An Energy Efficient Garbage Collector for Java Embedded Devices [*]

Paul Griffin
Electrical and Computer Enginerring
Iowa State University
Ames, IA 50011
pgriffin@iastate.edu

Witawas Srisa-an
Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588
witty@cse.unl.edu

J. Morris Chang
Electrical and Computer Enginerring
Iowa State University
Ames, IA 50011
morris@iastate.edu

Abstract

This paper presents a detailed design and implementation of a power-efficient garbage collector for Java embedded systems. The proposed scheme is a hybrid between the standard mark-sweep-compact collector available in Sun's KVM and a limited-field reference counter. There are three benefits resulting from the proposed scheme. (a) the proposed scheme reclaims memory more efficiently and this results in less mark-sweep garbage collection invocations, (b) reduction in garbage collection invocations improves cache locality and reduces the number of main memory accesses, and (c) reduction in memory access ultimately results in lower energy consumption, since a memory access can consume a large amount of energy when compared with an instruction execution. The proposed scheme has been implemented into Sun's KVM, and has been shown to reduce the number of mark-sweep garbage collection invocations by up to 100% in some cases, and the number of level-1 cache misses by as much as 87% when compared to the default garbage collector. We also find that in some applications, the proposed scheme can reduce the power consumption by as much as 27% when compared to the default Sun's KVM.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors–memory management (garbage collection); C.3 [*Special-purpose and application-based systems*]: Real-time and embedded systems

General Terms Algorithms, Design, Performance, Experimentation, Languages

Keywords Garbage collection, embedded systems, Java mobile computing, virtual machines

* Partial support for this work was provided by the National Science Foundation through grants CCR-0098235, CCR-0296131, CCR-0219870 and CNS-0411043.

1. Introduction

Java has seen a rise in popularity within the embedded computing arena, due to its many attractive features that address the requirements of an embedded device. Among other features that reduce development time, such as sandbox-type security measures and distributed computing (Remote Method Invocation, or RMI,) Java offers a small memory footprint, and automatic *Garbage Collection* (*GC*). The small memory footprint is obviously crucial for memory-constrained devices, whereas the automatic GC dramatically reduces development time, since the software developer is freed from the time-consuming task managing dynamic memory [4]. As a result of these features, the number of Java-enabled mobile/embedded devices have already exceeded 750 million units with over 240 million users. [1, 22].

To provide support for Java computing in these mobile devices, Sun Microsystems provides the Micro Edition of the Java platform [21]. This edition includes one configuration defined as the Connected Limited Device Configuration (CLDC). It provides a foundation for the Java Runtime Environment (JRE) that targets small, resource-constrained devices, such as mobile phones, etc., and is suitable for devices with 16/32-bit microprocessors/controllers and as little as 160KB of available memory [21]. At the heart of the CLDC is Sun's Kilobyte Virtual Machine (KVM), a virtual machine designed from the ground up with the constraints of inexpensive mobile devices in mind [21]. Our preliminary studies find that the KVM fails to meet many of the requirements in embedded systems. A prime example of this is the KVM's use of simple mark-sweep-compact GC. This incurs unpredictable pause-times in execution, which precludes real-time operation, and allows unused objects to accumulate as far as possible in memory, which requires additional memory space, and frequent garbage collection calls. This in turn undermines the goal of minimal memory and energy usage.

The goal of this paper is to compare our hybrid garbage collection scheme with the mark-sweep-compact scheme used in the Sun's KVM. Our hybrid implementation combines limited-field reference counting with the KVMs default mark-sweep-compact collector. This scheme yields three important benefits.

1. It reclaims memory efficiently which result in less garbage collection invocations.
2. It also requires less memory accesses since less garbage collection invocations improve cache locality.
3. It results in lower energy consumption because memory access consumes a large amount of power.

It is a well-known fact that mark-sweep GC can pollute the cache during the marking phase [15], which suggests that reducing full-heap GC collections can improve cache performance. Reclaiming memory incrementally can further improve cache performance by reusing the same memory repeatedly, rather than continuously expanding heap usage. To compare the performance, we have compiled the default and proposed systems for SimpleScalar-ARM [5]. We then analyze a suite of benchmark programs representative of current embedded applications, measuring several metrics that include reference behaviors of Java objects, reference count, cache behaviors, instruction count, and garbage collection invocations.

Our results show that limited-field reference counting can be a very suitable reclamation scheme for embedded applications. Our study finds that cyclic structures, which degrade reference counting performance, are uncommon among most applications; therefore reference counting should work well in such an environment. We also find that typical reference-count values are low enough to allow highly compact reference-count storage, avoiding memory overhead without significant memory leakage. We compare the performance of our proposed scheme with the default collector and we find that the proposed scheme, with only a very primitive allocator/deallocator, is much more efficient and can reduce the number of garbage collection invocations by up to 100%, with average and median reductions of 97.5% and 97.9%, respectively. In addition, for a highly memory-constrained environment, the number of level-1 cache-misses is reduced by up to 90%, 48% on average, with the potential for further improvement in a more optimized implementation. As a result, the proposed scheme can reduce the energy consumption by as much as 27% in some applications.

The remainder of this paper is organized as follows. Section 2 discusses previous work in this area. Section 3 describes the design and implementation of the proposed reference counting. Section 4 provides a brief overview of the KVM and the basic behavior of the benchmark applications. Section 5 reports the experimental results. Section 6 compares the energy consumption of the proposed scheme and the default scheme. The last two sections discusses future work in this area and concludes the paper.

2. Related Work and Background

2.1 Related work

There are several on-going research efforts that concentrate on reducing the power consumption in Java embedded system. For example, Tomar et al. [23] introduced the use of local memory to facilitate efficient execution of Java program. In their work, on-chip memory is used to directly allocate frequently used objects. Their result indicates that memory energy consumption can be reduced by 12.7% when cache (4KB) and local memory (4KB) are used in conjunction. Chen et al. [8] also proposed that the performance of garbage collection could be tuned by selectively turning off unused memory banks. They also compare the energy consumption between mark-sweep and mark-sweep-compact. Their result indicates that savings of up to 43% can be gained through mode control and energy-aware active bank allocation. These, however, are hardware-based solutions, making them inherently inflexible, and each incurs its own overhead: The management of the small local space, in the case of Tomar et al. [23], and the time required to activate and deactivate banks of memory, in the case of Chen et al. [8].

Unlike the work mentioned in the previous paragraph where author ”tweak” with the default collector, Diwan et al. [11] compares the energy consumption between four different memory management strategies: No deallocation, explicit deallocation, conservative mark-sweep, and conservative incremental mark-sweep. He finds that incremental mark-sweep can consume as many as 40 times more energy than explicit deallocation. They also find that incremental approach can consume up to four times more energy than the non-incremental mark-sweep approach.

Work by Velasco et al. [25] discussed the differences in energy consumption between multiple garbage collectors (mark-sweep, copying, generational) currently supported by Jikes RVM [2]. The study found that generational garbage collection may yield the least energy usage among the different algorithms. The study was based on SPECjvm98 benchmark programs which are not representative of applications in today’s Java embedded devices. In addition, RVM is a compiled only VM which is quite different than virtual machines for embedded devices such as the KVM which is an interpretation only virtual machine.

2.2 Background

Reference Counting (*RC*) is one of the two main approaches in automatic dynamic memory management, the other approach being tracing, as in mark-sweep collection. In reference counting, each object has a reference count field, initialized to zero when the object is first allocated. Each time a pointer to that object is copied, the reference count field is incremented, and each time a pointer to that object is removed, the reference count is decremented. When the reference count is decremented to zero, the object is freed and reclaimed. This approach suffers from the inability to reclaim cyclic structures, or any structure reachable from a cyclic structure. A cyclic structure is any structure in which one can begin at an object A, and by recursively tracing along pointers, arrive back at A. Fortunately, studies have found that the frequency of such structures is generally quite low [7], but any such structure represents a piece of memory that will not be reclaimed during the program’s execution. Because of this limitation, and others that will be introduced, reference counting, if employed at all, is generally accompanied by a back-up tracing collector [15] or a complex algorithm to break up and detect the cyclic structures, typically employing local tracing as opposed to a full-heap trace [6].

Empirical studies of Lisp and other functional languages have shown that a majority of objects have only a small number of references and a very short lifetime, which is an optimal scenario for reference counting [15, 20]. This observation has led to the notion of using very limited fields for reference counting (e.g. one-bit, two bits, etc.). Hence, a considerable amount of research effort was spent on studying the effectiveness of limited field reference counting in functional programming languages, in the late 1970s, throughout the 1980s, and during the early 1990s. In Wise and Friedman [12], the authors suggested restricting the reference count field to a single bit [12, 20] which has subsequently been called multiple-reference bit (MRB) [9, 14]. This approach, while resulting in one additional bit for each reference (if no bits in the reference are known to be unused), can reduce the number of accesses to the object since the reference count on any object is known as soon as it becomes accessible [18]. Thus, the cache performance is improved. Moreover, the object can be recycled without even touching it [18]. In [20], the authors reported that by using one-bit reference counting, the garbage collection performance improvement can range from 35% to 70%. However, the authors did not report reference behaviors such as the percentage of multiple reference objects. In [18], Roth and Wise also reported that one-bit reference counting can reduce the number of collections by 33% to 85%. Again, no information about reference behaviors was reported. This work studies the power-efficiency of limited-field reference counting on a power-constrained platform, and investigates the relevant object behaviors and supporting system modules that reference counting calls for.

Our goal is to use reference counting in conjunction with the KVMs default mark-sweep-compact to lower the amount of energy consumed by reducing the number of garbage collection invocations and thusly the number of memory accesses. The proposed hybrid scheme has been implemented as part of Suns KVM. We have also collected a suite of benchmark programs that are representative of current embedded applications. Our work can complement existing work in several ways. First, efficient memory management can reduce the average heap footprint. This would allow work in the area of [CSK 02] to be even more effective. In addition, our work also extend the study performed by [11] where reference counting was not discussed. Due to space limitation, we will not go into detail of memory management in KVM. Such information is available from [8, 23].

3. Our Garbage Collector Implementation

Our garbage collector is **a hybrid between limited field reference counting (3-bit) and the default mark-sweep-compact used in the KVM**. The adopted reference counting algorithm is a deferred technique introduced by Deutsch and Bobrow [10]. This approach is a systematic run-time method of deferring reference count adjustment ignoring reference manipulations to the local variables and stack-allocated compiler temporaries [15]. With this approach reference counts only reflect the number of references from other heap objects. Studies of languages such as Lisp and ML found that this approach can eliminate the reference count manipulation operations by as much as 99% [15]. Since stack references are not accounted for, one major challenge in this approach is to determine whether an object with a reference count of zero is still reachable from the stack and local variables. To overcome this challenge, any objects with the reference count of zero are temporarily stored into the zero-count table (ZCT). When the ZCT is full or memory cannot be allocated, the ZCT is reconciled so that any references on the ZCT that are not found by scanning the stack are collected [10, 15]. In our implementation of this algorithm, we have made two additional modifications to improve the algorithm efficiency—pointer-tagging and frame-tagging.

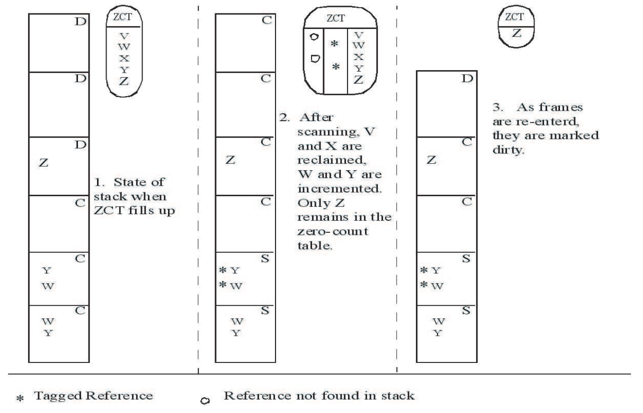
ZCT
V W X Y Z
1. State of stack when ZCT fills up
2. After scanning, V and X are reclaimed, W and Y are incremented. Only Z remains in the zero-count table.
3. As frames are re-enterd, they are marked dirty.
* Tagged Reference
Reference not found in stack

Figure 1. Illustration of the Proposed Algorithm

Frame-tagging simply refers to the use of a 2-bit field in a method frame to denote one of three possible states for a frame: dirty, clean, and saved. As frames are pushed onto the stack (newer frames being higher than old ones,) their states are initialized to dirty. During ZCT-updating, the dirty and clean frames are scanned for heap references. All dirty frames scanned are then tagged as clean, and clean frames that are scanned are tagged as saved. When a group of frames are tagged as saved, all heap objects for which references exist in those frames are incremented. This allows the system to ignore frames that have already been repeatedly scanned while resolving the ZCT in the future. Frames are reset to dirty when the mutator re-enters them after popping higher frames off of the stack. When a saved frame is re-entered, its references must be decremented, to keep reference counts correct. Because of this, only one-half of clean frames that are scanned during a ZCT-resolve routine are saved, so as to minimize repeated saving and re-entering of the same frame.

Every frame higher than the highest saved frame is scanned, and every scanned frames tag is set to clean, and only frames that were previously clean can be saved, and thusly no dirty frame can be higher than a clean frame, nor can a clean frame be higher than a saved frame. This optimization may potentially result in much more frequent increments/decrements than is typically seen in Deutsche-Bobrow reference counting, which would severely degrade the efficiency benefits. This problem is addressed with pointer-tagging.

Pointer-tagging simply refers to the storing of a small number of bits (in this case 1 bit) along with a pointer. These could be overtop of unused bits in a pointer, such as the least significant bits, so as to exploit data alignment, or the most significant bits, so as to exploit an address space that is much greater than the size of the heap and libraries combined. In this algorithm, this tag associated with a pointer tells the virtual machine whether or not an identical reference at a lower spot in the same execution stack has been incremented. This removes the need for wasteful incrementing/decrementing as the ZCT is updated. By default, all pointers are un-tagged (the tag bit is 0.) Instead of simply incrementing all objects referenced by each frame as frames are saved, the scanning begins at the lowest clean frame, and progresses up the stack. For each reference encountered, if that reference was previously encountered and incremented in the same thread stack and during the same ZCT-resolve routine, then that reference is tagged, and otherwise it is incremented. Accordingly, as frames are popped off the stack later, if a saved frame is re-entered and marked dirty, then only objects referenced by the untagged pointers in that frame are decremented. This combination of pointer-tagging and frame-tagging modifications prevents redundant stack-scanning, and redundant increments/decrements. Figure 1 depicts the workings of this algorithm, at three points in time: immediately before the ZCT is updated, immediately after the stack-space is scanned/saved, and shortly after the ZCT-update completes.

Although stack-based references can be ignored during program execution, they must be accounted for before any objects can actually be reclaimed, as is the case for any GC algorithm. Locating references in the execution stack requires a complex and potentially expensive process known as stack-mapping. This process analyzes the instructions of a Java method, and ascertains where references lie in the target stack-frame. Compared to mark-sweep, reference counting performs object reclamation incrementally, and thus much more frequently, and so stack-mapping also must be performed much more frequently, making it a performance issue for reference counting. In a direct implementation, each scanning of a single frame would require a stack-mapping operation, but this is potentially inefficient. Depending on how often the ZCT is updated, some frames in the execution stack space may be scanned for references repeatedly without the resultant stack-map changing. These repeated stack-mapping operations are redundant, and our implementation eliminates these by storing the generated stack-map of a given method-frame inside the frame itself, for later reuse. The im-

Application	Description	Objects - Part of cycles		Objects - Referenced by cycles	
		% of total objects	% of total space	% of total objects	% of total space
MVideo	A mobile MPEG viewer	0	0	0	0
Xview	A mobile, zoom-enabled large-image viewer	0	0	0	0
SFMap	An interactive map of San Francisco	0.316%	0.589%	2.46%	3.84%
Mexel	A mobile spreadsheet program	0.400%	0.406%	2.50%	2.33%
CellHTML	A mobile HTML viewer,	0.00154%	0.000986%	0.00461%	0.00211%
SmarTicket	On-line ticket purchasing program	0.0958%	0.0351%	0.334%	0.0842%

Table 1. Description of Benchmark Programs

pact of this optimization is analyzed in Section 5. Optimization of stack scanning is also addressed in [6], however, their algorithm is geared toward concurrent garbage collection, and entails periodic increments and decrements of objects referenced by the execution stack, which increases code-size and can potentially degrade performance.

Aside from implementing the RC algorithm itself, some supporting modules must also be modified, to capitalize on the capacity of RC to prevent garbage accumulation. The most crucial of these modules is the memory allocator/deallocator. In the default, mark-sweep-compact algorithm, allocations are performed repeatedly and consecutively, interrupted only by a full-heap garbage collection, which frees all unused memory in contiguous blocks. This calls for only a very simple allocator, and the default implementation uses a simple first-fit algorithm. RC results in intermixed allocations and deallocations, which introduces additional complexity fragmentation of space. To reduce this fragmentation, we have extended the allocator to maintain two sets of free-lists referred to as the *wilderness* list and the *FLcache* list to promote more efficient reuse of objects and defer coalescing. In the wilderness list, free objects are segregated by size, with the range of possible sizes exponentially increasing. The FLcache maintains a cache of frequently used sizes, and for each size an accompanying free-list of objects. Thus, FLcache would be the first place that the allocator would look for free a object.

Also critical in highly dynamic memory-management is coalescing, in which adjacent free chunks of memory are combined, removing fragmentation. This is achieved implicitly during the sweep phase of mark-and-sweep, but must be performed explicitly with incremental deallocation. If memory cannot be allocated after the ZCT is updated, coalescing is performed to consolidate memory. Our coalescing function performs two scans of all free objects, clears the lists in the FLcache, and dumps the coalesced memory chunks into the wilderness free-lists.

Application	Percentage of Objects with Reference Watermark of:						
	1	2	3	4	5	6	7+
MVideo	88.8	10.9	0.18	0.07	0.01	0.09	0.01
Xview	60.6	39.1	0.28	0	0	0	0
SFMap	4.14	94.9	0.67	0.10	0.12	0.04	0.06
Mexel	53.6	43.6	1.96	0.63	0.16	0.02	0.02
CellHTML	55.2	37.2	4.41	3.01	0.08	0.06	0.06
SmarTicket	23.4	55.2	10.4	3.78	1.08	5.68	0

Table 2. Reference Behavior in the Benchmark Programs

4. Experimental Environment

4.1 Computing Platform

The proposed scheme has been implemented in the KVM version 1.0.3, compiled into the MIDP version 1.0.3 platform. The experiment was conducted on a simulated ARM platform, with cache configurations taken from low-power, java-targeted ARM processor cores, primarily the ARM1026EJ-S. The I-cache and D-cache are fully symmetric and 4-way associative in all tested configurations. The number of sets and the line-size have been varied, as the cache of the ARM1026 core is highly configurable [3]. The three configurations presented are: 64 sets x 32 bytes/line (8 KBytes), 128 sets x 16 bytes/line (8 KBytes), and 128 sets x 32 bytes/line (16 KBytes). The MIDP-KVM was compiled to the ARM architecture, and run on SimpleScalarARM (sim-cache build) [5]. SimpleScalar has been modified to support realistic, controlled user input for interactive benchmarks. The heap size selected is 120% of the minimal heap size, to allow the default KVM room to accumulate garbage in between GC invocations.

Benchmark	Reduction (%)
Mvideo	99.97
Xview	100
SFMap	96.08
Mexel	97.79
CellHTML	98.00
SmarTicket	93.06

Table 3. Reduction in the Number of GC Invocations

Since the default KVM implementation uses pure mark-sweep-compact garbage collection, it does not need write barrier. Reference counting on the other hand, requires write-barrier to perform reference increment or decrement whenever references to heap objects are written or removed. There are 33 instances of this write-barrier required in the MIDP-KVM, and 20 explicit deallocations required to reclaim temporary, internally-allocated buffers. The total code-size dedicated to reference counting is approximately 1000 lines, and the size of the SimpleScalar executable build is increased by 0.495%.

4.2 Benchmark Characteristic

In order for RC to operate efficiently, the target platform must satisfy two conditions. The first condition is that the reference count field stored on each object must not introduce significant memory overhead. A 16 or 32-bit integer field may seem to be a natural choice, but this much space on each object may be excessive, if the upper bits are virtually always unused. It is unlikely that the reference-count of a single object would reach a few dozen, let alone 256, which would justify more than a single byte. So, to keep memory overhead to a minimum, the reference-count field should be exactly as large as is needed. This minimum field-size must be determined from the target applications.

The second condition for efficient RC is that the amount of memory that RC cannot reclaim must be very low. Unreclaimable objects can be categorized into three groups: Objects with saturated reference-counts, for which the reference-count becomes as large as the count field can hold, cyclic objects, which can reach themselves through pointer-tracing, and cyclic-referenced objects,

Application	Default KVM		Hybrid KVM	
	Instruction Count (Billion)	Data Access (Billion)	Instruction Count (Billion)	Data Access (Billion)
Mvideo	95.64	51.30	91.47	49.32
Xview	14.00	7.00	14.04	7.02
SFMap	4.34	1.95	3.47	1.83
Mexel	4.29	2.38	4.24	2.47
CellHTML	4.24	2.67	4.30	2.72
SmarTicket	3.28	2.19	3.45	2.26

Table 4. Comparing Instruction Count and Data Accesses

which are reachable from cyclic objects. Cyclic objects are then a subset of cyclic-referenced objects, which represent the total amount of data leaked due to cyclic references.

We have collected a suite of six Java applications for embedded devices, representing commercial grade interactive programs with GUI and network-intensive functionality. These applications are then executed on a modified KVM, combined with Suns Mobile Information Device Profile (MIDP) [21], that records basic information such as reference counts and the number of cyclic structures. The basic description of each application and its cyclic data characteristics is given in Table 1. As shown in that table, the amount of cyclic data is quite manageable, reaching a maximum of 3.84% and an average of 1.04%. The breakdown of the maximum reference-counts (reference watermarks) that the benchmarks objects reached is given in Table 2. For all applications tested, the reference watermark is 2 or less for nearly all objects, and 6 or less for over 99%. Since values of 7 or more are very rare, a reference-count field of 3 bits can be considered sufficient. Each class-instance also requires a fourth bit, for internal GC-related record-keeping, to allow for object finalization. A field this small can be embedded in the object header, coopting unused bits in the object size and type fields. This eliminates per-object memory overhead, since the object header itself is required regardless of the GC scheme.

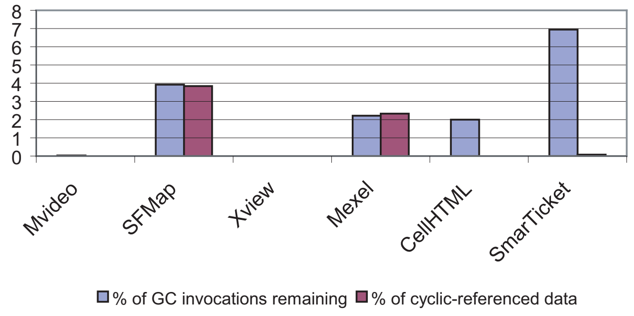

Figure 2. Comparison between cyclic data and remaining GC requirement

5. Experimental Results

5.1 Algorithm Effectiveness

The stack-map-saving optimization of the Deutsche-Bobrow algorithm described in Section 3 reduces stack-map routines by an average of 36.5%, and 46.4% if Xview and SmarTicket are excluded, due to their very low number of ZCT-resolution operations.

We measure the reduction in garbage collection of our proposed scheme as a percentage of the garbage collection invocation in the default mark-sweep scheme, and the results are given in Table 2. As seen in Figure 2, the fraction of GC invocations remaining

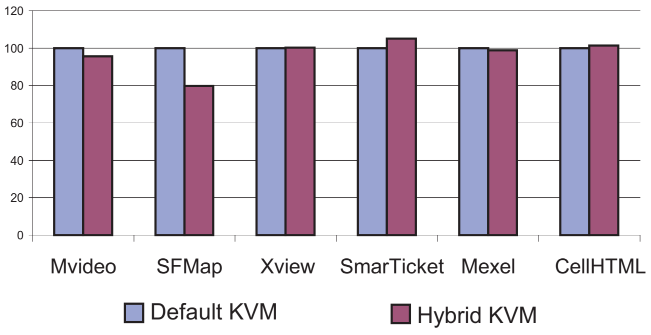

Figure 3. Comparison of the instruction count between the default and the proposed schemes

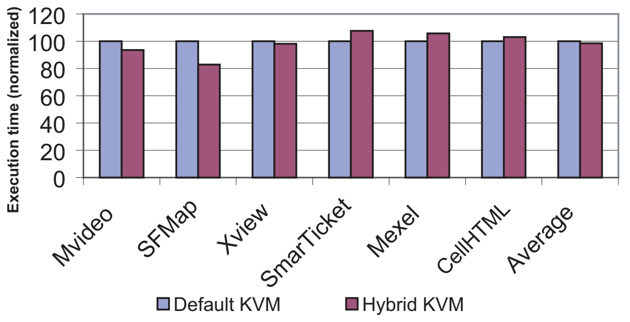

Figure 4. Comparison of the execution time between the default and the proposed schemes

closely parallels the fraction of cyclic data, in four of the six benchmarks tested. This implies that the GC requirement is now directly driven by the amount of cyclic data created, which suggests that the allocator in place is sufficient to avoid fragmentation. The two benchmarks with a GC requirement greater than their cyclic data frequency, CellHTML and SmarTicket, have relatively low memory usage, and with reference counting, only require 3 and 5 GC invocations respectively. As a result, their GC requirement could be inflated due to fragmentation resulting from cyclic objects, even without a matching amount of cyclic space.

The cache of frequently-used sizes exhibited an average miss-rate of 28.4%, 10.11% if Mvideo and Xview are discounted. Xview allocates very few objects, only 2148, and so its miss-rate is very likely inflated due to the start-up cost of populating the FLcache. MVideo has much larger average object sizes than the other benchmarks (excluding Xview), and this results in a very small, heavily used set of large memory chunks. This results in an understocked freelist in the corresponding FLcache entry, which in turn results in an inflated FLcache miss-rate. Since this cache is four-way as-

Application	Cache Configuration	Default KVM I-Cache Miss (Million)	Default KVM D-Cache Miss (Million)	Default KVM Writeback (Million)	Hybrid KVM I-Cache Miss (Million)	Hybrid KVM D-Cache Miss (Million)	Hybrid KVM Writeback (Million)
Mvideo	64x32	74.69	**516.57**	261.22	78.93	**318.65**	229.58
	128x16	122.85	**896.07**	488.13	127.04	**572.47**	442.13
	128x32	54.30	**438.56**	238.73	53.39	**276.38**	212.72
Xview	64x32	2.13	23.05	3.09	2.91	21.65	2.58
	128x16	2.25	17.84	3.23	2.32	14.52	2.68
	128x32	0.069	6.09	1.17	0.068	6.33	1.06
SFMap	64x32	9.19	**148.59**	6.19	18.03	**14.26**	3.12
	128x16	11.30	**130.53**	6.76	22.36	**14.30**	3.77
	128x32	1.66	**95.00**	4.30	4.59	**7.13**	1.85
Mexel	64x32	46.23	72.23	11.33	52.90	61.25	12.28
	128x16	61.08	71.18	13.56	71.63	59.02	14.76
	128x32	12.09	35.77	7.66	16.72	30.64	7.93
CellHTML	64x32	35.12	9.55	1.51	18.56	9.21	2.04
	128x16	21.35	8.20	1.99	4.92	7.44	2.50
	128x32	0.60	3.98	1.04	0.69	3.88	1.35
SmarTicket	64x32	49.32	7.29	1.42	18.57	8.54	1.86
	128x16	36.29	7.57	2.02	5.92	7.91	2.34
	128x32	1.31	3.98	1.07	1.33	4.27	1.26

Table 5. Comparing Cache Misses

sociative, it is unlikely that miss-rates are elevated due to simple associativity conflict.

Very few instructions and cache-misses occur during coalescing, less than 1% of the total instructions and cache-misses for all benchmarks. However, the number of coalesce operations required is surprisingly high, averaging at roughly a third of the number of GC invocations required by the default KVM. This suggests that object reuse is below optimal, causing frequent splitting of large objects to create smaller ones, This can cause data-cache misses outside of the coalesce operation itself. Object reuse can be improved primarily through a more sophisticated freelist-coalescing strategy.

5.2 Performance Comparison

Performance measurements for the SimpleScalarARM platform, including instructions executed and cache misses, have been made for all benchmarks, in both versions of the MIDP-KVM. Table 4 depicts the cache-independent instruction-count and data-load count. Table 5 gives the level-1 cache misses and write-backs for all cache configurations tested.

Notice that these applications are highly memory intensive. Dividing the number of data accesses by the instruction count, we find that the percentage of load/store instructions ranges from 43% to 61% in the default KVM and 45% to 60% in the hybrid KVM. Studies [13, 16] have found that the energy consumption in memory can be up to 200 times higher than the consumption resulting from cache hits [13]. Thus, reduction in cache misses has the potential to greatly improve power efficiency in embedded systems.

Figure 3 depicts the normalized instruction count of the proposed scheme and the default scheme used in the KVM. It is worth noticing that in three of the six applications, the proposed scheme reduces the number of instructions by as much as 20.1%. In the other three applications, the instruction counts of the proposed scheme increase by 0.3, 1.5, and 5.1% (Xview, CellHTML, and SmarTicket, respectively). On average, instructions decrease by 3.13%, with an average increase of 2.83% if we exclude SFMap, which displays extreme results.

Prior to performing the SimpleScalar-based simulation, both versions of the MIDP-KVM were compiled for Linux (Pentium 4 1.8 GHz, Red Hat Linux, version 7.3), and the time required to run all benchmarks was measured. The hybrid reference-counting scheme was found to actually reduce the total execution time, by an average factor of 1.5%, as shown in Figure 4. As seen in Figures 2 and 3, execution time correlates almost exactly with instructions executed, except that the normalized execution time is consistently lower than the normalized instruction count. This can most likely be attributed to the sometimes dramatic reduction of level-1 cache-misses, as seen in Figure 5.

Figure 5 demonstrates the improvements in cache-behavior yielded by reference counting. All benchmarks except for Mexel demonstrate a substantial reduction in cache misses for the 8-KByte cache configurations. Mexel fails to benefit due to promiscuous duplication of references, as shown in Table 6. Compared with the other benchmarks, Mexel performs a much larger number of reference increments/decrements. This hurts the DL1-cache savings, through line-replacements as well as write-backs, as objects in memory must be repeatedly accessed and altered. It also results in more frequent updating of the ZCT content, which increases IL1 cache-misses.

Benchmark	RC Operations	Object Allocated	RC Operations per Object
Mvideo	319,970	930,849	0.344
Xview	3,384	2,014	1.68
SFMap	982,698	980,980	2.01
Mexel	1,741,698	489,214	1.78
CellHTML	386,044	210,514	1.83
SmarTicket	214,942	89,621	2.40

Table 6. Reference Count Operations Required

Contrary to what is expected from a reference counting implementation, and to what we observe in other benchmarks, DL1 cache misses actually slightly increased for SmarTicket on all cache configurations, rather than decreasing. This appears to be due to SmarTickets unique memory characteristics. Although it has far fewer reference increments/decrements than Mexel or SFMap, its ratio of increments/decrements to objects allocated is significantly higher, as seen in Table 6. To add to this, SmarTicket has the highest minimum memory requirement, and allocates very little space. In short, SmarTicket has too large a memory footprint, compared to the memory that it allocates, to see the object-reuse benefits of RC.

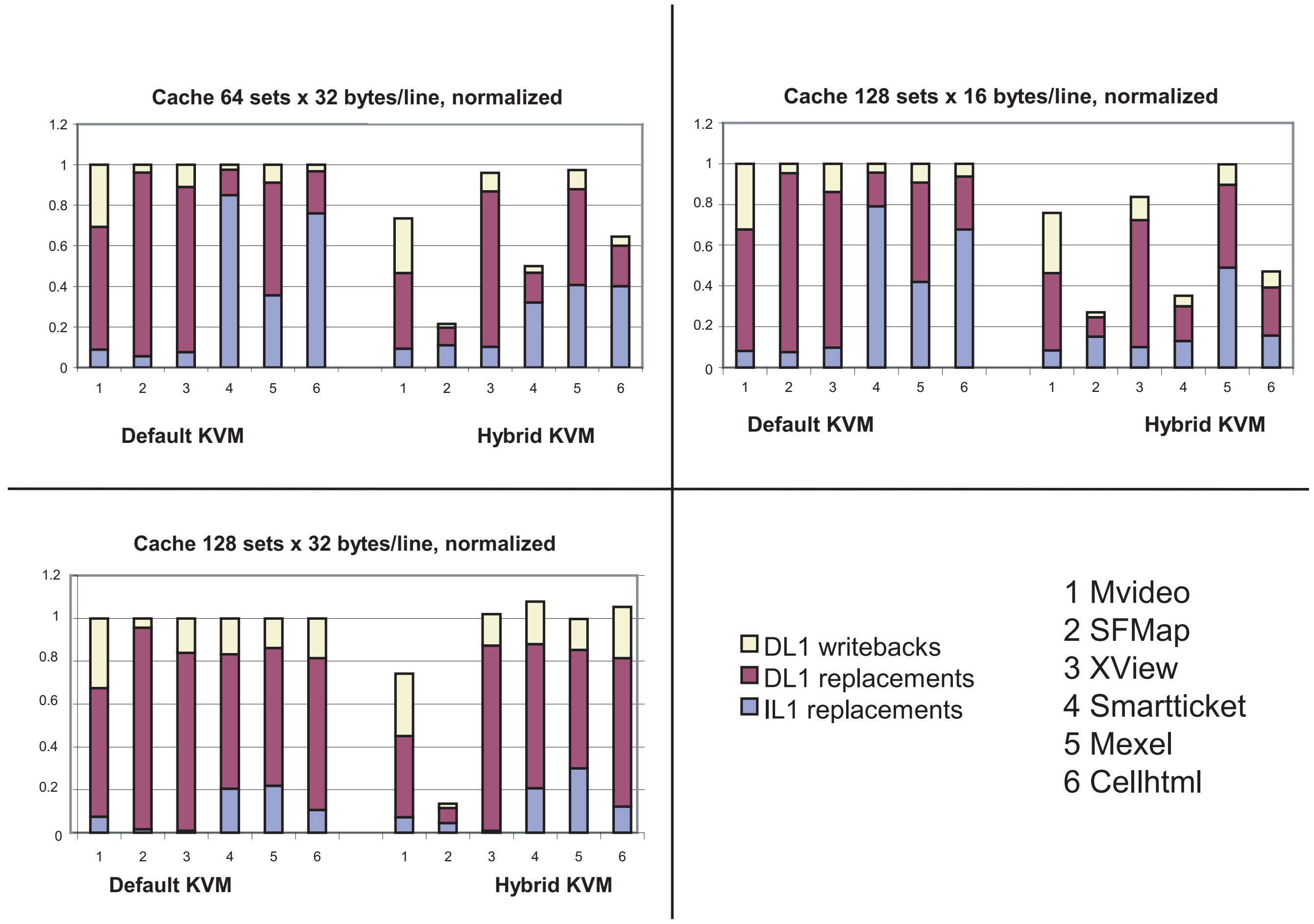

Figure 5. Comparison in Level-1 Cache Misses and Writebacks

For four of the six benchmarks, IL1 cache-misses increase for the hybrid KVM, due to the more frequent garbage collection interruptions. For SmarTicket and CellHTML, however, IL1 cache-misses are dramatically reduced for both 8k cache configurations. Follow-up experimentation revealed this to be a direct result of the structure of the allocator module. Although, as mentioned in Section 3, reference counting requires an allocator that is more complex overall, the common case (the FLcache) is actually much simpler, since no list-searching or splitting of free chunks is required. These two deviant benchmarks have low miss-rates in the FLcache, and also update the ZCT infrequently. With a previous allocator very similar to the default allocator, IL1 cache misses in the reference counting implementation are slightly greater than those of the default KVM. For the 16k cache-configuration, the IL1 misses of these two benchmarks in the default KVM drop below those of the hybrid KVM, presumably because reduced contention removes cache-misses caused by the default KVMs allocator.

6. Study of Energy Consumption

For this analysis, we classify three major sources of energy expenditure that are related to the proposed modification. They are the amount of energy spent on executing the instruction by the CPU core, accessing caches (data and instruction), and accessing memory (during misses). The amount of energy consumed is greatly dependent on the architecture and therefore, we will provide equations that can be used to estimate the amount of energy consumed by these three functions. Here are the equations:

$$\begin{aligned} Energy_{exe} &= IC_{non\_mem} \cdot 1 + IC_{mem} \cdot 2.37 \quad (1)\\ Energy_{cache} &= 0.447 \cdot (IC + DA) \quad (2)\\ Energy_{memory} &= 15 \cdot (miss_{instr} + miss_{data})\\ &\quad +28 \cdot writeback \quad (3)\end{aligned}$$

Notice that *IC* and *DA* represent instruction count and data access, respectively. In (equation 1), we use 1 nanoJoule (nJ) as the average energy consumption for non-memory access instructions and 2.37 nJ for memory access instructions (e.g. load and store) [17]. In (equation 2), we use 0.447 nJ as the amount of energy consumed during one cache access [13]. For (equation 3), the two constant coefficients, 15 nJ and 28 nJ represent the average amount of energy consumed for memory access and write-back, respectively [24]. Since SimpleScalar reports the number of data cache hit and miss separately, *DA* is equal to $cache_{hitData} + cache_{missData}$. The SimplePower simulator [26] provides a more standard analysis tool, but the current release only assumes “Perfect Cache” and so is unsuitable for our research. Table 7 gives the energy consumption of main memory accesses for all cache configurations, and Table 8

Application	Cache configuration	Default KVM Memory Access Energy (J)	Hybrid KVM Memory Access Energy (j)	Saving Percentage
Mvideo	64x32	16.18	12.39	23.43
	128x16	28.95	22.87	21.00
	128x32	14.08	10.90	22.55
Xview	64x32	0.46	0.44	5.04
	128x16	0.39	0.33	16.35
	128x32	0.13	0.13	-0.52
SFMap	64x32	2.54	0.57	77.49
	128x16	2.32	0.66	71.70
	128x32	1.57	0.23	85.50
Mexel	64x32	2.09	2.06	1.81
	128x16	2.36	2.37	-0.40
	128x32	0.93	0.93	-0.03
CellHTML	64x32	0.71	0.47	33.49
	128x16	0.50	0.27	48.83
	128x32	0.10	0.11	-8.72
SmarTicket	64x32	0.89	0.46	48.38
	128x16	0.71	0.27	61.79
	128x32	0.11	0.12	-9.05

Table 7. Comparison of Memory Access Energy Consumption

depicts the cache-configuration-independent energy consumption of instruction execution and L1 cache hits.

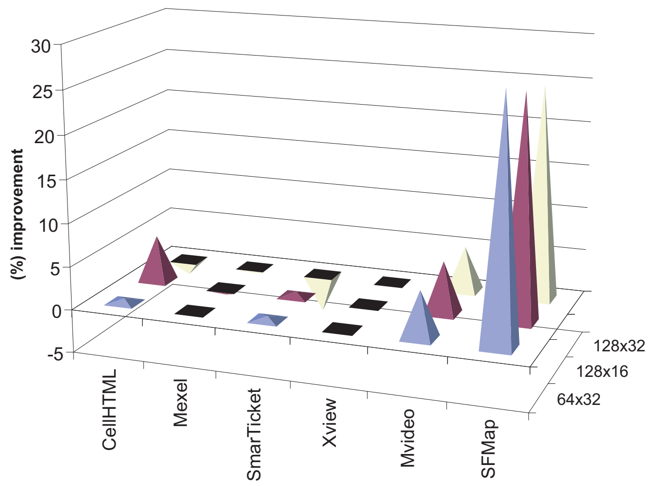

Figure 6. Energy Reduction with the Proposed Scheme

We are able to conclusively reduce the energy consumption in two out of six applications. For the remaining four applications, there are almost no differences between the default and the proposed algorithms as some applications show slight reduction and some show slight increase. Figure 6 depicts the amount of saving and overhead resulting from the proposed scheme. For the 16K cache-configuration, we find that the amount of energy can degrade by as much as 4.5% in one application (SmarTicket). This stems nearly entirely from the elimination of IL1 savings in SmarTicket and CellHTML, as described at the end of Section 5. For Mvideo and SFMap, we achieve conclusive energy reduction across all cache configurations of 6% and 27%, respectively. It is worth noticing that the substantial improvement on data cache misses with the proposed scheme is the major reason for the reduction of energy consumption for SFMap and Mvideo as clearly shown in Table 5.

Application	Default KVM Execution (Joule)	Default KVM Data Access (Joule)	Hybrid KVM Execution (Joule)	Hybrid KVM Data Access (Joule)
Mvideo	165.79	85.41	158.91	78.32
XView	23.56	9.71	23.63	9.71
SFMap	7.01	4.96	5.97	2.85
Mexel	7.54	4.78	7.61	4.79
Cellhtml	7.88	3.52	8.01	3.41
Smarticket	6.27	3.01	6.54	2.83

Table 8. Comparison of Energy Consumption

7. Future Directions

It is worth noting that the ARM1026EJS, among other ARM cores, is equipped with the Jazelle feature, which enables it to natively execute most Java bytecodes, including the most frequently used [19, 3]. These natively supported bytecodes do not include those which must be modified for most garbage collection schemes. This could introduce dramatic changes in the cache behavior, particularly if the Java bytecodes are consequently relocated to the instruction cache, although garbage collection and other complex aspects of Java execution would be unaffected.

Although the allocator provided is sufficient to prevent detrimental data fragmentation, it is possibly not as efficient as can be hoped for. More intelligent management of free-lists could potentially improve the reuse of memory, thereby reducing the required number of coalesce operations, and thereby improve the instruction count, and more significantly, data-cache performance.

8. Conclusions

Our goal of this work is to use reference counting in conjunction with the KVMs default mark-sweep-compact to lower the amount of energy consumed by reducing the number of garbage collection invocations and improving the cache locality during memory allocation. It is a well known fact that mark-sweep garbage collection can pollute the cache during the marking phase [15]. Therefore, reducing the number of GC invocations can improve the cache performance. Since memory access consumes a large amount of energy, reducing cache misses in the same program can result in less energy consumption.

Given the results observed, it can be concluded that for embedded applications, particularly data-intensive multimedia programs, limited-field reference counting offers a viable software-based means of satisfying stringent environmental constraints, without significantly degrading the overall execution time. An efficient reference counting implementation requires an efficient and dynamic segregated free-list manager in order to facilitate the GC reduction that reference counting is capable of. For such a real-world implementation, cyclic objects account for only a very small fraction of the total memory allocated, and as such can most likely be disregarded and deferred to a backup tracing collector for non-hard-real-time systems. Also, for a realistic implementation, a 3-bit saturating reference-count field can be expected to suffice for over 99% of all objects allocated.

As demonstrated in the paper, the there are three affected areas related to energy consumption with the proposed modifications: instruction fetch, instruction execution, and data access. We provide three basic equations to calculate the amount of energy consumed by each area. Our study based on the information reported by [13, 17, 24] indicates that the proposed scheme can potentially reduce energy consumption in some applications by as much as 27% over the default KVM, which employs mark-sweep-compact.

This is mainly due to the improvement in data cache hit rates. As a result, we find that the proposed scheme can reduce the energy consumption in four out of six applications with 8-KByte cache configuration. However, the proposed scheme may increase energy consumption with larger cache configuration (16-KByte in this case).

References

[1] Acotel. J2me. White Paper, 2003.

[2] B. Alpern, M. Butrico, T. Cocchi, J. Dolby, S. Fink, D. Grove, and T. Ngo. Experiences porting the Jikes RVM to Linux/IA32. In *Proceedings of 2nd Java(TM) Virtual Machine Research and Technology Symposium (JVM'02)*, San Francisco, California, August 1-2, 2002.

[3] ARM. ARM1026EJ-S. Technical Reference Manual, 2003.

[4] R.W. Atherton. Moving Java to the factory. *IEEE Spectrum*, pages 18–23, 1998.

[5] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[6] D. F. Bacon, C. R. Attanasio, H. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 92–103, 2001.

[7] J. M. Chang and E. F. Gehringer. Evaluation of an object-caching coprocessor design for object-oriented systems. In *Proceedings of IEEE International Conference on Computer Design*, pages 132–139, Oct. 3-6 1993.

[8] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection in an embedded Java environment. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA'02)*, Cambridge MA, February 2-6 2002.

[9] T. Chikayama and Y. Kimura. Multiple reference management in flat GHC. In *Proceeding of the 4th International Conference on Logic Programming*, pages 296–293, 1987.

[10] L. P. Deutsch and D. G. Bobrow. An efficient incremental automatic garbage collector. *Communications of ACM*, 19:522–526, 1976.

[11] A. Diwan, H. Lee, and D. Grunwald. Energy consumption and garbage collection in low-power computing, *Technical Report*, University of Colorado, 2002.

[12] D. P. Friedman and D.S. Wise. The one-bit reference count. *BIT*, 17:351–9, 1977.

[13] R. Fromm, S. Perissakis, N. Cardwell, C. E. Kozyrakis, B. McGaughy, D. A. Patterson, T. E. Anderson, and K. A. Yelick. The energy efficiency of IRAM architectures. In *ISCA*, pages 327–337, 1997.

[14] Y. Inamura, N. Ichiyoshi, K. Rokusawa, and K. Nakajima. Optimization techniques using the MRB and their evaluation on the MULTI-PSI/V2. In *E. L. Lusk & R. A. Overbeek, Logic Programming, Proc. of North American Conference*, pages 907–921, Cambridge, MA, 1989. MIT Press.

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

[16] L. Li, I. Kadayif, Y. Tsai, N. Vijaykrishnan, M. T. Kandemir, M. J. Irwin, and A. Sivasubramaniam. Leakage energy management in cache hierarchies. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 131–140. IEEE Computer Society, September 2002.

[17] S. Nikolaidis. Instruction-level energy characterization of an ARM processor. In *Proceedings of the 2nd MARLOW workshop*, September 2003.

[18] D. J. Roth and David S. Wise. One-bit counts between unique and sticky. In *Proceedings of Intl. Symp. on Memory Management (ISMM '98)*, pages 49–56, 1998.

[19] S. Steele. Accelerating to meet the challange of embedded Java. In *Arm Limited White Paper*, 2001.

[20] W. Stoye, T. Clarke, and A. Norman. Some practical methods for rapid combinator reduction. In *Proceeding of the Symposium on LISP and Functional Languages*, Austin TX, August 1984.

[21] Sun. J2ME: Building blocks for mobile devices. White Paper, May 2000.

[22] Sun. Java technology is everywhere, surpasses 1.5 billion devices worldwide. Press Release, February 2004.

[23] S. Tomar, S. Kim, and N. Vijaykrishnan. Use of local memory for efficient Java execution. In *Proceedings of IEEE International Conference on Computer Design*, Austin Texas, Sep. 23-26 2001.

[24] J. Trajkovic and A. Veidenbaum. Reducing SDRAM energy consumption in embedded systems. Technical Report TR04-02, University of California, Irvine, 2004.

[25] J. M. Velasco, D. Atienza, L. Pinuel, and F. Catthoor. Energy-aware modelling of garbage collectors for new dynamic embedded systems. In *Proceedings of First Int'l Workshop on Power-Aware Real-Time Computing*, Pisa, Italy, September 2004.

[26] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: a cycle-accurate energy estimation tool. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 340–345, New York, NY, USA, 2000. ACM Press.