Active Memory Processor: A Hardware Garbage Collector for Real-Time Java Embedded Devices

Witawas Srisa-an, *Member*, *IEEE*, Chia-Tien Dan Lo, *Member*, *IEEE*, and J. Morris Chang, *Member*, *IEEE*

Abstract—Java possesses many advantages for embedded system development, including fast product deployment, portability, security, and a small memory footprint. As Java makes inroads into the market for embedded systems, much effort is being invested in designing real-time garbage collectors. The proposed garbage-collected memory module, a bitmap-based processor with standard DRAM cells is introduced to improve the performance and predictability of dynamic memory management functions that include allocation, reference counting, and garbage collection. As a result, memory allocation can be done in constant time and sweeping can be performed in parallel by multiple modules. Thus, constant time sweeping is also achieved regardless of heap size. This is a major departure from the software counterparts where sweeping time depends largely on the size of the heap. In addition, the proposed design also supports limited-field reference counting, which has the advantage of distributing the processing cost throughout the execution. However, this cost can be quite large and results in higher power consumption due to frequent memory accesses and the complexity of the main processor. By doing reference counting operation in a coprocessor, the processing is done outside of the main processor. Moreover, the hardware cost of the proposed design is very modest (about 8,000 gates). Our study has shown that 3-bit reference counting can eliminate the need to invoke the garbage collector in all tested applications. Moreover, it also reduces the amount of memory usage by 77 percent.

Index Terms—Garbage collection, embedded systems, real-time systems, Java virtual machine, active memory.

1 Introduction

The increasing popularity of Java in embedded system environments has created the need for a high-performance *Garbage Collection* (*GC*) system for embedded devices. Often times, embedded systems have real-time constraints; therefore, nondeterministic allocation and garbage collection pause times can severely degrade runtime performances and may even result in system failures.

Java possesses many advantages for embedded system development, including fast product deployment, portability, security, and small memory footprint. As Java makes inroads into the market for embedded systems, much effort is being invested in designing real-time garbage collectors. Such efforts include Sun's Java Embedded Server [15], Esmertec's JBED (Java real-time operating system) [8], and Newmonic's PERC products [10]. Embedded systems are experiencing explosive growth due to the increasing pervasiveness of smart devices, such as Internet appliances, *Personal Digital Assistance* (*PDA*), and portable phones. For example, Allied Business Intelligence (www.alliedworld.com) reported that the Internet appliance market is expected to grow dramatically over the next few years with shipments rising from 21.4 million units in 2000 to 174.4 million units by 2006, when it will represent a $39 billion market [13]. Industry observers also predict that there will be 10 times more embedded system developers than general-purpose software developers by the year 2010 [1].

Additionally, the introduction of network-centric computing platforms such as Jini emphasizes the importance of garbage collection in small-embedded devices. In Jini, devices or services are part of a confederated computing network. Any component connected to this network possesses a truly plug and play characteristic. The Jini architecture was developed entirely in Java and all devices that are Jini-ready operate on code written in Java. Remote Method Invocation is used to pass along serializable objects from one device to the next. In this environment, Jini-ready devices can be anything from household appliances to thin client devices [7]. These devices will operate on a limited amount of physical memory. Thus, a fast and efficient garbage collector will determine the overall performance and success of such devices.

In this paper, we introduce the design of the Active Memory Processor, which supports dynamic memory allocation, limited-field reference counting, and mark-sweep garbage collection in the hardware. The proposed design utilizes a set of bitmaps to maintain heap information and reference counting information. Typically, one-bit in a bitmap represents a fixed size memory (e.g., 16 bytes).

- *W. Srisa-an is with the University of Nebraska-Lincoln, Computer Science and Engineering, Ferguson Hall, Lincoln, NE 68588. E-mail: witty@cse.unl.edu.*
- *C.-T. Dan Lo is with the University of Texas-San Antonio, Department of Computer Science, 6900 N. Loop 1604 West San Antonio, TX 78249. E-mail: danlo@cs.utsa.edu.*
- *J. Morris Chang is with Iowa State University, Electrical and Computer Engineering, 3231 Coover Hall, Ames, IA 50011. E-mail: morris@iastate.edu.*

Manuscript received 8 Feb. 2002; revised 25 Oct. 2002; accepted 16 Dec. 2002. For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number 4-022002.

There are bitmaps for allocation status, size information, reference count information, and marking information. These bitmaps are manipulated by combinational components. We use limited-field reference counting to efficiently reclaim no-longer-used memory. Our study finds that, in Java applications for embedded devices, the majority of reference counts are less than seven. Therefore, we adopt three-bit reference counting in the study to analyze the effectiveness of limited-field reference counting. Our study finds that the majority of objects can be reclaimed through reference counting. Thus, the need to invoke the garbage collector is postponed.

In addition, the design also capitalizes on the locality of bitmaps to reduce the hardware cost. Our study shows that there is a good locality in accessing the bitmaps. Therefore, only a portion of bitmaps is manipulated at one time. Based on this finding, a caching mechanism is adopted so that the hardware cost to construct the combinational components would be very modest.

The remainder of this paper is organized as follows: Section 2 discusses previous work in this area. Section 3 provides an overview of the proposed Active Memory Processor. Section 4 reports our finding on a reference counting study on various Java applications. Section 5 analyzes the performance of the proposed caching mechanism (referred to as Allocation Look-aside Buffer). The last section concludes this paper.

2 Previous Work

One of the main goals in garbage collection research is to make real-time garbage collectors. Some researchers investigate hardware-assisted schemes while others concentrate on garbage collection scheduling and algorithms. In this section, we will concentrate on hardware approaches to improve the performance of automatic heap management.

Heap allocation is required for data structures that may survive the procedures that created them. In such cases, it is very difficult for the compiler or the programmers to determine the exact time to explicitly reclaim these objects. In such situations, the automatic garbage collection is necessary to properly maintain the heap [11]. In the Java programming language, the automatic garbage collection is a language requirement. The Java virtual machine determines when objects are no longer needed and reclaims them. The Java garbage collector runs as a low priority thread and operates during idle time or when the memory has exhausted. Much effort has been spent on improving the performance of garbage collection. In most case, the researchers attempt to make enhancement through the software approaches. There are also several approaches to implement garbage collection function in the hardware.

Over the years, there have been several attempts to incorporate hardware support as a way to reduce runtime overhead, increase throughput of the system, and limit the worst-case latencies. The first use of hardware to support garbage collection was found in Lisp machines. In these machines, special microcode accompanies the implementation of each memory fetch or store operation. The result was the improvement in worst-case latencies. However, the runtime overhead and the throughput were not improved. As a matter of fact, the throughput got worst by 30 percent. Since the target audience for this architecture was very small, the improvement that should have been made to the design (i.e., pipeline, superpipeline, or superscalar) was not made [16]. This project became an economic failure.

While Lisp machines fail economically, its limited success inspired many researchers to develop hardware support for object-oriented programming. Two of such projects are *Smalltalk on a RISC* (*SOAR*) [25] and Mushroom [27]. Both systems are targeted at improving the throughput, but not the worst-case latencies of garbage collection. The underlying rationale for their research is the observation that Smalltalk programs run up to 20 times slower than comparable C programs. Unfortunately, their designs are not suitable for low-level languages such as C and C++. For example, the hardware support in the Mushroom system assumes that all pointers are represented by a combination of two values: the object base address and the offset within that object. Moreover, the object in Mushroom system cannot be larger than 1K bytes and the header space for every word (4 bytes) is 1 byte. This translates to 20 percent overhead for the header alone. Apparently, their garbage collection schemes are very specialized and have limited functionality. These reasons make it difficult to justify for their costs.

One of the approaches is Chang and Gehringer's second-level cache [3]. In their design, a coprocessor used to allocate objects in its cache is introduced. Once objects are created in the cache, reference counting is used to manage those objects. Their simulation shows that the coprocessor can remove up to 70 percent of dead objects in the cache and, thus, great amounts of bus traffic can be reduced [3].

Another interesting hardware-assisted approach was done by Nilsen and Schmidt [18]. Their proposed scheme utilized Baker's real-time copying algorithm [2] and is targeted for real-time systems. Baker's copying collector algorithm is adopted where the entire heap is split into two semispaces, only one region can be active during runtime. Copying also rearranges the allocation order. This can be important in some applications where spatial locality should be preserved. In order to reserve the proper space for live objects, the *Object Space Manager* (*OSM*) is used to perform constant-time lookup of the object size. The size is encoded on every object and only available to the collector. OSM calculates the actual memory needed for all live objects, and the local processor would reserve the needed memory at the bottom of the heap. In doing so, flipping of semispaces can be done instantly and the new to-space is ready to be accessed by the mutator immediately. The copying routine is done in the background using the hardware routine. A pacing threshold that limits the allocation rate is used to balance the allocation and copying throughput. It is worth noting that pacing is an issue unique to this algorithm. Pacing is not a part of tracing overhead (tracing is done in a traditional fashion to locate live objects). It acts as a threshold for allocation rate.

In [26], a hardware self-managing heap memory, *Reference Counting Memory* (*RCM*) is presented. It includes data memory (ordinary RAM) and reference count memory inside an 8 MB memory bank. Each back of memory maintains a set of free lists, which is used to locate free memory chunk during a memory allocation request. In the tested system, there are a maximum of one million objects

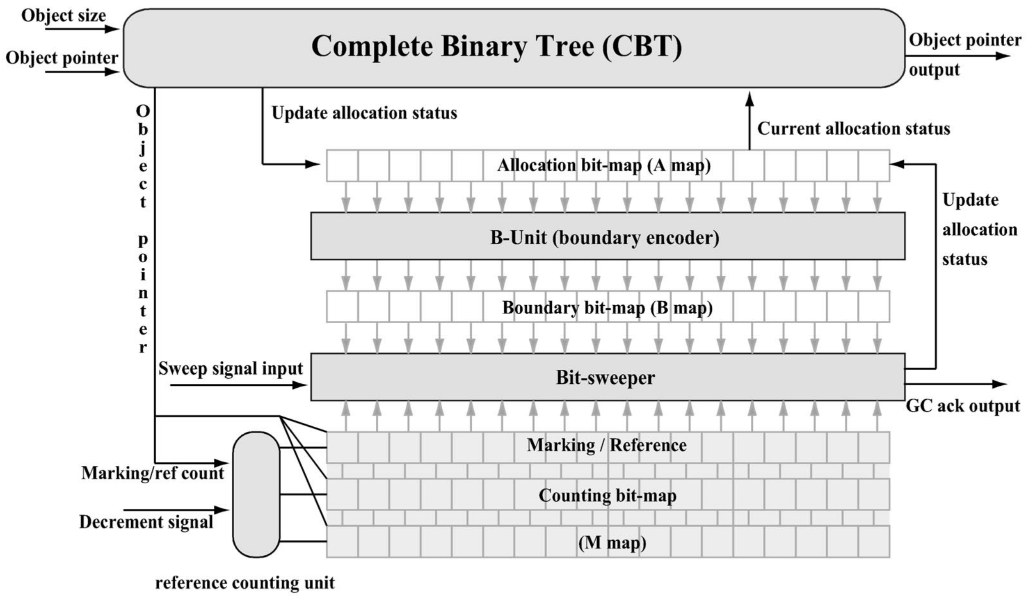


Fig. 1. The top-level design of the *active memory processor*.

per 8-MB of memory. It also supports Deutsch-Schorr-Waite (pointer reversal) mark-phase for constant space operation [20], [12]. The initial testing suggests that it is potentially effective [11]. Reference Counting is performed at no cost to the mutator program. The RCM mark-sweep can collect at twice the speed of a software copying collector. Moreover, the sweeping can be done concurrently with the mutator. After the collection, the free list is reconstructed to reflect the current availability of each memory bank. When the work was published, it was claimed to be the only device that provides hardware support for both reference counting and garbage collection.

Recently, reference counting has made its way into real-time systems. In [19], the author uses reference counting to achieve hard real-time for Java systems. The author did not explore the possibility of using limited-field reference count and, therefore, the overhead for each object is quite high (16 bytes/object). In addition, there are 24 million reference counting operations added on to the processing cost. We believe that a dedicated hardware with modest cost can be more efficient in performing reference counting operation and garbage collection for two reasons. First, the processing of reference count is off loaded to a coprocessor. Therefore, the main processor can operate in parallel the coprocessor such as the proposed *Active Memory Processor* (*AMP*). Second, to perform reference counting and garbage collection using the main CPU, the main processor can consume a large amount of power due to frequent memory accesses and complexity of the main CPU. A coprocessor is a more effective choice. Therefore, the proposed design represents a cost effective and power efficient solution for dynamic memory management in real-time Java embedded devices.

3 Introducing the Active Memory Processor

At the heart of the proposed system, there are three or more bitmaps used to record objects information. If one bit-reference counting is used, then there will be four bitmaps. For each bit added to the reference count field, one bitmap is added to the system. The Fig. 1 illustrates the internal components of the Active Memory Processor. It is worth noting that three-bit reference counting is used in the illustration.

The function of each bitmap is explained as follows:

- The *Allocation bitmap* (*A-map*) represents the allocation status of the entire heap memory.
- The *Boundary Bitmap* (*B-map*) records the size information (record by boundaries) of all objects in the heap.
- The *Marking/Reference Counting bitmaps* (*M-maps*) update reference count field based on the starting address. For example, if an object at address A is created, address A is sent to the reference counting unit, which places a mark on the first bit in the reference counting field (001). If another reference is made to the object, the starting address is then given to the reference counting unit and the corresponding reference count would be (010). To decrement a reference count, the starting address and the decrement signal are sent.

Each bit in a bit-vector represents the smallest block of memory that can be allocated. According to the results of our simulations [22], [4], 16-byte per block is the most suitable. Allocation is performed using a pair of complete binary trees built "on top of" the A map.

- One binary tree—the **or**-gate tree—determines if there is a large enough chunk to satisfy the allocation request (step (a) in Fig. 2).
- The other binary tree—the **and**-gate tree—finds the beginning address of such a chunk (step (b) in Fig. 2).

Each node in the **or**-gate tree is simply a two-input or gate. The leafs of the tree are connected to the bitmaps which represent the memory space. The output of the or gate shows the allocation status of the subtree (either free

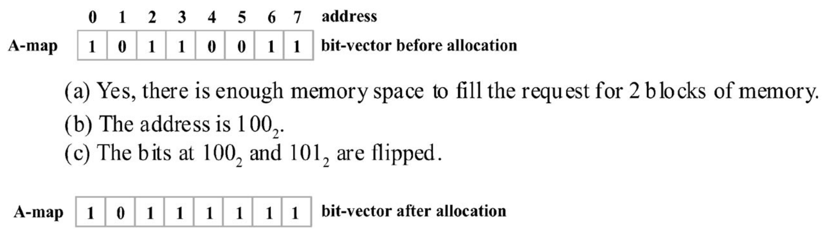

0 1 2 3 4 5 6 7 address
A-map 1 0 1 1 0 0 1 1 bit-vector before allocation
(a) Yes, there is enough memory space to fill the request for 2 blocks of memory.
(b) The address is 100_2.
(c) The bits at 100_2 and 101_2 are flipped.
A-map 1 0 1 1 1 1 1 1 bit-vector after allocation

Fig. 2. A simple example in allocating two blocks memory from an 8-bit bit-vector.

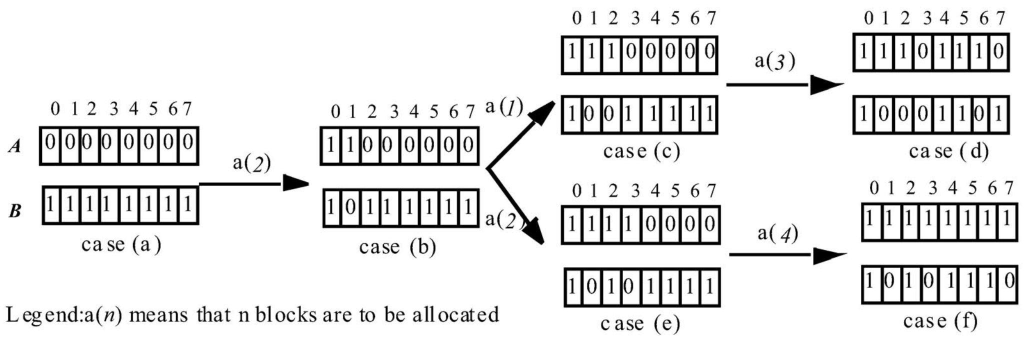


Fig. 3. Example of using the B bit-vector and A bit-vector to allocate from an 8-block region.

[0] or not completely free [1]). For a memory with 2^N blocks, this requires 2^{N-1} **or**-gates. The levels of the **or**-gate tree will be numbered 0 through N, with the root being level 0. To calculate the starting address of the block, the level-l outputs of the **or**-gate tree are fed into a second binary tree, constructed from and gates. This **and**-gate tree propagates availability information upward toward the root, in such a way that the location of the first zero at level l can be found by a nonbacktracking search. Suppose the **and**-gate tree determines the address of the first zero (at level l of the **or**-gate tree) to be n. Then, the beginning address of the free memory block in the bit-vector is $n \cdot 2^{N-l}$. Fig. 2 provides a simple example of allocating two blocks of memory. It is worth noting that the initial value of the A map is "0." When an object is allocated, all the corresponding bits (bits at address 4 and 5 in Fig. 2) are set to "1."

To complete step (c) in Fig. 2, a combinational hardware component called the *bit-flipper* is used to invert certain bits in the bit-vector, from 0 to 1 for allocation. The inputs to the *bit-flipper* are the starting address of the bits that need be flipped, and the number of bits to flip. Unlike the **or**-gate tree and the **and**-gate tree, the *bit-flipper* propagates signals from the root of the tree to the leaves. This is reasonable, since the bit-vector is made up of the leaves of the tree. Building a sophisticated node that can intelligently propagate information has the advantages of limiting the hardware complexity to $O(n)$ and the potential propagation delay to $O(logn)$, where n is the total number of bits in the bit-vector.

The size information is recorded on the B-map using object boundary. It uses logic 0 in the B-vector to record the boundary between an allocated block and a free region, or between two allocated blocks. Initially, the entire B-vector is set to logic high, which represents a contiguous free-memory chunk. During the allocation process, bits are set to 0 to represent boundaries of objects that are allocated. In Fig. 3, an A-vector and a B-vector are used to show the allocation status and size encoding for eight consecutive blocks.

A boundary (represented by a 0 in the B-vector) can be the interface between an allocated memory chunk and a free memory chunk (e.g., bit 1 in step (b)) or between two allocated memory chunks (e.g., bit 1 in step (c)). These boundaries are used as sentinels in the deallocation process. Suppose that after step (d), a deallocation instruction was passed to the parameter 4, which means to free the object beginning at bit 4. This is the object that was allocated between step (c) and step (d). By searching for the first logic 0 (i.e., the boundary) from bit 4 of the B-map, we would find that the object size is three blocks. It is worth noting that the B-unit, a combinational hardware component, is used to perform boundary encoding.

Initially when an object is created, it automatically has one reference. Thus, the corresponding bits in the M-Maps are set to "01." These bits are set using the reference counting unit. If there are more references made to this object, the reference counting unit would increment the reference count by updating the corresponding bits in the M-maps. For example, in a system that supports 2-bit reference counting, when an object is created, the corresponding bit in the first M-map is set to "1" and the one in the second M-map is set to "0." If an additional reference is made, the reference counting unit would update the corresponding bit in the second M-map to "1" and the first M-map to "0." It is worth noting that the *bit-flipper* is used to set the write enable of the corresponding bits in the M-maps. Fig. 4 illustrates reference counting in the proposed system. Once the count reaches the capacity of the reference count field (e.g., three references in a 2-bit count field), the object is referred to as "sticky."

Once an object becomes "sticky," it would remain in this status even if the actual reference count has been

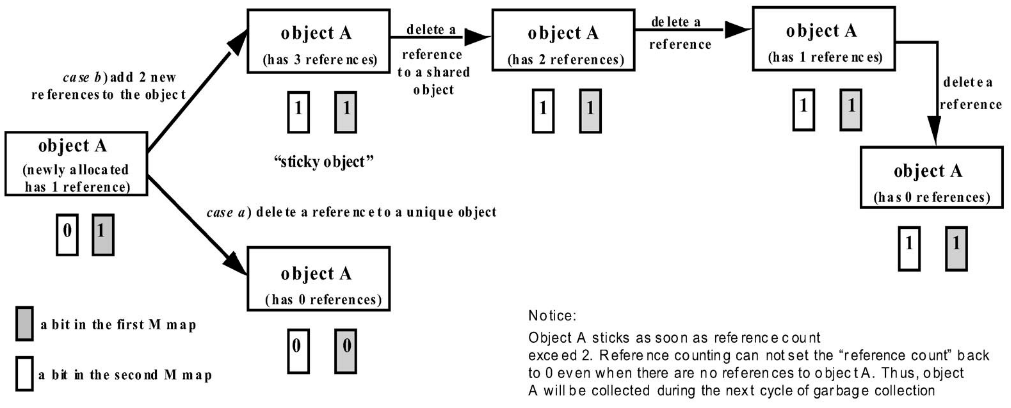

object A (newly allocated has 1 reference)
0 1
case b) add 2 new references to the object
object A (has 3 references)
1 1
"sticky object"
delete a reference to a shared object
object A (has 2 references)
1 1
delete a reference
object A (has 1 references)
1 1
delete a reference
object A (has 0 references)
1 1
case a) delete a reference to a unique object
object A (has 0 references)
0 0
a bit in the first M map
a bit in the second M map
Notice:
Object A sticks as soon as reference count exceed 2. Reference counting can not set the "reference count" back to 0 even when there are no references to object A. Thus, object A will be collected during the next cycle of garbage collection

Fig. 4. Two-bit reference counting.

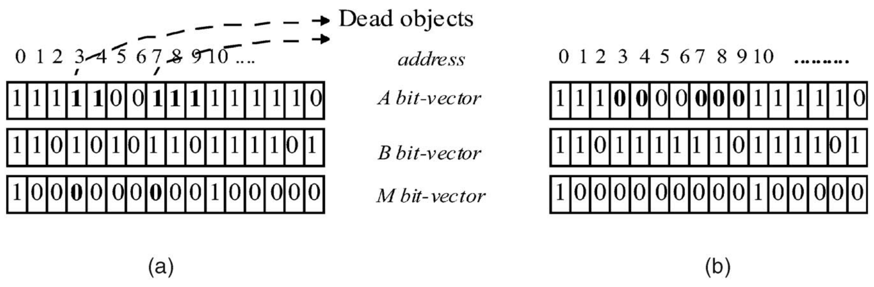


Fig. 5. An example of a garbage collection cycle. (a) Prior to garbage collection and (b) after garbage collection.

decremented to "0." The only way to collect this object is through a full mark-sweep collection. When mark-sweep is invoked, tracing of live objects is done through software. First, all entries in the M-maps are set to "0." For each live object found, a signal is sent to the AMP so that the reference count is updated. Any object with a positive reference count is considered live. Once the marking is done, the sweeping phase would begin. The bit-sweeper is a combinational hardware that can work in two ways. First as a deallocator, where it simply resets the allocation status bits in the A-map starting from the beginning address until the first boundary in the B-map is reached. This approach is used to immediately reclaim memory through reference counting. Second, as a sweeper where it will propagate signals through the entire bitmaps so that all the dead objects are reset. An example of mark-sweep garbage collection in the proposed scheme is given in Fig. 5.

From the M bit-vector, we can see that there are two live objects (at address 0 and 10); however, there are also two dead objects at address 3 and 7. By looking at the M bit-vector, it is apparent that there is no pointer pointing to the objects at address the 3 and 7. These two objects are, in effect, garbage and should be collected. Again, if multiple M-maps are used for reference counting, any object with positive reference count is considered live. Fig. 6 depicts a snapshot of all four bitmaps.

According to Fig. 6, object X has the size of four blocks (as indicated by the first "0" in the B-map). Initially, all bits in the B-map are set to "1." When an object is allocated, the corresponding bit that represents the object's boundary would be set to "0." Therefore, the first "0" in the above B-map indicates the boundary between object X and unallocated space. The first "1" in the M-map_1 (address 10) also indicates that object X is alive. At the same time, since the bit at address 10 in the M-map_2 is also "1," object X is a "sticky" object. On the other hand, object Y and object Z are not "sticky" since the reference count are "1" and "2," respectively. Object W is also a "sticky" object. Let's assume that all the references to it have been deleted. This object is now garbage, but it cannot be reclaimed through reference counting due to its "sticky" status. When the next marking phase is completed, the bits on the two M-maps for object W will be set to "0" to indicate that there is no reference made to this object and it will be collected by the bit-sweeper.

Let us assume that eventually, the heap space will be full and marking is necessary. The first step required before marking would be to reset both M-maps to "0." The first reference to a live object will then be mark on the M-map_1. The subsequent references will be counted until the limit (in this case, 3) is reached. The marking process continues until all objects are accounted for. After the completion of the marking phase, the correct reference behavior of all objects will be restored. It is worth noting that reference counting can be invoked 1) by the compilers or interpreters or 2) in the hardware [26]. For example, in a *Java Virtual Machine* (*JVM*) running in interpreter mode, reference counting can be incorporated by modifying the interpretation of bytecodes that perform object references. A bytecode such as PUTFIELD, which stores a value from the stack of a method into a field inside an object, would require one more step in addition to the common PUTFIELD operation—it needs to check the content of the destination field prior to writing the new reference. If the destination field contains a reference, then the reference count for that object needs to be decremented. Then, the new reference is incremented. If it is to be done in the hardware, a small processor can be used to trap all the reference writes to the heap (write barrier). By

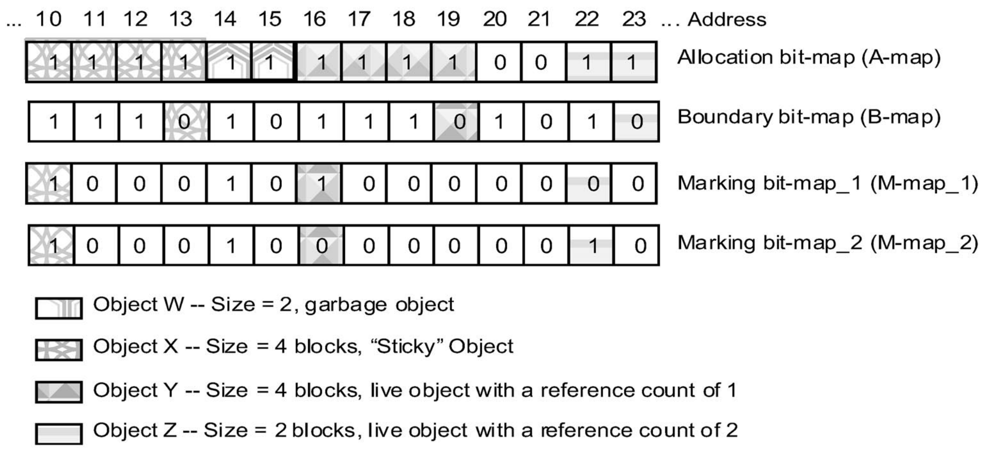

... 10 11 12 13 14 15 16 17 18 19 20 21 22 23 ... Address
1 1 1 1 1 1 1 1 1 1 0 0 1 1 Allocation bit-map (A-map)
1 1 1 0 1 0 1 1 1 0 1 0 1 0 Boundary bit-map (B-map)
1 0 0 0 1 0 1 0 0 0 0 0 0 0 Marking bit-map_1 (M-map_1)
1 0 0 0 1 0 0 0 0 0 0 0 1 0 Marking bit-map_2 (M-map_2)
Object W -- Size = 2, garbage object
Object X -- Size = 4 blocks, "Sticky" Object
Object Y -- Size = 4 blocks, live object with a reference count of 1
Object Z -- Size = 2 blocks, live object with a reference count of 2

Fig. 6. Maintaining object status with 2-bit reference counting.

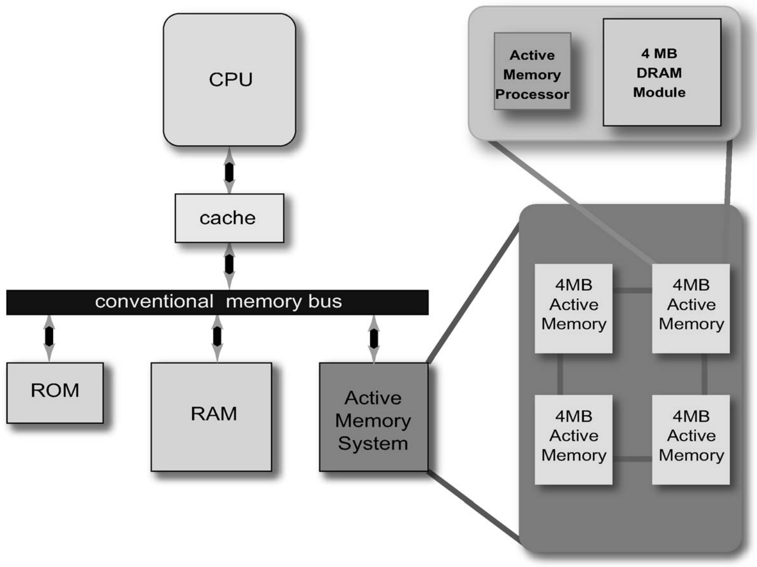


Fig. 7. AMP system integration.

doing so, there is no need to modify the code of the mutator. At the same time, the reference count can be manipulated concurrently with the running application [26].

3.1 System Architecture of the Active Memory Processor

Since garbage collection is only done in the heap, we can separate the traditional DRAM Memory Module from the Active Memory System. Thus, the Active Memory will only be used to allocate dynamic objects through the allocation function (e.g., new operator). It is worth noting that the approach to separate the heap memory from the main memory is similar to Nilson and Schmidt's Garbage Collected Memory Module design [18] and Wise et al.'s RCM design [26]. This means that the operating system must allow allocation and garbage collection requests to be sent directly to the Active Memory Processor. The basic system integration of the proposed system is given in Fig. 7.

Inside each memory module, there are two major components, the *Active Memory Processor* and the DRAM itself. For example, a 4 MB module would consist of 4 MB DRAM and an AMP. The AMP is used to maintain the heap status, to manipulate reference count, and to perform the garbage collection [21]. Since most embedded system works in the physical memory domain and not the virtual memory domain, we do not have to consider the issue of page translation. However, the allocation and garbage collection functions must be designed to provide information such as requested sizes and object references directly to the AMP. It is worth noting that a larger system can be constructed by using multiple Active Memory Modules. For example, a 16 MB system can be constructed out of 4x4-MB modules. As stated earlier, bitmaps (A-map, B-map, and M-map(s)) are used to record all necessary information. These bitmaps reside as part of the Active Memory Processor.

4 A Study of Reference Count in Embedded System Application

Hardware support can provide an efficient way to reference count an object. In the AMP, only one hardware instruction is needed to perform reference counting. Additionally, reference counting in the hardware can also improve the cache locality because the objects themselves are not touched. On the other hand, if mark-sweep is invoked in the proposed scheme, the complexity of tracing live objects is unbounded. Studies have shown that tracing is by far the most time consuming part in a nonincremental approach. To achieve a truly bounded latency for garbage collection, the issue of unbounded tracing time must be overcome. This section studies the effect of using limited-field reference counting to incrementally reclaim dead objects. We organize this section as follows: Section 4.1 provides an overview of tracing and reference counting. Section 4.2 reports the experimental results of the proposed scheme. Section 4.3 presents the reduction of the number of garbage collection invocations. The last section compares the memory footprint of mark-sweep and limited field reference counting.

4.1 Tracing versus Reference Counting

The complexity of tracing largely depends on the number of live objects in the heap. As the number becomes larger, the tracing time would grow longer as well. In a typical tracing scheme (variations of copying or mark-sweep), the number of live objects is not known prior to the invocation of the garbage collector. The marker processes or threads usually suspend other applications and begin an exhaustive scanning of the memory for references (from global, stack, etc.). Once all the live objects are discovered, the sweeping process would begin. In the proposed scheme, the sweeping process is bounded; however, the determinism of the marking phase depends on the tracing algorithm used.

Reference counting, on the contrary, distributes the cost of memory management throughout the computation. Management of active and garbage objects is interleaved with the execution of the user program [11]. Additionally, reference counting also reclaims garbage objects immediately. Thus, fewer page faults may be generated. On the other hand, reference counting can suffer from problems such as high processing cost, additional space to store reference count, and cyclic structures. Moreover, the coalescing cost when garbage objects are returned to the free pool can also be expensive.

Many researchers have proposed using a small number of bits as the reference count field [9], [24], [6] and these bits can be tagged to each pointer [24] in the same way that runtime tags are used for type-checking [23], [11], or integrate to the hardware as part of the AMP. In the proposed design, the marking bitmaps record an object as "live" as soon as it is created. An object that is not "sticky" can be reclaimed as soon as its sole reference is deleted. On the other hand, "sticky" objects are reclaimed through garbage collection. Because a JVM is stack-based, there can be multiple references to an object inside a method (multiple references from within the stack and local variables). In the proposed scheme, reference count is adjusted when there is a reference coming from outside the method that created the object. It is preferable to monitor reference write in the hardware because there is no overhead on the running program. The write barrier is done concurrently in the hardware. However, for this experiment, we monitor reference write by monitoring a number of bytecodes such as PUTFIELD, AASTORE, AALOAD, GETFIELD, ARETURN, etc. If such bytecodes write a reference to an object outside the method, the corresponding value in the M-map would be set to update.

TABLE 1
Description of the Test Programs

Application	Description
Calculator*	a simple calculator program
Dragon*	displays fractal curves, the complexity of which are controlled by user input
Kvideo	MPEG player, available from the JShape project, www.jshape.com
Manyballs*	a highly multi-threaded program, that displays a user-controlled number of balls moving about the screen, each ball given its own thread
Missiles*	a simple game program
Scheduler	simulates the behavior of a personal calendar/organizer
StarCruiser*	a simple game program

*these applications are included with the CLDC version 1.0.2

4.2 Performance Analysis of Limited-Field Reference Counting

This section presents a study on the number of object references. The information is obtained by trapping all the bytecodes that create objects (*NEW, ANEWARRAY, MULTIANEWARRAY*) and bytecodes that allows an object to survive the method that creates it (e.g., PUTSTATIC, PUTFIELD, AASTORE, etc.). The JVM used is Sun's KVM [14]. It is worth noting that KVM is used due to its simplicity and popularity in Java embedded devices. In addition, the garbage collection scheme is a simple mark and sweep with occasional compaction. The applications used in this study are mostly from Sun's *Connected Limited Device Configuration* (*CLDC*) 1.02 and they have been used in previous study of virtual machine for embedded devices [5]. The basic descriptions and characteristics of each program are presented in Table 1. It is worth noting that six out of seven applications are *Graphical User Interface* (*GUI*) based. The exception is Scheduler, which only simulates a GUI application. The allocation behaviors of each application is then given in Table 2.

As reported in Table 3, more than 99 percent of objects have reference counts of 6. Therefore, a three-bit reference count field will be sufficient to maintain reference count information. It is worth noting that in four out of seven programs, at least 30 percent of objects allocated have a reference count of one. Such objects, referred to as unique references, require minimal bookkeeping in the proposed scheme.

Table 3 also indicates that in most applications, about 50 percent of objects have reference counts of 2 or less. The exception is Kvideo where only 42 percent of objects has reference counts of less than or equal to 2. Thus, two-bit reference counting may also be an alternate solution since it can reclaim about half of the objects in most applications. In this section, we want to compare the effectiveness of 2-bit

TABLE 2
Allocation Behavior of Each Application

Application	Number of Allocations	Average Object Size	Maximum Object Size	Total Memory Request
Calculator	333928	49.25 bytes	152 bytes	16.53 MB
Dragon	13560	150.32 bytes	16396 bytes	2.03 MB
Kvideo	12901	158 bytes	16396 bytes	2.05 MB
Manyballs	12129	33.4 bytes	140 bytes	405.22 KB
Missiles	341	32.5 bytes	212 bytes	11.08 KB
Scheduler	87906	33.2 bytes	140 bytes	2.92 MB
StarCruiser	40024	27.13 bytes	140 bytes	1.09 MB

reference counting versus 3-bit. Obviously, the 2-bit scheme requires smaller amounts of hardware. If the two schemes can work efficiently, the frequency of full collection invocations should be much lower. Table 4 describes the distributions of "sticky" garbage for all applications.

It is clear that three-bit reference counting can reclaim nearly 100 percent of all objects. However, the two-bit approach can only reclaim about 56 percent of objects. Table 5 reports the effectiveness of 2-bit reference counting in terms of memory space. The results clearly indicate that the amount of memory to be collected by full-collection can be reduced by 53-93 percent. As stated earlier, reference counting cannot collect any garbage objects that are "sticky." The amount of unclaimed memory is displayed in column [B]. The last column depicted the percentage of the amount of memory that can be collected by reference counting. On average, the 2-bit reference counting can reclaim more than 71 percent of unused memory. It is worth noting that we decide not to report the effectiveness of 3-bit reference counting in a table format. The 3-bit reference counting is so effective that it reclaims over 99 percent of space in all applications.

4.3 Reduction in the Garbage Collection Invocations

In this section, we compare the number of full-collection invocations between mark-sweep approach and the proposed two-bit and three-bit reference counting schemes. As a reminder, full collection is used to collect unreachable sticky objects and cyclic structures in the proposed scheme. A full collection cycle is exactly the same as a traditional mark-sweep cycle. One of the goals for the proposed scheme is to reduce the number of full collection invocations. This section reports our experimental results.

TABLE 3
Accumulated Percentage of Objects by Reference Count

Application	Reference Count					
	1	2	3	4	5	6
Calculator	0.3	64.89	88.75	99.96	99.99	99.99
Dragon	32.01	48.28	57.98	59.12	63.19	99.79
Kvideo	10.44	42.64	65.64	93.62	98.92	99.57
Manyballs	55.56	55.66	55.75	83.54	83.57	99.98
Missiles	48.68	58.65	68.91	92.66	96.18	98.24
Scheduler	2.59	50.29	90.07	95.78	99.50	99.99
StarCruiser	30.01	61.63	93.88	99.97	99.99	99.99

We simulate the conditions where garbage collection would occur in the traditional mark-sweep and the proposed scheme. In all applications, the number of full-collection invocations is greatly reduced when reference counting is applied. This is because reference counting can efficiently keep the heap residency below the GC triggered point. Table 6 summarizes the invocations of full collection in mark-sweep, mark-sweep with 2-bit reference counting, and mark-sweep with 3-bit reference counting.

With 2-bit reference counting, the reduction in garbage collection invocation ranges from 75 percent to 98 percent. This should allow the overall performance of garbage collection to be greatly improved. Remarkably, 3-bit reference counting can completely eliminate the need for full-collection invocations and, therefore, proves to be a very suitable approach for real-time applications.

4.4 Memory Usage Comparisons

The previous section has shown that 3-bit reference counting is very effective in reclaiming unused memory. Efficiently recycled memory can lead to lower memory footprint and power consumption. Table 7 reports the maximum amount of active memory (memory in used) within the 128KB heap space. We can see that 3-bit reference counting reduces the maximum amount of active memory cells by an average of 77 percent. Thus, 3-bit reference counting is a more energy efficient approach than mark-sweep because lower heap footprint. Currently, there are research efforts that attempt to minimize power consumption by turning off memory area that is unused [5]. By using 3-bit reference counting, only a small portion of the memory needs to be turned on through out the execution of a program.

5 Architectural Support for the Active Memory Processor

Since the bitmaps are usually much too large to be held in dedicated hardware, small segments of each bitmap, called bit-vectors, are held in the hardware of the AMP. Portions of the bitmaps that are not in the AMP bit-vectors are instead held in the bitmaps, which is a software structure. This is very similar to how a hardware *Translation Look-aside Buffer* (*TLB*) is used to cache software page-table information. Each AMP entry holds one bit-vector from each of the bitmaps (A bitmap, B bitmap, and M bitmap(s)). Associated with each AMP entry is the largest_available_size of a block that could

TABLE 4
Distribution of Sticky Garbage Objects with Two-Bit and Three-Bit Reference Counting

Application	Referenced objects [A]	Garbage caused by stuck bit (2-bit) [B1]	Uncollected objects(2-bit) [B1/A]	Garbage caused by stuck bit (3-bit) [B2]	Uncollected objects(3-bit) [B2/A]
Calculator	333928	117135	35.07%	2	0%
Dragon	13560	5035	37.13%	2	0%
Kvideo	12901	7405	57.40%	3	0%
Manyballs	12129	5387	44.41%	2	0%
Missiles	341	149	43.70%	6	1.76%
Scheduler	87906	43714	49.73%	3	0%
StarCruiser	40024	15394	38.46%	3	0%
Average			43.70	–	0%

TABLE 5
The Effectiveness of 2-bit Reference Counting in Reclaiming Memory

Application	Accumulated referenced object space (MB) [A]	Unclaimed "sticky" garbage space (MB) [B]	Amount of memory reclaimed by reference counting [1 - B/A]
Calculator	16,448,928	2,492,740	84.85%
Dragon	2,038,396	129,056	93.66%
Kvideo	2,048,152	622,344	69.61%
Manyballs	405,220	188,024	53.60%
Missiles	11,076	4,424	60.06
Scheduler	2,917,036	971,544	66.70
StarCruiser	1,086,004	285,600	73.70%
Average			71.74%

TABLE 6
Reduction in Full Collection Invocations

Application	Mark-sweep invocations	2 bit reference counting		3 bit reference counting	
		Mark-sweep invocations	% reduction	Mark-sweep invocations	% reduction
Calculator	761	164	78.44	0	100
Dragon	76	1	98.68	0	100
Kvideo	165	14	91.51	0	100
Manyballs	4	1	75	0	100
Scheduler	82	23	71.95	0	100
StarCruiser	14	2	85.71	0	100
Average			83.55	–	100

be allocated from the memory for which it records allocation information, and also the beginning of the address to which it pertains. Together, the AMP entries and this additional book-keeping information make up the *Allocation Lookaside Buffer* (*ALB*). It is worth noting that the ALB was first introduced in [4]; however, the performance studies did not include the hit ratio of reference counting and marking operations. Moreover, the applications tested were not a very large set of Java programs. In this paper, the simulator is reconstructed to include reference counting and marking. Sweeping can be done incrementally as a bit-vector is brought into the AMP. Fig. 8 presents the operation of the proposed architectural support for the AMP.

When a memory allocation request is received (step 1), the requested size is compared against the largest_available_size of each bit-vector in a parallel fashion. This operation is similar to the tag comparison in a fully associated cache. However, it is not an equality comparison. There is a hit in the AMP, as long as one of the largest_available_size is greater or equal to the request size. If there were a hit, the corresponding bit-vector is read out (step 2) and sent to the *Complete Binary Tree* (*CBT*). The CBT

TABLE 7
Maximum Active Memory Cells when the Heap Size is 128KB

Application	Maximum heap usage (bytes)	
	Mark-sweep	3-bit reference counting
Calculator	128,000 (100%)	1,156 (0.90%)
Dragon	127,356 (99.5%)	50,144 (39.18%)
Kvideo	127,992(100%)	23,848 (18.63%)
Manyballs	127,996 (100%)	2,852 (2.22%)
Missiles	11,076 (8.65%)	2,612 (2.04%)
Scheduler	128,000 (100%)	2,924 (2.28%)
StarCruiser	127,988 (100%)	3,568 (2.79%)
Average	111,201 (86.87%)	12,443 (9.72%)

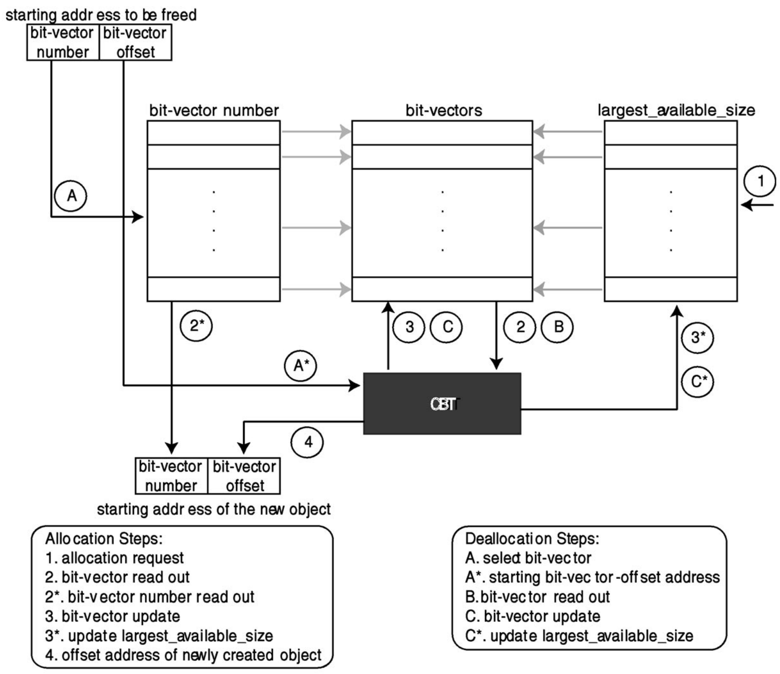
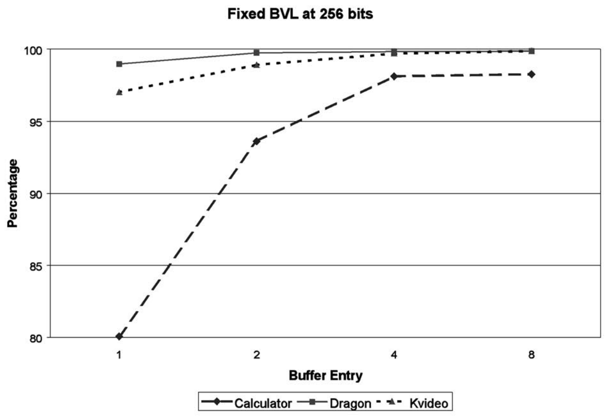


Fig. 8. The allocation and deallocation processes inside the AMP.



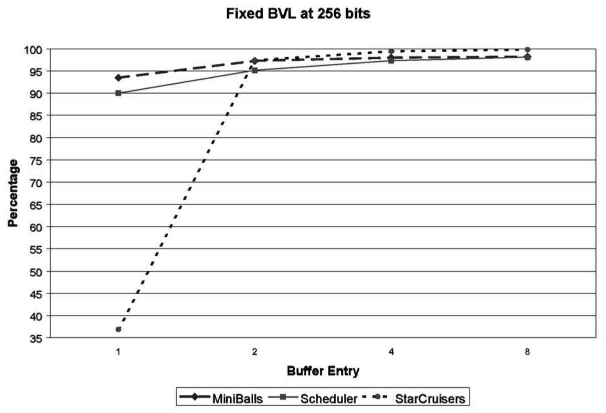


Fig. 9. Buffer size versus hit ratio (BVL = 256).

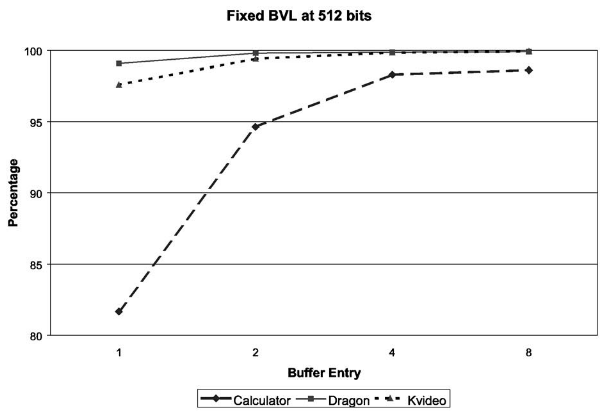
is a hardware unit that perform allocation/deallocation on bit-vector.

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

For the deallocation, when the AMP receives a deallocation request, the bit-vector number of the object pointer is used to select a bit-vector (step A). This process is similar to the tag comparison in cache operation. At the same time, the bit-vector offset is sent to CBT as the starting address to be freed (step A*). The corresponding bit-vector is read out (step B) and sent to CBT. The CBT will free the designated number of blocks (based on the information from B bit-vector) starting at the address provided by the step A* and update the bit-vector (step C) and the largest available size field (step C*). The page number, bit-vectors, and the largest_available_size are placed in a buffer, called *Allocation Look-aside Buffer* (*ALB*).

There are two important parameters that determine the performance and the cost of the AMP. First, the *Bit-Vector Length* (*BVL*), which is defined as the length of the bit-vector that is directly managed by the AMP. Second, is the ALB size which is the number of entries containing in the ALB. This section studies the various configurations of these two parameters so that the proposed system would yield high-performance at a reasonable cost. It is worth noting that parameters including the block size (block_size), which is the number of bytes represented by each bit, is selected based on the investigation reported in [22]. The study found that 16 bytes/block yields a good balance of low fragmentation and smaller bitmap size.

The performance of the ALB (hit ratio) is also determined by the locality of accesses by different memory operations which include allocation, making, increment reference, and decrement reference. We perform the ALB performance



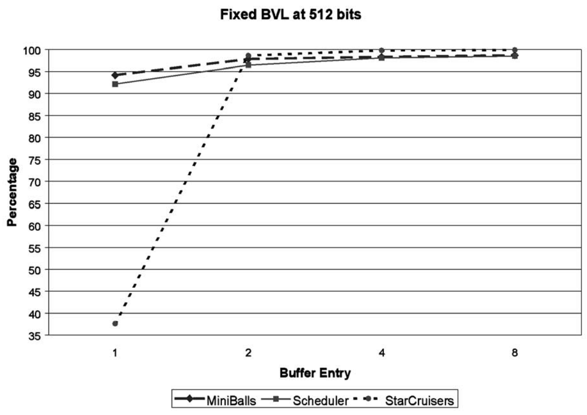
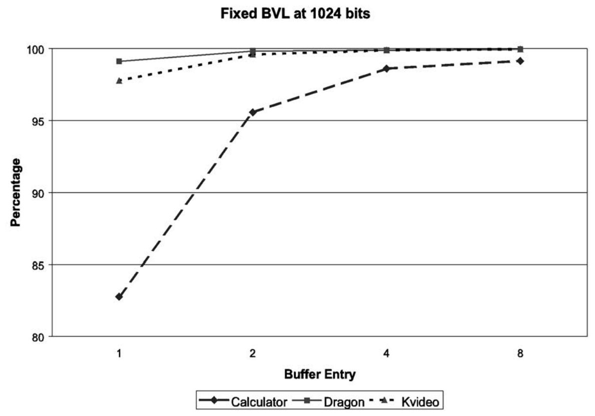


Fig. 10, Buffer size versus hit ratio (BVL = 512).



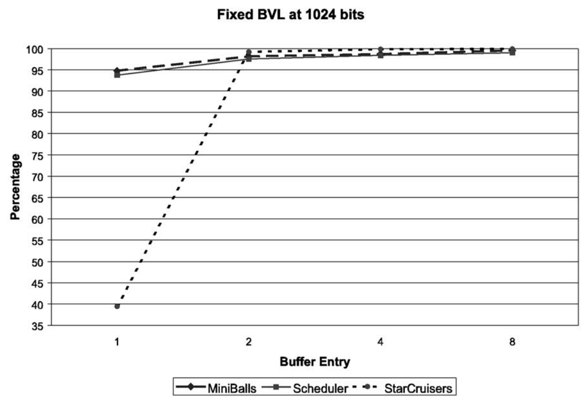
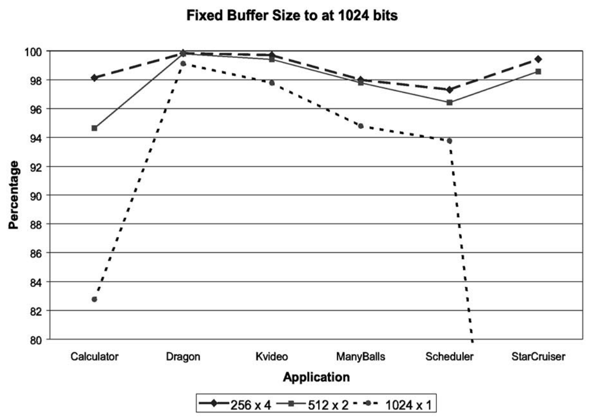


Fig. 11. Buffer size versus hit ratio (BVL = 1,024).



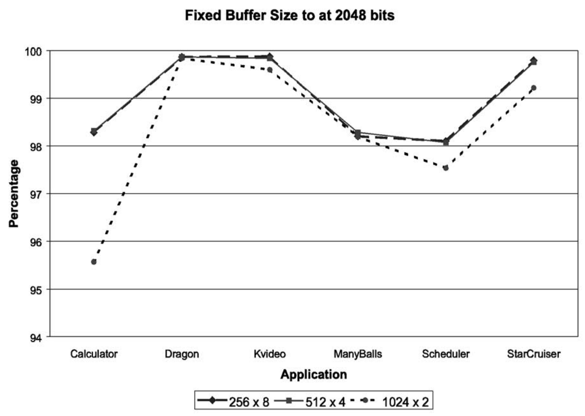


Fig. 12. Fixed buffer size at 1 K bits and 2 K bits.

evaluation through trace driven approach. The trace files are generated by running Java programs on a modified KVM. The information obtained are allocation, marking, reference update (increment and decrement), and freeing. The simulator is written in C++ and takes command line arguments that include the ALB size and the bit-vector-length. The result is presented in the next section.

5.1 ALB Performance Evaluation

We investigate the performance of the ALB through two approaches. First, we fix the size of the *Bit-Vector Length* (*BVL*) and increase the number of entries (this also increases the buffer size). In doing so, we can find a good saturation point where the hit ratio of all or most of the programs begin to stabilize. It is worth noting that we set the system to have the heap size of 128 KB. The results are illustrated in Figs. 9, 10, and 11.

In most applications, the good saturation point with the fixed BVL approach is at four entries. It is also worth noticing that with the BVL of 256 bits, the hit ratio of all applications are more than 95 percent. This translates to the buffer size of 512 bytes (4 bitmaps $\times$ 256 bits $\times$ 4 entries / 8 bits per byte). We also find that reasonable performance can be obtained with 256 bits at two entries. It is worth noting that we do not

TABLE 8
Propagation Delays for the Proposed System

Components	Longest Path (logic gates)	Complexity (node)	Total delay (# of gates)
CBT	2	8	16
B-unit	3	1	3
Marker	2	9	18
Sweeper	3	256	768
Reference counter	9	1	9

include missiles in the simulation. Missiles only allocate small amounts of objects and, therefore, infrequently invoke the garbage collector.

In the second approach, we set the buffer size to the value provided by the first approach (in this case 256 bits × 8 entries and 256 bits × 4 entries). Then, we would investigate the effect of buffer configuration (number of entries × BVL) on the hit ratio. For the buffer size of 2K blocks, we can have the following configuration, 4 × 512 bits, 8 × 256 bits, and 2 × 1,024 bits. For the buffer size of 1K blocks, we can have the following configuration, 2 × 512 bits, 1 × 256 bits, and 1 × 1,024 bits. As indicated in Fig. 12, the hit ratio decreases as the number of buffer entries decreases. This indicates that the areas with frequent accesses are scattered over the entire bitmaps and locality may be temporal. Therefore, more buffer entries would outperform longer buffer line.

5.2 Estimating Hardware Cost and Propagation Delay

By utilizing the Active Memory Processor to manage a heap space, the additional hardware cost to construct this processor is one of the factors that can determine the feasibility of this design. Our earlier study also indicates that the suitable block size for most applications is 16-byte/block [22]. Thus, for a 2-bit reference counting configuration that includes 256 bit BVL and 8 ALB entries, the amount of memory needed to construct the ALB is:

$$mem_{ALB} = \frac{3 \cdot BVL \cdot ALB_{entry}}{8\left(\frac{bit}{byte}\right)} = \frac{4 \cdot 256 \cdot 8}{8} = 1KB. \quad (1)$$

This translates to 1KB of SRAM. For a 128 KB module, an additional DRAM storage of 4KB is also needed for the entire bitmap.

For the propagation delay, we will provide the number of gate delays instead of estimated time in seconds. This is because the time delay is determined by the semiconductor used to build the AMP. Table 8 describes the propagation delay for each unit of the component. The BVL is set as 256 bits. The allocator requires two stages. First, the allocation of a memory block is done through the Complete Binary Tree (CBT, please refer to the previous work section for more information). In each node of the CBT, the propagation delay is two gates. If the BVL is 256 bits, there would be eight levels of the binary tree.

The hardware cost to construct the combinational components is illustrated in Table 9. For a 3-bit reference counting configuration and a BVL of 256 bits with eight entries, we will need 8,000 gates and 1.3K bytes of SRAM. The hit ratio of the configuration would be about 98 percent on average. It is worth noting that in all applications, the hit ratio is above 95 percent. It is also worth noting that the current memory buses can provide very high peak bandwidths (GB/sec). For example, Intel Pentium III has the bus bandwidth of 800 MB/sec, Sun UltraSPARC II offers 1.9 GB/sec, and Compaq Alpha 21256 has 2.6GB/sec bandwidth [17]. If the BVL is 256 bits, the miss penalty would be less than 100 seconds in a 0.8GB/sec bandwidth.

6 Conclusions

The Active Memory Processor provides a cost effective solution for real-time Java embedded devices. Presently, real-time embedded system developers avoid using Java because of its nondeterminism. Since other aspects of Java (e.g., small memory footprint, portability, security, and quick product deployment) are already suitable for embedded systems, the use of the proposed Active Memory Processor would allow dynamic memory management to be performed in a deterministic and bounded fashion.

Our study also finds that by adopting three-bit reference counting, we can eliminate the need for garbage collection invocation. In addition, three-bit reference counting can also reduce the amount of memory usage by as much as 77 percent. Therefore, the proposed design is also power efficient due to a smaller number of garbage collection and smaller memory usage.

Acknowledgments

Stephen Hines developed the simulations of the Active Memory Processor. Paul Griffin generated traces of Java embedded systems. The authors would like to thank the referees for the valuable comments, which greatly im-

TABLE 9
Number of Gates Needed to Construct the AMP

Component	Number of Gates per node	Complexity (N = 256 bits)	Number of Gates (N = 256 bits)
Complete Binary Tree	7	N - 1	1,785
B unit	13	N - 1	3,315
Bit-sweeper	6	N	1,536
Miscellaneous	–		1,300
Total			7,936

proved the presentation of this work. This work has been partially supported by the US National Science Foundation under grants CCR-0219870, CCR-0296131, and CCR-0113335.

REFERENCES

[1] R.W. Atherton, "Moving Java to the Factory," *IEEE Spectrum,* pp. 18-23, 1998.

[2] H. Baker, "List Processing in Real Time on a Serial Computer," *Comm. ACM,* vol. 21, pp. 280-294, 1978.

[3] J.M. Chang and E.F. Gehringer, "Evaluation of an Object-Caching Coprocessor Design for Object-Oriented Systems," *Proc. IEEE Int'l Conf. Computer Design,* pp. 132-139, 1993.

[4] J.M. Chang, W. Srisa-an, and C.D. Lo, "Architectural Support for Dynamic Memory Management," *Proc. IEEE Int'l Conf. Computer Design,* pp. 99-104, 2000.

[5] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and M. Wolczko, "Tuning Garbage Collection in an Embedded Java Environment," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture (HPCA'02),* 2002.

[6] T. Chikayama and Y. Kimura, "Multiple Reference Management in Flat GHC," *Proc. Fourth Int'l Conf. Logic Programming,* pp. 296-293, 1987.

[7] K. Edwards, *Core Jini.* Prentice Hall, 1998.

[8] Esmertec, Jbed, http://www.esmertec.com, 2001.

[9] D.P. Friedman and D.S. Wise, "The One-Bit Reference Count," *BIT,* vol. 17, pp. 351-359, 1977.

[10] Newmonics Inc., Perc available from Newmonics Inc., http://www.newmonics.com, 2003?

[11] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1998.

[12] D.E. Knuth, *The Art of Computer Programming III: Sorting and Searching.* Reading, Mass.: Addison-Wesley, 1973.

[13] Sun Microsystems, http://java.sun.com/products/consumer-embedded, 2003.

[14] Sun Microsystems, Java CLDC and K Virtual Machine, http://java.sun.com/products/cldc/, 2003.

[15] Sun Microsystems, Java Embedded Server available from Sun Microsystems, http://www.sun.com/software/embeddedserver, 2003.

[16] D.A. Moon, "Architecture of the Symbolics 3600," *Proc. 12th Ann. Int'l Symp. Computer Architecture,* pp. 76-83, 1985.

[17] S.S. Mukherjee and M.D. Hill, "Making Network Interfaces Less Peripheral," *Computer,* Oct. 1998.

[18] K. Nilsen and W. Schmidt, "A High-Performance Hardware-Assisted Real-Time Garbage Collection System," *J. Programming Languages,* pp. 1-40, 1994.

[19] I.T. Ritzu, "Hard Real-Time Reference Counting without External Fragmentation," *Proc. Java Optimization Strategies for Embedded Systems Workshop,* 2001.

[20] H. Schorr and W.M. Waite, "An Efficient Machine-Independent Procedure for Garbage Collections in Various List Structures," *ACM Comm.,* vol. 10, pp. 501-506, 1967.

[21] W. Srisa-an, C.D. Lo, and J.M. Chang, "Active Memory: Garbage-Collected Memory for Embedded Systems," *Proc. Second Ann. Workshop on Hardware Support for Objects and Microarchitectures for Java,* pp. 11-15, 2000.

[22] W. Srisa-an, C.D. Lo, and J.M. Chang, "A Performance Analysis of the Active Memory Module (AMM)," *Proc. IEEE Int'l Conf. Computer Design,* pp. 493-496, 2001.

[23] P. Steenkiste and J. Hennessy, "Tags and Type Checking in Lisp: Hardware and Software Approaches," *Proc. Second Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS II),* Mar. 1987.

[24] W. Stoye, T. Clarke, and A. Norman, "Some Practical Methods for Rapid Combinator Reduction," *Proc. Symp. LISP and Functional Languages,* Aug. 1984.

[25] D. Ungar, "The Design and Evaluation of a High Performance Smalltalk System," *ACM Distinguished Dissertations,* 1987.

[26] D.S. Wise, B. Heck, C. Hess, W. Hunt, and E. Ost, "Research Demonstration of a Hardware Reference-Counting Heap," *Lisp Symbolic Computing,* vol. 10, pp. 159-181, 1987.

[27] M. Wolczko and I. Williams, "The Influence of the Object-Oriented Language Model on a Supporting Architecture," *Proc. 26th Hawaii Int'l Conf. System Sciences,* 1993.

Witawas Srisa-an (M'98) received the BS degree in science and technology in context, and the MS degree in computer science from the Illinois Institute of Technology (IIT). In 1999, he received the Dean's Scholarship to pursue his PhD degree and joined the Computer Systems Laboratory at IIT under the advisory of Dr. Morris Chang. He received the PhD degree in computer science from the Illinois Institute of Technology in 2002. He is currently an assistant professor in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln. He has taught courses on such subjects as operating system kernels, computer architecture, and performance analysis of object-oriented systems. His research interests include computer architecture, object-oriented programming, dynamic memory management and garbage collection, dynamic instrumentation, hardware support for garbage collection, Java, and C++ programming languages. He is a member of the IEEE and the IEEE Computer Society.

Chia-Tien Dan Lo (M'98) received the the BS degree in applied mathematics from the National Chung-Hsing University, Taiwan, the MS degree in electrical engineering from the National Taiwan University, Taiwan, and the PhD degree in computer science from the Illinois Institute of Technology (IIT), Chicago, in 1990, 1992, and 2001. In 2002, he joined the Department of Computer Science at the University of Texas at San Antonio, San Antonio, Texas, as an assistant professor. From 1992 to 1994, he served as a second lieutenant for the military and joined the Data Automation Project Group for the Army. After 1994, he spent two years as a research assistant at the Institute of Information Science, Academia Sinica, Taipei, Taiwan. His work there was implemented to a verifier that can be used in real-time concurrent systems. In 1995, he founded a computer company specializing in embedded system design, small business networking services, and database applications. At the beginning of 1998, he received the Dean's scholarship and joined the computer system group at IIT. He started his teaching career in 1999 and has taught courses that include the Unix systems programming and the computer network. His research interests include VLSI design, operating systems, neural networks, concurrent algorithms, computer communication networks, artificial intelligence, computer architecture, and computing theory. His current research thrusts are dynamic memory management in Java, C and C++, hardware support for java virtual machine, reconfigurable computing, low-power embedded systems, process migration, and multithreaded systems. He is a member of the IEEE and the IEEE Computer Society.

Ji-en Morris Chang (M'85) received the BS degree in electrical engineering from Tatung Institute of Technology, Taiwan, the MS degree in electrical engineering, and the PhD degree in computer engineering from North Carolina State University in 1983, 1986, and 1993, respectively. In 2001, Dr. Chang joined the Department of Electrical and Computer Engineering at Iowa State University where he is currently an associate professor. His industrial experience includes positions at Texas Instruments, Microelectronics Center of North Carolina, and AT&T Bell Laboratories. He was on the faculty of the Department of Electrical Engineering at the Rochester Institute of Technology, and the Department of Computer Science at the Illinois Institute of Technology (IIT). In 1999, he received the IIT University Excellence in Teaching Award. Dr. Chang's research interests include: wireless networks, object-oriented systems, computer architecture, and VLSI design and testing. Dr. Chang has been serving on the technical committee of the IEEE International ASIC Conference since 1994. He also served as the secretary and treasurer in 1995 and Vendor Liaison Chair in 1996 for the International ASIC Conference. He was the conference chair for the 17th International Conference on Advanced Science and Technology (ICAST 2001), Chicago, Illinois. He is a member of the IEEE and the IEEE Computer Society.