations. When time passes and the technology evolves, many reasons that support old designs and implementations may no longer be valid, but new designs and implementations may still follow old reasons to develop without asking why. The evolution in microprocessor architecture design is under this situation. During the last few decades, although the computer architecture design evolved from CISC to RISC and VLIW, the basic model is still von Neumann, which was developed based on procedural programming model. When the mainstream programming model in software development moved from procedural paradigm to object-oriented paradigm, many of the old reasons to support the von Neumann model are no longer valid. There are new reasons established in control flow, protection and memory management of the object-oriented paradigm. It comes to a point that we have to rethink how the computer architecture should be developed based on these new scenarios.

Manuscript received June 5, 2007
Manuscript revised June 20, 2007

Among these new reasons, memory management plays an important part, because [2] and [3] stated that a C++ program performs an order of magnitude more memory allocations than comparable C program. For Java program, the situation is even worse that the amount of memory allocation requests is much more than a C++ program, and the allocated objects need to be collected automatically by the runtime environment, [1] stated that 15.58% of CPU time is used in handling memory allocation requests and 28.08% of the CPU time is used in garbage collection. If the memory management can be handled by hardware in an effective way, the performance of Java execution can be greatly enhanced.

2. Dynamic Memory Usage in Java

In order to obtain the statistics on the memory allocation and deallocation behavior of Java programs, a tracing tool was developed to collect memory allocation and deallocation events during Java program execution for further analysis. This tracing tool is a profiler agent developed according to the specification of the Java Virtual Machine Profiler Interface (JVMPI). When executing a Java program using an instance of the Java Virtual Machine (JVM) with the profiler agent installed, all the memory allocation and deallocation events can be captured for further analysis.

The choice of the Java applications under test should covered different areas including desktop applications and server applications, therefore the applications shown in Table 1 are chosen for the evaluation.

Table 1 Java Applications chosen for Evaluation

Java Application	**Description**
SPEC JVM98 benchmark	Standard benchmark suite
Java2D demo	Typical desktop Java application (single-user, multi-threads)
Pet Store	Typical enterprise Java application (multi-users, multi-threads)

By executing and profiling these applications, the memory allocation and deallocation events were obtained and passed to an analyzer to analyze the data in two different dimensions:

1. Size of block requested
2. Life time of a block

Fig. 1 to 4 show the 3-dimensional plots of the distribution of blocks requested against the block size and block life. All 3 tested applications show similar distribution in the plots that memory blocks requested by the applications concentrate on small sizes and short life times. This locality finding is a good seed for developing an efficient memory management system for Java.

3. Hardware Support of Memory Management for Java

3.1 Description of Prior Art

In order to improve the overall performance of object-oriented program execution, a sensible approach is to improve the efficiency of the memory management system of the execution environment. Many researchers had proposed many hardware approaches to improve the efficiency of Java memory management system [4] [5] [6] [7] [8] [9]. These approaches all uses tree based combinational logic to locate free memory blocks and to mark blocks used/freed. Chang and Srisa belongs to the same research team and proposed an modified buddy system as the core of their solution; while Cam utilizes the basic idea of Chang's suggestion and proposes another structure which can generate less fragmentation than Chang's method, but it requires much more logic gates to implement.

Both methods can do memory allocation and deallocation requests in a single-cycle, but they can only detect free blocks with sizes in the power of 2. In addition, the trees will become too complex to implement if the total number of memory units is large. For example, if the basic unit for allocation is 16 bytes and the total memory is 128MB, the size of the bit-vector is 8M bits. To implement such a system using Chang's design requires a tree with (28M)/2 nodes. If Cam's design is applied, even more nodes are needed. Apparently it is impractical to implement such a design in a chip when the number of nodes is this much.

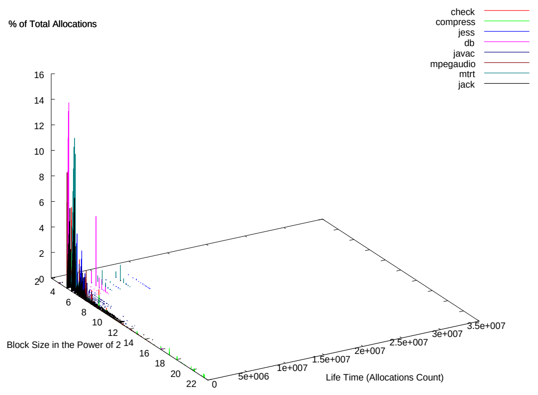

Fig. 1 Dynamic Memory Usage Characteristics of JVM-98 Benchmarks

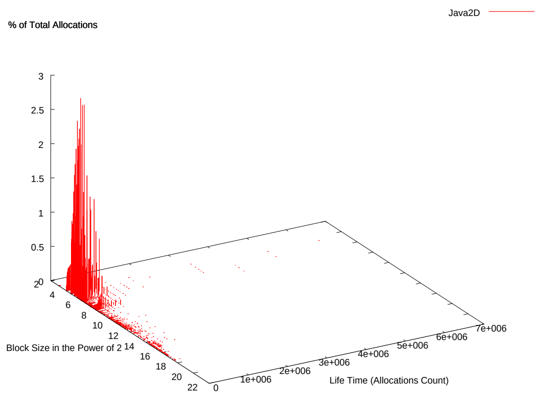

Fig. 2 Dynamic Memory Usage Characteristics of Java2D Demo

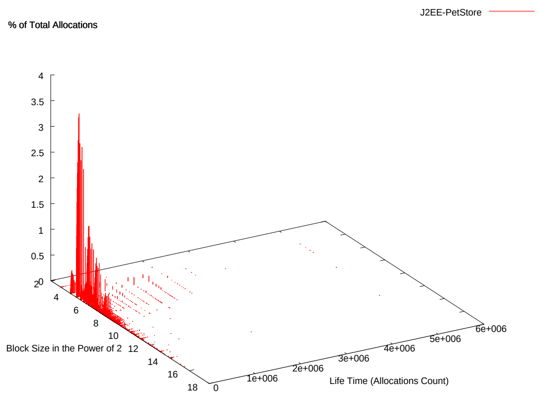

Fig. 3 Dynamic Memory Usage Characteristics of J2EE PetStore Demo

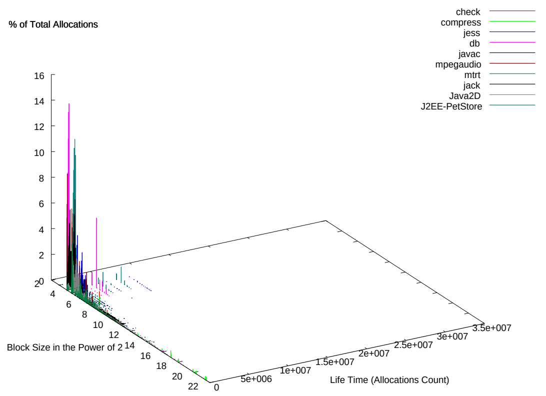

Fig. 4 Dynamic Memory Usage Characteristics of All Applications

To overcome this problem, larger units may be used to reduce the total number of blocks, but this will lead to greater internal fragmentation. Another approach is to partition the memory into many regions so that the hardware tree is used for managing only one region and the operating system is responsible to switch the active region for the hardware to work on from time to time. This method ruins the performance of the hardware approaches, as a lot of software overhead is required in augmenting the hardware.

3.2 Hardware/Software Memory Management System

3.2.1 Locality Characteristics that Driven the Design

Based on our study on the dynamic memory usage behavior in Java programs, we have concluded three locality characteristics:

1. Dynamic memory allocations are heavily used by Java programs. A simple Othello game applet generates about 600K memory allocation requests for one game play [10].

2. Dynamic allocation requests are concentrate on small block sizes. Around 90% of the total allocation requests are with block sizes less than 256 bytes and around 99.5% of the total allocation requests are with block sizes less than 1K bytes.

3. Most of the blocks allocated have short lifetime, and blocks with small sizes have a higher probability in having short lifetime as well.

These three characteristics conclude that the memory allocation and deallocation behavior of Java programs have certain localities. These localities focus on small sized memory allocations/deallocation requests. Therefore to improve the efficiency of memory allocation/deallocation request handling for Java runtime environment, a hardware/software co-design can be applied. For small block sizes, a relatively simple hardware can be used to gain better performance; while for larger block sizes, a traditional malloc-like software method can be used. This approach can facilitate the memory allocations and deallocations to be efficiently done and the performance can be greatly enhanced.

3.2.2 Detailed Description of the Design

Allocation requests are first classified into small size and large size types. This process is implemented by a combinational logic to detect which range is the size resides in and triggers appropriate process. If an allocation request belongs to small size type, the block size will be passed to the small size allocation hardware and generate a block reference. If an allocation request belongs to large size, a software trap will occur and the control will be passed to the appropriate software routine to handle the request depends on which type the request belongs to. Based on the statistical results, the threshold to divide small size requests from large size is 1K bytes. Deallocation requests use the same type information assigned during the allocation process and use the same components to do a reverse operation to free up the memory units.

To provide effective memory management in hardware, the main memory is divided into fixed sized regions, so that the memory address is divided into a region address and a region offset, see Fig 5. The region address is used to identify a specific region, while the region offset is used to address data values within a region.

Memory Address

Region Address	Region Offset

Fig. 5 Region Addressing

The Small Object Heap (SOH), which is a collection of regions partitioned from the main memory, and serves the purpose of allocating small blocks. It does not require to be contiguous, but it is built up by many fragments such that each fragment is a contiguous set of regions, see Fig 6. This heap is maintained by the operating system and the overall size can be enlarged or shrunk according to the demands of small size allocation requests.

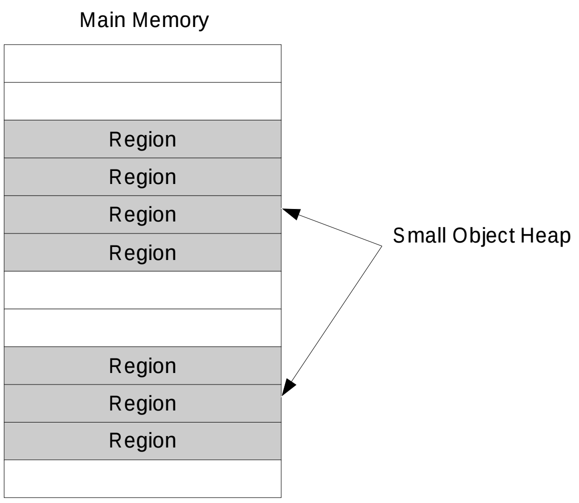

Fig. 6 Small Object Heap Fragments

Each region in the Small Object Heap (SOH) has a bit-vector, which records the used/freed information of each memory units within a region. Each unit is the minimum distinctive entity for memory management. The size of a memory unit should be somewhat smaller than the size of a region, but it cannot be too small (e.g. 1 byte) otherwise the memory management will not be effective as the bit-vector for a region will be too long. In this way, the region offset is divided into a unit address and a unit offset, see Fig 7. The unit address is used to address each unit within a region; while the unit offset is used to address each byte of data values within a memory unit. The size of the bit-vector associated with a region is governed by the following formula:

$$\textit{bit-vector length} = 2^{\textit{length of unit address}} \text{ bits}$$

When a small size allocation request arrives, the size of the request is quantized into number of memory units. Then a suitable region in the Small Object Heap (SOH) is selected for handling this request and a search is conducted on the bit-vector associated to this region to find out the first contiguous block of units with the quantized size. The unit address of the block is generated and combined with the region address to form the memory address. The memory address of the block is then returned to fulfill the request and the allocated units are marked as used in the bit-vector. When a small size deallocation request arrives, the process is reversed. First the block address is sliced to produce a region address and a unit address. Then the region address is used to find out which region the block resides in. Using the unit address and the quantized size of the block, deallocated units are marked as freed in the bit-vector.

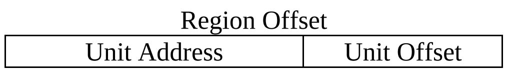

Fig. 7 Unit Addressing

To speed up this algorithm, a hardware structure named Memory Management Cache (MMC) is used so that an allocation or deallocation request takes only 1 cycle or a few pipeline stages to process. The Memory Management Cache (MMC) is a fast storage that holds the information of a subset of regions within the Small Object Heap (SOH). See Fig 8.

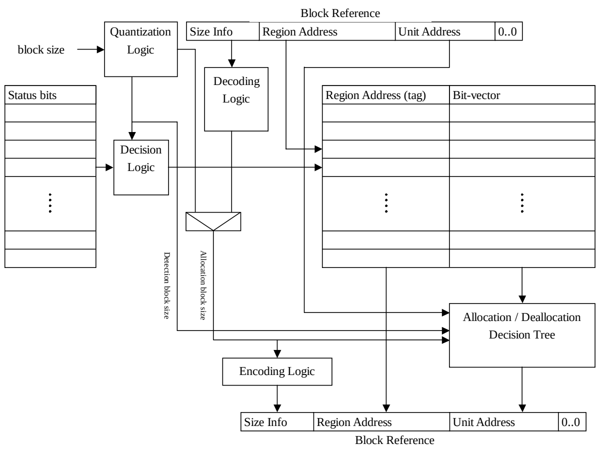

Fig. 8 Memory Management Cache

Each cache line represents a region and consists of three parts: Region Address, Bit-vector, and a status bit. The region address defines which region this cache line is representing and uses to match with the region address provided by a deallocation request to select appropriate cache line for deallocation. The bit-vector stores the used/freed information associated with the region. The status bits store information that helps to select appropriate region for handling an allocation request. It consists of the Most Recently De-allocated (MRD) information, the Most Recently Allocated (MRA) information and the information whether the region has any free blocks of each of the detectable sizes given by the allocation/deallocation decision tree.

The Decision Logic is used to select which region for allocation. It uses the information provided by the status bits of all cache lines and the block size of the allocation request to make the decision. The allocation/deallocation decision tree is responsible for locating free blocks, marking the bit vector, generating the region offset and updating the status bits. To make a better search of free blocks, we proposed a design that is capable to detect free blocks with more discrete sizes and not restricted to sizes of power of 2. The way to do this is to group multiple bits in the size to address the level of the decision tree. If we group 2 bits at a time, we can detect the following discrete units:

1, 2, 3, 4, 8, 12, 16, 32, 48, 64

An implementation of the prefix circuit based on grouping 2 bits is shown in Fig 9.

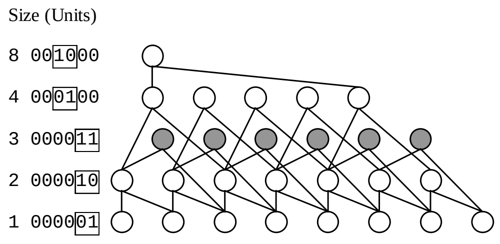

Fig. 9 Multi-bit Prefix Circuit

When a small size allocation request arrives, the block size will be quantized into 2 different sizes using the quantization logic: the detection block size and the allocation block size. The detection block size is used by the detection logic for cache line selection and the allocation/deallocation decision tree for free block lookup; while the allocation block size is used by the encoding logic and the allocation/deallocation decision tree for marking bit-vector. The allocation block size is generated by quantizing the block size in terms of memory units, while the block detection size is generated by quantizing the block allocation size into one of the detectable sizes of the allocation/deallocation decision tree.

After quantization of sizes, the detection block size will be passed in the decision logic, and it will suggest a region for handling the allocation request using the status bits of all cache lines. As the status bits already contain information if any free blocks can be detected with a given detection block size, the region selected guarantees a free block with the given size can be detected.

After a cache line is selected, the bit-vector will be passed to the allocation/deallocation decision tree along with the detection block size and allocation block size. The decision tree will locate the free block and generates its unit address, which is then combined with a unit offset with zero value to produce the region offset. Then the units allocated are marked used by consulting the unit address and the allocation block size. In addition, a new copy of status bits will be generated by the decision tree and updates the old copy in the cache line.

Finally the allocation block size is encoded into a size info and combined with the region address, the unit address and the zeroed unit offset to form a block reference, see Fig 10. If a region cannot be found in the very beginning for allocation, the control will be passed to the operating system to kick out some cache lines and bring in some new lines.

Block Reference

Size Info.	Memory Address

Fig. 10 Block Reference

When a small size deallocation request arrives, the block reference will be decomposed into a size info, a region address, an unit address, and a zeroed unit offset. Then, the size info will be passed into the decoding logic and produce the allocation block size. The region address will be used to match all the region addresses of the cache lines. If a miss occurs, the control will be passed to the operating system and kick out a cache line and bring in the required line. If a hit occurs, the deallocation process continues. First the hit cache line is enabled and the bit-vector will be passed to the allocation/deallocation decision tree. With the bit-vector, the allocation block size, and the unit address, the decision tree will mark the units de-allocated as freed in the bit vector. Also, a new copy of status bits will be generated by the decision tree and updates the old copy in the cache line.

4. Performance Evaluation of the Hardware Support

4.1 The Experiment

In order to evaluate the performance of the proposed hardware/software approach in handling memory allocation and deallocation requests of Java programs. A high level behavioral model was built for evaluation by injecting events of memory allocation/deallocation when executing benchmark programs into the model. The benchmark programs chosen are the same used for obtaining the statistical behavior. The model was setup with the hardware proposed and a simple LRU replacement algorithm for handling the memory management cache miss. The hardware parameters chosen in the simulation is shown in Table 2.

Table 2 Optimized Configurable Parameters

Parameter	Value
No. of bits grouped in multi-bit prefix circuit	2
No. of bits of the unit offset	3
No. of bits on the region offset	16
Maximum allocatable size	1K bytes
No. of cache lines	16

In this simulation, the events captured in the previous experiment were reused to feed in the simulation model and captured the behavior of the operation of the proposed hardware. The information captured includes:

1. The cache hit rate during memory allocation
2. The cache hit rate during memory deallocation
3. The overall cache hit rate
4. Average internal fragmentation of regions
5. Average region utilization during kick out from cache
6. The oversize rate of allocation requests that the block sizes of allocation requests exceeded the limit and will not be processed by the hardware.

The hit rates captured are used to evaluate the efficiency of the cache, while the internal fragmentation and region utilization during kick out are used to evaluate the efficiency of the usage of memory. The oversize rate of allocation requests are used to calculate the effective time required to process a request.

4.2 The Simulation Results and Analysis

The analysis shows that the hardware proposed gives a high overall hit rate at around 98.4% in handling memory allocation and deallocation requests when using the default values in configurable parameters of the simulation model. The default values of the configurable parameters are obtained by choosing the optimal values in different set of tests. The average internal fragmentation is around 14.8%. The average region utilization during kick out is 75.3% and the oversize rate is around 0.2%.

Table 3 Hit Rate

Benchmark Prog.	Alloc. Hit Rate	Free Hit Rate	Total Hit Rate
JVM98 200 check	100.00%	100.00%	100.00%
JVM98 201 compress	100.00%	100.00%	100.00%
JVM98 202 jess	99.94%	99.81%	99.87%
JVM98 209 db	99.97%	99.94%	99.96%
JVM98 213 javac	99.84%	96.88%	98.42%
JVM98 222 mpegaudio	99.98%	99.19%	99.63%
JVM98 227 mtrt	99.93%	99.30%	99.62%
JVM98 228 jack	99.95%	99.56%	99.75%
Java2D Demo	99.93%	99.67%	99.80%
J2EE PetStore	99.86%	99.18%	99.53%

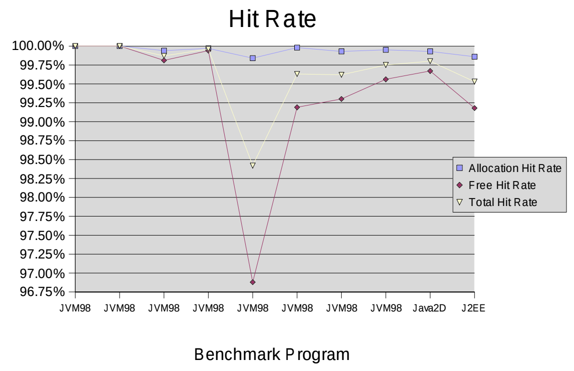

Fig. 11 Hit Rate

Table 4 Internal Fragmentation

Benchmark Program	Internal Fragmentation
JVM98 200 check	26.28%
JVM98 201 compress	24.58%
JVM98 202 jess	10.89%
JVM98 209 db	2.88%
JVM98 213 javac	10.77%
JVM98 222 mpegaudio	18.89%
JVM98 227 mtrt	3.52%
JVM98 228 jack	13.33%
Java2D Demo	15.03%
J2EE PetStore	21.78%

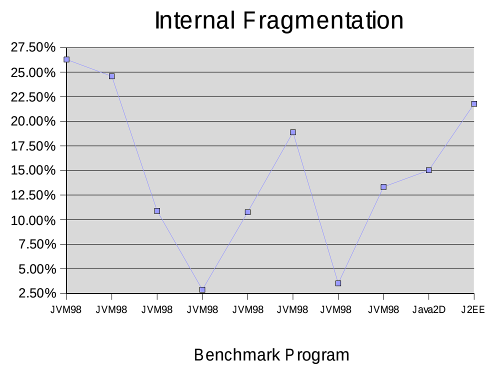

Fig. 12 Internal Fragmentation

Table 5 Region Utilization during Kick Out

Benchmark Program	Region Utilization
JVM98 200 check	
JVM98 201 compress	
JVM98 202 jess	76.43%
JVM98 209 db	69.61%
JVM98 213 javac	76.68%
JVM98 222 mpegaudio	78.96%
JVM98 227 mtrt	75.93%
JVM98 228 jack	82.68%
Java2D Demo	69.21%
J2EE PetStore	72.68%

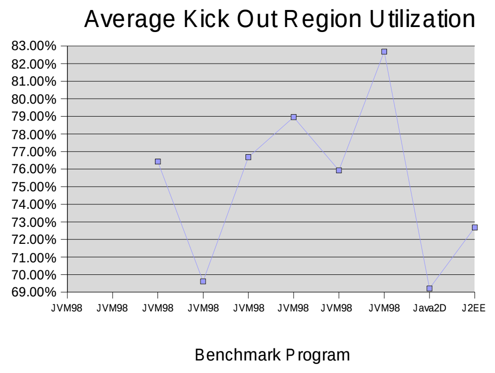

Fig. 13 Region Utilization during Kick Out

Table 5 Oversize Rate

Benchmark Program	Oversized Rate
JVM98 200 check	0.32%
JVM98 201 compress	5.08%
JVM98 202 jess	0.01%
JVM98 209 db	0.01%
JVM98 213 javac	0.05%
JVM98 222 mpegaudio	0.82%
JVM98 227 mtrt	0.00%
JVM98 228 jack	0.11%
Java2D Demo	0.37%
J2EE PetStore	0.26%

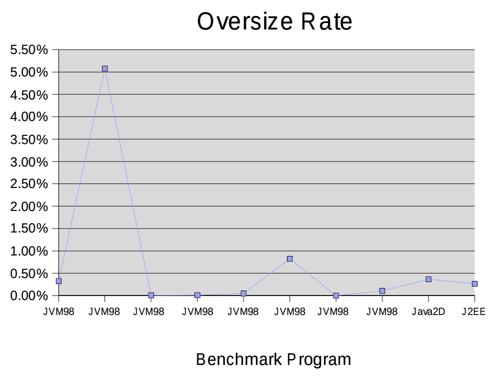

Fig. 14 Oversize Rate

When a Java program is executed under our memory management scheme, around 99.8% of all memory allocation and deallocation requests are handled by hardware while around 0.2% of memory allocation and deallocation requests are handled by software. Among these 99.8% memory allocation and memory deallocation requests, around 98.4% will generate a hit in the memory management cache during request processing. These numbers imply that around 98.2% of all memory allocation and deallocation requests can be handled by the hardware in a single cycle. 1.6% of the requests will have a penalty on the miss in the memory management cache, and 0.2% of the requests will rely on software method to fulfill the requests.

During a memory management cache miss, the system needs to kick out the least recently used bit vector with status bits to the memory and brings in a suitable bit vector with status bits to the memory manage cache. Using the default configuration parameters, the length of a bit vector is 2(16 - 3) = 8,192 bits, the length of status bits is 11 bits. Therefore during a miss in memory management cache the memory management cache needs to read and write a total (8,192 + 11) / 8 = 1,026 bytes from and to memory. Assuming reading/writing memory need a cycle for each 32-bit word, reading and writing the bit-vector with status bits from and to memory requires around 514 cycles. By adding around 500 cycles for the decision logic to decide which bit vector to bring in, the total penalty of cache miss is around 1,014 cycles.

For traditional software approach, [7] states that the best malloc software algorithm needs around an average 400 cycles to process a single memory allocation request. The effective speed required to process a memory allocation request using the proposed hard/software approach is approximately equals to:

$$\begin{aligned} \textit{Effective time for allocation} &= 98.2\% \times 1\,cycle \\ &\quad + 1.6\% \times 1014\,cycles \\ &\quad + 0.2\% \times 400\,cycles \\ &= 18\,cycles \end{aligned}$$

Comparing with the 400 cycles required by best malloc software algorithm, the performance of the proposed solution give a gain of 400 / 18 = 22 times. According to the information given in [1], around 15.58% of the Java program execution time is used for memory allocation. The overall speedup in Java program execution when using our hardware/software solution is:

$$\begin{aligned} \textit{Speedup factor} &= \frac{1}{\frac{0.1558}{22} + (1 - 0.1558)} - 1 \\ &= 17\% \end{aligned}$$

As a summary, our proposed solution can give a performance gain of 22 times in doing memory allocations, which will lead to a gain of 17% in the overall execution of Java programs.

5. Conclusion

Based on the locality characteristics of dynamic memory usage of Java programs, a hardware/software combined memory system is proposed. The hardware is responsible for handling memory allocation/deallocation requests of small size (<= 1K bytes) blocks, while traditional software malloc routine is used for handling requests of large block sizes (> 1K bytes).

The proposed hardware gives a significant efficiency in handling memory allocation and deallocation requests. It can handle 99.8% of all memory allocation and deallocation requests and gives an overall hit rate of 98.4% with internal fragmentation 14.8% and region utilization 75.3% during kick out. Using the proposed design, each memory allocation requests need an average 18 cycles to process. When comparing to pure software approach which need 400 cycles to handle each memory allocation requests, the performance gain in processing memory allocation requests is 22 times of pure software method. This performance gain can lead to a 17% gain in the performance of overall Java program execution.

Acknowledgments

The work described in this paper was partially supported by the City University of Hong Kong, Strategic Research Grant 7001847.

References

[1] Lo, C.-T.D.; Srisa-an, W.; Chang, J.M., "Who is Collecting Your Java Garbage?", In *IT Professional*, Volume: 5, Issue: 2, pages 44-50. IEEE Computer Society Press, Mar-Apr 2003.

[2] Detlefs, D.; Dosser, A.; Zorn, B., "Memory Allocation Costs in Large C and C++ Programs", In *Software - Practice and Experience*, pages 527-542. , June 1994.

[3] Calder, B.; Grunwald, D.; Zorn, B., "Quantifying Behavioral Differences Between C and C++ Programs", In *Technical Report CU-CS-698-95*, pages . Department of Computer Science, University of Colo, Jan 1995.

[4] Chang, J. M.; Gehringer, E. F., "A High Performance Memory Allocator for Object-oriented Systems", In *IEEE Transactions on Computers*, Volume: 45, Issue: 3, pages 357-366. IEEE Computer Society Press, Mar 1996.

[5] Chang, J.M.; Hasan, Y.; Lee, W.H., "A high-performance memory allocator for memory intensive applications", In *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on* , Volume: 1, pages 6 - 12 vol.1. IEEE Conference, May 2000.

[6] Cam, H.; Abd-El-Barr; Sait, S. M., "A High-Performance Hardware-Efficient Memory Allocation Technique and Design", In *International Conference on Computer Design, 1999 (ICCD '99)*, pages 274-276. IEEE Computer Society Press, Oct 1999.

[7] Srisa-An Witawas; Chia-Tien Dan Lo; J Morris Chang, "A performance analysis of the active memory system", In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pages 493 - 496. , Sep 2001.

[8] Srisa-an, W.; Lo, C.-T.D.; Chang, J.M., "Performance Enhancements to the Active Memory System", In *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2002*, pages 249-256. IEEE Computer Society Press, Sep 2002.

[9] Srisa-an, W.; Lo, C.-T.D.; Chang, J.-M., "Active memory processor: a hardware garbage collector for real-time Java embedded devices", In *Mobile Computing, IEEE Transactions on*, pages 89 - 101. , Apr-Jun 2003.

[10] Li Richard C. L.; Fong Anthony S.; Chun H. W.; Tam C. H., "Dynamic Memory Allocation Behavior in Java Programs", In *Proceedings of the ISCA 16th International Conference in Computers and Their Applications, 2001.*, pages 362-365. The International Society for Computers and Their Applications - ISCA, 2001.

Li, Richard C. L. received the first class honor BEng in Computer Engineering and the M.Phil. degrees in City University of Hong Kong in 1996 and 1999, respectively. During 1999-2006, he stayed in City University of Hong Kong to continue his PhD. study. He has been awarded one United States patents on computer architecture and design. At present he is finishing his PhD. Study and working in a self-founded software company.

Dr. Fong, Anthony S. S. received his M.Sc. degree in Computer Science from the State University of New York at Buffalo. He was awarded Ph.D. degree from University of Sunderland. He has been awarded eight United States patents on computer architecture and design and published more than sixty papers on computer architecture and design, and database. At present he is Associate Professor in City University of Hong Kong and working on a computer system project HISC for object-oriented computing.