**Java Memory Compression**

APPROVED BY SUPERVISING COMMITTEE:

Dr. Chia-Tien Dan Lo, Chair

Dr. Kay A. Robbins

Dr. Tom Bylander

Dr. Dakai Zhu

Dr. C. L. Philip Chen

ACCEPTED:

Dean, Graduate School

# Java Memory Compression

**by**

**Mayumi Kato, B.A.**

**DISSERTATION**

Presented to the Graduate Faculty of

The University of Texas at San Antonio

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY DEGREE IN COMPUTER SCIENCE**

THE UNIVERSITY OF TEXAS AT SAN ANTONIO

College of Science

Department of Computer Science

May 2007

## Acknowledgments

I would like to thank Dr. Chia-Tien Dan Lo, Dr. Kay A. Ribbins, Dr. Tom Bylander, Dr. Dakai Zhu, Dr. C. L. Philip Chen, and anonymous reviewers.

The research was supported by the Center for Infrastructure Assurance and Security (CIAS) under grant CIAS-26-0200-62 and by a Fellowship and travel awards from the Department of Computer Science and the College of Science at the University of Texas at San Antonio.

May 2007

**Java Memory Compression**

Mayumi Kato, Ph.D.
The University of Texas at San Antonio

Supervising Professor: Dr. Chia-Tien Dan Lo

Java enabled mobile/wireless handheld devices now dominate the market, and a variety of attractive entertainment services are targeted to these devices. However, applications on these devices are limited by their memory capacity and battery life. Memory compression is a key to overcoming these limitations. We will study in-memory compression for object-oriented languages such as Java together with its runtime environment.

A difficulty with in-memory/in-heap object compression is that objects and their object headers are compressed. The cost of decompression and compression increases that of accessing objects. We introduce a page-based approach and hardware solution to make time overhead negligible. A new Java Runtime Environment (JRE) is designed and implemented. A Java virtual machine with memory compression is developed and evaluated through real workload benchmark programs.

Experimental results show that up to 45% of memory is reduced without activating garbage collection and some of the programs run faster (speedup 1.125 in demo (HTTP)). The performance for web applications is, however, poor in both memory saving and speed. Since we targeted battery-powered handheld devices, our initial design (configuration I) is further examined using power and energy. Experimental results show that power/energy saving is very good for web applications with very small energy overhead due to compression and decompression. We conclude that configuration I outperforms the base configuration and works well in the addressed domain.

The experimental results further suggest that configuration I will further reduce

memory demand if used in combination with a garbage collector designed for a compressed heap (configuration II). The performance of the Java virtual machine has been evaluated by activating garbage collection and using benchmark applications. Experimental results show over 50% memory saving. This leads to a 53% power saving and 57% energy saving in the heap memory. We believe that in-memory object compression (runtime optimization) is feasible in Java-enabled, resource-limited, battery-powered handheld devices. Further improvement is done using system and code analyses based on space, power/energy, and time and/or in combination with rich off-line optimization.

# Table of Contents

# List of Tables

# List of Figures

# CHAPTER 1: Introduction

## Introduction

Technology advances and low cost have made mobile telephone service a boom industry. For example, prepaid cellular phones are set up to make consumer's budgets feasible: there is no access fee, no deposit and no monthly fee. Popular cellular phone vendors, NOKIA and MOTOROLA, preinstall programs for prepaid transactions. The average talk time is 5.38 hours, and the standby time is 307.25 hours (Table 1: Details in [1]). These cellular phones not only have functionality to talk but also have popular games and web utility applications. Time used on entertainment is getting longer than the talking time, and battery life is becoming a limiting factor. For example, a battery pack lasts for 5 to 6 hours when users enjoy game and web applications. Therefore, power consumption is an important issue in mobile/wireless small handsets.

Table 1: Battery Life of Popular Prepaid Wireless Devices [1]

| Device | Talk Time | Standby Time |
|---|---|---|
| NOKIA 1100 | 3 hours | 384 hours |
| MOTOROLA C139 | 8 hours | 384 hours |
| MOTOROLA C155 | 6.5 hours | 300 hours |
| 5NOKIA 2600 | 3.5 hours | 240 hours |
| MOTOROLA V170 | 6.5 hours | 300 hours |
| MOTOROLA V176 | 6 hours | 300 hours |
| MOTOROLA 0261 | 6 hours | 300 hours |
| NOKIA 2600 | 3.5 hours | 250 hours |

The current generation of mobile/wireless small handsets employs Java technology to implement a variety of phone services. Java technology is predicted to be integrated into 74% of wireless phones shipped in 2007 [2] because of its flexible user interfaces, robust security models, broad ranges of built-in network protocols and support for mobile/wireless applications. The acceleration of Java technology in mobile/wireless embedded systems has been studied at software and hardware levels. Java promises an optimal balance between speed,

user experience, memory footprint, power consumption, reliability and security. Current mobile Java solutions include Just-in-time (JIT)/Adaptive compilation (AOT) technologies, optimized interpretors, co-processors, dedicated Java processor, and Jazelle technologies [3]. Table 2 summarizes their brief descriptions (strength and weakness).

Table 2: Current Java Technologies for Mobile/Wireless Embedded Devices

| Current solution | Description |
|---|---|
| JIT/AOT compilation (software solution) | (strength) fast<br>(weakness) large memory overhead |
| optimized interpretor (software solution) | (strength) low memory overhead, easy to debug<br>(weakness) poor performance |
| co-processor (hardware solution) | (strength) reasonable fast<br>(weakness) large chip area, which leads to high power demand and system cost |
| dedicated Java processor (hardware solution) | (strength) excellent performance<br>(weakness) high implementation cost |
| Jazelle technology (hardware and software) | (strength) fast bytecode execution, low power consumption, low cost, fast product integration<br>(weakness) high memory demand |

Small Java-enabled mobile/wireless phones based on current solutions are able to access attractive mobile/wireless Internet services such as email, web, games, mobile commerce, and video/audio viewers. These devices also support Internet telephony that includes direct telephony control, datagram messaging, address book and calendar information, user profile access, power monitoring, and application installation. Unfortunately, attractive Java mobile/wireless applications consume a huge amount of power and the battery may drain out in several hours (5 to 6 hours in our experience). A solution for mobile/wireless embedded systems is to use low-power hardware for the CPU core and its peripheral and runtime power control. The adaptive computing machine (ACM) is one of the popular low-power

hardware platforms used for mobile/wireless embedded devices. Runtime power control is a new technique for low power consumption, which detects unnecessary units on hardware at runtime and powers them off, and similarly detects active units and powers on some of inactive ones as needed. Recently, a technique for minimizing the number of the active units was introduced for runtime power control [4]. Power control with runtime decompression has been proposed for Java language [4]. It reduces 20 % of the energy in executable code ( a Java classfile ) for an application by turning off unused memory banks. The leakage current on the inactive memory banks is eliminated as a result. This solution involves an optimized interpretor that runs applications with a memory management system and a decompression module [4].

We introduce runtime memory de/compression as a power control technique and propose computation mechanisms for hardware/software co-designed memory management of the Java heap. There are three goals in the proposed mechanism. The first goal is to reduce on-the-fly heap memory demands for large Java applications. Java applications can run on mobile/wireless devices with a small memory capacity. The second goal is to minimize the number of active memory banks by turning off unused memory banks and to eliminate the leakage current. This achieves low-power consumption on battery-powered mobile/wireless embedded devices. The last goal is to minimize the overhead in speed, memory, and power by hardware/software codesign.

Features of the proposed mechanisms are to handle objects, code and data in the Java heap and permanent memory space and to de/compress a group of objects, code and data on the fly. The approach minimizes the number of active memory banks, and eliminates their leakage currents. We design and implement our own interpretors optimized in speed, small memory footprint, low power, and energy consumption. The interpretors allow any third-party developed services to be deployed. Contributions of the work include integration of de/compression techniques into Java VM, and a Java compressed heap implementation that allows configuration with/without garbage collection. The implementation has a small memory footprint, low power and energy consumption, no speed overhead, and low network

bandwidth requirements.

The computation mechanism applied to Java heap and permanent memory space can also be applied to other fields such as disk-space compression, and network data compression. Its variations are expected to work for any type of memory management systems.

## Background

### Traditional Memory Management and Memory Compression

Operating systems [5] manage memory using variations of page segmentation. The operating systems allocate memory and free memory according to instructions from a program. Traditional programming languages [6], such as C, assume programmer-controlled memory management. Programmers determine points in a program at which a variable is allocated and deallocated. The program passes the information to the operating systems by calling a library function to allocate/deallocate variables. For example, given a memory request, memory space larger than the request is selected from the free list, and allocated. The memory space is put back to the free list when it is deallocated. Here, we discuss how in-memory compression is applied to the traditional memory management. The space in the free list is in compressed form, and the allocated space is also initially in the compressed form. For reading operations, the operating system, or the hardware decompresses the compressed space and sends it to the program. For writing operations, the operating system or the hardware compresses the data in the program and stores it in random access memory (RAM). For the deallocation, the operating system deallocates the memory space and put it back to the free list.

### Hardware Compression

High density of the very-large-scale integration (VLSI) technology makes it feasible to map software functions to hardware. In [7][8][9], a hardware dynamic memory management module is built to improve dynamic memory management performance to an extent

that may not be achievable by software. The new trend has created some common standards including 3G wireless protocols such as W-COMA, and encryption and security algorithms such as RSA and triple-DES. The Register Transfer Level (RTL) design [10] uses small area, consumes less power, and achieves high performance. Owning to the emergence of hardware description languages (HDLs) such as Verilog and very high speed integrated circuit Hardware Description Language (VHDL), RTL design and its verification can be done easily. Moreover, any design can be mapped to Field-programmable Gate Arrays (FPGAs) for further testing before mass production. FPGAs and HDLs make reconfigurable computing feasible.

Hardware implementation of compression algorithms has been studied extensively [11] [12] [13] [14]. De/compression algorithms can be mapped to hardware nicely. For example, the X-Match de/compression algorithm has been implemented in FPGAs and Application-Specific Integrated Circuits (ASICs) [14]. X-Match de/compression may have a performance improvement up to several fold over its software realization.

Figure 1 shows an hardware implementation of the X-Match compressor. The X-Match is a dictionary-based partial match compression algorithm. The dictionary entries are 4 bytes wide. A 4-byte pattern is considered to be a match if at least 2 characters are the same as some dictionary entry. If the input bytes match, the algorithm returns a tuple <ML, MT, Literal>, following a signal that indicates hit/miss. The first term ML is the match location, the second term MT is the match type, and the third term is Literal or empty. If an input byte does not match, the byte is transmitted literally. The input bytes are stored in the dictionary with a move-to-front (MTF) strategy: a new tuple is placed at the front of the dictionary while the rest move down one position. In the compression implementation, the data are put into a Content Addressable Memory (CAM) array to find a match. The match types are passed to the best-match decision logic. The resolved match type is sent to the Huffman coder, and the match location is sent to the phased binary coder with the next free location. The coded data is fed to the code assembler and sent to output ports.

The decompression implementation shown in Figure 2 performs a similar procedure

in a reversed order. As Figures 1 and 2 show, both decompression and compression are performed with the same number of cycles. The frequencies of decompression and compression are, however, different.

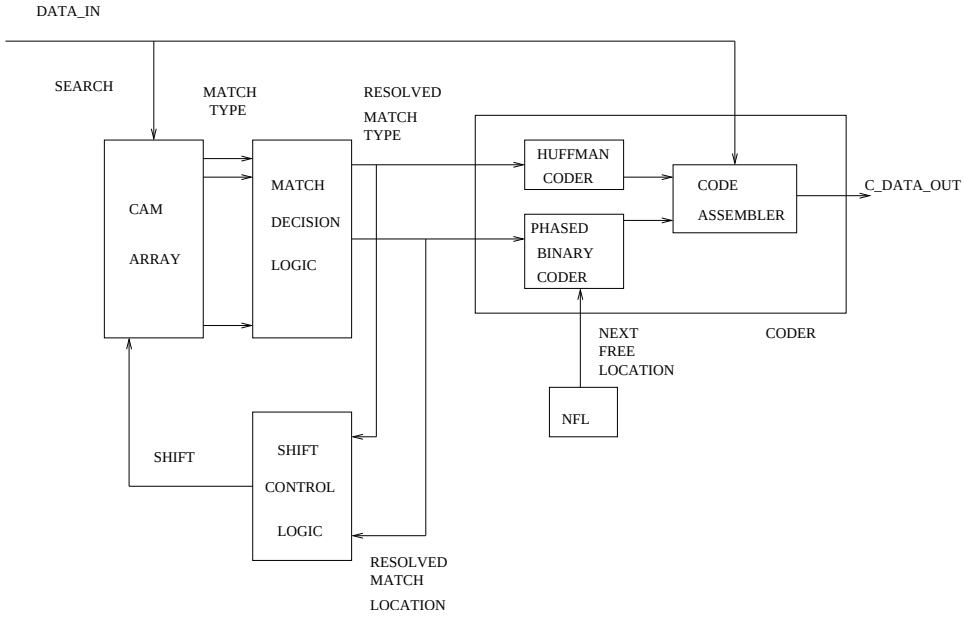



Figure 1: X-Match Compressor

## Power and Energy Consumption

Power consumption has been measured using the voltage drop across a register in the main power circuit or simulated using hardware description language (VHDL/Verilog) / C language. The typical design of a Java embedded system integrates the system, i.e., an operating system and a KVM (Kilobyte Virtual Machine), with firmware-based processors and other resources on a chip, called System on Chip (SoC). In a battery-powered environment, resource usage is constrained so as to save power. However, as the offered operations become more complex, their power consumption and processing time (execution time) will be more significant. For instance, processing animations and M-commerce applications obviously consume a huge amount of power because they usually need more memory and complicated

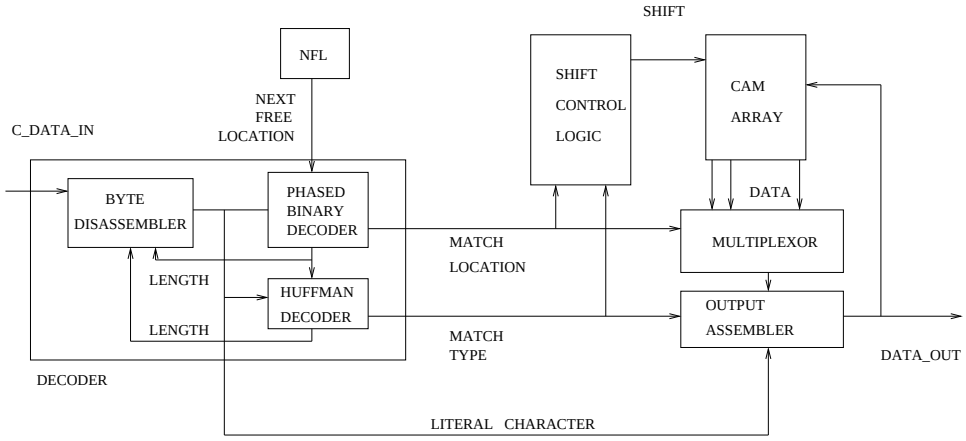



Figure 2: X-Match Decompresser

computations. System errors such as "out of power" or "out of memory" may occur. Some studies have shown that memory is one of the major power consumption sources in Java embedded systems [15]. Therefore, by compressing data and code, system power consumption can be reduced.

The most closely related work is the memory architecture proposed by Chen *et al.* [4][16][17]. Their design employed an SoC-based system and a technique to reduce energy consumption by turning off unused memory banks. The SoC with a decompressor for read-only memory achieves low leakage energy consumption by reducing the number of active (powered on) memory blocks that hold read only data, KVM code, and class libraries. The memory that contains read/write data is composed of memory banks, each of which can be independently turned on or off (a memory partition technique). A memory bank is called active if it is on; inactive if it is off. The total energy consumed by the partitioned memory is the sum of the energy consumed by each active bank. Figure 3 shows the high level view of this design, and Figure 4 details the modified X-Match decompressor.

The memory bank partitioning technique reduces the number of cells to be activated and mitigates power consumption. Chen *et al.* [4] study the trade-off between memory compression and power consumption for an embedded Java virtual machine (JVM and KVM) using the modified X-Match decompression algorithm. A simple round-robin strategy is

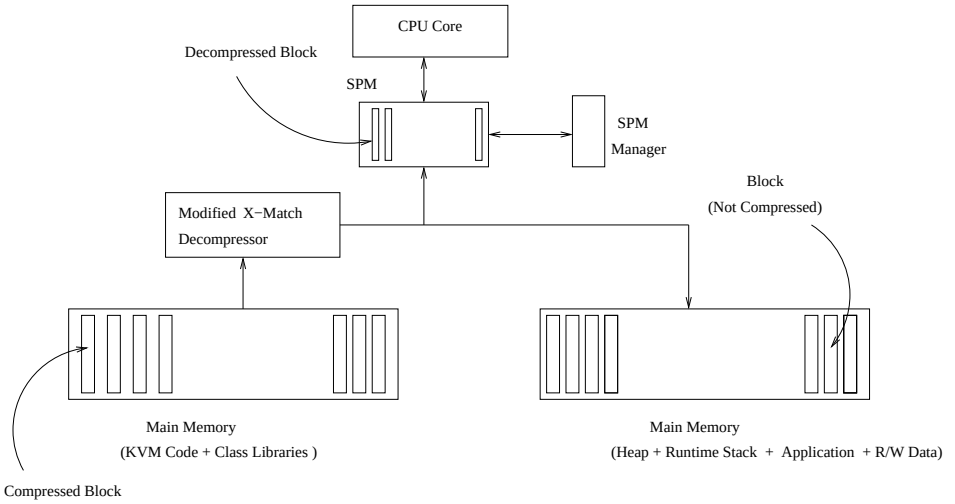



Figure 3: High-level View of the SoC Memory Architecture with Decompressor [4]

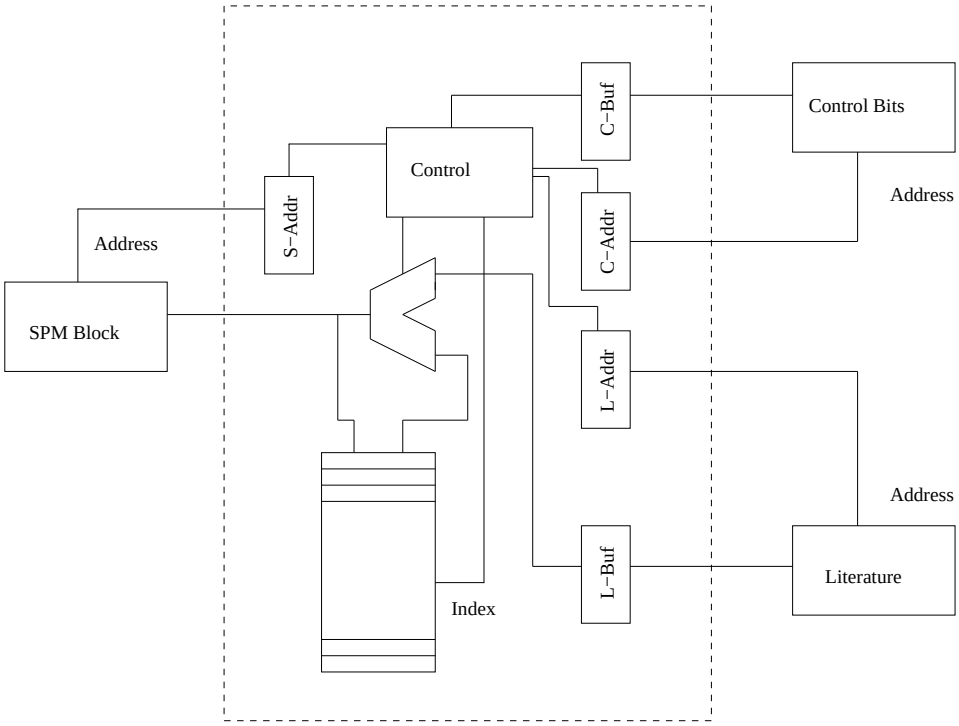



Figure 4: Details for the Modified X-Match Decompressor [4]

adopted in the modified X-Match algorithm instead of the move-to-front strategy. New tuples are appended to the end of the current dictionary. When the dictionary is full, the replacement pointer is moved to the last dictionary entry. When a data item belonging to a compressed memory block is requested, the whole block is decompressed and written into the Scratch Pad Memory (SPM). This strategy saves a lot of memory space because data in read-only memory is stored in compressed form. This reduces the leakage energy consumption in the read-only part of the main memory system. The amount of the saving is determined by the compression ratio and the algorithm used. Compression was found to be effective in reducing energy even when considering the runtime decompression overhead for most applications. In this approach, however, data stored in RAM are not compressed.

## Objectives

Java technology has been ported into mobile/wireless small handsets. Attractive Java mobile/wireless applications (email, web services, games, mobile commences, video/audio services, etc.) consume a huge amount of memory and power. We introduce a runtime memory de/compression with a memory bank partitioning technique in combination with power controls to the Java enabled mobile/wireless devices and solve the problems on the mobile/wireless small handsets.

The objectives of this work are threefold. First, on-the-fly heap memory demands are reduced for large Java applications. Java applications can run on mobile/wireless devices with a small memory capacity. Second, the number of active memory banks is minimized and the leakage current is eliminated by turning off unused memory banks. This achieves low-power consumption on the battery-powered mobile/wireless embedded devices. Last, the approach also minimizes the overhead in speed, memory and power by hardware/software codesign.

In the remaining chapters of this dissertation, CHAPTER 2 discusses features of object-oriented languages Java, a problem of in-memory object compression, and its solutions. CHAPTER 3 presents in-memory object compression without automatic mem-

ory management, which is one example of the design and implementation of the hardware/software codesigned Java runtime environment. CHAPTER 4 describes in-memory object compression with automatic memory management system, which is another example of the design and implementation of the hardware codesigned Java runtime environment. CHAPTER 5 compares the in-memory object compression with/without the automatic memory management between the standard Java/state-of-the-art technologies and our approach. Also depicted are differences. CHAPTER 6 concludes the research and its future directions.

.

# CHAPTER 2: Memory Compression for Object-Oriented Languages

In this chapter, we discuss objects, their memory management systems, a problem of in-memory object compression, and its solutions. System and code analysis is also modeled for performance evaluation of battery-powered embedded systems.

## Virtual Machines and Object-Oriented Languages

Traditional computer systems [5] run multiple processes controlled by an operating system. The operating system accesses the bare hardware in response to system calls from the processes. Optimal utilization of the hardware requires some changes in the operating system. The computer system may be unavailable for a while due to bugs. In these circumstances, a virtual machine system is introduced to solve the problem. This virtual machine approach provides a "virtual" copy of the underlying computer and allows system programmers to change the operating system. Every new development is performed on the virtual machine concurrently to the regular operation.

The virtual machine concept can also be extended to programming languages. For example, the Java language has a virtual machine called Java Virtual Machine (JVM) or Java interpretor. JVM is a stack-based execution engine that takes Java classfiles as inputs and executes Java bytecode on any type of operating systems. A basic unit of data is the object [6]. In practice, objects are defined as a class instance or an array and implemented using "struct (in C language)" [18][19]. The objects are interpreted by the Java Virtual Machine, and allocated/deallocated by its an automatic memory manager called a garbage collector which frees programmers from explicit memory allocation/ deallocation.

Figure 5 shows an example of an object with its object header and its $n$ object reference holders. The objects are classified for the garbage collection (GC). There are eleven object types and six GC operations (one for each group) in the current implementation of Java [20]. Table 3 lists the groups and their types. The objects are stored in permanent heap

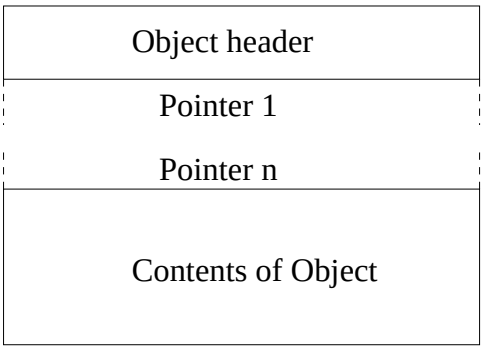



Figure 5: Object

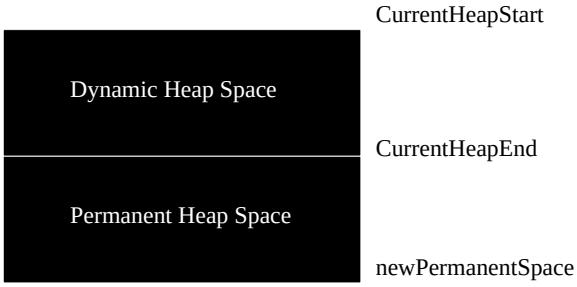



Figure 6: Alignment of Heap Space

space and/or in dynamic heap space (Figure 6). The operating system maps the heap spaces on the hardware according to instructions from the Java Virtual Machine. The Java Virtual Machine manages the dynamic and permanent heap spaces as virtualized memory spaces of the real memory in the hardware. The real memory itself is managed in the traditional way by the operating system or the hardware. Systems and policies for the virtual memory spaces are similar to the operating system's, but they have extra support for automatic memory management. The type information discussed here is set to the type field in the object header (Figure 7) and is extracted during garbage collection. The garbage collector performs the operation that corresponds to the type information.

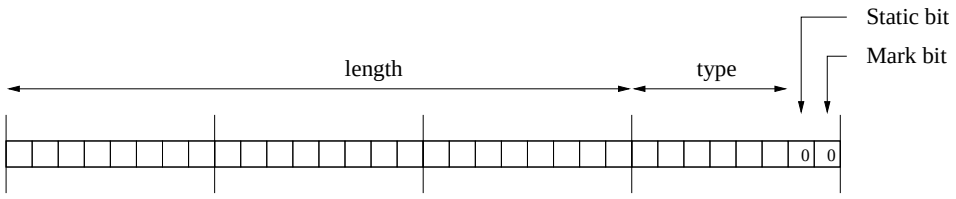



Figure 7: Data Structure of Object Header used in Implementation

Table 3: Java Object Types

| Group | Type | Description | Dynamic (Dy) / Permanent (Pe) |
|---|---|---|---|
| 1 | GCT_FREE | garbage collection-free | - |
| 2 | GCT_NOPOINTERS | no pointers inside them and can be ignored safely during garbage collection. | Dy/Pe |
| 3 | GCT_INSTANCE | Java object instances. Java-level objects which may have mutable pointers inside them. | Dy |
| | GCT_ARRAY | Java arrays with primitive values | Dy/Pe |
| | GCT_OBJECTARRAY | Java arrays with object references | Dy |
| 4 | GCT_METHODTABLE | pointers to Java code and handlers. | Dy/Pe |
| 5 | GCT_POINTERLIST | internal V objects which may have mutable pointers inside them. | Dy/Pe |
| | GCT_EXECSTACK | | Dy |
| | GCT_THREAD | | Dy |
| | GCT_MONITOR | | Dy |
| 6 | GCT_WEAKPOINTERLIST | a weak pointer list that will be marked/copied after all other objects. | Dy |

## Automatic Memory Management System

Automatic memory management systems, garbage collection, have been studied for several decades [21]. There are three types of garbage collection algorithms in the literature: reference counting, mark-sweep, and copying.

Java memory management for mobile/wireless embedded devices employs a mark-sweep algorithm using a stop-start algorithm that suspends programs up to a few seconds. The mark-sweep algorithm relies on a global traversal of all live objects to determine which objects are ready for reclamation. If a request is made for a new object, "useful" processing is temporarily suspended while the garbage collector routine is called to mark objects reachable from a root and an active object, to sweep all currently unused objects from the heap and

to put them back into a pool of free memory. The behavior of the mark-sweep algorithm is not acceptable for interactive programs (e.g., web viewer and M-commerce) and programs that need hard real-time requirements (e.g., audio player and video watcher).

The reference counting scheme maintains an invariant [21]: "the reference count of each cell is equal to the numbers of pointers to that cell from roots or heap cells." The basic operation is to count the number of references to each cell from other, active cells or roots and to keep track of whether cells are in use or not. The virtue of reference counting lies in its simplicity of keeping track of cells and the distribution of its overhead throughout the program execution. The reference counting algorithm is faster than the mark-sweep algorithm.

The copying algorithm divides the heap equally into two semi-spaces (current data and the other data). It starts by flipping the role of the two spaces and traverses the active data structure in the old semi-space, and copies each live cell into the new semi-space when the cell is visited. The strength of the copying algorithm is an extremely low allocation cost: the out-of-space check is a simple pointer comparison, new memory is acquired simply by incrementing the free-pointer, and fragmentation is eliminated by compacting the active data. The weakness of the copying algorithm is that the memory space required is doubled compared with non-copying collection (e.g., mark-sweep, reference counting), and the copying algorithm incurs more page faults than mark-sweep garbage collection.

Low power consumption comes from a small memory space. The copying garbage collector uses twice as much memory compared with a non-copying garbage collector (mark-sweep, a reference counting) and is not suitable for low power consumption on small memory handsets. The mark-sweep algorithm traverses the entire heap and sweeps unmarked objects when there is not enough heap space for an object allocation. During the garbage collection, the main stream of operations are suspended.

On every object reference assignment, the reference counting algorithm updates the counters associated with objects and objects reachable from object reference holder(s). The garbage collection time of the reference counting algorithm is distributed over the entire

program execution. It is attractive, but we are not interested in the reference counting algorithm because it cannot overcome a problem that objects inside a given object is compressed (Figure 8). Accessing one object may require several page accesses and thus several decompression and compression calls. Writing back an object in the memory may require several page accesses and thus several decompression and compression calls. When the automatic memory management updates object liveness, object headers of each object and objects pointed by its object reference holder(s) must be decompressed and compressed again. The de/compression is very expensive. The problem discussed here cannot be solved by simply changing the type of the garbage collection algorithm because the problem comes from a data structure of objects. We therefore use the garbage collection algorithm (mark-sweep algorithm with compaction) used in the current Java for mobile/wireless embedded devices.

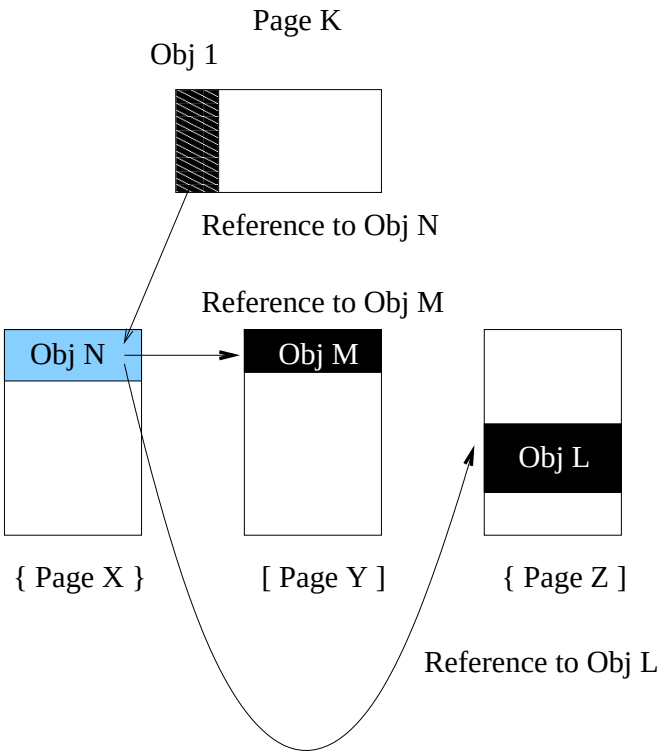



Figure 8: Problem on Compressed Heap

## Mark-Sweep with Compaction Algorithm

Garbage collection deallocates storage for objects that are not live (dead). The liveness of objects is defined as follows:

Definition: Define $\rightarrow$ as the "points-to" relation. For any code or (computation) root M and node N, M $\rightarrow$ N if and only if M holds a reference to N. The set of live nodes is the transitive referential closure of the set of roots under this relation, i.e., the least set live where

$$live = \{ \text{ N} \in \text{Nodes} \mid (\exists r \in Roots.r \rightarrow N) \vee (\exists M \in live.M \rightarrow N \ ) \ \}$$

An object is live if its address is held in a root, or there is a pointer to it, which is held in another live object (node).

The garbage collector detects live objects and eliminates dead ones to make storage space for new objects. This guarantees efficient memory management, but still suffers from internal fragmentation (memory that is internal to a partition of the allocated memory block, but is not being used). Java technology employs a compaction technique as an option to resolve the internal fragmentation problem. The compaction mechanism relocates live objects to one side of heap memory and eliminates its internal holes.

A mark-sweep garbage collection algorithm with compaction is employed by [20] for mobile/wireless embedded systems and consists of three phases: mark, sweep, and compaction.

1. mark phase: all objects reachable from the root node and other live objects are marked as live objects,
2. sweep phase: objects unmarked in the mark phase are swept from Java heap as garbage, and
3. compaction phase: live objects are relocated to one side of the Java heap.

The pseudocode presented in table 4 through 14 shows how the garbage collector traverses and processes objects. Table 4 shows the top level procedure of the garbage collection algorithm (garbageCollection). This procedure calls garbageCollectorForReal(Table 5) to mark, sweep, and compact.

Table 4: GarbageCollection

```
    garbageCollection(moreMoemory)

1.  if gcInprogress != 0
2.      fatalVMError(KVM_MSG_CIRCULAR_GC_INVOCATION)
3.  gcInProgress ← gcInProgress + 1
4.  RundownAsynchroniousFunctions()
5.  MonitorCache ← NULL
6.  if CurrentThread
7.      storeExecutionEnvironment(CurrentThread)
8.  garbageCollectForReal(moreMemory)
9.  if CurrentThread
10.     loadExecutionEnvironment(CurrentThread)
11. RestartAsynchronousFunctions()
12. gcInProgress ← 0
```

The mark phase consists of markRootObjects (Table 6), markNonRootObjects (Tables 7 and 8), markWeakPointerLists (Table 9). The sweep phase is done in sweepTheHeap (Table 10) and the compaction phase has compactTheHeap (Table 11), updateRootObjects (Table 12), updateHeapObjects (Table 13) and MARK_AND_TAIL_RECURSE (Table 14) as major procedures. The mark-sweep with compaction is featured in marking and updating live objects. Objects directly reachable from a root are marked and/or updated using object reachable graphs and their object types. During the operation, objects inside an object are also marked and/or updated. Typical operations are shown in markChildren (Table 8). If an object is a Java object instance (GCT_INSTANCE), its possible monitor is marked and all fields of the object are scaned. If a field is a non-static pointer field, objects kept

Table 5: garbageCollectorForReal

```
    garbageCollectorForReal(realSize)

1.  markRootObjects()
2.  markNonRootObjects()
3.  markWeakPointerLists()
4.  firstFreeChunk ← sweepTheHeap(&maximumFreeSize)
5.  #if ENABLE_HEAP_COMPACTION
6.      if realSize > maximumFreeSize
7.          freeStart ← compactTheHeap(&CurrentTable, firstFreeChunk)
8.      if currentTable.length > 0
9.          updateRootObjects(&currentTable)
10.         updateHeapObjects(&currentTable, freeStart)
11.     if freeStart < CurrentHeapEnd -1
12.         firstFreeChunk ← freeStart
13.         firstFreeChunk→size ←
14.                 ( CurrentHeapEnd - freeStart − HEADERSIZE ) << TYPEBITS
15.         firstFreeChunk→next← NULL
16.     else
17.         firstFreeChunk ← NULL
18. #endif
19. FirstFreeChunk ← firstFreeChunk
```

by object reference headers inside the object are marked in a tail recursive manner. This operation is repeated toward the superclass of the object. If an object is a Java array with primitive value (GCT_ARRAY), its possible monitor is marked. If an object is a Java array with object references (GCT_OBJECTARRAY), its possible monitor is marked and objects in the array are marked live in a tail recursive manner (Table 14). If an object is a stack (GCT_EXECSTACK), nothing will be done in markChildren of markNonRootObjects. The executing stack is scanned in markRootObject. If an object is a monitor (GCT_MONITOR), nothing will be done in markChildren. If an object is the type of GCT_NOPOINTER, the object does not have any pointer in it and nothing will be done in markChildren. Similar

operations are also shown in updateHeapObjects (Table 13).

Tables 15 and 16 summarize the object types and operations of dynamic and permanent heap space where M indicates the mark phase and C denotes the compact phase. The first column shows the object types. The second column clarifies the object reference holder(s) inside the object. The third column indicates if the mark phase and/or the compaction phase uses the object reference holder(s). The fourth column describes how an object and object(s) inside the object are marked/updated. There are eleven types of objects to be stored in dynamic heap space and three types of objects to be stored in permanent heap. For example, if the type of a given object is GCT_INSTANCE, there are four object reference holders. One is for the given object. The others are for objects reachable from the given objects. In the table, the given object corresponds to "object" in the second column. Similarly, the other three objects are shown in the second column (object → ofClass[→fieldTable→u.offset], object → data[offset].cellp, object → superClass). The objects are accessed in both the mark phase and the compaction phase, and labeled with M and C in the third column. In the mark and compaction phases, the monitor of the "object" is marked live and then the objects inside the "object" are marked/updated by walking through all the fields of the "object" and see if they contain any pointer (object reference holder). If it is a non-static pointer field, pointers are marked/updated (using a tail recursive manner in the mark phase). The operation is repeated toward its superClass. The tail recursive access of objects is the major cause of the problem in the in-memory object compression.

## Memory Compression

Two memory types are used for embedded systems. One is Read-Only Memory (ROM) and another is Random Access Memory (RAM) that includes dynamic and permanent heap spaces. A Java Virtual Machine for mobile devices (VM) uses the ROM and the RAM to run a Java application in general. The ROM stores Java class files, and the contents of the ROM space can be optimized off-line. For example, Java class files have been

Table 6: markRootObjects

```
    markRootObjects

1.  *heapSpace ← CurrentHeap
2.  *heapSpaceEnd ← CurrentHeapEnd
3.  ptr ← &GlobalRoots[0]
4.  endptr ← ptr + GlobalRootsLength
5.  for ptr to endptr
6.        MARK_OBJECT_IF_NON_NULL(*(ptr→cellpp))
7.  ptr ← &TemporaryRoots[0]
8.  endptr ← ptr + TemporaryRootsLength
9.  for ptr to endptr
10.       location ← *ptr
11.       if location.cell == -1
12.             MARK_OBJECT_IF_NON_NULL(ptr[2].cellpp)
13.         ptr += 2
14.   else
15.         MARK_OBJECT_IF_NON_NULL(*(location.cellpp))
16. stringTable ← InternStringTable
17. if ROMIZING || stringTable != NULL
18.       count ←stringTable→bucketCount
19.       while -count >= 0
20.         instance ← stringTable→bucket[count]
21.         for instance != NULL // instance ← instance →next
22.             checkMonitorAndMark(instance)
23. if ROMIZING || ClassTable != NULL
24.       FOR_ALL_CLASSES(clazz)
25.         checkMonitorAndMark(clazz)
26.         if !IS_ARRAY_CLASS(clazz)
27.             iclazz ← clazz
28.             statics ← iclazz→staticFields
29.             methodTable ← iclazz→methodTable
30.             MARK_OBJECT_IF_NON_NULL(iclazz→initThread)
31.       if clazz→accessFlags & ACC_ROM_CLASS
32.         continue
33.       if USESTATIC
34.         MARK_OBJECT_IF_NON_NULL(iclazz→constPool)
35.         MARK_OBJECT_IF_NON_NULL(iclazz→ifaceTable)
36.         MARK_OBJECT_IF_NON_NULL(iclazz→fieldTable)
37.         MARK_OBJECT_IF_NON_NULL(iclazz→methodTable)
38.       if statics != NULL
39.         count ← statics→length
40.         while -count == 0
41.         MARK_OBJECT_IF_NON_NULL(
42.         statics→data[count].cellp))
43.       if iclazz→status == CLASS_VERIFIED
44.         continue
45.       FOR_EACH_METHOD(thisMethod, methodTable)
46.         if !thisMethod→accessFlags & ACC_NATIVE
47.             checkValidHeapPointer(
48.             thisMethod→u.java.stackMaps.verifierMap)
49.       MARK_OBJECT_IF_NON_NULL(
50.         thisMethod→u.java.stackMaps.verifierMap)
51.       END_FOR_EACH_METHOD
52.   END_FOR_ALL_CLASSES
53.   for each thread in AllThreads
54.       // thread = thread→nextAliveThread
55.         MARK_OBJECT(thread)
56.         if thread→javaThread != NULL
57.             MARK_OBJECT(thread→javaThread
58.             if thread→stack != NULL
59.             markThreadStack(thread)
```

Table 7: markNonRootObjects

```
    markNonRootObjects

1.  do
2.        WeakPointers ← NULL
3.        initializeDeferredObjectTable()
4.        for scanner ← CurrentHeap to endScanPoint
5.            // scanner ← scanner + SIZE(*scanner) + HEADERSIZE
6.        if ISMARKED(*scanner)
7.            *object ← scanner + 1
8.            markChildren(object, object, MAX_GC_DEPTH)
9.  while deferredObjectTableOverflow
```

Table 8: markChildren

```
      markChildren(object, limit, remainingDepth)

 1.   *heapSpace ← CurrentHeap
 2.   *heapSpaceEnd ← CurrentHeapEnd
 3.   *nextObject ← NULL
 4.   remainingDepth ← remainingDepth - 1
 5.   for(;;)
 6.      *header ← object - HEADERSIZE
 7.      gctype ← TYPE(*header)
 8.      switch (gctype)
 9.      case GCT_INSTANCE:
10.         instance ← object
11.         clazz ← instance→ofClass
12.         checkMonitorAndMark(instance)
13.         while (clazz)
14.          FOR_EACH_FIELD(thisField, clazz→fieldTable)
15.           if ( thisField→accessFlags
16.           & ACC_POINTER | ACC_STATIC) == ACC_POINTER
17.           offset ← thisField→u.offset
18.           *subobject ← instance→data[offset].cellp
19.           MARK_AND_TAIL_RECURSE(subobject)
20.          END_FOR_EACH_FIELD
21.          clazz ← clazz→superClass
22.         break // end for
23.      case GCT_ARRAY:
24.         checkMonitorAndMark(object)
25.         break
26.      case GCT_POINTERLIST:
27.         list ← object
28.         length ← list→length
29.         ptr ←&list→data[0].cellp
30.         goto markArray
31.      case GCT_WEAKPOINTERLIST:
32.         object→gcReserved ← WeakPointers
33.         WeakPointers ← object
34.         break
35.      case GCT_OBJECTARRAY:
36.         checkMonitorAndMark(object)
37.         length←object→length
38.         ptr →&object→data[0].cellp
39.      markArray:
40.          while –length >= 0
41.          *subobject ←*ptr++
42.          MARK_AND_TAIL_RECURSE(subobject)
43.   6mm break
44.      case GCT_METHODTABLE:
45.         FOR_EACH_METHOD(thisMethod, object)
46.         if ( thisMethod→accessFlags & ACC_NATIVE ) == 0
47.   MARK_OBJECT(thisMethod→u.java.code)
48.   MARK_OBJECT_IF_NON_NULL(
49.   thisMethod→u.java.handlers)
50.          END_FOR_EACH_METHOD
51.      case GCT_MONITOR:
52.          break
53.       case GCT_NOPOINTERS:
54.          break
55.       case GCT_EXECSTACK:
56.          break
57.       case GCT_THREAD:
58.          break
59.      default:
60.         fatalVMError(
61.         KVM_MSG_BAD_DYNAMIC_HEAP_OBJECTS_FOUND)
62.   if nextObject != NULL
63.         object ← nextObject
64.         nextObject ← NULL
65.   else if ( remainingDepth == MAX_GC_DEPTH - 1
66.      && deferredObjectCount > 0)
67.      object ← getDeferredObject()
68.   else
69.      break
```

Table 9: markWeakPointerLists

```
      markWeakPointerLists()

 1.   if CurrentThread
 2.          currentNativeLp ← CurrentThread→nativeLp
 3.   for list ← WeakPointers to NULL // list ← list→gcReserved
 4.          (*finalizer)(INSTANCE_HANDLE) ← list→finalizer
 5.          *ptr ← &list→data[0]
 6.          *endPtr ← ptr + list→length
 7.          for ptr ← endPtr
 8.                object ← ptr→cellp
 9.                if object ! = NULL
10.                  if !ISKEPT( (object)[-HEADERSIZE] )
11.                    ptr→cellp ← NULL
12.                    if finalizer
13.                      if CurrentThread
14.                        CurrentThread→nativeLp←&object
15.                      finalizer(&object)
16.          if CurrentThread
17.                CurrentThread→nativeLp ← currentNativeLp
```

Table 10: sweepTheHeap

```
    sweepTheHeap(*maximumFreeSizeP)

 1. firstFreeChunk ← NULL
 2. *nextChunkPtr ← &firstFreeChunk
 3. done ← FALSE
 4. *scanner ← CurrentHeap
 5. *endScanPoint ← CurrentHeapEnd
 6. maximumFreeSize ← 0
 7. do
 8.       while scanner < endScanPointer && ISKEPT(*scanner)
 9.           *scanner ← *scanner &  MARKBIT
10.           scanner ← scanner + SIZE(*scanner) + 1
11.       lastLive ← scanner
12.       while scanner < endScanPointer && !ISKEPT(*scanner)
13.           scanner ← scanner + SIZE(*scanner) + 1
14.       if scanner == endScanPoint
15.           if scanner == lastLive
16.             break
17.           else
18.           done ← TRUE
19.       thisFreeSize ← (scanner - lastLive - 1)
20.       newChunk→size ← thisFreeSize << TYPEBITS
21.       newChunk← lastLive
22.       *nextChunkPtr ← newChunk
23.       nextChunkPtr ← &newChunk→next
24.       if thisFreeSize > maximumFreeSize
25.           maximumFreeSize ← thisFreeSize
26.   while !done
27. *nextChunkPtr ← NULL
28. *maximumFreeSizeP ← maximumFreeSize
29. return firstFreeChunk
```

Table 11: compactTheHeap

```
    compactTheHeap(*currentTable, firstFreeChunk)

 1. copyTarget ← CurrentHeap
 2. currentHeapEnd ← CurrentHeapEnd
 3. lastRoll ← 0
 4. freeChunk ← firstFreeChunk
 5. *table ← NULL
 6. for scanner ← CurrentHeap, count ← -1
 7.     live ← scanner
 8.     if freeChunk ! = NULL
 9.         liveEnd ← freeChunk
10.         scanner ← liveEnd + SIZE (*liveEnd) + HEADERSIZE
11.         freeChunk ← freeChunk→next
12.     else
13.         liveEnd ← scanner ← currentHeapEnd
14.     if count < 0
15.         copyTarget ← liveEnd
16.     else
17.         liveSize ← PTR_DELTA(liveEnd, live)
18.         if count == 0
19.             for i ← 0 to liveSize
20.                 CELL_AT_OFFSET(copyTarget, i) ← CELL_AT_OFFSET(live, i)
21.             table←scanner - 1
22.         else
23.             extraSize ← PTR_DELTA(scanner, liveEnd)
24.             table ← slideObject(copyTarget, live, liveSize, extraSize, table, count, &lastRoll)
25.         table[count].address ←live
26.         table[count].offset ←PTR_DELTA(live, copyTarget)
27.         copyTarget ← PTR_OFFSET(copyTarget, liveSize)
28.     if scanner >= currentHeapEnd
29.         break
30. if lastRoll > 0
31. sortBreakTable(table, lastRoll)
32. currentTable→table ←table
33. currentTable→length ← count + 1
34. return copyTarget
```

Table 12: updateRootObjects

```
     updateRootObjects(*currentTable)

 1.  ptr ← &GlobalRoots[0]
 2.  endptr ← ptr + GlobalRootsLength
 3.  for ptr < endptr
 4.        updatePointer(ptr→cellpp, currentTable)
 5.  ptr ← &TemporaryRoots[0]
 6.  endptr ← ptr + TemporaryRootsLength
 7.  for ; ptr < endptr; ptr++
 8.        location← *ptr
 9.        if location.cell == -1
10.              offset ← *(ptr[1].charpp) - ptr[2].charp
11.              updatePointer(&ptr[2], currentTable)
12.              *(ptr[1].charpp) ← ptr[2].charp + offset
13.              ptr += 2
14.        else
15.        updatePointer(location.cellp, CurrentTable)
16.  stringTable ← InternStringTable
17.  if ROMIZING || stringTable != NULL
18.        count ←stringTable→bucketCount
19.        while –count >= 0
20.              instance ← stringTable→bucket[count]
21.        for instance != NULL // instance ← instance →next
22.              updateMonitor(instance, currentTable)
23.  if ROMIZING || ClassTable != NULL
24.        FOR_ALL_CLASSES(clazz)
25.              updateMonitor(clazz, currentTable)
26.              if !IS_ARRAY_CLASS(clazz)
27.                iclazz ← clazz
28.                statics ← iclazz→staticFields
29.                initThread ← iclazz→initThread
30.                if initThread != NULL
31.                  updatePointer(&initThread, currentTable)
32.                  setClassInitialThread(iclazz, initThread)
33.                if clazz→accessFlags & ACC_ROM_CLASS
34.                  continue
35.                if USESTATIC
36.                  updatePointer(iclazz→constPool, currentTable)
37.                  updatePointer(iclazz→ifaceTable, currentTable)
38.                  updatePointer(iclazz→fieldTable, currentTable)
39.                  updatePointer(iclazz→methodTable, currentTable)
40.                if statics != NULL
41.                  count ← statics→length
42.                    while –count == 0
43.                      updatePointer( &statics→data[count].cellp, CurrentTable)
44.                if iclazz→status == CLASS_VERIFIED
45.                  continue
46.                FOR_EACH_METHOD(thisMethod, iclazz→methodTable)
47.                  if !thisMethod→accessFlags & ACC_NATIVE
48.                    if !USESTATIC || inCurrentHeap(thisMethod)
49.                      updatePointer(
50.                        &thisMethod→u.java.stackMaps.verifierMap, currentTable)
51.                    else
52.                      oldValue ←
53.                      thisMethod→u.java.stackMaps.verifierMap
54.                      newValue ← oldValue
55.                      updatePointer(&newValue, currentTable)
56.                      if oldValue != newValue
57.                        cp ← iclazz→constPool
58.                          offset ← &thisMethod
59.                          →u.java.stackMaps.pointerMap - cp
60.                          modifyStaticMemory(
61.                          cp, offset, &newValue, sizeof)POINTERLIST)
62.                END_FOR_EACH_METHOD
63.        END_FOR_ALL_CLASSES
```

Table 13: updateHeapObjects

```
updateHeapObjects(
*currentTable, endScanPoint)

for(scanner ← CurrentHeap;
scanner < endScanPoint;
scanner ← scanner + SIZE(*scanner) + HEADERSIZE )
      *header ← scanner
      *object ← scanner + 1
      gctype ← TYPE(*header)
      switch (gctype)
      case GCT_INSTANCE:
            instance ← object
            clazz ← instance→ofClass
            updateMonitor(instance, currentTable)
            while (clazz)
              FOR_EACH_FIELD(thisField, clazz→fieldTable)
                if (thisField→accessFlgs
                & ACC_POINTER | ACC_STATIC)
                == ACC_POINTER
                  updatePointer(
                  &instance→data[thisField→u.offset].cellp,
                  currentTable)
              END_FOR_EACH_FIELD
              clazz ← clazz→superClass
            break // end for
      case GCT_ARRAY:
            updateMonitor(object, currentTable)
            break
      case GCT_POINTERLIST:
            list ← object
            length ← list→length
            ptr ←&list→data[0].cellp
            goto markArray
      case GCT_WEAKPOINTERLIST:
            list ← object
            length ← list→length
            ptr &list→data[0].cellp
            goto markArray:
      case GCT_OBJECTARRAY:
            array ← object
            updateMonitor(array, currentTable)
            length←object→length
ptr → &array←data[0].cellp
  markArray:
     while –length >= 0
        updatePointer(ptr, currentTable)
        ptr++
       break
  case GCT_METHODTABLE:
     FOR_EACH_METHOD(thisMethod, object)
       if (thisMethod →accessFlags & ACC_NATIVE) == 0
        updatePointer(thisMethod→u.java.code, currentTable)
        updatePointer(thisMethod→u.java.handlers, currentTable)
     END_FOR_EACH_METHOD
    break


  case GCT_MONITOR:
     monitor ← object

     updatePointer(&monitor←owner, currentTable)
     updatePointer(&monitor←monitor_waitq, currentTable)
           updatePointer(&monitor←condvar_waitq, currentTable)
           break
  case GCT_NOPOINTERS:
     break
  case GCT_EXECSTACK:
     break
  case GCT_THREAD:
     thread ← object
     updatePointer(&thread→nextAliveThread, currentTable)
     updatePointer(&thread→nextThread, currentTable)
     updatePointer(&thread→javaThread, currentTable)
     updatePointer(&thread→monitor, currentTable)
     updatePointer(&thread→nextAlarmThread, currentTable)
     updatePointer(&thread→stack, currentTable)
     break
  default:
     fatalVMError(
     KVM_MSG_BAD_DYNAMIC_HEAP_OBJECTS_FOUND)
// end of switch statement
```

Table 14: MARK_AND_TAIL_RECURSE

```
#define MARK_AND_TAIL_RECURSE(child)
if inHeapSpaceFast(child)
      _tmp_ ← OBJECT_HEADER(header)
      OBJECT_HEADER(child) ← _tmp_ | MARKBIT
if child < limit
      if nextObject ! = NULL
            RECURSE(nextObject)
      nextObject ← child
```

Table 15: Dynamic Heap Space: Object Type and Operations (1)

| Object Type | Reference Holder | Phases | Operations |
|---|---|---|---|
| GCT_INSTANCE | object<br>object → of Class<br>[→ fieldTable<br>→ u.offset]<br>object → data[offset].cellp<br>object→ superClass | M, C | Mark the possible monitor to be alive<br>Walk through all the fields of the object and<br>see if they contain pointer.<br>For each field, if it is a non-static pointer field,<br>mark pointers to be alive<br>(using a tail recursive function in mark phase).<br>The operation is repeated toward superclass. |
| GCT_ARRAY | object | M, C | Check Monitor and Mark. |
| GCT_OBJECTARRAY | object | M, C | Check Monitor and Mark.<br>Mark objects in the array alive (using<br>a tail recursive function in mark phase). |
| GCT_EXECSTACK | object | | This is handled by the thread that the stack belongs to. |
| GCT_MONITOR | <br>object<br>object → owner<br>object → monitor_waitq<br>object → condvar_waitq | M?<br>C | The only pointers in a monitor will get handled elsewhere<br>Update the pointers |
| GCT_NOPOINTERS | object | | The object does not have any pointers in it.<br>Do nothing |
| GCT_THREAD | <br>object<br>object → javaThread<br><br>object → stack<br>object → nextAliveThread<br>object → nextThread<br>object → monitor<br>object → nextAlarmThread | <br>M, C<br>M, C<br><br>M, C<br>C<br>C<br>C<br>C | <br>Update the pointers<br>Update thread and stack |
| GCT_WEAKPOINTERLIST | object | M, C | Push the object onto the linked list of weak pointers.<br>Keep objects in the list alive.<br>( It does not mark the children. If the objects are<br>updated as being alive because of<br>other references, we only want to update the pointers. ) |
| GCT_METHODTABLE | object<br>object → u.java.code<br>object → u.java.handlers | M, C | For each method in the table,<br>mark the object (code) and<br>mark the object (handlers) if it is non null. |
| GCT_POINTERLIST | object | M, C | Mark objects in the list alive (using<br>a tail recursive function in mark phase) |

Table 16: Permanent Heap Space: Object Type and Operations (2)

| Object Type | Reference Holder | Phases | Operations |
|---|---|---|---|
| GCT_NOPOINTER | object | M, C | The object does not have any pointers in it.<br>Do nothing |
| GCT_POINTERLIST | object | M, C | <br>Keep objects in the list alive. |
| GCT_METHODTABLE | object<br>object → u.java.code<br>object → u.java.handlers | M, C | For each method in the table,<br>mark the object (code) and<br>mark the object (handlers) if it is non null. |

compressed by traditional optimization techniques (compiler and programming language optimization [22] [23] [24] [25] and/or file-to-file de/compression techniques) and are statically stored in compacted/compressed ROM (read-only memory). This involves a special Java interpretor with runtime decompression. The cost is high, but the small memory consumption achieved leads to another benefit in battery-powered small handsets. Offline optimizations used with runtime memory bank partitioning will achieve low power consumption and reduce the number of active memory banks. The unused memory banks are powered off to eliminate their leakage current. Chen *et al.* [4] reported a 20% power consumption reduction using memory bank partitioning, runtime hardware decompression and offline compression for the ROM.

We introduce a runtime compression for the RAM with the memory bank partitioning technique. The RAM is used as the heap space for Java objects. The behavior of the heap space is not static, which is different from that of the permanent memory space and thus cannot be optimized by traditional off-line optimization techniques described above.

Instead, new methods must be developed for optimizing the memory footprint of the Java heap (memory demand) and the power consumption. Fortunately, memory/cache compression methods share similar characteristics to Java heap. Memory/cache compression techniques compress a compressed memory to store instructions and data, which are dynamic and done at software level, hardware level or a combination. We develop a Java heap compression technique that introduces a virtual compressed heap and de/compresses a group of objects that are created and accessed at runtime. The difficulty comes from the

dynamic behavior of objects and automatic garbage collection at runtime. We do not know anything about objects and their behavior until an application is actually executed and objects are instantiated. How to capture them and how to feed them back to efficient memory management is a key issue.

We have discussed the compressed ROM space and the compressed RAM (heap) space and have seen their benefits in *memory saving* and *power consumption*. We believe that the Java heap compression is a new direction of object-oriented languages for small battery-powered mobile/wireless embedded devices.

## Hardware/Software Codesigned Java Runtime Environment

The in-memory compression for the object-oriented language Java has been dis-

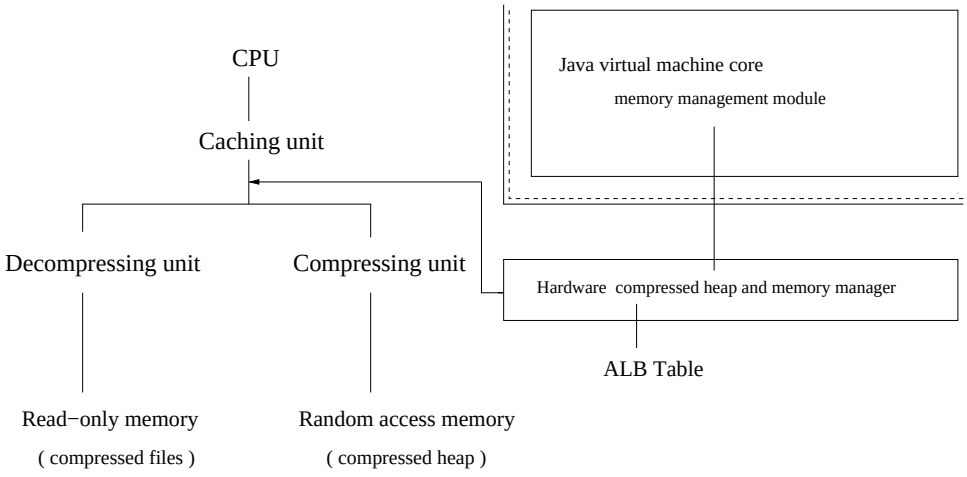



Figure 9: Hardware/software Codesigned Architecture for Java Runtime Environment

cussed in the previous sections. In this section, we propose a hardware/software codesigned architecture for Java Runtime Environment. Figure 9 shows an overview of hardware/software codesigned architecture with the compression and decompression units. The architecture consists of a hardware part and a software part. The hardware part includes a CPU, a caching unit, a compressing unit, a decompressing unit, a read only memory (ROM) that holds compressed files, a random access memory (RAM) that constructs a compressed

heap, an Address Lookaside Buffer (ALB) table, and a hardware memory manager. The software part consists of a Java Virtual Machines (VMs), a software memory manager that has an interface to Java VMs and an interface to the hardware memory manager. Figure 10

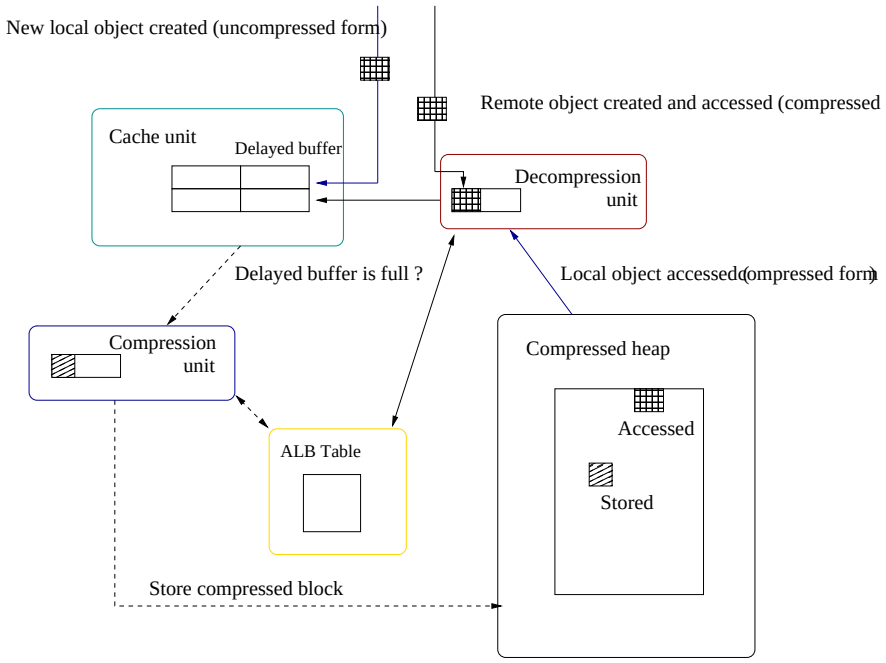



Figure 10: Details of the Proposed Architecture

illustrates details of the architecture. The compression unit includes a compression buffer. The decompression unit consists of a decompression buffer. The caching unit is composed of buffers (one works as a delayed cache buffer to accumulate objects (a page size)), and the ALB table holds page information. The architecture uses a different address mapping scheme from the original one. In the address mapping scheme (Figure 11), an object is represented by an object identifier that consists of page-number bits, and offset bits in the page (user defined optimization bits are provided as option). Given an object request, the software memory management module extracts the page-number bits from the object identifier and gets the page number. The hardware memory manager finds the address of the page from the ALB table (page header includes size bits) and loads the page (the size of the compressed page is found from the ALB table as well. The software memory manager then

uses the offset bits and determines an object boundary in the page with the object size in the object header.

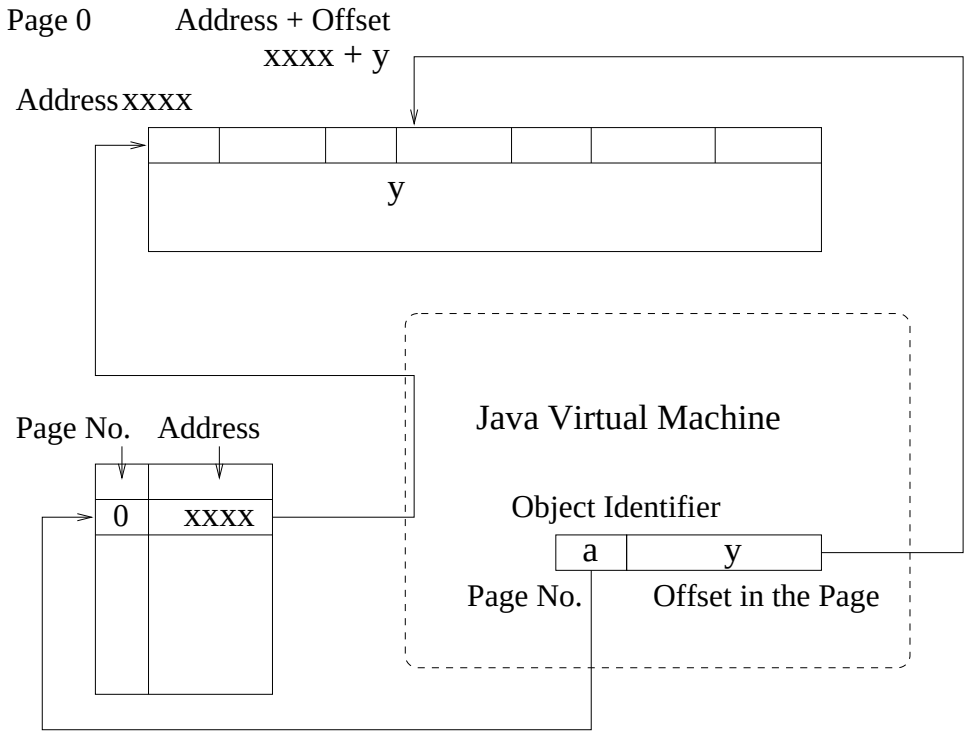



Figure 11: Address Mapping

De/compression techniques are used in combination with the Java memory management system. The memory management module calls the de/compression unit upon object creation and object accesses as required. We will discuss how the compression technique is applied to Java computation mechanism.

Case 1: when the delayed buffer is not full, the memory management module sends the object directly to it.

Case 2: when the delayed buffer is full and some of the caching buffers are not used yet, one of the unused buffer is set as the delayed buffer. The memory management module sends the object to the delayed buffer and updates the ALB table based on the move-to-front strategy.

Case 3: when the delayed buffer and all the caching buffers are full. In the first

cycle, the caching unit sends the page in the delayed buffer as a victim to the compression unit. In the second cycle, the memory management module sends the object directly to the delayed buffer and updates the ALB table based on the move-to-front strategy. In the same cycle, the compression unit compresses the victim page. In the third cycle, the compression unit stores the compressed page in the compressed heap and updates the ALB table.

In all cases, the object in the caching unit is forwarded to the Java core when it is requested. Operations for remote objects are handled in a similar manner to the operations described above, except that a remote object is decompressed on the fly at the local decompression unit and sent to the delayed buffer in the caching unit.

## Hardware Compression and Decompression

Hardware compression and decompression occur when a Java Virtual Machine (a

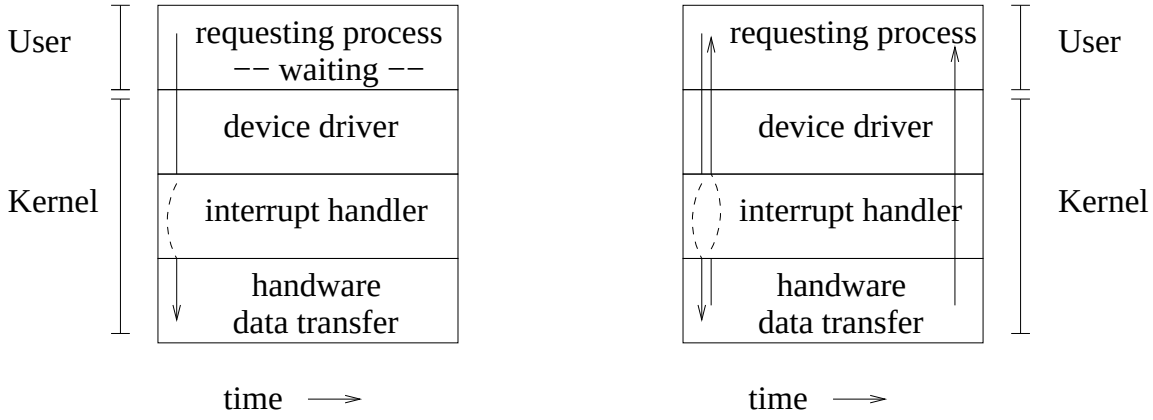
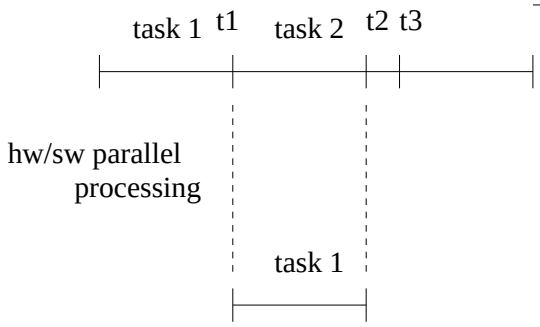



Figure 12: Example of I/O Interruption

user process) requests de/compression and works in a similar way to I/O interruption (Figure 12). The left side of Figure 12 shows a synchronous I/O operation. When there is an I/O request, a user process will wait until data come from the hardware. The right side of the figure illustrates an asynchronous I/O operation. When there is an I/O request, a

user process is interrupted but resumed without waiting the I/O request. Similar to the I/O interruption, two courses of action are possible for the de/compression. In the synchronous de/compression, the Java Virtual Machine stops its operation and waits for an object. Asynchronous de/compression returns control to the Java Virtual Machine after a request without waiting for the de/compressor to complete. The de/compression is executed along with other operations. Asynchronous compression is always possible, but sometimes asynchronous decompression is not impossible if an object is needed on subsequent operations. Decompression must be synchronous. Figures 13 and 14 show examples of asynchronous compression and synchronous decompression, respectively.




| Time | Task | Operation |
|---|---|---|
| t1 | task 1 | migrates into HW |
| | task 2 | starts |
| t2 | task 2 | interrupted |
| | task 1 | migrates from HW into SW |
| | | write compressed data into RAM |
| | task 1 | passes compressed data from HW |
| | | to the memory management system |
| | | and write it into RAM |
| t3 | task 2 | resumes |

Figure 13: Asynchronous Compression

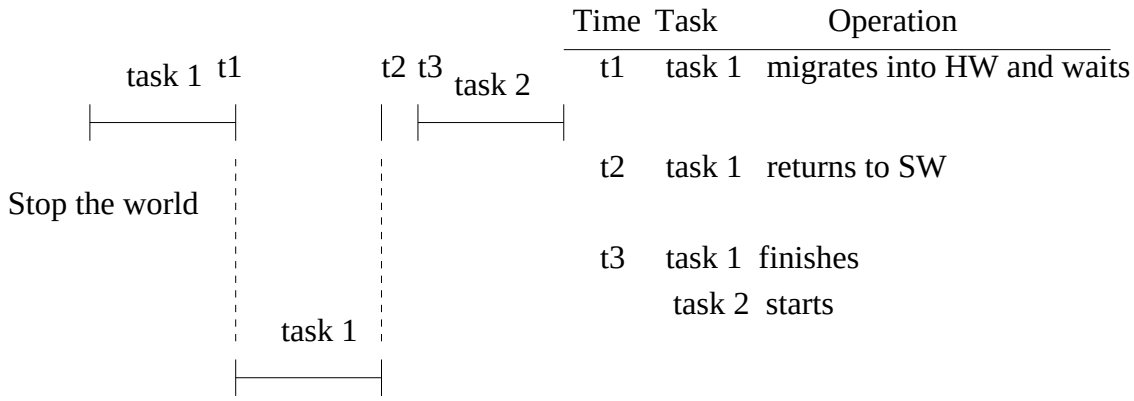



Figure 14: Synchronous Decompression

## Power and Energy Consumption

Power and energy is critical issues in battery-powered, handheld devices and has been estimated in several ways [36][37][38][26]. In this section, we discuss energy estimation on System-on-Chip (SoC) [37][38] and power estimation with Xilinx Xpower [26].

## Energy Estimation on System-on-Chip (SoC)

We design an SoC memory architecture with a compressor and a decompressor as shown in Figure 15. Energy is estimated using parameters and values presented in [4]. Simulation results are reported with a software compressor, a software decompressor, and the heap memory image dumped from the Java runtime environment.

Experiments for energy consumption are conducted using manyballs (a Java game application downloaded from the Internet). Table 17 shows an energy consumption reduction with the X-match algorithm. The configuration with compression consumes less energy than the original one.

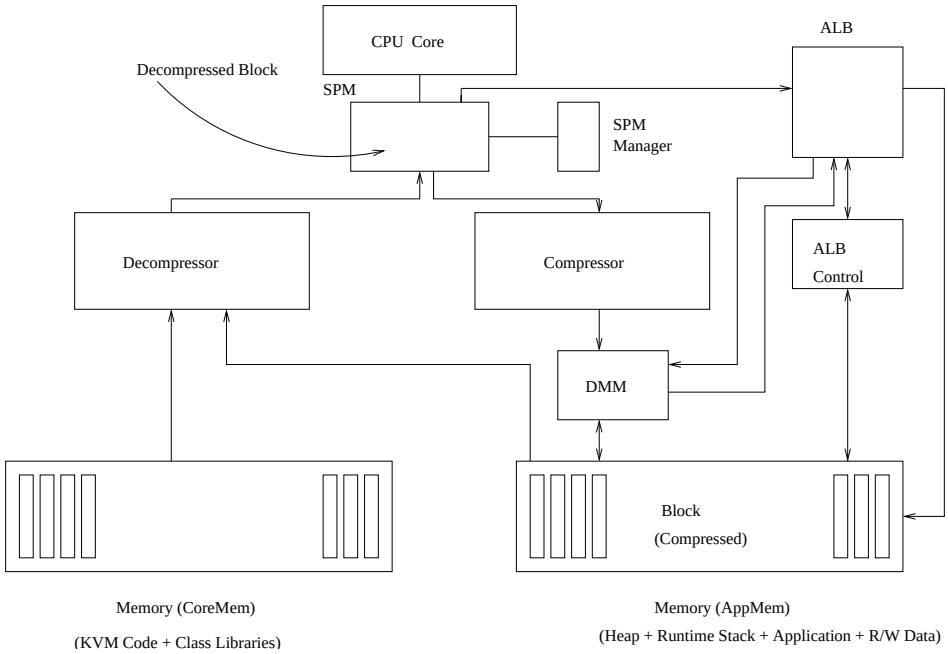



Figure 15: SoC Architecture

Table 17: Energy Compression Reduction with X-Match Algorithm

| Application | $E_{RAM}$ | $E_{RAM_{de/compression\_module}}$ | Total |
|---|---|---|---|
| **manyballs** (w/ compression) | 5.8 mJ | 1.1 mJ | 6.9 mJ |
| **manyballs** (w/o compression) | 31.45 mJ | - | 31.45 mJ |

## Power Estimation with Xilinx XPower

Xilinx provides an ISE simulator for hardware (including FPGA) design, which includes a power estimation capability. XPower Software is a utility for estimating the power consumption and junction temperature of FPGA and CPLD devices:

1. XPower Software reads an implemented design (NCD file) and timing constraint data
2. The user supplies activity rates:
   (a) Clock Frequencies
   (b) Activity rates for nets, logic elements, and output pins
   (c) Capacitive loading on output pins
   (d) Power supply data and ambient temperature
   (e) Detailed design activity data from simulator (VCD file)
3. XPower calculates total average power consumption and generates a report.

XPower calculates power as a summation of the power consumed by each switching element in the design. The power consumed by each switching element in the design is given by

$$P = C * V^2 * E * F * 1000 \tag{1}$$

where P is power in mW, C is capacitance in farads, V is volts, E is switching activity (average number of transitions per clock cycle), and F is frequency in Hz.

Capacitance is determined by the users. Voltage is a fixed value for a specific device set by default in the XPower interface. E * F is the total number of transitions for a specific element. Frequency, or the activity rate of each signal in a design, is the most variable element of the above equation for the XPower tool. The switching activity is defined by the user based on the global default activity rate, the simulation results generated through a VCD file, and activity rates manually entered through the GUI.

A compressor and a decompressor are implemented using *Verilog* and the power is estimated using the XPower with activity rates and the implemented code as its input.

## Modeling for Performance

The space efficiency of a configuration is defined as

$$\textbf{SpaceEfficiency} = \frac{W}{W_{after}} \tag{2}$$

where $W$ is the watermark (the minimum amount of memory for running an application) on the base configuration and $W_{after}$ is the watermark on the proposed configuration.

The speedup of a configuration is defined as

$$\textbf{Speedup} = \frac{T}{T_{after}} \tag{3}$$

where $T$ is the total execution time (including garbage collection time) on the base configuration, and $T_{after}$ is the total execution time (including de/compression times and/or garbage collection times) of each application on the proposed configuration.

$T_{after}$ is calculated by

$$T_{after} = Cost_{base}[+Cost_{gc}] + Cost_{de/comp} \tag{4}$$

$$Cost_{de/comp} = \sum Cost_{case1} + \sum Cost_{case2} \tag{5}$$

Cases 1 and 2 are related to an object access that is not in the cache. Case 1 is defined a situation where the caching unit has not been full and a page containing the object to be accessed can be loaded into the caching unit. One decompression is invoked. Case 2 is defined as a situation where the caching unit is already full. One of the cache blocks (pages) is

selected and compressed. Then, the compressed page is stored. One compression is invoked. Sequentially, a page containing the accessed object is fetched from the compressed heap into the decompressing buffer, and is decompressed. One decompression is invoked.

The power is calculated using equation (1) or the following equation

$$\mathbf{P} = \mathbf{I} \times \mathbf{V} \tag{6}$$

where I is current in mA, and V is voltage in volts.

The power consumption saved is calculated by subtracting power consumed on the proposed configuration from power consumed in the base configuration.

The power efficiency is defined as

$$\mathbf{PowerEfficiency} = \frac{\mathbf{P}}{\mathbf{P_{after}}} \tag{7}$$

where P is power in mW on the base configuration and $P_{after}$ is power in mW on the proposed configuration.

The energy is estimated by

$$\mathbf{E} = \mathbf{P} \times \mathbf{S} \tag{8}$$

where E is energy in mJ, P is power in mW, and S is time in seconds.

The energy consumption saved is calculated by subtracting energy consumed on the proposed configuration from energy consumed in the base configuration.

The energy efficiency is defined as

$$\mathbf{EnergyEfficiency} = \frac{\mathbf{E}}{\mathbf{E_{after}}} \tag{9}$$

where E is energy in mJ on the base configuration and $E_{after}$ is energy in mJ on the proposed configuration.

A performance model is designed for system and code analyses in battery-powered, resource-limited embedded device. Performance is evaluated using the system and code analyses to be described in CHAPTER 3, CHAPTER 4, and CHAPTER 5.

# CHAPTER 3: Memory Compression without Automatic Memory Management

## Configuration I

The garbage collection (GC) increases free memory space by collecting unused objects in memory and compacting live objects. Compression techniques also work to reduce a program's memory footprint, but are faster than GC. We introduce Java heap compression as an alternative to garbage collecting and propose a Java compressed heap based on the hardware/software codesigned architecture discussed in CHAPTER 2 (hereinafter, it is referred to as configuration I). Configuration I with hardware de/compressor is referred to as configuration $I^h$,

## In-memory Compression Algorithms

Compression techniques are classified into two groups: lossless compression and lossy compression. Lossless compression includes Huffman coding, arithmetic coding, predication, dictionary-based compression, vector quantization, differential encoding, and transform coding. In practice [27] [28], gzip (Lempel-Ziv coding LZ77), Zip (a combination of the UNIX commands tar and compress that is compatible with Phil Katz's ZIP for MSDOS systems) and compress (adaptive Lempel-Ziv coding) are well-known as lossless compression algorithms implemented in software libraries to compress files on Unix, Linux, and Windows computer systems. Hardware implementations include adaptive lossless data compression ALDC (LZ1 derivation by IBM), data compression Lempel Ziv DCLZ (LZ2 derivation by Hewlett packer and AHA), Lempel Ziv stac. LZS (LZ1 derivation by STAC/Hifn), and X-match (fixed-width dictionary-based algorithms developed by Loughborough University). Recently, small handsets have become popular and require in-cache/in-memory compression because of their limited resources, especially, in the battery. The most popular compression algorithms (Lempel-Ziv (LZ) family) are, however, designed for human readable text

and are not suitable for data in memory/cache because of their regularity patterns . In-memory/cache data consist of word-aligned integers and pointers, containing many repeating zero values. In general, the majority of its information is found in its low bits. The X-match [11][12][13], zero-removal [29], and Wilson and Kaplan (WK) [30][31] algorithms have been designed to capture these features and are used for in-memory/in-cache compression. X-match is implemented in hardware and others are usually implemented in software. We briefly compare the three algorithms.

The X-match algorithm uses a fixed-width dictionary of previously seen data that consist of four entries: match, address, match type and literals. The algorithm attempts to match or partially match the current data element with an entry in the dictionary. The hardware implementation of X-match achieves 0.5 compression ratio [11][12][13].

Zero-removal compression [29] encodes all zero words with a single "0" bit, and other words are encoded with a "1" bit and literal bits. Typically, the zero-removal algorithm has a 0.108 compression ratio for Java heap [4].

The WK algorithm is similar to X-match but differs in that its coding specification consists of match type, dictionary index, 10 low bits, and 22 upper bits. The match type is always included in the compressed form and the remaining is optional. Software-simulated WK algorithms achieve a compression ratio of 0.5.

Both the X-match and the WK algorithms achieve compression ratio of about 0.5. Recall, however, that the memory/cache data have many repeating zero values. The WK algorithm has special encoding type called "ZERO". If the input word is zero, it is encoded with a ZERO tag (2 bits). The size of the encoded result is much smaller than that of the corresponding encoding for X-match because the X-match needs at least ( match + address + match type ) to encode ZERO [1]. We focus on the WK algorithm and its hardware implementation.

[1] [+ literal bits] will be zero if there is no zero value in the dictionary

## Wilson and Kaplan Algorithm

The WK algorithm is based on a dictionary, compares input symbols to recently seen symbols stored in the dictionary, and models the similarity of values that are spatially local. The compression operation has two phases:

Table 18: Coding for WK Algorithm

| Match type | Coding specification |
|---|---|
| ZERO | <0x0> |
| EXACT | <0x1> < 4 bits dictionary index > |
| PARTIAL | <0x2> < 4 bits dictionary index > < 10 low bits > |
| MISS (NO Match) | <0x3>< 32 bits literal pattern > |

1. Packing phase: the majority of information is contained in low bits of input words, which are compacted as much as possible by extracting and combining their information parts only. For example, if information is stored in the lower eight bits, each lower eight bits of four words can be combined into a new single word by a set of shift and bitwise XOR operation.

2. Compression phase: each packed word is compared against values in the dictionary, and all information (tag and other bits) is assembled into a compressed form. Coding varies on match types as is illustrated in Table 18.

The basic decompression operation is to extract a tag from the compressed input and decompress the whole input using the tag information and the coding specification (Table 18). Figure 16 shows how the WK algorithm works.

## Hardware In-memory Compression Algorithms

We modify the Wilson and Kaplan (WK) algorithm to meet hardware requirements (input and output ports, bus width, etc.) and implement it in Verilog and C language. Figures 17 and 18 show diagrams of the hardware compressor and decompressor.

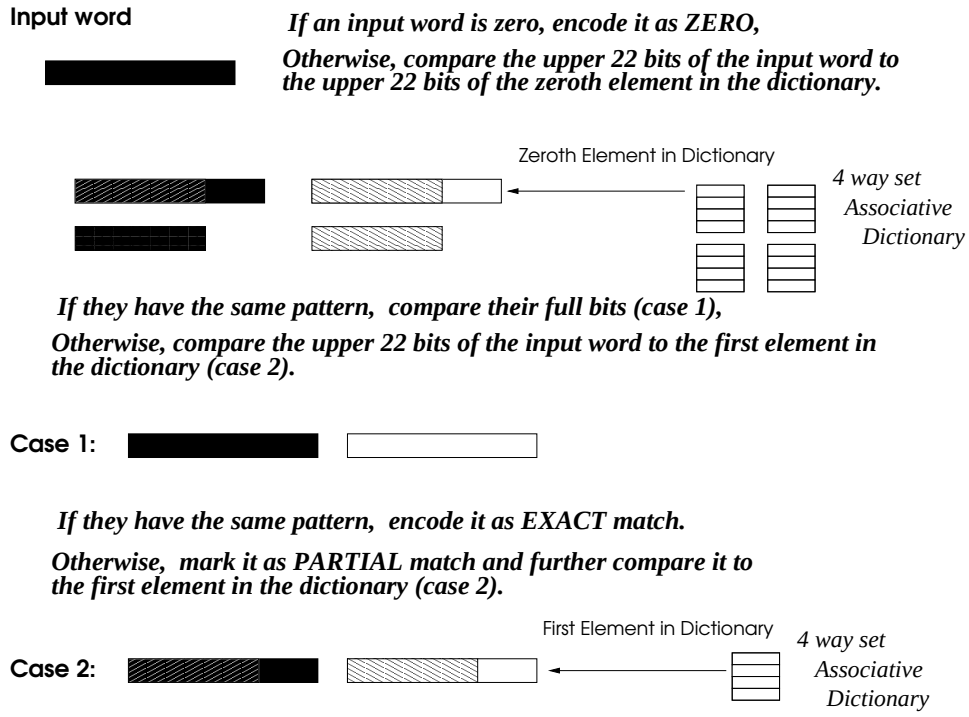



Figure 16: Operation Example of the WK Algorithm [40]

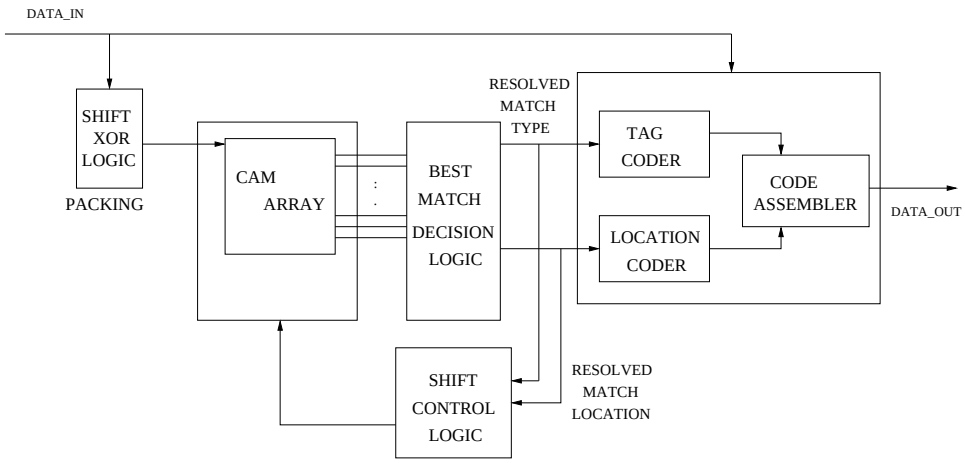



Figure 17: WK Compressor

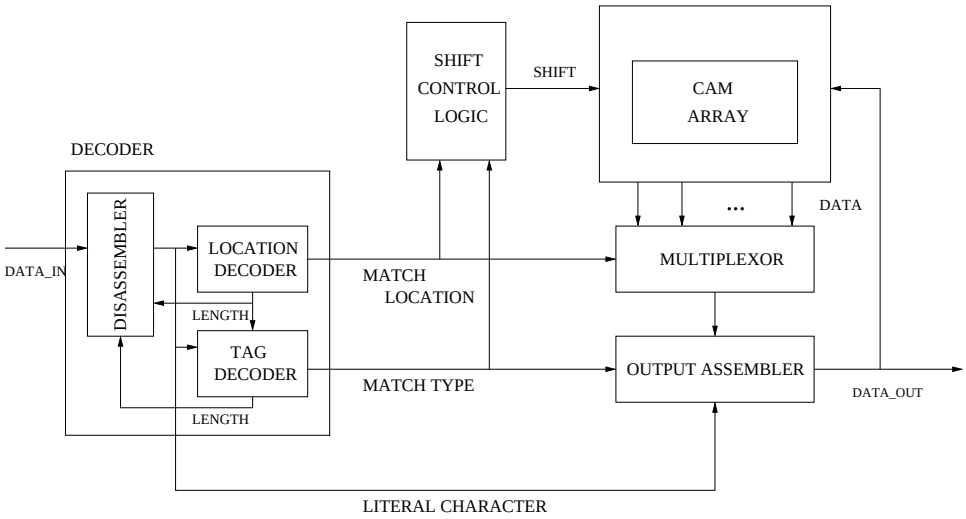



Figure 18: WK Decompressor

## Experiments

We examine compression techniques over mobile/wireless devices and show their impact on space efficiency, execution time, and power/energy consumption. The system is designed using a client and server architecture. The Java 2 Platform, Micro Edition (J2ME) is used as a base configuration. The heap size is 500,000 bytes (maximum). The page size is 4 KB by default, and the cache size is 64 KB. The compaction mode is on. We implement a modified Wilson-Kaplan (WK) algorithm in software and hardware. The hardware compressor and decompressor are simulated with an Xilinx ISE simulator on a Virtex IIpro device. The compressed heap is simulated on the Solaris 9 platform (SunOS 5.9).

Table 19 lists the benchmark applications selected to evaluate the implementation. In experiments, time and power are measured through hardware simulation. The software simulator reports the number of Garbage Collection (GC) cycles, the execution time of GC, the number of cache misses (that are related to the number of compression and decompression operations), and the execution time of de/compression [2]. Memory is composed of 16 1-Mbit Random Access Memory (RAM) modules with 39 mW in power, 20 mA in current, and 1.95 V in voltage. A heap is mapped on the RAM. The space efficiency, speedup, and power/energy consumption and efficiency saved are calculated using Eq. (1) to Eq. (9) described in CHAPTER 2.

Table 19: Benchmarks

| MIDlet suite | Description |
|---|---|
| **demos(HTTP)** | Technical demonstration program of http for MIDP ( comes with Sun Java KVM ). |
| **stock** | Get stock quotes from a publicly available website, and set alerts ( comes with Sun Java KVM ). |
| **audiodemo** | Listen to sounds using Mobile Media APIs audio building block ( comes with Sun Java KVM ). |
| **WebViewer** | A fast HTML Web browser for Java-enabled (J2ME) mobile devices. ( www.reqwireless.com ) |

[2]Each object is traced when it is created and accessed

## Results

Table 20: Space Efficiency, Speedup, and Power (Configuration I)

| Application | Space efficiency | | Speedup | Power saved in heap (mW) | Power efficiency (heap) |
|---|---|---|---|---|---|
| | application | all | | | |
| **demo(HTTP)** | 1.81 | 0.984 | 1.118 | 0.548 | 1.82 |
| **stock** | 1.66 | 0.799 | 1.125 | 0.488 | 1.67 |
| **audiodemo** | 2.13 | 0.885 | 0.941 | 0.646 | 2.13 |
| **WebViewer** | 1.23 | 1.389 | 0.808 | 0.926 | 1.23 |

Table 21: Energy Saved in Configuration I and Energy Overhead

| Application | Energy saved (mJ) | SW de/comp energy (mJ) | HW de/comp energy (mJ) | Energy efficiency (heap) |
|---|---|---|---|---|
| demo (HTTP) *64KB heap* | 442 | 0.554 | 0.008 | 1.68 |
| stock *64KB heap* | 678 | 2.233 | 0.074 | 1.55 |
| audiodemo *64KB heap* | 545 | 0.792 | 0.010 | 2.29 |
| WebViewer *256KB heap* | 1245 | 42.970 | 0.344 | 1.32 |

The configurations have been evaluated using benchmark applications (*HTTP demo*, *stock*, *audiodemo*, and *WebViewer*). Table 20 summarizes experimental results. *HTTP demo* and *stock* have good speedups (1.118, 1.125) and good memory saving (45%, 40% in the application part, 1.6%, 20% of memory overhead for the user interface which allows users to select an application). *audiodemo* is bad in speedup (0.941), but good in space (53% of memory saved in the application part, 12% of memory overhead for the user interface). *WebViewer* has poor performance in both speedup (0.808) and space efficiency (19% of memory saved in the application part, 28% of memory saved in the user interface). We suspect that configuration I may not work well for web applications that have large memory demands.

The targeted devices of our research Java-enabled, battery-powered, handheld devices. We further analyze configuration I using power and energy. Results are also shown in Table 20. Power efficency is very similar to the space efficiency on the heap, but the amount of power saved in the heap seems to be proportional to the heap size. *WebViewer* has the best value for power saved in the heap. This indicates that configuration I work well for

web appliations that have large memory demands. We further examine configuration I using energy. Table 21 shows minimum energy saving on the heap and energy overhead because of a software de/compression and hardware de/compression. The energy efficiency is very similar to the space and power efficiencies, but the amount of energy seems to be proportional to the heap size. *demo*($HTTP$) and *stock* have good energy efficiency (1.68, 1.55) and good energy saving (442 mJ, 678 mJ). *audiodemo* has very good energy efficiency (2.29), but only has very small amount of energy saving (545 mJ) which is similar to *demo*($HTTP$) and *stock*. *WebViewer* has poor energy efficiency (1.32), but has the best energy saving (1245 mJ). The energy overhead is very small accross all benchmark applications, especially with the hardware de/compression, and is neglibile. We conclude that configuration I can save energy for application programs that require a large heap. The conclusion suggests that configuration I will further reduce memory demand in combination with garbage collection for the compressed heap.

# CHAPTER 4: Memory Compression with Automatic Memory Management System

## Configuration II

In Configuration II, Java virtual machines (with heap compression and garbage collection) will be studied. The configuration I, presented in CHAPTER 3, is modified to have the benefit of the garbage collection and to maximize memory and power saving. The configuration II with hardware compression and decompression is referred to as configuration $II^h$. The proposed architecture is similar to configurations I and $I^h$, but garbage collection is activated in the Java memory management system. We will discuss in-memory compression, object access, object creation, and garbage collection.

## Object Creation and Object Access

When a new object is created, its object identifier and the object identifiers of the object reference holder/holders inside the object are encoded and added to a list of object identifiers for the current page. The ALB table keeps the object identifiers per page. When an object is requested, its corresponding object identifier is decoded. In Figure 19, the object identifier is $a\ y\ z\ t$ where $a$ indicates a page number, $y$ is the offset in the page, $z$ is a mark bit, and $t$ is type bits. The memory manager looks up the ALB table to get the memory address and decodes the position of the object by adding the offset $y$ to the memory address (xxxx). The object size is found from the object header (the first word) from the location.

## Memory Compression and Garbage Collection

Chen *et al.* [32] have proposed a garbage collection strategy for Java compressed heap that compresses each object (Figure 20) but leaves object headers uncompressed. The problem with this approach is that it achieves low compression ratio and high time overhead. To have better compression ratio and high memory and power saving, we introduce a page-

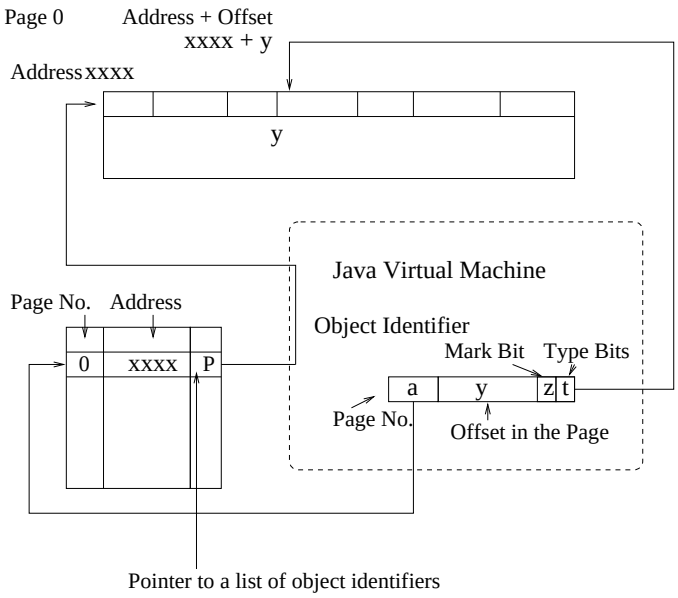



Figure 19: Address Mapping for Compressed Heap with Garbage Collection

based compression scheme that uses a page of objects as a basic unit (Figure 21). The difficulty with this approach is that the mark and compact phases must traverse compressed headers of objects (Figure 22). We examine current implementation details of the mark-sweep algorithm with compaction and propose solutions to the problem. The solution is implemented as configurations II and II$^h$.

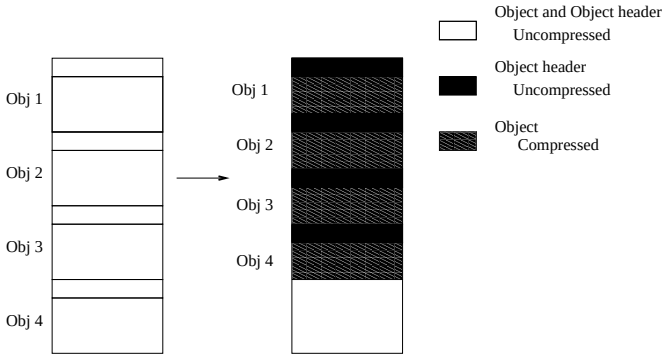



Figure 20: Uncompressed Header

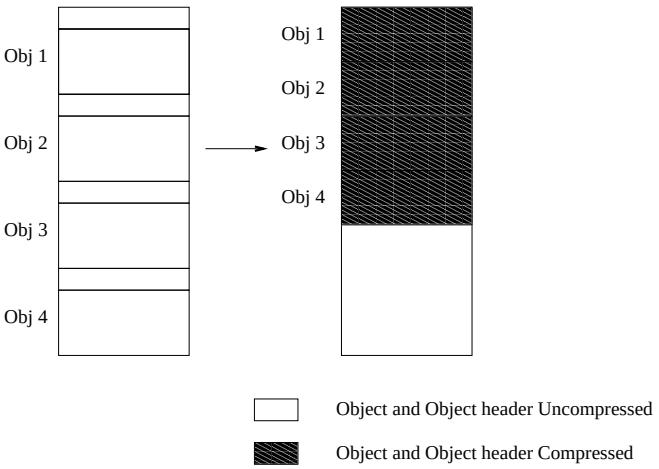



Figure 21: Compressed Header

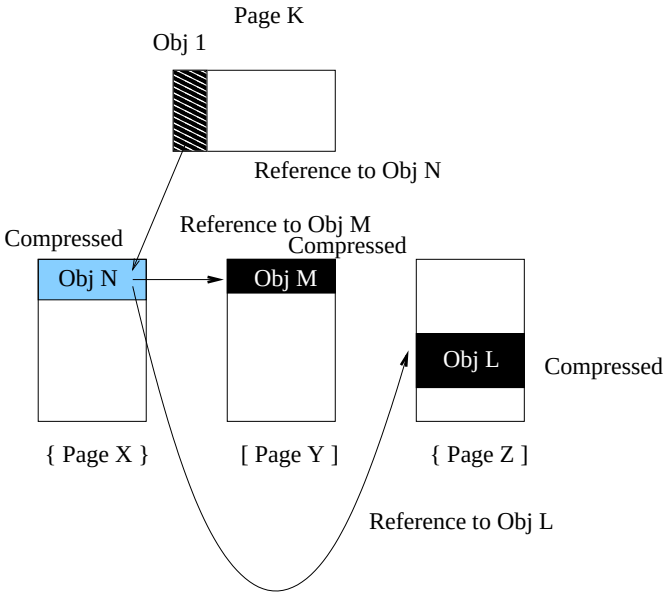



Figure 22: Difficulty in Page-based Compression

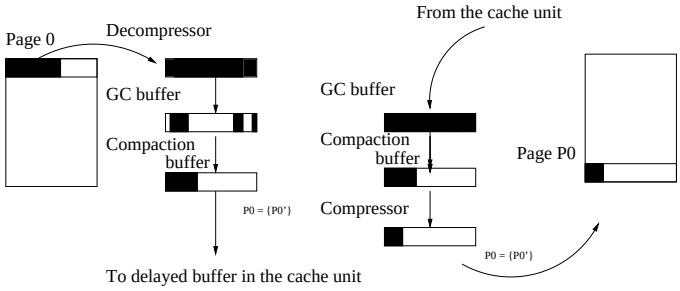



Figure 23: Example of De/Compression Steps in Garbage Collection for Compressed Java Heap

## Java Runtime Environment for Small Memory Footprint and Low Power Consumption

We redesigned garbage collection for the compressed heap. The redesigned garbage collector is a modified page-based mark and sweep algorithm with compaction and uses the de/compression units that have a garbage collection (GC) buffer and a compaction buffer. Garbage collection information (mark and type bits) is added to the user defined optimization bits in the object identifier, and a list of the object identifiers is added to the Address Lookaside Buffer (ALB) table per page. Each object identifier in the ALB table has object reference holder(s). In our design and implementation, the object reference holder(s) are also object identifiers. The mark phase is similar to the original GC except that mark bits are stored in the object identifiers (Figure 19). The sweep and compaction phases are migrated into de/compression modules, and delayed until an object is accessed and/or a cache miss occurs. Figure 23 shows how the proposed garbage collector works.

## Implementation Details

In this section, we discuss implementation details that include major data structures, the basic operations, object creation, object access, and the garbage collection algorithm with Java memory compression (mark, sweep and compaction phases). The typical mark-sweep algorithm handles objects based on object types (GCT _FREE, GCT _NOPOINTERS,

GCT _INSTANCE, GCT _ARRAY, GCT _OBJECTARRAY, GCT _METHODTABLE, GCT _POINTERLIST, GCT _EXECSTACK, GCT _THREAD, GCT _MONITOR, and GCT _WEAKPOINTERLIST). When the garbage collector is called, the mark phase checks the type of a given object. In this stage, the garbage collector knows from its specification if there are some object reference holders inside the given object. For example, if the object type is GCT_INSTANCE, there is three object reference holders inside the object. Table 22 shows an example of its implementation. In general, we can access objects of the object reference holders using object → ofClass [→ fieldTable →u.offset], object → data[offset].cellp, and object →superClass.

Table 22: Instance Data Structure

```
    struct instanceStruct
1.  COMMON_OBJECT_INFO(INSTANCE_CLASS)
2.  union {
3.      cell *cellp;
4.      cell cell;
5.  } data[1];
```

```
    #define COMMON_OBJECT_INFOR(_type_)
    _type_ ofClass;
    monitorOrHashCode mhc;


    class.h
```

The method that a garbage collector uses to access object reference holders inside a given object is a key to the garbage collection algorithm for a compressed heap. In our implementation, we duplicate the object reference holders inside the given object and store them in the Address Lookaside Buffer (ALB) table along with the corresponding object identifier shown in Figure 19.

In the sweep and compaction phases, the garbage collector finds a list of identifiers for a given page in the ALB table, then sweeps garbage objects using the mark bit informa-

tion (sweep phase) and relocates live objects toward the top of the page as well as objects reachable from the live objects. In the mark phase, the garbage collection updates mark bits in their corresponding object identifiers in the ALB table without touching the pages and maintains object liveness.

## Major Data Structures

There are five major data structures; an ALB table, an object identifier, a caching unit that includes buffers and their headers, a compressor, and a decompressor. Table 23 shows an ALB table data structure, and Table 24 presents a caching unit data structure. The object identifier is a simple word that consists of page bits, offset bits, a mark bit and type bits as discussed previously. The compressor and decompressor are software functions or hardware functions that are mapped on the reconfigurable hardware. For the memory management, a free list data structure is also added (Table 25).

Table 23: Address Lookaside Buffer Table Structure

```
1.  struct ALBunit

2.  ALBstruct ALBentry[PageSize+1];
```

```
1.  typedef struct

2.  unsigned page_number:20
3.  unsigned long* page_addr
4.  unsigned long* objectIdentifierP

5.  ALBstruct;
```

Table 24: Caching Unit Structure

```
1.   struct bufferHeader_

2.   unsigned page_number:20
3.   unsigned long headOfFreeList
4.   unsigned freeSIZE:16
5.   unsigned largest_size:16
6.   unsigned long *buffer;
```

Table 25: FreeListStruct

```
1.   typedef struct FreeListStruct

2.        struct FreeListStruct_ *pre
3.        struct FreeListStruct_ *next
4.        unsigned status:16
5.        unsigned size:16
```

Table 26: bufferHeader

```
1.   struct bufferHeader_

2.        // keeps the corresponding page in compressed heap
3.        unsigned page_number:20
4.        // keeps the head of the linked list of free blocks
5.        unsigned long headOfFreeLinkList
6.        // keeps the largest size of free blocks in the buffer
7.        unsigned free_size:16
8.        // keeps the largest size of free blocks
9.        unsigned largest_size:16
10.       // points to actual buffer which is OK
11.       unsigned long *buffer
```

## Basic Operations

The basic operations consist of JVM initialization and finalization, heap initialization (Table 27), heap finalization (Table 28), ALB initialization (Table 31), ALB finalization (Table 32), caching-unit initialization (Table 29), caching-unit finalization (Table 30), object creation (Tables 33, 34), object access (Table 35), and de/compressor initialization and finalization. The Java VM initialization and finalization are similar to the original one, but the ALB table and the caching-unit need to be initialized additionally. The heap initialization and finalization are quite different from the original one. Heap space is not allocated until all buffers in a caching unit are filled up. So, the heap is created in object creation (ALLOCATE_HEAP_OBJECT 33) once the caching unit is full. The software compressor initialization is simply a function call for the software compressor in the software JVM memory manager, and the software decompressor initialization is a function call for the software decompressor. Similarly, the software de/compressor finalization corresponds to the end of the operations in the software de/compressor. The hardware de/compressor initialization reconfigures the hardware de/compressor. The hardware de/compressor finalization releases the hardware de/compressor.

## Object Creation

An object is created using a MALLOC_HEAP_OBJECT function. Given the size and type, the function calls ALLOCATE_FREE_CHUNK. Once the memory is allocated, the type is put into the object header. Then, an object identifier is encoded for the object and is returned.

In the ALLOCATE_FREE_CHUNK (Table 34), which is not similar to the original one, the object is allocated in one of the buffers in the caching unit. Then, the size is set in its object header, and the entry of the ALB table, which corresponds to a page that includes the object, is updated.

Table 27: INITIALIZE_HEAP

```
    INITIALIZE_HEAP ()

1.  ALLOCATEBUFFER(sizeOfBuffer, numberOfBuffer)
2.  // corresponding to HeapStart
3.  BufferStart ← Buffer[0].buffer
4.  // corresponding to CurrentHeap
5.  CurrentBuffer ← BufferStart
6.  // corresponding to HeapEnd
7.  BufferEnd ← PTR_OFFSET(BufferStart, sizeOfBuffer)
8.  // corresponding to CurrentHeapEnd
9.  CurrentBufferEnd ← BufferEnd
10. INITIALIZEALB ()
11. bufferHeader ← (BUFFERHEADER)CurrentBuffer
12. bufferHeader→page_number ← 1
13. bufferHeader→free_size ← sizeOfBuffer
14. bufferHeader→largest_size ← sizeOfBuffer
```

Table 28: FINALIZE_HEAP

```
    void FinalizeHeap(void)

1.  freeHeap(TheHeap)
```

Table 29: ALLOCATE_BUFFER

```
ALLOCATE_BUFFER (sizeOfBuffer, numberOfBuffer)

1.  Buffer ← MALLOC(sizeOfBuffer * numberOfBuffer)
2.  for i ← 0 to numOfBuffer - 1
3.        do Buffer[i].buffer ← MALLOC( ceil(sizeOfBuffer / numberOfBuffer) )
```

Table 30: DEALLOCATE_BUFFER

```
void DEALLOCATE_BUFFER

1.  freeBuffer(TheBuffer)
```

Table 31: INITIALIZE_ALB

```
INITIALIZE_ALB ()

1.  ALB ← MALLOC(sizeOfALB)
2.  // 0 in Buffer, 1 in Heap
3.  ALBtable→ALBentry[1].page_status ← 0
4.  ALBtable→ALBentry[1].buffer_addr ← 1
5.  ALBtable→ALBentry[1].comp_addr ← Buffer[0].buffer
6.  ALBtable→comp_size ← 0
```

Table 32: FINALIZE_ALB

| | |
|---|---|
| 1. | ALBstruct **ALBtable |
| | FINALIZE_ALB |
| 1. | freeALB(ALBtable) |

Table 33: MALLOC_HEAP_OBJECT

| | |
|---|---|
| | MALLOC_HEAP_OBJECT (size, type) |
| 1. | thisChunk ← ALLOCATE_FREE_CHUNK (size) |
| 2. | *thisChunk ← *thisChunk \| ( type << TYPE_SHIFT ) |
| 3. | *thisChunk ← *thisChunk \| (size - HEADERSIZE ) << TYPEBITS |
| 4. | objectIdentifier ← CREATE_OBJECT_IDENTIFIER(thisChunk + HEADERSIZE, offset, size) |
| 5. | return objectIdentifier |

Table 34: ALLOCATE_FREE_CHUNK

```
    ALLOCATE_FREE_CHUNK ( size )

1.  if bufferHeader→free_size - size < 0
2.      currentBufferIndex ← ( currentBufferIndex + 1 ) mod numOfBuffer
3.      bufferHeader ← Buffer + CurrentBufferIndex
4.      bufferHeader→free_size ← sizeOfbuffer / numberOfBuffer
5.      updateALBtable()
6.  overhead ← bufferHeader→free_size + HEADERSIZE - size
7.  bufferHeader→free_size ← overhead - HEADERSIZE
8.  dataArea ← Buffer[currentALBIndex-1) mod numOfBuffer].buffer + overhead
9.  *dataArea ← ( size - HEADERSIZE ) << TYPEBITS
```

## Object Access

An object is represented by an object identifier inside the Java Virtual Machine (VM). When an object is accessed, its object identifier is decoded into a page number, an offset in the page, a mark bit, and a type. The address of a page indicated by the page number is looked up from the ALB table, and the address of the object is calculated by adding the offset to the page address. ObjectAccess (Table 35) shows pseudocode of the object access.

## Garbage Collection on Compressed Java Memory

The garbage collection algorithm has been discussed. The garbage collection is implemented in garbageCollection and its subfunction called garbageCollectForReal. The garbageCollectorForReal calls seven procedures (markRootObjects, markNonRootObjects, markWeakPointerLists, sweepTheHeap, compactTheHeap, updateRootObjects, and updateHeapObjects). Our garbage collector uses procedures and functions similar to the original one. However, our procedures and functions have a preprocessing step before each assignment and a postprocessing step after each assignment. In our architecture, objects are represented

Table 35: ObjectAccess

```
ObjectAccess(objectIdentifier)

1.  pageNumber ← STRIPE_PAGE_NUMBER(objectIdentifier)
2.  offset ← STRIPE_OFFSET(objectIdentifier)
3.  mark ← STRIPE_MARK(objectIdentifier)
4.  type ← STRIPE_TYPE(objectIdentifier)
5.  address ← ALBtable → ALBentry[pageNumber]
6.  objectAddress ← address + offset
7.  return objectAddress
```

by objectIdentifiers and the objectIdentifiers must be decoded to find the real objects. For example, the address of a page that includes an object represented by an objectIdentifier, is found and then the page is loaded into the caching unit before subsequent procedures and functions are called. An address of the object is calculated by adding its offset in the page to the starting address of the page in the caching unit and the object is used for assignments. Once the assignments are finished, the results are postprocessed. The objects are encoded into objectIdentifiers again. Table 36 shows an example of how the preprocessing step and the postprocessing step handle the objectIdentifer. Tables 37 and 38 give pseudocode of each. Note: The preprocessing step is a simplest version. The unCompressedPage is 4KB which is the same as the buffer size. If uncompressed page is less than the buffer size, and the delayed buffer has enough space, the uncompressed page is merged into an existing page in the delayed buffer.

## Mark Phase

The object identifiers are not allocated in a compressed heap and an object graph (a data structure that holds objects reachable from a computational root) stores the object

Table 36: How Preprocessing and Postprocessing Handle ObjectIdentifier

```
markChildren

:


switch(gctype)
  case GCT_INSTANCE:
    instance ← object;
  preprocess(object, object1, pageNumber1, offset1)
    instance1 ← object1 // instance ← object
    but for object reference holder inside object
    preprocess(instance→ofClazz, object2, pageNumber2, offset2)
    clazz ← object2 // clazz ← instance → ofClazz
    checkMonitorAndMark(instance) // pass object identifier
  while(clazz)
    FOR_EACH_FILED(thisField, clazz→fieldTable) // pass object identifier
      preprocess(thisField, object3, pageNumber3, offset3)
      preprocess(object3→u.offset, object4, pageNumber4, offset4)
      offset ← object4 →u.offset // offset ←thisField→u.offset
      preprocess(object1→data[offset].cellp, object5, pageNumber5, offset5)
      *subobject ← object5 // *subobject ← instance →data[offset.cellp;
      postprocess(*subobject, offset5, pageNumber5)
      MARK_AND_TAIL_RECURSE(subobject)
    END_FOR_EACH_FIELD
    preprocess(clazz→ superClazz, object6, pageNumber6, offset6)
    clazz ← clazz → superClazz
    postprocess(offset, offset4, pageNumber4)
    postprocess(thisField, offset3, pageNumber3)
    postprocess(clazz, offset2, pageNumber2) // postprocess(clazz, offset6, pageNumber6)
    postprocess(object, offset1, pageNumber1)
  break
:
```

Table 37: Preprocess

```
    preprocess(objectIdentifier, object, pageNumber, offset)

1.  pageNumber ← STRIPE_PAGE(objectIdentifier)
2.  offset ← STRIPE_OFFSET(objectIdentifer)
3.  pageAddress ← ALBtable− >ALBentry[pageNumber].pageAddress
4.  if !inBuffer(pageAddress)
5.      if inAnyHeap(pageAddress)
6.          compressedPage
7.          ← compressor(Buffer[nextVictimEntry].buffer
8.          store(compressedPage)
9.          unCompressedPage ← decompressor(pageAddress)
10.         // decompressor will check a compressed page size
11.         Buffer[nextVictimEntry].buffer ← unCompressedPage
12.         object ← Buffer[nextVictimEntry].buffer + offset
13.     else VMError(pageAddress) // end of if inAnyHeap...
14. object ← pageAddress + offset
```

Table 38: Postprocess

```
    postprocess(objectIdentifier, offset, pageNumber)

1.  objectIdentifier ← pageNumber « x | offset « y
```

identifiers in place of memory addresses of objects in our architecture. In the mark phase, the garbage collector traverses objectIdentifiers reachable from the roots in the graph. The operation is done without decompressing the real objects. When the garbage collector encounters an objectIdentifier in a reachable graph, it updates a mark bit in its objectIdentifier and in the ALB table. If the object contains object reference holder(s), the garbage collector also updates its/their mark bit/bits. The garbage collector traverses the objectIdentifiers reachable from the encountered objectIdentifier using information from their object reference holders. The objectIdentifiers reachable from the encountered objectIdentifier are marked in this stage. The garbage collector resumes processing for the objectIdentifiers directly reachable from the roots. This operation is repeated until all objectIdentifiers directly and indirectly reachable from the roots are processed.

### Sweep Phase

The sweep phase is performed per page in our architecture. First, the sweep phase gets a list of objectIdentifiers for a page entry from the ALB table and checks their mark bits. If the mark bit indicates the object is not live, the objects are deallocated. The sweep phase is called in the compressing and decompressing units. Table 39 shows pseudocode of the sweep phase.

### Compaction Phase

The compaction phase is called following to the sweep phase in the compressing and decompressing units. Thus, the basic unit is a page. First, a list of object identifiers for a page in the decompressing or compressing unit is obtained from the ALB table and then all objects indicated by the object identifiers are moved toward the start of the page in a sliding manner. Thus, the internal fragmentation of the page is removed. Table 40 shows pseudocode of the compaction phase.

Table 39: sweepTheHeap

```
    sweepTheHeap()

1.  List ← get_ObjectIdentifierList(ALB, pageNumber)
2.  objectIdentifier ← getNext(List)
3.  do
4.          if !IsMarked(objectIdentifier)
5.              free(objectIdentifier)
6.          objectIdentifier ← getNext(List)
7,  while objectIdentifier != NULL
```

Table 40: Compaction Phase

```
    compactTheHeap()

 1.  List ← get_ObjectIdentifierList(ALB, pageNumber)
 2.  objectIdentifier ← getNext(List)
 3.  pageNumber ← STRIPE_PAGE(objectIdentifier)
 4.  nextObjectStart ← ALBtable→ALBentry[pageNumber].pageAddress
 5.  do
 6.      Slide(objectIdentifier, nextObjectStart)
 7.      objectSize ← STRIPE_SIZE(objectIdentifier)
 8.      nextObjectStart ← nextObjectStart + objectSize
 9.      objectIdentifier ← getNext(List)
10,  while objectIdentifier ! = NULL
```

## Experiments

We examine compression techniques on mobile/wireless devices and show their impacts on space efficiency, speedup, and power/energy consumption. The system is designed as a client-server model. The Java 2 Platform, Micro Edition (J2ME) is used for the base configuration. The heap size is set to 32 KB, 64 KB, or 256 KB. The page size is 4 KB by default, and the cache size is 64 KB. The compaction mode is on. The compression algorithm is the modified Wilson-Kaplan (WK) algorithm, implemented both in software and hardware. The hardware de/compressor is simulated in a Xilinx ISE simulator on the device XC2VP30ff896. The compressed heap is simulated on a Solaris 9 platform (SunOS 5.9). Table 41 lists a set of benchmark applications for this experiment. In the experiments, the hardware simulation reports time and power. Memory is composed of 16 1-Mbit Random Access Memory (RAM) modules with 39mW in power (20 mA in current and 1.95V in voltage). 1-Mbit RAM contains 64 4-KB memory banks, 32 8-KB memory banks, 4 64-KB memory banks or 1 128-KB memory banks. Memory management includes a memory bank partitioning technique and a power saving mechanism. The memory bank partitioning technique virtually separates unused memory banks from used memory banks, and the power saving mechanism powers off the unused memory banks. The in-memory/in-heap object compression is used to increase the number of unused memory banks and to achieve low power consumption. The software simulator measures the number of Garbage Collection (GC) cycles, the execution time of GC, the number of cache misses (that are related to the number of de/compression), and the execution time of de/compression. Each object is traced when it is created and accessed. The space efficiency, speedup and power/energy consumption and efficiencies are calculated using Eq. (1) to Eq. (9) as described in CHAPTER 2.

The performance of the configurations has been evaluated using benchmark applications. Table 42 shows the space efficiency and speedup. Four out of five benchmark applications have space efficiency greater than 2.0. A space efficiency of 2.0 indicates that the in-heap compression technique can save 50% of the heap memory demand or more on

Table 41: Benchmarks

| MIDlet suite | Description |
|---|---|
| **demos(HTTP)** *64KB heap* | Technical demonstration program of http for MIDP ( comes with Sun Java KVM[3] ). |
| **stock** *64KB heap* | Get stock quotes from a publicly available website, and set alerts ( comes with Sun Java KVM ). |
| **audiodemo** *64KB heap* | Listen to sounds using Mobile Media APIs audio building block ( comes with Sun Java KVM ). |
| **manyballs** *32KB heap* | Game program (comes with Sun Java KVM). |
| **EmailViewer** *256KB heap* | Wireless email service program for Java-enabled (J2ME) mobile devices ( www.reqwireless.com ). |

Table 42: Space Efficiency, Speedup, Power Consumption Saved (Configuration II)

| Application | Space efficiency | Speedup |
|---|---|---|
| **demo(HTTP)** *64KB heap* | 1.80 | 1.00 |
| **stock** *64KB heap* | 2.50 | 1.00 |
| **audiodemo** *64KB heap* | 2.20 | 1.00 |
| **manyballs** *32KB heap* | 2.09 | 1.00 |
| **EmailViewer** *256KB heap* | 2.05 | 0.99 |

Table 43: Power Saved and Power Efficiency in Configuration II (heap and RAM)

| Application | Power efficiency (heap) | Power saved (heap) | Power efficiency (RAM) | Power saved (RAM) |
|---|---|---|---|---|
| **demo(HTTP)** *64KB heap* | 2.27 | 0.536 | 72.76 | 38.46 |
| **stock** *64KB heap* | 2.50 | 0.488 | 79.92 | 38.51 |
| **audiodemo** *64KB heap* | 1.85 | 0.658 | 59.27 | 38.34 |
| **manyballs** *32KB heap* | 1.92 | 0.317 | 123.02 | 38.68 |
| **EmailViewer** *256KB heap* | 1.96 | 2.486 | 15.69 | 36.51 |

Table 44: Energy Saved in Configuration II (heap) and Energy Overhead

| Application | Energy saved (mJ) | SW de/comp energy (mJ) | HW de/comp energy (mJ) | Energy efficiency (heap) |
|---|---|---|---|---|
| **demo(HTTP)** *64KB heap* | 413 | 1.235 | 0.007 | 1.79 |
| **stock** *64KB heap* | 427 | 12.321 | 0.004 | 2.50 |
| **audiodemo** *64KB heap* | 581 | 1.532 | 0.001 | 2.17 |
| **manyballs** *32KB heap* | 106 | 0.614 | 0.001 | 2.04 |
| **EmailViewer** *256KB heap* | 739 | 20.892 | 0.088 | 2.08 |

average. The speedup of *demo*(*HTTP*), *stock*, *audiodemo* and *manyballs* is 1.00, which means the total execution time of the proposed configuration is almost the same as the execution time of the base configuration. Thus, the time overhead is negligible by introducing the hardware de/compressor and the memory management (The hardware de/compressor is currently ten thousands times faster than its software counterparts.). In configuration I, *audiodemo* shows the time overhead (0.941 in speedup for a user interface part). The time overhead is negligible. *EmailViewer* shows a speedup of less than 1.00 (0.99). There is some time overhead. It seems to be caused by a large heap size (256KB) and is left for future studies. Table 43 shows power efficiency (configuration II over the base configuration) in heap and RAM, and presents power saved in heap and RAM where the maximum amount of power in RAM is 39 mW. *demo*(*HTTP*), *stock* and *audiodemo* use a 64KB heap and have 2.27, 2.50, and 1.85 of power efficiencies, respectively. The size of the heap for manyballs is 32 KB and its power efficiency is 1.92. Similarly, the size of the heap for EmailViewer is 256 KB and its power efficiency is 1.96. There is no relationship between the power efficiency in heap and its size.

Configuration II is further examined in energy saving in combination with its sensibility to the size of memory banks. Table 44 shows minimum energy saving on 16 1-Mbit RAM (random access memory) in configuration II over the base configuration, energy overhead due to a software de/compressor, and a hardware de/compressor. The minimum energy saving varies between 106 to 739 mJ, while the energy overhead ranges from 0.614 mJ to 20.892 mJ in the software de/compressor, and 0.001 mJ to 0.088 mJ in the hardware de/compressor. The energy overhead is very small, especially for the hardware de/compressor. This shows that introducing the de/compressor to Java runtime environment (JRE) is feasible.

Tables 45, 46, 47, and 49 summarize energy consumption with variable-sized memory banks. Table 45 shows the energy consumption of the benchmark applications over the base configuration. The energy is in a range of 200 to 1550 (mJ) over 16 Mbits RAM.

Table 46 and Figure 24 shows energy consumption in the base configuration with

Table 45: Energy Consumption in Base Configuration

| Application | Energy (mJ) |
|---|---|
| **demo (HTTP)** *64KB heap* | 783.218 |
| **stock** *64KB heap* | 1067.176 |
| **audiodemo** *64KB heap* | 1263.052 |
| **manyballs** *32KB heap* | 216.188 |
| **EmailViewer** *256KB heap* | 1539.893 |

the memory management that includes a memory bank partitioning technique with a power saving mechanism where energy consumption is calculated using several memory bank sizes (4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB). Experimental results show that 32 KB is a memory bank size that can minimize the energy for four benchmark applications. manyballs requires 32 KB memory space as its minimum heap and 64 KB-memory banks saved more energy than 128 KB-memory banks. 64 KB-memory space is the minimum heap size for the demo (HTTP), stock and audiodemo and 64 KB-memory banks cannot save more energy than 128 KB-memory banks. For EmailViewer, a 16 KB-memory bank is better. The minimum heap size of EmailViewer is 256 KB. The optimum size of a memory bank is inversely proportional to the minimum heap size required for executing benchmark applications.

Table 46: Energy Consumption for Different Memory Bank Partitioning

| Application | 4 KB (mJ) | 8 KB (mJ) | 16 KB (mJ) | 32 KB (mJ) | 64 KB (mJ) | 128 KB (mJ) |
|---|---|---|---|---|---|---|
| **demo (HTTP)** *64KB heap* | 23.093 | 23.093 | 23.093 | 23.093 | 46.186 | 46.486 |
| **stock** *64KB heap* | 33.383 | 33.383 | 33.383 | 33.383 | 66.767 | 66.767 |
| **audiodemo** *64KB heap* | 39.511 | 39.511 | 39.511 | 39.511 | 79.022 | 79.022 |
| **manyballs** *32KB heap* | 3.381 | 3.381 | 3.381 | 3.381 | 6.763 | 13.526 |
| **EmailViewer** *256KB heap* | 84.102 | 84.102 | 84.102 | 96.342 | 144.513 | 192.684 |

Table 47 and Figure 25 show energy consumption in configuration II where energy is calculated using several memory bank sizes (4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB). The optimum memory bank size is 32 KB, similar to the experiment whose results are shown in Table 46 and Figure 24, but the memory space required in configuration II is much smaller than the base configuration and energy consumption with 64 KB-memory banks is still better than energy consumption on the 128 KB-memory bank. Configuration II is more sensible in terms of block size than the base configuration and achieves lower

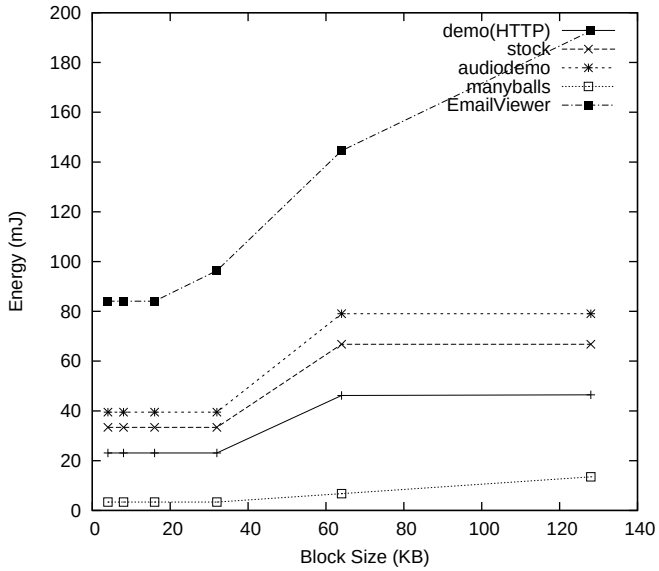



Figure 24: Energy Consumption for Different Memory Bank Partitionings in Base Configuration

energy consumption. In this case, energy consumption in configuration II compared to the base configuration with the memory banks is proportional to the size of the active memory (the total size of the active memory banks).

Table 47: Energy Consumption in Configuration II for Different Block Sizes

| Application | 4 KB (mJ) | 8 KB (mJ) | 16 KB (mJ) | 32 KB (mJ) | 64 KB (mJ) | 128 KB (mJ) |
|---|---|---|---|---|---|---|
| **demo (HTTP)** *64KB heap* | 11.546 | 11.546 | 11.546 | 11.546 | 23.092 | 46.184 |
| **stock** *64KB heap* | 14.504 | 16.693 | 16.693 | 16.693 | 33.387 | 66.774 |
| **audiodemo** *64KB heap* | 19.755 | 19.755 | 19.755 | 19.755 | 39.510 | 79.019 |
| **manyballs** *32KB heap* | 1.663 | 1.663 | 1.663 | 3.381 | 6.762 | 13.525 |
| **EmailViewer** *256KB heap* | 42.240 | 42.240 | 48.161 | 48.161 | 48.161 | 96.322 |

Table 48 and Figure 26 show energy saved in Java heap on configuration II. The base configuration implements memory bank partitioning and power control. In Table 48, energy consumption saved is calculated using several memory bank sizes (4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB). Energy saved in configuration II compared to the base configuration with the memory banks depends on benchmark applications. 64 KB-memory banks can greatly save energy in configuration II with benchmark applications demo(HTTP), stock,

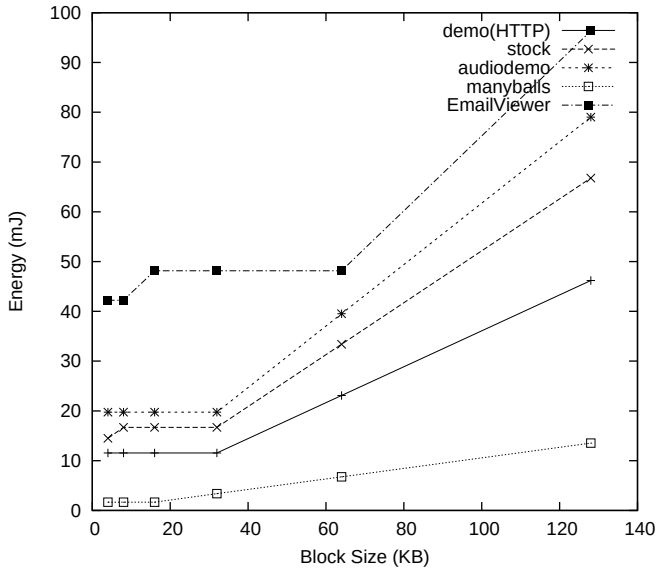



Figure 25: Energy Consumption in Configuration II as a Function of Block Size

audiodemo, and manyballs. As for EmailViewer, 128 KB memory banks show slightly higher saving, but the difference is not significant. In the experiment with stock, configuration II with 128 KB-memory banks shows lower energy saving than the base configuration with 128 KB-memory banks. The characteristics of the benchmark application **stock** seem to be different from others.

Table 48: Energy Saved in Configuration II compared to Base Configuration with Different Memory Bank Partitionings

| Application | 4 KB (mJ) | 8 KB (mJ) | 16 KB (mJ) | 32 KB (mJ) | 64 KB (mJ) | 128 KB (mJ) |
|---|---|---|---|---|---|---|
| **demo (HTTP)** *64KB heap* | 11.547 | 11.547 | 11.547 | 11.547 | 23.094 | 0.002 |
| **stock** *64KB heap* | 18.879 | 16.690 | 16.690 | 16.690 | 33.380 | -0.007 |
| **audiodemo** *64KB heap* | 19.756 | 19.756 | 19.756 | 19.756 | 39.512 | 0.003 |
| **manyballs** *32KB heap* | 1.719 | 1.719 | 1.1719 | 0.001 | 0.001 | 0.001 |
| **EmailViewer** *256KB heap* | 41.862 | 41.862 | 35.941 | 48.181 | 96.352 | 96.362 |

Table 49 and Figure 27 show energy saved in configuration II compared to the base configuration (Table 45) where energy consumption saved is calculated using several memory bank sizes (4 KB, 8 KB, 16 KB, 32 KB, 64 KB, 128 KB). Energy consumption saved in configuration II compared to the base configuration ranges from 200 mJ to 1500

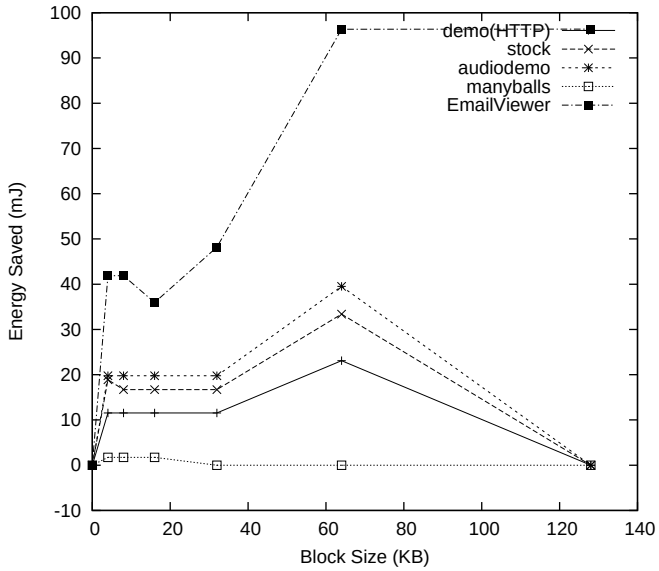



Figure 26: Energy Saved in Configuration II compared to Base Configuration with Different Memory Bank Partitioning

mJ and is not sensitive to the size of the memory bank. Through several experiments on memory-bank-size sensibility, we conclude the following two cases:

1. application programs with small memory demands in a system with a bigger memory capacity require the size of the memory bank somewhere between half to 3/4 of their heap size and can have power consumption reduction mainly with the memory bank partitioning technique.

2. application programs with large memory demands in a system with memory capacity smaller than their demands can have power consumption reduction mainly as a result of in-memory/in-heap compression with the memory bank partitioning technique. Using small memory banks make unused banks in this case.

Memory demands of application programs are predicted to increase every year as rich functionality is a trend (e.g., 2D, 3D image processing and HotSpot technology). The current memory capacity of mass-products is not enough to support these applications. The second case may increase. However, there are a variety of application programs that the first and

second cases are mixed. The optimal size of the memory bank is determined by local and global minimum and maximum amount of memory required to run application programs. Reconfigurable virtual memory banks seem to be effective and are to be investigated in the future.

Table 49: Energy Saved in Configuration II compared to Base Configuration and Block Size Sensibility

| Application | 4 KB (mJ) | 8 KB (mJ) | 16 KB (mJ) | 32 KB (mJ) | 64 KB (mJ) | 128 KB (mJ) |
|---|---|---|---|---|---|---|
| **demo (HTTP)** *64KB heap* | 726.673 | 726.673 | 726.673 | 726.673 | 715.127 | 692.035 |
| **stock** *64KB heap* | 1052.672 | 1050.483 | 1050.483 | 1050.483 | 1033.789 | 1000.403 |
| **audiodemo** *64KB heap* | 1243.297 | 1243.297 | 1243.297 | 1243.297 | 1223.543 | 1184.033 |
| **manyballs** *32KB heap* | 214.252 | 214.525 | 214.525 | 212.807 | 209.426 | 202.664 |
| **EmailViewer** *256KB heap* | 1497.654 | 1497.654 | 1491.732 | 1491.732 | 1449.732 | 1443.571 |

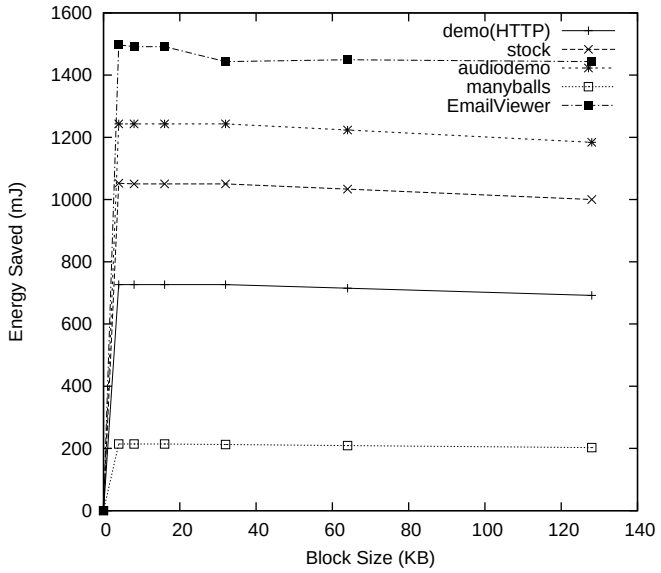



Figure 27: Energy Saved in Configuration II compared to Base Configuration and Block Size Sensibility

# CHAPTER 5: Discussion

We compare configurations I and II with a base configuration. Configuration I works well for the Hypertext Transfer Protocol (HTTP), which is 11.8 % faster than the base configuration with the garbage collection. Configuration I, however, shows a variety of behaviors (good or bad) for HTTP applications. Entirely, configuration II shows significant memory saving (52.8 % on average) over all benchmarks with negligible time overhead using the hardware compressor and decompressor and the memory management. The significant memory saving adds extra advantage in random access memory (RAM) power consumption (up to 0.926 mW of power consumption was saved). This corresponds to 73.9 mJ energy saving at maximum with small energy overheads of compressor and decompressor (experiment shows 0.614 mJ to 20.892 mJ in the software de/compressor, and 0.001 mJ to 0.088 mJ in the hardware de/compressor). Figures 28 and 29 show space, power and energy efficiencies in configurations I and II. Power efficiency and space efficiency are very close among benchmark applications in configuration I, but energy efficiency varies in different application programs. demo(HTTP) and stock have low efficiencies in energy compared to the power and space, while audiodemo and WebViewer have higher efficiencies in energy among all. The implementations of audiodemo and WebViewer are energy-efficient. The real amount of energy saved supports the statement for WebViewer, but the real amount of power saved supports the statement for both audiodemo and WebViewer. Figures 30 and 31 show energy saved (mJ) and power saved (mW) in configuration I. In configuration II, there are four relationships in power, space, and energy among five benchmark applications. Figures 32 and 33 show energy saved (mJ) and power saved (mW) in configuration II. audiodemo and manyballs are space-efficient compared to power and energy. demo(HTTP) is power-efficient and EmailViewer is energy-efficient. Among all benchmark programs, stock is the best in the space, power, and energy efficiency. The implementation of stock is optimal, although the real amount of energy saved (mJ) in stock is 427 mJ, which is the third, and the real amount of power saved (mW) is 0.488 (mW), which is the fourth. In practice, the real amount of power saved is very important because the targeted devices are resource-limited in battery.

In our analysis, EmailViewer is energy-efficient, saving 2.486 mW of power. We are currently studying the relationships between the efficiency and the real amount of power saved.

Lower power consumption is a critical issue in mobile/wireless embedded systems. We believe that the runtime compression technique can contribute to space and power reduction with low time overhead in very small handsets and support Java applications that demand large memory. We also believe that our modeling for performance (space, power/energy efficiencies, power/energy saved, time) can contribute to the analysis of the application development (energy-efficient, space-efficient, power-efficient, and/or time-efficient) and will provide guidance for their future improvements.

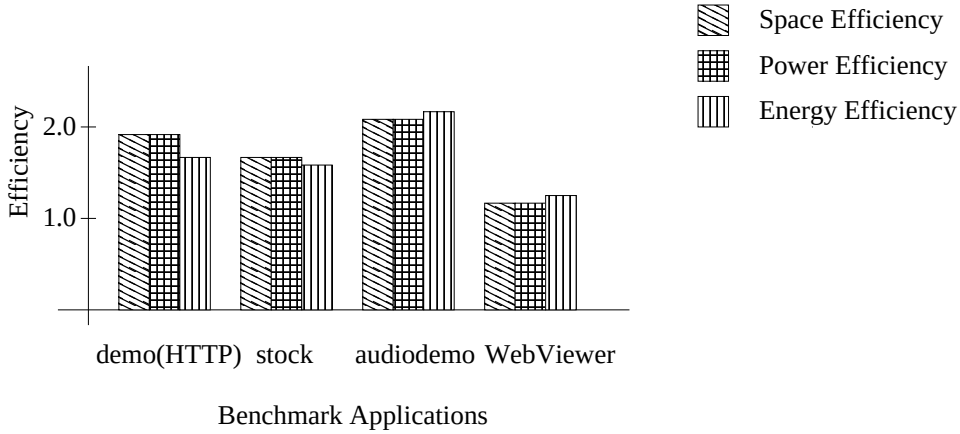



Figure 28: Space, Power, Energy Efficiencies in Configuration I

Our approach is built on a standard JRE with a different type of memory and memory management. The standard JRE assumes standard memory (uncompressed), while our JRE assumes a virtual compressed memory for Java heap and uncompressed buffers for caching. In the standard JRE, an object is represented using a memory address inside the Java VM, and the object is accessed using the memory address. In our JRE, objects are represented by object identifiers. An object is accessed by decoding its identifier and loading a page that contains the object into a caching unit after decompressed. Once the object is sent to the Java VM in an uncompressed form, it is processed in a way similar to that in the standard JRE. However, object liveness is handled differently. There is a

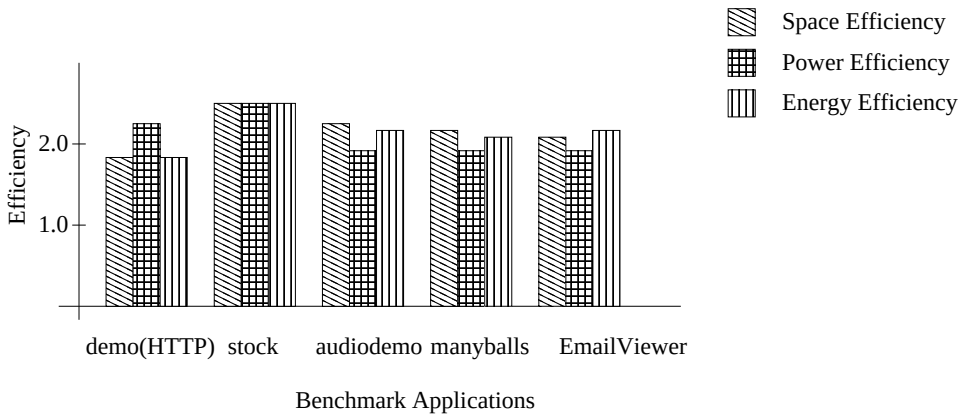



Figure 29: Space, Power, Energy Efficiencies in Configuration II

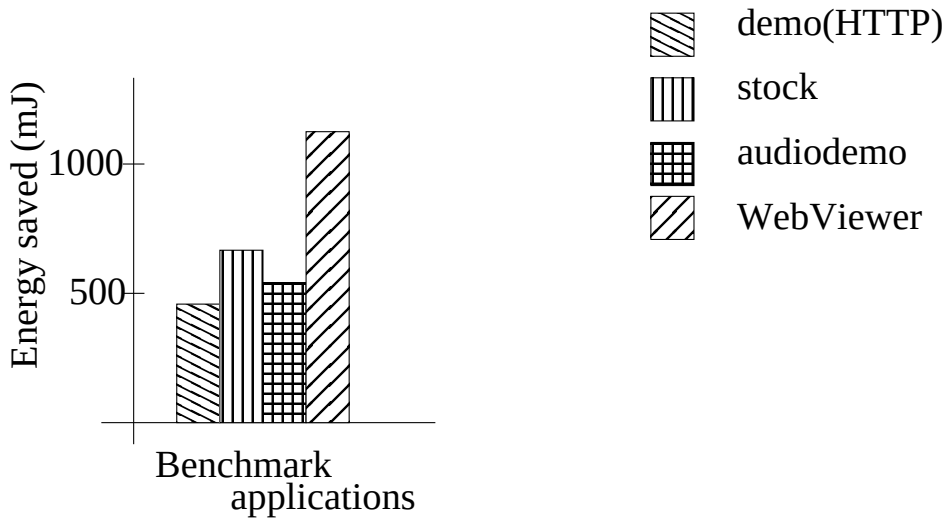



Figure 30: Amount of Energy Saved in Configuration I

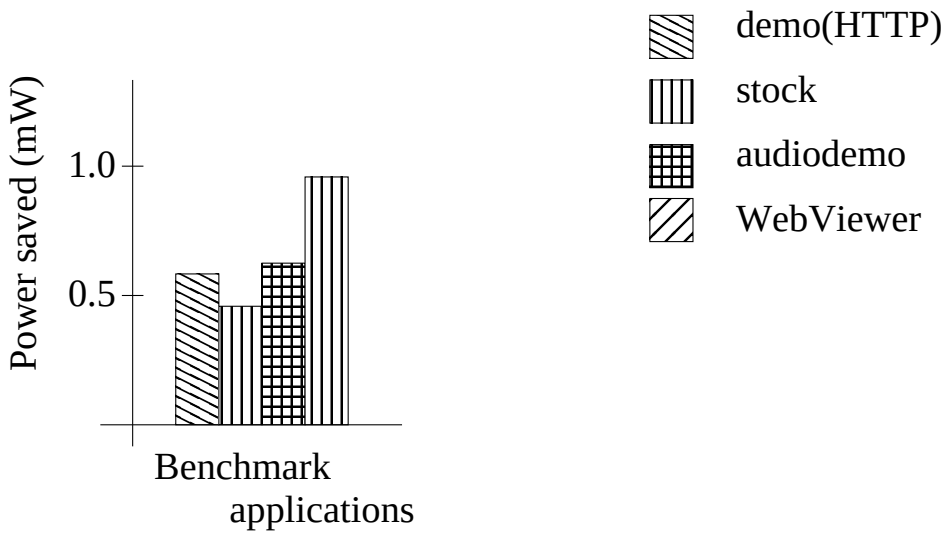



Figure 31: Amount of Power Saved in Configuration I

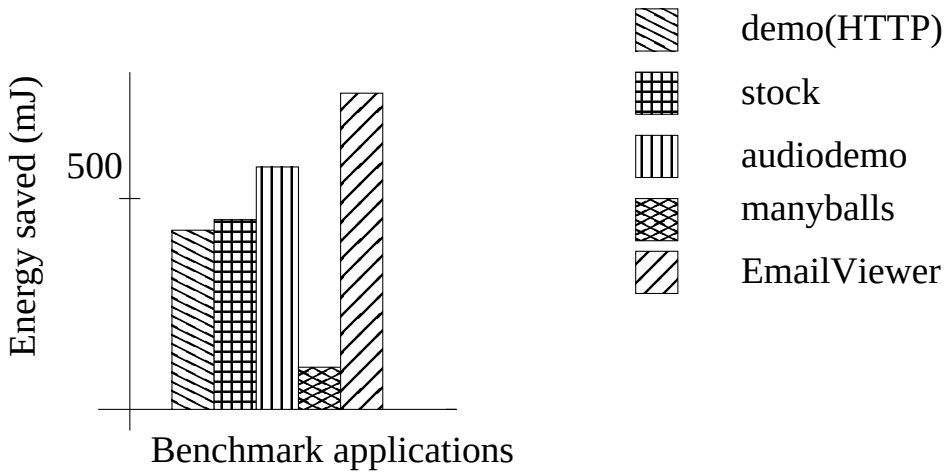



Figure 32: Amount of Energy Saved in Configuration II

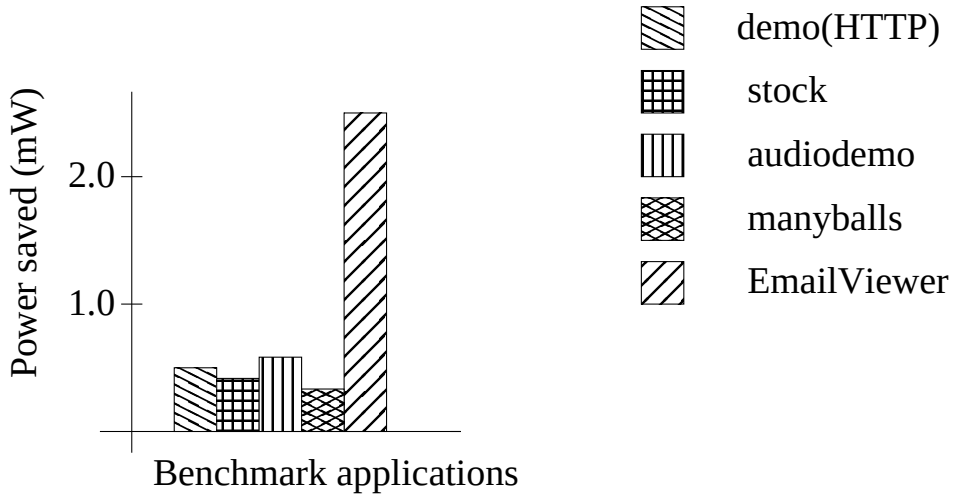



Figure 33: Amount of Power Saved in Configuration II

garbage collection mechanism in the standard JRE. Our JRE configurations I and $I^h$ do not activate garbage collection. So, the automatic memory management has the in-memory compression only and object liveness is not maintained during the execution. On the other hand, configuration II and $II^h$ handle object liveness and have both in-memory compression and garbage collection activated. In the standard JRE, object liveness is kept by a garbage collection mechanism based on a mark-sweep algorithm (with compaction). In configuration II and $II^h$ of our JRE, the garbage collection also keeps object liveness information, but does not use traditional way of the mark-sweep algorithm (with compaction). There is a problem in Java compressed heap memory that the garbage collection cannot traverse objects directly. In configuration II and $II^h$, an object graph holds object identifiers in place of object references (memory addresses), and the mark and compact phases handle object liveness using the object identifiers. Therefore, the mark and compact phases do not need to touch actual objects in Java compressed heap memory. Everything is done within the object identifiers.

The standard JRE for mobile/wireless embedded system is implemented in software and has a software Java VM, while our JRE is a hardware/software codesigned architecture that has a software Java Virtual Machine (VM), a software compressed-memory manager,

and a hardware memory manager, a software/hardware compressor, a software/hardware compressor, a caching unit and an Address Lookaside Buffer (ALB) Table. There are several additional units compared to the standard JRE, but the benefit of our JRE is significant. Small memory demand and low power consumption with negligible time overhead are benefits of in-memory compression over the standard JRE and are evaluated by the simulation and experiments in CHAPTER 3 and CHAPTER 4. Table 50 summarizes the comparison between the standard JRE and our approach.

Table 50: Comparison between the standard JRE and Our approach

| Different parts | standard JRE for mobile/wireless embedded system | Our approach |
|---|---|---|
| software? hardware? | software Java runtime environment (JRE) that consists of a Java virtual machine (JVM) | software and hardware codesigned Java runtime environment (JRE) that consists of a Java virtual machine (JVM), a SW memory management module, a HW memory management module, an Address Lookaside Buffer (ALB) table, a compressing unit, a decompressing unit, and a caching unit |
| Object Representation inside Java virtual Machine | physical address | object identifier |
| Memory Assumption | standard memory (uncompressed) | uncompressed buffers and compressed heap memory |
| Automatic Memory Management | mark and sweep algorithm (compaction is option) | configuration I and $I^h$ does not have any garbage collection algorithm and has in-memory compression. configuration II and $II^h$ has in-memory compression and also uses a modified mark and sweep algorithm with compaction as a garbage collection algorithm |

Tables 51 and 52 shows a comparison between Chen *et al.* [32] and our approach. In their approach, each object is de/compressed individaly, but its object header is uncompressed for the garbage collection. In our approach, a group (page) of objects is de/compressed

together. The major advantage of our page-based in-memory compression is that fewer compression calls are required, and smaller time overhead is involved. A page of objects may include many objects requiring for computation because of temporal and spatial locality of the objects. Page-based in-memory compression reduces the number of de/compression compared to object-based in-memory compression. Therefore, the time penalty due to de/compression is smaller than that of the object-based in-memory compression on a given architecture in general. The in-memory compression is sensitive to the type of a compression algorithm used to achieve high compression ratio. Chen *et al*. use a zero-removal compression algorithm. We modify the Wilson and Kaplan (WK) compression algorithm in the JRE. Our simulations and experiments suggest that the WK compression algorithm outperforms the zero-removal in a domain of Java mobile/wireless applications. There are some major shortcomings of the page-based in-memory compression compared to the object-based compression in garbage collection, but our approach overcomes the shortcoming. In the page-based in-memory compression architecture, the mark and compact phases cannot retrieve object headers and object holders inside objects because they are compressed (CHAPTER 2). In our page-based in-memory compression architecture, objects are represented by object identifiers that are a newly designed data structure used inside our modified Java Virtual Machine (VM) and stored in the Address Lookaside Buffer (ALB) table. For garbage collection, the mark and type bits of objects are kept in the object identifiers. The mark and compact phases then retrieve the object identifiers and mark/update the object identifiers. The sweep phase collects garbage objects using the information in the object identifiers. Introducing object identifiers and storing them in uncompressed form removes the major shortcoming of the page-based in-memory object compression over the object-based compression. Both the object-based compression and our approach use the mark and sweep garbage collection algorithm with compaction, but they are completely different from each other. In the work proposed by Chen *et al*, the garbage collection algorithm incorporates de/compression, which compresses each object, In our approach, configuration II and $II^h$ have garbage collection, the mark phase works over object identifiers as discussed. The sweep and compact phases

are moved into the compression and decompression modules and called per page. Calling the mark phase is invoked periodically or when the Java VM requires more memory. But, the sweep and compaction phases are called every time compression or decompression is called to a given page and/or may be called for each compressed page in a back-ground thread.

Our approach differs from that of Chen *et al.* in object creation and access as summarized in Tables 51 and 52. In their approach, when an object is created, it is stored in the heap in uncompressed form until the heap reaches a predetermined size. Once the heap grows to the size, objects in the heap are compressed during the garbage collection. In our approach, each object is temporarily stored in one of the buffers in a caching unit when it is created. Address information of each object is encoded into its object identifier, and each object identifier is added to the Address Lookaside Buffer (ALB) Table. More specifically, it is added to an entry (in the ALB table) related to a page that includes a corresponding object. Once the caching unit gets full, one of the buffers is selected and its content is compressed. The compressed content is stored in the Random Access Memory (RAM) and the space reused for newly coming objects in the caching unit. However, in the approach used by Chen *et al.*, each object is accessed in a similar way to that of the original JRE. Once objects are compressed, each object is decompressed when accessed. Each object after decompressed is not buffered. In our approach, a page of objects is buffered in a buffer after decompression for future uses. Four buffers are created in a caching unit to hold objects (totally four pages) to reduce the number of de/compressions (one vs. the number of objects). Chen *et al.* also did not address the causes of power consumption of the garbage collector with compression and the impact of hardware de/compression. In contrast, our work includes analysis of power consumption and the impact of hardware de/compression.

*Kermany and Petrank* ([33] 2006) focus on a compaction algorithm for the garbage collection and its speedup only and demonstrate the feasibility of a page-based approach to Java [33]. Their work is motivated by the fragmentation problem that web application servers frequently face. In general, work is motivated by the long compaction time (to eliminate the fragmentation but stop the application for a long time). *Kermany and Petrank* present

Table 51: Comparison between Chen et al. [32] and Our approach (1)

| | Approach proposed by Chen et al. | Our Approach |
|---|---|---|
| De/Compression basis | A single object is de/compressed in each call for the de/compression. | A group of objects (page basis) is de/compressed in each call for the de/compression. |
| Object header | Object headers are modified but uncompressed. The garbage collection can retrieve objects. | Original object header Compressed per page |
| Compression algorithms | Zero-removal | WK algorithm |
| Garbage collection | Mark and Sweep with compaction and compression.<br><br>In the garbage collection, a compression is used.<br><br>The header is retrieved in a similar manner to the original one. | Mark and Sweep with compaction, but the sweep and compaction phases are migrated into compression and decompression units.<br><br>The object header is compressed and the mark phase retrieves object identifiers and update mark bits of them and in the ALB table. |
| Buffering | n/a | A caching unit is used to hold uncompressed/decompressed objects. The objects are supplied to Java virtual machine and/or sent to memory. The caching unit consists of four buffering units. One works for holding incoming objects. |

a novel one-pass parallel/concurrent compaction algorithm that reduces pause time. The compactor does not compact the heap into itself. It compacts the heap into a second space. First, the compactor finds a page and then move the objects on this page to their new locations in the second space using object liveness information in a mark bit vector and new addresses of the objects from an offset table. The mark bit vector is typically output by any marking procedure and sets bits corresponding to the first and last words of each live object. The offset table holds new object addresses that are first computed based on the mark bit vector without accessing the actual objects in the heap. Both the mark bit vector and the offset table are well-known techniques and their feasibility is discussed in [35] [34].

Table 52: Comparison between Chen et al. [32] and Our approach (2)

| | Approach proposed by Chen et al. | Our Approach |
|---|---|---|
| Object creation | original way, but objects are compressed during the garbage collection as required. | an object identifier is created for each object and objects are accumulated to make a page in a caching unit. When the caching unit gets full, a page of objects are compressed |
| Object access | original way, but the contents of objects must be uncompressed. | Given an object request, if a page that contains the object is in the caching unit, the object is sent to the JVM as it is because it is uncompressed. If it is not in the caching unit, a page that contains the object is decompressed and the object (uncompressed) is sent to the JVM. |

The compactor of *Kermany and Petrank* is parallel/concurrent because it does not need to use any object dependency and touches each page once only.

The advantage of *Kermany and Petrank* comes from the page-based approach, parallelization, and concurrency. The disadvantage is that it requires a double-sized heap space and is therefore not suitable for resource-limited mobile/wireless embedded devices. Our approach uses a mark, sweep, and compaction algorithm with in-memory compression. The goal is to minimize the power and memory consumption with negligible time overhead. The compaction algorithm is used in combination with the sweep algorithm and compacts live objects in a page. Our compactor only requires an one-page-sized buffer for its operation and does not suffer from the extra memory problem. As for the parallelization and concurrency, in our case, object reference graphs are maintained in the mark phase. The mark phase can output object-dependency information held in the ALB table. Then, the sweep and compaction phases, which are integrated into the de/compression units, perform one-page-sweep/compact tasks as back-ground threads using the object-dependency information. Notice that *Kermany and Pertrank* have not applied their page-based approach to mark and sweep phases yet. In contrast, we have applied our page-based approach to the entire garbage collection.

We have focused on traditional Connected Limited Device Configuration (CLDC) implementation and have discussed why Java compressed heap is an emerging issue in Java technology for mobile/wireless embedded systems. It is for low power consumption. Here, we want to discuss a CLDC implementation that supports fast application execution, which is called "CLDC HotSpot implementation." The CLDC HotSpot is a high-performance Java virtual machine for resource-constrained wireless phones and communication-type devices and supports the traditional CLDC specification. The virtual machine of the CLDC HotSpot implementation runs in a mixed mode that uses an adaptive compiler to optimize the frequently used, time-critical pieces of the applications and employs an optimized bytecode interpretor for infrequently used code. The white paper [20] reports that the CLDC HotSpot implementation can achieve a performance gain in speed of appropriately 8 to 10 times when compared to a traditional bytecode interpretor. However, the generated code takes 4 to 8 times as much space as the original bytecode. To simply port the HotSpot technology would result in a memory footprint far too large for mass-market, battery-powered devices. We believe that our Java heap compression could be applied to the CLDC HotSpot implementation to allow an application to run on the small battery-powered devices with low power consumption. The Java heap compression can extendedly support the CLDC HotSpot implementation, especially its large complied code.

Although our work focuses on client side J2ME CLDC implementations, it is also applicable to J2SE and J2EE server side technologies. Database-intensive operations require a huge amount of memory in, for example, biological databases, and memory compression may be attractive. Patterns in a database will be a key to having a good compression ratio. Object compression can also be extended to the remote invocation using an approach similar to memory/heap compression discussed in this work. Renote architecture requires the RMI runtime environment to be modified. The benefit of user's compressed objects over the network is a reduction of the network bandwidth and traffic.

We have studied the in-memory/in-heap object compression. The de/compression time overhead has been minimized by reducing the number of memory accesses (page-based

approach). The content of the memory and heap may have some regularity and the in-memory/in-heap object compression algorithm can fully utilize access patterns and patterns in the contents of objects to improve compression ratio. The patterns have been studied at source level in compilers and programming languages [22][23][24][25]. We believe this is a promising future research direction.

Table 53: Comparison between *Kermany and Petrank* [33] and Our approach

| | Approach proposed Kermany and Petrank | Our Approach |
|---|---|---|
| garbage collection | The a-page-based compaction is added to the standard mark and sweep algorithm<br><br>Their goal is to improve the speed of the traditional compaction algorithm<br><br>Their compaction algorithm uses two virtual spaces and copies object from one to the other. Their roles will change thereafter. | The a page-based mark, sweep, and compaction algorithm with in-memory compression.<br><br>The goal is to minimize the power and memory consumption with negligible time overhead<br><br>The compaction algorithm is used in combination with the sweep algorithm and compact live objects in a page. One-page-sized space is used for the compaction. The page-based approach is also used in the mark and sweep phases. |
| how to provide a small memory consumption | a parallel stop-the-world compactor and a concurrent, incremental, and parallel compactor | a compressor and a decompressor |
| basic unit | a block and page | a block called a page |
| block size | 512 bytes | 4KB<br>The block is also called the page |
| page size | 4KB | 4KB |
| # of pages per call | one or several pages | one page |
| major data | a mark bit vector is typically output by some marking procedure and has set bits corresponding to the first and last words of each live object.<br><br>an offset table holds new object addresses that are first computed based on the mark bit vector without accessing the actual objects in the heap. | a caching unit that consists of buffers. one of the buffers works as a delayed buffer to accumulate incoming objects and/or pages.<br><br>An Address Lookaside Buffer (ALB) table structure that contains physical addresses of pages and a list of object identifiers per page.<br><br>Each object identifier includes # of page in which its corresponding object is stored, the offset in the page, a mark bit and type bits of its corresponding object. Objects may be objects corresponding to the object identifiers in a page or objects pointed by object reference holder/holders inside each object in the page. |

# CHAPTER 6: Conclusions and Future Work

Memory compression is a key technique to supporting the Java runtime environment on wireless/mobile devices. Memory is saved with runtime compression and power is reduced by a memory bank partitioning (powering off unused memory banks). High performance can also be achieved by hardware acceleration.

Java is an object-oriented language. Some objects are defined as having other objects. In-memory compression cannot handle their object feature efficiently. In the in-memory compression schemes, objects and their headers are compressed. Every time an object is accessed, the object and objects reachable from it must be decompressed and compressed back as required. Similarly, when the memory is cleaned up, the object must be decompressed and compressed again. Cost for their runtime de/compression is expensive.

The state-of-the-art technology leaves the object headers uncompressed to reduce the time overhead during the memory clean-up. However, these approaches do not address object access and memory compaction. The cost of de/compression is still high.

We achieve high performance by developing techniques to take advantage of a hybrid software/hardware implementation. We introduce a page-based approach and hardware implementations to make time overhead negligible. Several new data structures including object identifiers are developed to support the approach, and a new Java Runtime Environment (JRE) is hardware/software codesigned and implemented. We consider two configurations that trade off compression and garbage collection. Garbage collection (GC) increases free memory space by collecting garbage in the memory and the compacting space of live objects spatially located. Compression techniques work to reduce memory demands as the GC does, but are faster than GC. Configuration I has in-memory object compression but automatic memory clean-up (garbage collection) is deactivated. Configuration II integrates in-memory object compression and activates garbage collection.

Configuration I has been examined using benchmark applications and shows performance improvements in space efficiency and speedup for many of the applications. However, *WebViewer* shows poor performance in both the space efficiency and the speedup. We sus-

pect that configuration I does not work well for web applications that have large memory demands. The research is targeted to battery-powered small devices, and we further examine the performance using power and energy. *WebViewer* has the best power/energy saving with negligible time overhead derived from the de/compressor, although its space efficiency and speedup are poor in the previous experiments. Based on the power and energy saving, we conclude that configuration I does work well for web applications that have large memory demands. These findings also indicate configuration I will further reduce memory demand in combination with a garbage collection mechanism designed for a compressed heap.

Configuration II is similar to configuration I, but garbage collection is redesigned for compressed pages that hold objects. The object identifier is modified to keep information for the garbage collection (mark bit and type bits), and the address lookaside buffer (ALB) table is expanded to hold a pointer to a list of the object identifiers for each page. The performance of configuration II has been evaluated using benchmark applications. Time overhead is negligible by introducing the hardware de/compressor and more than 50% of the heap memory is saved. This leads to 53% of power saving and 57% of energy saving in the heap memory. In practice, random access memory (RAM) contains 16 1-Mbit memory and the heap sizes of the benchmark applications selected are small. Over 90% of the power and energy for running the benchmark applications is reduced using in-memory/in-heap compression along with the memory bank partitioning and power control techniques. As the size of heap increases, the saved power and energy will decrease. The upper limit is estimated as the amount of the power and energy saved in the heap. The merits of configuration II are small memory demand, lower power consumption, and negligible time overhead in battery-powered very small handsets.

We analyze application programs using performance modeling in terms of space, power/energy efficiencies, power/energy saved, and time. Web applications have large memory demands but are energy-efficient in both configurations I and II. The targeted devices are battery-powered handheld devices, and good energy efficiency is critical. We conclude that configurations I and II work well for application programs, even those with large memory

demands such as the web applications. We believe the efficiency analysis for system and code in the execution environment will help developers improve their programs in space, power/energy, and speed.

There are several issues unsolved in Java memory compression and left for future work. Porting the CLDC HotSpot technology to the current existing mobile/wireless embedded devices may result in a memory footprint far too large for mass-market, battery-powered devices. Integration of CLDC HotSpot implementation is to be studied. It would also be interesting to investigate a Java virtual machine (JVM) implemented solely or within Web browsers that runs across different hardware platforms and operating systems. Recently, the JVM is also implemented directly on hardware to avoid the overhead imposed by running Java on general-purpose operating systems. The effect of memory compression using Java hardware remains to be studied. The continued utility of the virtual machines and Java memory management discussed here can be extended to other traditional virtual machines.

# Bibliography


[1] TracFone Wireless Inc., URL: www.tracfone.com.

[2] G. Foley and F. O'Reilly: Software distribution for wireless devices: a reconfigurable approach, in *Proc. of IFIP/IEEE Eight International Symposium on Integrated Network Management*, June 2003, pp. 469–472.

[3] ARM Jazelle Technology, URL: www.arm.com/products/esd/jazelle_home.html.

[4] G. Chen, M. Kandemir, N. Vijaykrishnan, and W. Wolf: Energy saving through compression in embedded Java environments, in *Proc. CODES'02* (IEEE Computer Society Digital Library), 2002, pp. 163–168.

[5] Silberschatz Galvin: Operating system concepts, fifth Edition, Addison-Wesley, 1998, pp. 23-47, and pp. 239-336.

[6] David Galles: Modern compiler design, Scott/Jones Inc., 2004, pp. 1-357.

[7] J. M. Chang, W. Srisa-an, C. D. Lo, and E. F. Gehringer: Dmmx: Dynamic memory management extensions, ( *The Journal of Systems and Software ( Elsevier Science, 63(3)*), Sep. 2002, pp. 187–199.

[8] W. Srisa-an, C. D. Lo, and J. M. Chang: A hardware implementation of realloc function, ( *Integration, The VLSI Journal, ( Elsevier Science)*, 1999, PP. 173–184.

[9] W. Srisa-an, C. D. Lo, and J. M. Chang: Scalable hardware algorithms for object resizing and reclamation, *International Journal of Microprocessors and Microsystems ( Elsevier Science)* , 2002, pp. 1214–1281.

[10] Wayne Wolf: Modern VLSI design, Third Edition, Peason Education, 1997, pp. 1–32.

[11] J. Nunez and S. Joues: Gbit/s lossless data compression hardware, in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, volume 11:3, June 2002, pp. 499–510.

[12] J. Nunez and S. Joues: Lossless data compression programmable hardware for high-speed data networks, in *Proc. of IEEE FPT'02*, Dec. 2002, pp. 290–293.

[13] J. Nunez and S. Joues: The X-matchLITE FPGA-based data compressor, in *Proc. of 25th EUROMICRO Conference*, Sept. 1999, pp. 126–132.

[14] J. Nunez and S. Joues: The X-matchpro 100 mbytes/second FPGA-based lossless data compressor, in *Proc. of Design, Automation and Test in Europe, DATE Conference*, 2000, pp. 129–142.

[15] N. Vijaykrishnan, M. Kandemir, S. Tomar, S. Kim, A. Sivasubramaniam, and M. J. Irwin: Energy characterization of Java applications from a memory perspective, in *Proc. The USENIX Java Virtual Machine Research and Technology Symposium*, April 2001, pp. 207–220.

[16] G. Chen, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and M. Wolczko: Adaptive garbage collection for battery-operated environments , in *Proc. The 2nd USENIX Java Virtual Machine Reseach and Technology Symposium* (JVM'02), 2002, pp. 1-12.

[17] G. Chen, R. Sherry, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and M. Wolczko: Tuning garbage collection in an embedded Java environment, in *Proc. The 8th International Symposium on High Performance Computer Architecture* (HPCA'02), 2002, pp. 92–103.

[18] J. Gosling, B. Joy, and G. Steele: The $Java^{TM}$ Programming language: Third Edition, ADDISON-WESLEY, 1996, pp. 29–49.

[19] K. Arnold, J. Gosling, and D. Holmes: The $Java^{TM}$ language specification, ADDISON-WESLEY, 2000, pp. 35–63.

[20] Sun microsystems Inc.: The CLDC HotSpot implementation virtual machine, Sun Microsystems Inc., 2001-2002.

[21] R. Jounes and R. Lins: Garbage collection: John Wiley & Sons, 1999, pp. 19–140.

[22] W. Pugh: Compressing Java class files, in *Proc. PLDI'99* (ACM Press), 1999, pp. 247–258.

[23] L. Clausen, U. Schultz, C. Consel, and G. Muller: Java bytecodes compression for low-end embedded systems (ACM transactions on programming languages and systems), 2000, pp. 471–489.

[24] W. Evens and C. Fraser: Bytecode compression via profiled grammar rewriting, in *Proc. PLDI'01* (ACM Press), 2001, pp. 148–155.

[25] W. Evans and C. Fraser: Grammar-based compression of interpreted code (Communications of the ACM), 2003, pp. 61–66.

[26] Xilinx Inc., URL: www.xilinx.com.

[27] T. C. Bell, J. G. Cleary, and I. H. Witten: Text compression: Prentice-Hall, Inc. 1990, pp. 1–318.

[28] Khalid Sayood: Introduction to data compression: Second Edition, Harcourt India Private Limited, 1996, 2000, pp. 1–636.

[29] L. Rizzo: A very fast algorithm for ram compression, *ACM SIGOPS Operating Systems Review*, 31 (2), April 1997, pp. 36–45.

[30] P. Wilson, S. Kaplan, and Y. Smaragdakis: The case for compressed caching in virtual memory systems, in *Proc. USENIX Annual Technical Conference* (USENIX Association), 1999, pp. 6–11.

[31] S. F. Kaplan: Compressed caching and modern virtual memory simulation, in *PhD. Thesis* (The University of Texas at Austin), 1999, pp. 1–348.

[32] G. Chen, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and B. Mathiske: Heap compression for memory constrained Java environments, in *Proc. OOPSLA 2003* (ACM Press), 2003, pp. 282–301.

[33] Haim Kermany and Frez Petrank: The Compressor: concurrent, incremental, and compaction, in *Proc. PLDI'06* (ACM Press), 2006, pp. 354–363.

[34] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein: An efficient parallel heap compaction algorithm, *OOPSLA'04*, October 2004, pp. 224–236.

[35] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank: An on-th-fly mark and sweep garbage collector based on sliding view, *OOPSLA'03*, November 2003, pp. 269–281.

[36] D. Zhu, R. Melhem, and D. Mosse: The Effects of energy management on reliability in real-time embedded systems, in *Proc. of International Conference on Computer Aided Design ICCAD*, Nov. 2004, pp. 35–40.

[37] C. D. Lo and M. Kato: Power consumption reduction in Java embedded systems, in *Proc. CCCT'03 and ISAS'03* (IIIS), 2003, pp. 123–128.

[38] C. D. Lo and M. Kato: Power consumption reduction in Java embedded systems, in *Proc. ICS 2006*, pp. 230–235.

[39] M. Kato and C. D. Lo: A heap de/compression module for wireless Java, in *Proc. PPPJ'04* (the ACM International Conference Proceedings series, Computer Science Press), 2004, pp. 91–99.

[40] M. Kato and C. D. Lo: Impact of Java compressed heap on mobile/wireless communication, in *Proc. ITCC'05* (IEEE Computer Society Press), 2005, pp. 2–7.

[41] M. Kato and C. D. Lo: Hardware solution to Java compressed heap, in *Proc. IEEE FCCM'05* (IEEE Computer Science Press), 2005, pp. 307–308.

## Vita

Mayumi Kato was born in Ikaho, Japan on June 17, 1965, the daughter of Saburo Kato and Tsuneko Kato. She attended Shibukawa Girl's High School in Japan, graduating in 1984. In 1988, she graduated from the Gunma Women's University, receiving a Bachelor of Art degree with a major in English and American Literature focusing on the Linguistics. She worked at Fujitsu Kiden Ltd. in Japan (currently, Fujitsu Frontech Ltd.) from 1988 to 1990 and Suzuye & Suzuye in Japan (Suzuye Patent and Trademark Agent) from 1990 to 1994. In 1995, she resumed her study at East Central University and Oklahoma State University, OK, USA with a major in Computer Science. In 2000, she entered the PhD. program in computer science at The University of Texas at San Antonio. She is a member of IEEE and ACM, and is a recipient of CS fellowships (2000-2005) and CIAS scholarships (2003-present), and several CS and COS travel and conference grants.

Papers/Activities in Computer Science

M. Kato: Low-Power Memory-Enhanced Java Runtime Environment with Compression, accepted in *the ACM SIGPLAN Conference Programming Language Design and Implementation (PLDI) 2007 Student Research Competition.*

M. Kato and C. D. Lo: Power Consumption Reduction in Java-enabled, Battery-powered Handheld Devices through Memory Compression, pre-accepted in *the IEEE International Symposium on Consumer Electronics 2007.*

M. KATO and C. D. Lo: Compression for Low Power Consumption in Battery-powered handsets, in *Proc. DCC 2007*, (IEEE Computer Society Press), 2007, pp. 386.

M. KATO and C. D. Lo: Impact of Java compressed heap on mobile/wireless communication, in *Proc. ITCC'05* (IEEE Computer Society Press), 2005, pp. 2–7.

M. KATO and C. D. Lo: Hardware solution to Java Compressed Heap, in *Proc. IEEE FCCM'05* (IEEE Computer Science Press), 2005, pp. 307–308.

M. KATO and C. D. Lo: A heap de/compression module for wireless Java, in *Proc. PPPJ'04* (the ACM International Conference Proceedings series, Computer Science Press), 2004, pp. 91–99.

M. KATO and C. D. Lo, "Discovering a Strategy for Professional Communication with New Frontier Through Classroom Presentation", IPCC 2004, Minneapolis, Minnesota, U.S.A, sponsored by IEEE Professional Communication Society and the Boeing company, IEEE 445 Hoes Lan Piscataway, NJ 08854-4150, USA, September-October 2004, pp. 223-234.

M. KATO and C. D. Lo, "Growing Adaptation of Computer Science in Bioinfomatics, ISICT 2004, Las Vegas, Nevada, U.S.A, the ACM International Conference Proceeding Series, Computer Science Press, Trinity College Dublin, Ireland, June 2004, pp. 226-231 (ISICT 2004 : information technology session chair).

C. D. Lo and M. Kato: Power Consumption Reduction in Java Embedded Systems, in *Proc. ICS*, 2006, pp. 230–235 .

C. D. Lo and M. Kato: Power Consumption Reduction in Java Embedded Systems, in *Proc. CCCT'03 and ISAS'03* (IIIS), 2003, pp. 123–128.

C. D. Lo and M. Kato: Hardware/Software Codesign in Supporting Security in Embedded Java, in *SCISS*, 2003, http://cops.csci.unt.edu/sciss/2003/02/index.html