Hardware Concurrent Garbage Collection for Short-lived Objects in Mobile Java Devices

Yau Chi Hang, Tan Yi Yu, Fong Anthony S., Yu Wing Shing

Department of Electronic Engineering, City University of Hong Kong
Tat Chee Avenue, Hong Kong
Anthony.Fong@cityu.edu.hk

Abstract. jHISC is an object-oriented processor for embedded system aiming at accelerating Java execution by hardware approach. Garbage collection is one of the critical tasks in a Java Virtual Machine. In this paper, we have conduct a study of dynamic object allocation and garbage collection behavior of Java program based on SPECjvm 98 benchmark suite and MIDP applications for mobile phones. Life, size, and reference count distribution of Java objects are measured. We found most Java objects die very young, small in size and have small number reference counts. Reference counting object cache with hardware write barrier and object allocator is proposed to provide the hardware concurrent garbage collection for small size objects in jHISC. Hardware support on write barrier greatly reduces the overhead to perform the reference count update. The reference counting collector reclaims the memory occupied by object immediately after the object become garbage. The hardware allocator provides a constant time object allocation. From the investigation, over half of Java objects can be garbage collected by the object cache that makes it unnecessary for these objects to copy to the main memory.

1. Introduction

Java is a widely used programming technology for developing applications on different kinds of mobile devices. Fast product development, portability and secure environment are the advantages that Java offers for the mobile devices. Connected Limited Device Configuration (CLDC) and Mobile Information Device Profile (MIDP) provide the core application functionality required by mobile applications, in the form of a standardized Java runtime environment and a rich set of Java APIs. Sun Microsystems announce that there are over 250 million handsets already in people's hands which support Java, over 200 different Java handset models (globally) and around 77 network operators' deploying Java based Web services in worldwide.

Java KVM is the traditional JVM on mobile devices, simple mark-sweep algorithms are used to collect garbage objects. The mark-sweep collector traverses the heap from the root to determine the reachability, and then sweep the heap by collect the reclaimed memory into a freelist. The major weakness of the classic mark-sweep GC method is that all other threads within JVM have to be suspended while the gar-

bage collector is running. The suspension time is directly proportional to the number of objects on the heap.

jHISC processor is proposed by Fong [1][2], which applies object-oriented concept into the architecture of the processor. This paper introduces a hardware garbage collection method based on reference counting algorithm, for jHISC processor.

The remainder of the paper is organized as follows: Section 2 is an overview of the related research. Section 3 is a research of the object behavior. Section 4 describes the design of the proposed garbage collector. Evaluation and further study are on the last two sections.

2. Background

2.1 Reference Counting

Reference-counting is a traditional garbage collection algorithm. The concurrent nature makes it suitable to be used in real-time embedded system. The traditional method reference counting was first developed for Lisp [13]. It was widely used in SmallTalk-80 [14] and Perl [15]. The basic theory of the reference-counting algorithm is that each object has an associated count of the references (pointers) to it. When a variable is assigned a reference to that object, the reference-count is incremented by one. Similarly, whenever a reference pointing to an object is deleted, its reference-count is decremented by one. When the reference-count becomes zero, it implies that the object is not referenced by other objects and the executing process cannot reach it. The memory occupied by this object is garbage and can be reclaimed by the collector.

The space overhead to store the reference counts is the weaknesses of the reference counting method. Extra space in each object is needed to store the reference count. For the worst case, the reference count field would have to be large enough to hold the total number of object references in the memory space [4]. Whenever a reference to an object is made, copied or deleted, it is necessary to access memory to update the object's reference count field. Furthermore, computation to update reference count degrades mutator performance.

There are researches try to apply reference counting algorithm into Java Virtual Machine (JVM). In JVM architecture, references to an object are created form three areas, which are other form objects heap, Java stacks and pc registers. Deutsch and Bobrow noted that most of the overhead is on counter updates from the frequent updates of local references (in stack and registers) [12]. Deferred reference counting is suggested by Blackburn to ignore frequent pointers mutations to the stack which eliminates most of their reference counting load [8]. It thus postpones all reclamation until it periodically enumerates the stacks.

Cyclic objects cannot be garbage collected because of its cyclic structure. Most of the research work of reference counting is used in generational garbage collection. Bacon and Rajan developed a current and localized algorithm to solve this disadvan-

tage to make RC unnecessary to work with other garbage collection algorithms [11]. The algorithm is capable of collecting garbage even in the presence of simultaneous mutation and never needs to perform a global search of the entire object heap. Local search is performing after a possible root is found. That algorithm combines the tracing and reference counting collection. It is only an incremental garbage collection algorithm because pause occurs when there is local search.

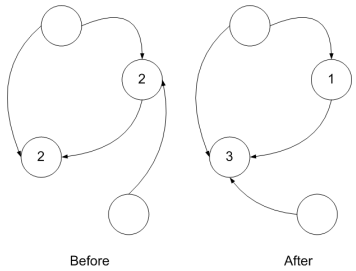

Fig. 1. Reference Count updates when a pointer is move.

2.2 Garbage Collection in Mobile Java

Due to the popular of the Java enabled mobile phone, many researches are conducted to improve the performance of Java execution in the limited resource on mobile device. Active memory processor is introduced by Chang to reduce the overhead of memory management in Java [6]. A bitmap-based processor is used to work with standard DRAM cells. Memory availability and GC information are stored in a bitmap. There are Compete Binary Tree (CBT) and a bit-sweeper inside the processor to provide constant time garbage sweeping and object allocation. Limited reference field is supported in the processor. According to the study of Chang, it can reduce the amount of the memory usage by 77%.

Table 1. Description of MIDP Benchmark Applications

Applications	Description
WebViewer	Simple Web Browser
WormGame	Popular game on mobile photo
PhotoAlbum	Image viewer that can used to view image file like JPEG, GIF and PNG
AudioDemo	Audio player include in the MIDP reference implementation

The disadvantage of the idea of Chang is that the design of the hardware memory manager is complex. It increases the cost of the hardware and power consumption. The CBT design cannot utilize the memory space well and causes internal fragmentation in the heap.

3. Behavior of Java objects

The Mobile Information Device Profile (MIDP) is a set of Java APIs which together with the Connected Limited Device Configuration (CLDC) to provide an environment for mobile devices, such as PDA and mobile phone. In this study, the object behavior of Java applications running on embedded devices based on MIDP 2.0 and CLDC 1.0.4 are investigated. There are no standard benchmark suites for MIDP applications available yet. Four applications are chosen form PennBench [3], which is a benchmark suite for embedded Java, defined by Java researchers.

In the MIDP study, we focus on the object life. We have modified the source code of the KVM to trigger garbage collector before every object allocation. The garbage collection scheme used on KVM is the simple Mark-Sweep algorithm. It is a Stop-the-World algorithm. Only garbage collector thread is active during garbage collection in progress, and other threads are suspended [5]. Object allocation is not allowed during garbage collection in progress. After the garbage collection is completed, no newly created object is in the object that makes sure all the memory used by the garbage objects are freed and available immediately for further object allocation.

Table 2. Accumulative Distribution of Object Size on Java MIDP applications

Applications	Percentage of objects which the size is smaller than <= 48 bytes
WebViewer	87.01%
WormGame	91.00%
PhotoAlbum	91.28%
AudioDemo	92.75%
Overall	90.51%

3.1 Size distribution

The object size is the first category we are concerned. On KVM runtime environment, each object has an object header which is 2 words (a word is 32-bit) in size. The minimum size of an object in KVM is 8 bytes. Each data inside Java object would use a word to store, regardless if it is an integer or a character. In our study, we found that most object sizes are small. Over 90% of objects are equal to or smaller than 48 bytes (12 words).

3.2 Age distribution

The relative age of an object we defined is the number of objects allocated between the object allocation and de-allocation events. According to the example of figure 2, there are two object allocation events between the allocation and de-allocation of object A. Based on the definition, the relative age of object A is 3. Applying the same rule on object C, the life of object C is zero.

We found that more than half of the Java objects (~50%) have relative ages of 0, and over 90% of objects have relative ages less than 16. We can make a conclusion that the life span of a typical object in embedded Java environment is short. Half of the Java objects can be garbage collected immediately before an object allocation event.

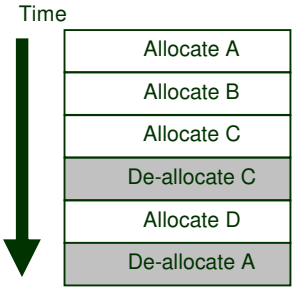

Fig. 2. Definition of relative age for Java objects

Table 3 Accumulative Distribution of Object AGE on Java MIDP applications

Applications	AGE = 0	AGE <=15
WebViewer	30.92%	83.54%
WormGame	55.80%	92.41%
PhotoAlbum	62.95%	95.59%
AudioDemo	58.15%	92.81%
Overall	51.95%	91.09%

3.3 Reference-count distribution

Jikes RVM (Research Virtual Machine) is used to study the reference-count of Java objects. It is designed to execute Java programs that are typically used in research on fundamental virtual machine design issues [9].

Our experiment is mainly targeted on the reference-counting collector. We are interested in the reasonable maximum reference-count of java objects. Through the experiment of running JVM SPEC 98 [17] in RVM, the reference-count of each object is obtained. We have modified the object header of the RVM in order to record the maximum reference-count of the object. The maximum value of the maximum reference-count is set to 255. If it is overflow, the reference-count will be reset to zero, and the object is subject to garbage collection.

According to the measurement, the average value of the maximum references is small (<= 2). There are over 99% of objects of which the maximum reference-count is

less than or equal to two. Therefore, four bits for reference-count for each object are sufficient to cover 99% of Java objects.

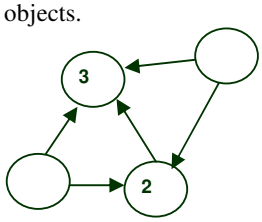

Fig. 3. Definition of reference count for Java objects.

4. jHISC architecture

In Java bytecode specification, there are several instructions defined to perform the tasks of the object-oriented features such as invokevirtual and putstatic. The ratio of the object-oriented bytecodes (including access constant pool instruction and method call instructions) on SPEC JVM 98 benchmark is 31.87% [17]. Object-oriented instructions are added into jHISC instructions set in order to accelerate the object-manipulation operations by hardware support.

In jHISC architecture, three-address register-to register instruction form is used. jHISC processor have a register engine which contains 32 general registers of32 bits wide. Beside the 32-bit data, each register contains a 4-bit type field to describe the type of the data inside the register. During the process of the method invocations caused by ivk instruction, a method frame is allocated in the register. The method frame acts as the stack frame in Java virtual machine which is the method code working space. The register engine can be used to translate all the stack operations, which defined in Java bytecode, into 3 addressing jHISC instructions to accelerate the execution of Java applications.

5. Garbage Collection for young object

Generational garbage collection is used. The whole object heap is divided into two generation spaces. They are young space and mature space. All the new objects will be allocated in young space. Objects will be promoted into mature space after it is still alive in the young space for a period of time. In order to support real-time application, concurrent garbage collection will be used in both generation spaces. On this paper, we are focusing on the young space.

5.1 Reference counting object cache

The main design concern of the memory management in young space is to provide fast object allocation. According to the research about the java object size and life, we have found that in most Java programs the vast majority of objects (often >95%) are very short-lived (i.e. are used as temporary data structures), small in size (90% objects =< 48 bytes). The object allocation occurs frequently.

The purpose of the scheme is to hold objects with their respective reference-counts in the cache to speedup the algorithm. The reference-count is modified directly in the cache to avoid accessing the main memory. For each instruction which will copy, delete or set a reference between two objects, it will check if the objects are held on the cache or not. If all the objects involved by this instruction are on the cache, the reference-count of these objects will be modified concurrently while the instruction is executing.

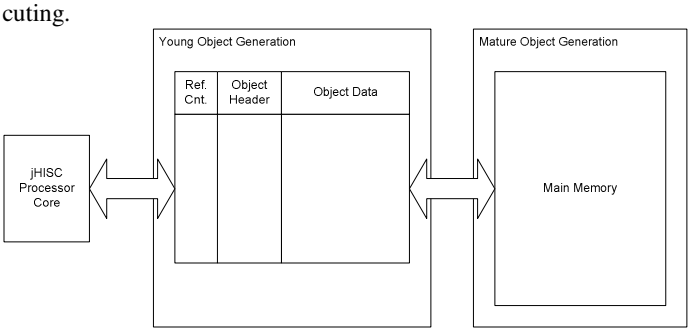

Fig. 4. Overview of jHISC Garbage Collection scheme

When the reference-count becomes zero, this implies that the object is not referenced by other objects and the running program cannot reach it. The cache line occupied by this object can then be reclaimed by the collector. When the reference-count of an object stays on cache overflow, it will be promoted to the mature space. Its reference-count field will be invalidated and it will be ignored by the reference-counting collector. Beside newly allocated instance objects, objects residing in the main memory are allowed to be brought into the cache. In these cases, their reference-counts are set to all "1"s and they will not be collected by the reference-counting collector.

There is a 3-bit reference-count field to store the reference-count. Zero represents that the object is garbage and all '1's represents overflow. There is an M-bit to represent that the object has references form the mature space's object.

One of the main drawbacks of reference-counting is its computational expense. Therefore, we only maintain reference-count of an object in the cache. After an object is moved out from the cache, it will not be collected by the reference-counting collector and the reference-count on the cache will be invalidated. It reduces the computational expense for adjusting the reference-count in main memory.

5.2 Object allocation for small size objects

Object creation takes place in the object-cache directly instead on the main memory. As the lifetimes of small-size object in Java are short and there are spatial localities in the memory allocation and deallocation patterns in Java, hardware object allocator is proposed to do the small block sizes memory allocations and deallocations. Many of objects will die in the cache, without ever being written to the main memory.

A separated object allocation bit vector and a decision tree are used to allocate an object in hardware level [6] [7]. Buddy System is used to find the appropriate contiguous free bits in a bit vector. This allocation scheme supports the objects sizes of power of two. In our instance object cache, each object will occupy a whole cache line. The young object space is divided into slots of same size. Design of the binary tree can be further simplified. The object allocator determines if there is an available slot or not. If so, the allocator locates the address.

Objects of data size smaller or equal to a cache line, are allocated in the cache directly by the hardware object allocator. For objects of size greater than a cache line, the allocation request will be transferred to the operating system. These objects will be allocated in the mature object space. A free list is maintained by the memory manager of the operating system. Suitable free space will be searched to adapt the allocation request.

5.3 Hardware write barrier for reference count update

During the execution time of the user Java program, we need to keep track of the action of the reference assignment and deletion operations and update the associated object's reference-count. Write barrier method is used to keep track of this operation.

In jHISC, we integrate write barrier into the instructions. There are four instructions responsible for reference update between objects. They are oo.pfld, oo.psfld, oo.pifld and array.store. While these reference update instruction is executing, we can update the reference-count of the object concurrently. When executing oo.pfld, an instance variable will be modified. If the data type of the variable is in reference type, write barrier will be activated. If the assigned references and the reference we are overwritten are in the object cache, the write barrier will update their reference-count. If the destination object is in the mature space and the object described by the assignment object is in the cache, the M-bit in the reference count field is set to 1.

Beside the references between objects, reference exists between stack /registers (register frame of jHISC) and objects. Reference-count update must be done on stack operations. Reference-count update process is invoked after the local registers movement operations ("data.move" instruction) if the type of the register are "reference".

6. Evaluation

A nearly real-time garbage collection with reference-counting has been done successfully. The reference-counting collector reuses the memory cell for further object allo-

cation immediately when the objects become garbage. Reclamation of short-lived garbage objects is done in-cache without requiring any accesses to main memory. One of the characteristics of reference-counting algorithm is that it can garbage collect the object as soon as the object becomes garbage.

Objects with relative age zero and size equal to or smaller than a cache line can be garbage collected on the cache and will not reside in the main memory. This kind of objects would be staying on object cache all their lives. The small-size size object which can fit into a cache line can be allocated on the object cache directly. In the object allocation process in the object cache, all objects are treated the same size which is equal to the size of each cache line. According to the object behavior we have studied, over 90% of objects (sizes smaller than or equal to 48 bytes) are allocated directly on the object cache. The objects which have zero relative age are all reclaimed by the reference-count garbage collector. There are 70% of objects dead on object cache.

Table 4. Percentage of small size and short-lived object on Java MIDP applications

Applications	Percentage of objects which the size is smaller than <= 48 bytes and relative age is equal to zero
WebViewer	83%
WormGame	72%
PhotoAlbum	43%
AudioDemo	76%
Overall	**68%**

7. Further Investigation

A concurrent garbage collector for small-size and short-lived objects is designed. This collector can be used as the collector on young space of generational garbage collection. A secondary collector is need on mature object space to collect the garbage object that cannot be reclaimed by the reference-counting algorithm such as cyclic objects and objects with many references. A heap compaction algorithm is another issue for future study. Fixed-size object allocation may cause the fragmentation in the object heap. Heap compaction is needed to remove the fragment in the heap. The existing solution is for secondary level garbage collection, and heap compaction is stop-the-world algorithm. For future investigation, we will concentrate to design a nearly concurrent algorithm to meet the real-time requirement for embedded systems.

Acknowledgements

The work described in this paper was partially supported by a grant from City University of Hong Kong (Strategic Research Grant Project No. 7001548).

References

1. Fong, A., S., HISC: A High-level Instruction Set Computer. In 7th European Simulation Symposium, pages 406-410. The Society for Computer Simulation, Oct 1995.
2. Mok Pak Lun, Richard Li, Anthony Fong. Method manipulation in an object-oriented processor. ACM SIGARCH Computer Architecture News archive. Volume 31, Issue 4, September 2003.
3. G.Chen, M.Kandemir, N.Vijaykrishnan, M.J. Irwin. PennBench: A Benchmark Suite for Embedded Java Proceedings of the 5th Workshop on Workload Characterization. Austin, TX, Nov, 2002
4. Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Management*, St. Malo, France, September 1992.
5. R. Jones, R. Lins. Garbage Collection Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons.1996
6. Witawas Srisa-an, Chai-Tien Dan Lo and J.M. Chang. Active Memory Garbage Collector for Real-Time Java Embedded Deveices. IEEE Transactions on Mobile Computing, volume 2, No. 2, pp. 89-101. 2003.
7. H.Cam, M.Abd-El-Barr, and S. M. Sait. A high-performance hardware-efficient memory allocation technique and design. International Conference on Computer Design, 1999. (ICCD '99), pp. 274-276. IEEE Computer Society Press, Oct 1999
8. S. Blackburn and K. Kathryn. Ulterior Reference Counting: Fast Garbage Collection without a Long Wait. OOPSLA' 03 ACM Conference on Object-Oriented Systems, Language and Applications, October 26-23, 2003.
9. B.Alpern, C.R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalalpeo in Java. In Processings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA'99, Denver, Colorado, November 1-5, 1999, volume 34(10) of ACM SIGPLAN Notices, pages 314-324, ACM Press, Oct. 1999.
10. H.Azatchi and E. Petrank. Integration generations with advanced reference counting garbage collectors. In International Conference on Compiler Construction, Warsaw, Poland, Apr. 2003.
11. D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In J. L. Kuden Knudsen, editor, Proceedings of 15th European Conference on Object-Oriented Programming Languages, New Orleans, Louisiana, January 15-17, 2003, volume 38(1) of ACM SIGPLAN Notices. ACM Press, Jan. 2003.
12. L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. Communications of the ACM, 19(9):522-526, September 1976.
13. Geoergo E.Collins. A method for overlapping and erasure of lists. Communications of the ACM, 3(12):655-657, December 1960.
14. Adele Goldberg and D. Robson. Smalltalk-80: The Language and its Implementation. Addison-Weslev, 1983.
15. Larry Wall and Randal L. Scheartz. Programming Perl, O'Reilly and Associates, Inc., 1991.
16. Sun claims it's winning developer space, http://www.theinquirer.net, May, 2004
17. Young-Min Lee, Byung-Chul Tak, Hye-Seon Maeng, and Shin-Dug Kim. Real-Time Java Machine for Information Appliances. IEEE Transactions, volume 46, issue 4, p.949-957. 2000.
18. SPEC JVM98. http://www.specbench.org/osg/jvm98/