Exploiting Frequent Field Values in Java Objects for Reducing Heap Memory Requirements*

Guangyu Chen, Mahmut Kandemir, and Mary J. Irwin
CSE Department
The Pennsylvania State University
University Park, PA 16802
{gchen,kandemir,mji}@cse.psu.edu

ABSTRACT

The capabilities of applications executing on embedded and mobile devices are strongly influenced by memory size limitations. In fact, memory limitations are one of the main reasons that applications run slowly or even crash in embedded/mobile devices. While improvements in technology enable the integration of more memory into embedded devices, the amount memory that can be included is also limited by cost, power consumption, and form factor considerations. Consequently, addressing memory limitations will continue to be of importance.

Focusing on embedded Java environments, this paper shows how object compression can improve memory space utilization. The main idea is to make use of the observation that a small set of values tend to appear in some fields of the heap-allocated objects much more frequently than other values. Our analysis shows the existence of such frequent field values in the SpecJVM98 benchmark suite. We then propose two object compression schemes that eliminate/reduce the space occupied by the frequent field values. Our extensive experimental evaluation using a set of eight Java benchmarks shows that these schemes can reduce the minimum heap size allowing Java applications to execute without out-of-memory exceptions by up to 24% (14% on an average).

Categories and Subject Descriptors

D.3.m [**Software**]: Programming Languages—*Miscellaneous*

General Terms

Languages

Keywords

Java Virtual Machine, heap, garbage collection, frequent field value

*This work is supported in part by NSF Career Award #0093082 and by GSRC.

VEE'05, June 11-12, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

There has been a continued growth in the sales of mobile and embedded devices. Mobile devices have become an integral part of day-to-day activities. Functionalities of multiple personal devices are often integrated into one. For example, cell phones integrate functionalities of handheld game boxes, audio players, digital cameras, and messaging devices. The support for dynamic content made possible by Java has spurred this growth. It is estimated that there are 250 million Java technology-enabled wireless devices offered by 31 major manufacturers currently in use [10].

The design of applications in these wireless devices is severely influenced by resource constraints. Battery power and memory limitations are two primary factors that constrain the applications that can be supported. In fact, memory size limitation is often cited as the main reason that mobile game applications, arguably the most popular segment of wireless applications, run slowly or crash [6]. Memory size constraints will continue to be of importance as the amount of memory in mobile devices influences their cost, power consumption, and form factor [14]. While technology improvements, both in processing and packaging, have enabled more memory to fit into the same form factor, applications supported by cell phones, for example, have been increasing in size and complexity even more rapidly. In fact, memory requirements for the applications running on cell phones have been doubling every fifteen months [20]. A combination of more powerful applications and the economic factors constraining memory sizes makes it important to consider techniques to optimize memory usage.

Compression is a popular technique used to make more effective use of memory space [28]. Compression techniques exploit the redundancies in the original data and represent it as accurately as possible using the fewest number of bits. However, many memory compression schemes (e.g., [19, 14]) treat data entities in the memory as structureless streams. To retrieve information from a compressed data entity, we first have to decompress the entire (or part of) entity, which may incur both performance and space overheads. Consequently, data compression needs to be applied judiciously so that the benefits accrued are larger than the overheads imposed.

In many Java applications, a large fraction of objects in the heap are similar to each other. In fact, a small set of values tend to appear in some fields of the heap-allocated objects much more frequently than other values. This small set of values are the frequent values for these fields (or, fre-

quent field values for short). This similarity among objects can be exploited to reduce the heap space. For example, the schemes proposed in [25, 24, 13] compare the content of an object with a set of existing objects and replace the similar ones with a single copy in order to reduce the storage space. However, this is done mainly as a *manual operation*. Further, such schemes do not exploit the similarity that exists in individual fields but not across the entire object. The work presented in this paper first analyzes the similarity in fields across a set of objects. Based on this analysis, we then propose two frequent field value based object compression schemes that exploit field-level similarity to reduce heap space requirements of Java applications without manual optimization.

This paper makes the following contributions:

- The similarity in data values of the fields of heap allocated objects are quantified using the SpecJVM98 benchmark suite.
- We present a strategy to identify the fields that contain frequent values and explain how this information can be used for restructuring object formats to reduce the memory space occupied by Java objects.
- We propose and evaluate two object compression schemes that exploit frequent field values to reduce heap space consumption of Java applications. The first scheme is oriented towards compressing the fields that hold zero or null values. The second scheme further reduces heap space occupancy by allowing multiple objects to share memory space for the fields that contain non-zero frequent values.
- We present details of our implementation of these schemes. Our extensive experimental evaluation demonstrates that these schemes can reduce heap space occupancy by up to 24% (14% on an average). The performance overheads incurred by our schemes are within 2% for most of the benchmarks.

The rest of this paper is organized as follows. Section 2 presents a characterization of frequent field values. Section 3 presents our two schemes that exploit frequent field values for reducing heap space requirements, and gives their implementation details. Section 4 discusses our experimental results, focusing on reducing heap space requirements and performance overheads. Section 5 reviews related work, and finally, Section 6 concludes the paper with a summary of our major observations.

2. FREQUENT FIELD VALUE CHARACTERIZATION

This section characterizes the frequent field values in Java applications, and identifies opportunities for exploiting the results of this characterization for reducing the heap memory space required to store object instances in embedded Java environments. Our optimization targets only object instances, not arrays.

2.1 Experimental Setup

We use the SpecJVM98 benchmark suite [7] to study the existence of frequent field values. This benchmark suite consists of eight Java programs. These benchmark programs can be run using three different inputs, which are named as s1, s10, and s100. We present our field value characterization results for s1 and s10. Since the frequent field value characteristics with these two input sets are quite similar (as will be shown shortly), we present the results of our two proposed schemes using the s1 data set only, and perform a sensitivity analysis with s10.

We use an instrumented JVM based on Kaffe VM 1.1.4 [4] to collect the execution trace of each benchmark. The traces include detailed information about each object allocation and access. Our trace-based simulator simulates the execution of each benchmark, and provides information about the memory savings and performance overheads. The important characteristics of our applications are given in Table 1. The third column of this table gives the number of classes loaded, the fourth column shows the number of object instance creations, and the fifth column gives the average size of an object instance for each application. The sixth column shows the execution cycles obtained by executing our applications using Sun JDK 1.4 (with Hotspot execution engine client version [9]) on a Solaris system with SPARC V9 microprocessor. The execution cycles are obtained through the performance counters available in the microprocessor. These cycles are referred as the *base results* in the rest of this paper. We later quantify how much overhead the different schemes we evaluate incur over these base execution cycles. Finally, the last two columns give the maximum heap occupancy without and with arrays. In this work, the "maximum heap occupancy" is defined as the maximum total size of the live objects (and arrays) at any given point during execution. Note that this value determines the minimum heap size that the application requires to run without an out-of-memory exception.

2.2 Existence of Frequent Field Values

To quantify the extent of the frequent field values existing in Java applications, for each value v that may appear in the j^{th} field of class C_i, we maintain an occurrence counter $K_{i,j,v}$. At every 1KB of memory allocations, we scan the entire heap and, for each scanned instance o of class C_i where $o.f_j = v$, we increase the counter $K_{i,j,v}$ by 1. Note that a value that remains in a particular field of an object for longer than the sampling interval may be observed multiple times. Therefore, the number of occurrences is determined by both the number of objects that contain the particular value and the duration of time that this value remains in each object. For the j^{th} field of class C_i, let us assume that:

$$K_{i,j,v_{i,j}^{(1)}} \geq K_{i,j,v_{i,j}^{(2)}}, \geq, ..., \geq K_{i,j,v_{i,j}^{(n)}}$$

where

$$\{v_{i,j}^{(1)}, v_{i,j}^{(2)}, ..., v_{i,j}^{(n)}\}$$

is the set of n values that appear in the j^{th} field of an instance of class C_i. Particularly, $v_{i,j}^{(1)}$ is the most frequent value for the j^{th} field of class C_i. For a given program, we define the distribution of the k^{th} frequent value as:

$$D_k = \frac{\sum_{\forall i,j} K_{i,j,v_{i,j}^{(k)}}}{\sum_{\forall i,j,v} K_{i,j,v}}.$$

Figure 1 shows the distribution of the occurrences of the top five frequent values for each benchmark for the s1 and s10 input sets. In this figure, "1st", "2nd", "3rd", "4th", and

Benchmark	Description	Number of Classes	Number of Instances	Average Size of Instances	Execution Cycles (10^6)	Maximum Heap Occupancy	
						Instance	Array+Instance
compress	LZW-based compression	199	3101	24B	59665.7	54KB	10503KB
jess	An expert shell system	352	32005	24B	2019.6	156KB	374KB
raytrace	Single-threaded ray-tracer	213	236555	20B	6170.9	2395KB	3549KB
db	Database query	199	4545	24B	607.7	66KB	216KB
javac	Java compiler	352	21564	23B	1827.5	283KB	643KB
mpegaudio	MPEG audio decoder	238	4779	22B	7545.7	75KB	314KB
mtrt	Multi-threaded ray-tracer	213	236492	20B	6171.3	2395KB	3548KB
jack	Java parser generator	246	234161	27B	6675.5	317KB	782KB

Table 1: Important characteristics of the benchmarks used in this study.

"5th" represent D_1, D_2, D_3, D_4, and D_5, respectively. One can observe from these results that, for most of the benchmarks, the most frequent value accounts for about 90% of the total number of occurrences. Only in benchmarks raytrace and mtrt do we observe that the most frequent value accounts for only about 50% of the total. This implies that, for such applications, one may need to go beyond the most frequent values (if we are to compress a significant fraction of the objects in the heap). Among all the frequent field values, the value zero (or null for reference fields) is of particular interest because zero fields can be easily eliminated, thereby saving memory space. The bar-chart in Figure 2 gives the breakdown of zero and non-zero most frequent values for each benchmark in the SpecJVM98 suite. We see from these results that non-zero values account for 55% of the most frequent values on the average. This means that a scheme that tries to exploit frequent values should accommodate for both zero and non-zero values.

While these results are encouraging from the perspective of potential memory space optimizations, one might also be interested in understanding why such frequent field values exist. To answer this, we studied the application codes in the SpecJVM98 benchmark suite, and found that frequent field values can occur due to many different reasons. As an example, in the benchmark raytrace, we found that the instances of class spec.benchmarks._205_raytrace.OctNode are observed 24,917,836 times during profiling. Each instance of this class represents a region of three-dimensional space that contains a certain number of three-dimensional objects. An instance of this class has five fields, two of which, ObjList and NumObj, are of particular interest. ObjList is a pointer to the header of a link table of the three-dimensional objects in this region, and NumObj is the number of the objects. Since objects are typically distributed in the three-dimensional space sparsely, many regions in the space are actually empty. We observed that, for 36% of the observed times of the instances of this class, these two fields contain null and zero, respectively. And, throughout its execution, the application creates 3,583 instances of this class, up to 2,995 of which can co-exist in the heap at a given time. This example shows how frequent field values can exist in a typical Java application.

2.3 Opportunities for Storage Optimizations

An important question now is how one can exploit these results for memory space optimization. There are at least two ways to achieve this. The first method is based on providing feedback to the programmer. More specifically, using a suitable interface, the field value characteristics discussed above can be presented to the application programmer. The programmer in turn may rewrite/restructure the application code based on these characteristics. For example, if, for a given class, a subset of the fields always have the same value, the application programmer can consider making these fields *static,* and consequently make such fields associated with the class rather than each object instance. There are also automatic static tools, such as JAX [30], that statically analyze the application codes to remove from the class files the fields that are not used by the application. Such tools may also statically identify the fields that all read of these fields observe the same value across all the instances and make these fields static. We refer to the techniques in this category as *user-level* space optimization since they are performed at the user level, and they do not need special support from JVM. A common problem with the user-level optimization techniques is that the optimizations must be conservative, and consequently, they may not be able to catch all the space optimization opportunities. To evaluate the upper bound (i.e., the maximum potential memory savings) that could be achieved by such user-level space optimizations, one can assume that the profile represents the behavior of the application 100% accurately. Based on this assumption, we modified the fields that contain a single value across all the instances to be static, and we also removed the fields that are never accessed from the class. The heap occupancy behavior of our benchmarks with this user-level optimization will be presented later in the paper and compared to our two schemes. However, in this work, we mainly focus on exploiting the information about frequent field values within the virtual machine. That is, our schemes can reduce memory space requirements of Java applications without rewriting the application code. Note that the applications that are already optimized using user-level techniques can still benefit from our optimization schemes. In the next section, we discuss our optimization schemes in detail.

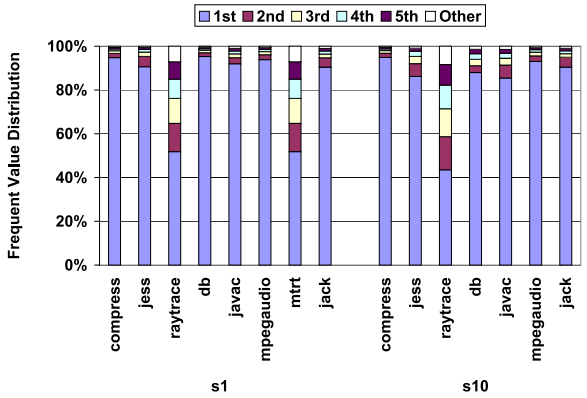

Figure 1: Frequent value distribution (the top five and the rest). Left: s1 input; right: s10 input.

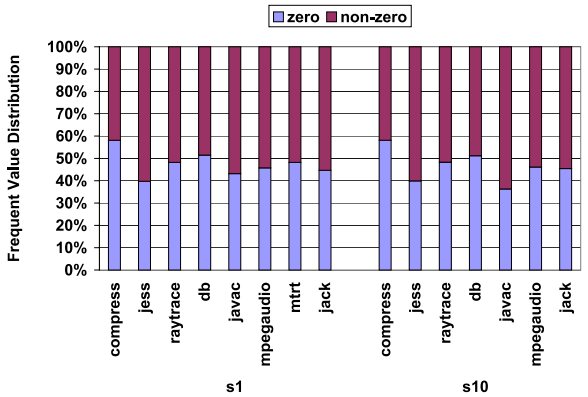

Figure 2: Zero and non-zero frequent values. Left: s1 input; right: s10 input.

3. OUR COMPRESSION APPROACH

A closer look at the problem of taking advantage of frequent field values reveals that, in order to do a good job, one needs two kinds of information: the fields that have small numbers of frequent values and the frequent values themselves. Our experience with different input sets (e.g., s1 and s10) indicates that, while the values themselves may change from one input set (execution) to another, the fields having frequent values do not change significantly. This is understandable since the fields with frequent values are usually shaped by the characteristics of the application, rather than the particular input set used. On the other hand, the field values themselves depend strongly on the input set used in a particular execution. As a result, one can use profiling with a typical input set to determine the fields that have small numbers of frequent values, and this information can then be encoded within so called *field description files* that could be distributed together with the class files of the application. We prefer using separate field description files instead of annotating fields in the class files because it is not possible to directly annotate the classes that belong to the class library. The virtual machine loads the field description files together with the class files to appropriately annotate the fields with frequent values. This is the approach employed by both the strategies presented in this paper. The details of the process used to determine the fields that hold frequent values are explained in Section 3.1.

3.1 Determining the Level of Each Field

In this subsection, we explain how we use profiling results to identify the fields that are likely to hold frequent values at runtime. Specifically, based on the profiling information, we classify the object fields into three levels:

- **Level-0**: the field does not have a dominant frequent value;
- **Level-1**: the field has a non-zero (or non-null for reference fields) frequent value; and
- **Level-2**: the field has a frequent value that is zero or null.

It should be noted that this classification is performed offline. The results of the classification are stored in the field description file and will be used in the the future executions of the program. Since the actual input to the application during the execution may be different from the input used during the profiling execution, the profiling results may not 100% accurately reflect the behavior of the application during the actual execution. The inaccuracy in the profiling results may affect the performance and the amount of the memory savings when using our schemes, however, it does not cause the program to run incorrectly.

Let us assume that class C_i has fields $F = \{f_1, f_2, \ldots, f_n\}$, and S_i is the set of subclasses of C_i. For a field $f_j \in F$, we define:

$$q(i,j) = \sum_{C_x \in S_i \cup \{C_i\}} K_{x,j,v^{(1)}_{x,j}}.$$

We select a subset of fields, $F^* \subseteq F$, such that:

$$|F^*| \min_{f_j \in F^*} q(i,j)$$

is maximized, where $|F^*|$ is the number of fields in F^*.[1] Fields in F^* are the candidates for the level-1 and level-2 fields. Another applicable rule for selecting candidates for the level-1 and level-2 fields will be discussed in Section 4.2.1. A field is considered to be a level-1 field if it belongs to F^* and the most frequent value of this field is non-zero. On the other hand, a field is considered to be a level-2 field if it belongs to F^* and the most frequent value of this field is zero (or null for pointer filed). In addition, field f_i is a level-k field in class C_i, it must be level-k in all the subclasses of C_i. Therefore, if the level in the subclass conflicts with that in the super-class, the level of the super-class overrides that of the subclass. This is necessary since the type of an object may be implicitly cast into its super-class.

We illustrate the procedure of field classification with an example. Let us assume that we have three classes, namely, C_x, C_y and C_z, and that both C_y and C_z are subclasses of C_x. Assume further that class C_x has fields $C_x.f_1$, $C_x.f_2$ and $C_x.f_3$, and both C_y and C_z have five fields, three of which (f_1, f_2 and f_3) are inherited from C_x. By profiling the application using our instrumented JVM, we obtain the value of q defined above for each field of each class. (see Table 2). We now show how we determine the level of each field of class C_x. Let us consider the following subsets of F (the field set of C_x):

$$F_1 = \{f_1, f_2, f_3\};$$
$$F_2 = \{f_1, f_2\};$$
$$F_3 = \{f_1\}.$$

Using the profile data in Table 2, we have:

$$|F_1| \min\{q(x,1), q(x,2), q(x,3)\} = 3 \times 2000 = 6000;$$
$$|F_2| \min\{q(x,1), q(x,2)\} = 2 \times 8000 = 16000;$$
$$|F_3| \min\{q(x,1)\} = 1 \times 10000 = 10000.$$

Since $16000 > 6000$ and $16000 > 10000$, for class C_x, we obtain:

$$F^*_x = F_2 = \{f_1, f_2\}.$$

Since $f_1 \in F^*_x$ and its most frequent value is 0, f_1 is classified as level-2. Field f_2, however, is level-1 as its most frequent value is non-zero. The fields that are not in F^*_x are made level-0. Similarly, for class C_z, we have:

$$F^*_z = \{f_1, f_2, f_3, f_4, f_5\}.$$

[1] F^* is actually the prefix of a list of the fields in the descending order of $q(i,j)$.

Class C_x				Class C_y: extends Class C_x				Class C_z: extends Class C_x			
Field	$q(x,i)$ Occurrences	$v^{(1)}_{x,i}$	Level	Field	$q(y,i)$ Occurrences	$v^{(1)}_{y,i}$	Level	Field	$q(z,i)$ Occurrences	$v^{(1)}_{z,i}$	Level
$C_x.f_1$	10000	0	2	$C_y.f_1$	7000	0	2	$C_z.f_1$	3000	0	2
$C_x.f_2$	8000	2	1	$C_y.f_2$	6000	2	1	$C_z.f_2$	2000	2	1
$C_x.f_3$	2000	0	0	$C_y.f_3$	2000	0	0	$C_z.f_3$	1900	3	0
				$C_y.f_4$	5000	4	1	$C_z.f_4$	2000	0	2
				$C_y.f_5$	4000	0	2	$C_z.f_5$	2000	1	1

Table 2: Determining the potential levels for the fields of three example classes.

Note that, although we have $f_3 \in F^*_z$, this field is still classified as level-0 since it has been determined to be so in the super-class C_x.

As discussed earlier, after this profiling, we create field description files and attach them to class files. During application execution, the VM checks the field description files and uses the information there to decide the object formats, which is discussed in detail in the rest of this section. Also, in Section 4.2.1, we explain and evaluate an alternate scheme for determining the level of fields.

3.2 Scheme-1: Eliminating Level-2 Fields

This scheme removes the level-2 fields from the objects whose level-2 fields contain only zeros to save memory space. Figure 3 shows the formats of an object in both uncompressed and compressed formats. An object is divided into two parts: the primary part containing level-0 and level-1 fields, and the secondary part containing level-2 fields. Each memory block allocated in the heap is associated with a one-bit flag (C). If $C = 0$, the block contains the primary part of a compressed object or the secondary part of an uncompressed object. The rest of the first word (four bytes) of this block is the GC Header (i.e., GCHeader1 in Figure 3), which contains information (such as the size of the block) needed by the garbage collector. If $C = 1$, the block contains the primary part of an uncompressed object. The remainder of the first word contains SPtr, a pointer to the secondary part of this object. The GC Header of the primary part of an uncompressed object is stored in the secondary part of this object (i.e., GCHeader2 in Figure 3).

When a "NEW" instruction in the program creates an object, only the primary part is allocated. The secondary part is lazily allocated when the first non-zero value is written into one of the level-2 fields of this object. During garbage collection, the collector removes the secondary parts of the uncompressed objects whose level-2 fields contain only zero values.

At any point during execution, an object can be either in uncompressed or in compressed format. Checking the current format of the object at each field access incurs some performance overhead. However, it should be noted that the level of each field is statically determined before the execution starts. A JIT compiler can use this information to avoid the format checking overheads in most of the cases. An interpreter can also avoid a significant portion of this overhead by marking each "getfield" or "putfield" instruction according to the level of the field being accessed. Such a marking is performed when this instruction is first executed. For example, when a "getfield#n" instruction (loading the n^{th} field from an object) is executed for the first time, we replace this instruction with a customized instruction "getfield_0_1#n" (or "getfield_2#n") if the n^{th} field of the object being accessed is of level-0 or 1 (or level-2). When a "getfield_0_1#n" is executed by the interpreter, we can simply load the value from the field using the object reference and the offset of the field without checking the current format of the object. To execute "getfield_2#n", however, we first load into a register the first word (containing the flag C and the pointer SPtr) of the primary part of the object being accessed, and then check the value of flag C. If $C = 0$, we know that the value in the field being accessed is zero; otherwise, we need to load the value of this field from the secondary part of this object. We mark "putfield#n" as "putfield_0_1#n" or "putfield_2#n" in the same manner. Similar to the case with "getfield" instructions, "putfield_0_1#n" instructions do not incur performance overhead due to compression. To execute a "putfield_2#n" instruction, however, we need to check the current format of the object. If the object is compressed and the value to be written is zero, we skip the write access since it is not necessary. If the object is not compressed, we write the value into the secondary part of the object. If the object is compressed and the value is not zero, we have to allocate the space for the secondary part of this object and then write the value into this part.

Compared to the original JVM implementation, our scheme incurs extra memory accesses in the following cases: (1) reading a level-2 field of an uncompressed object; (2) writing a value to a level-2 field of an uncompressed object; and (3) writing a non-zero value to a compressed object (this also involves allocating memory for the secondary part of the object being accessed). Since most level-2 fields contain zero, and most values written to level-2 fields are zero, these cases do not happen frequently. Therefore, we can expect the overall performance degradation due to our scheme to be low. Further, compared to the memory space allocated for each object in the original implementation of JVM, our scheme allocates two more words for each uncompressed object. This space overhead is amortized by the memory savings achieved by the compressed objects. That is, since most of the objects are compressed, our scheme reduces the overall heap memory requirement of an application.

Our scheme requires "putfield" and "getfield" be atomic operations. This incurs extra synchronization overheads for the JVM implementations where Java threads are mapped to native threads. Fortunately, most JVMs for embedded systems schedule Java threads by themselves without mapping Java threads to the native threads. For such JVMs, we can avoid the synchronization overheads by not preempting a Java thread when the thread is executing a "putfield" or "getfield" instruction. It should be noted that, most JVM implementations for embedded systems, such as KVM [3], schedule threads only between the boundaries of bytecode instructions, and thus never preempt a Java thread when the thread is executing an instruction.

3.3 Scheme-2: Sharing Level-1 Fields

The scheme explained in the previous subsection removes the level-2 fields from some objects to save memory space.

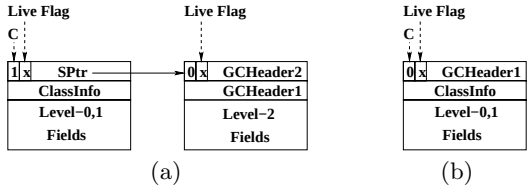

Figure 3: Object formats for Scheme-1. (a) Uncompressed. (b) Compressed.

In this section, we extend this scheme by sharing level-1 fields among multiple objects. Figure 4 shows the object formats used in our level-1 field sharing scheme: *uncompressed, compressed,* and *shared.* Figure 4(a) shows an uncompressed object. This object has two parts: the primary part containing the level-0 fields, and the uncompressed secondary part containing both level-2 and level-1 fields. The first word in the primary part contains two one-bit flags (U and C) and a pointer to the secondary part (SPtr). For an uncompressed object, we have $U = 1$ and $C = 1$, indicating that this object does not share the level-1 fields with any other objects, and that the secondary part of this object contains both level-1 and level-2 fields. It should be noted that the pointer SPtr points to the middle of the secondary part. The level-1 fields are stored in the locations with positive offsets, while the level-2 fields are stored in the locations with negative offsets. The level-2 fields of an object can be removed to save memory space if all these fields contain zeros. Figure 4(b) depicts the format of a compressed object, i.e., an object with its level-2 fields removed. In this format, the secondary part of this object contains only the level-1 fields. Multiple compressed objects whose level-1 fields contain the same values can share the same secondary part (Figure 4(c)) to reduce the overall memory space consumption.

The formats presented in Figure 4 allow us to read and write the level-0 fields of an object in the same manner, irrespective of the current format of the object. Accessing a level-1 field of an object involves loading into a register the pointer SPtr and the flags U and C from the primary part. To read a level-1 field, there is no need to check the current format of the object. To write a value into a level-1 field, however, we need to check flag U. If $U = 1$, the secondary part of this object is not shared and we can write the value to the field. If $U = 0$, on the other hand, the secondary part of this object is shared with other objects and we have to create an unshared secondary part (compressed, containing only level-1 fields) for this object. To do this, we allocate a memory block large enough to hold the level-1 fields of this object and copy the values of the level-1 fields to this block from the shared secondary part. After this, we write the value into the field in the newly-created secondary part. Note that, the pointer SPtr and the flag U in the primary part of this object should also be updated.

To read a level-2 field of an object, we first check the state of flag C. If $C = 0$, this indicates that the value of this field is zero. If $C = 1$, however, we have to load the value from the secondary part of the object. To write a value to a level-2 field, we first check the state of C. If $C = 0$ and the value to be written is zero, we skip the write operation since it is unnecessary. If $C = 1$, we write the value to the field in the secondary part of the object. If $C = 0$ and the value to be written is non-zero, we have to create an uncompressed

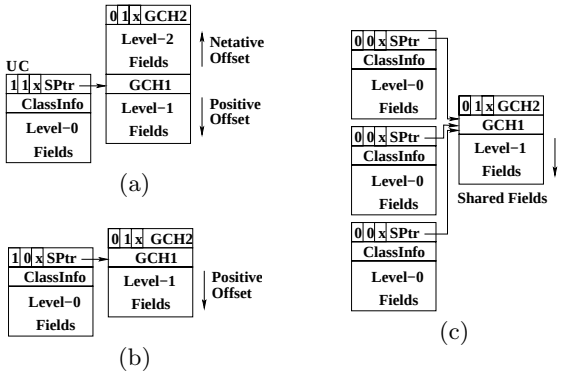

Figure 4: Object formats for Scheme-2. The first two bits of an object are C and U flags, respectively; and the third bit is the mark indicating if an object is live or dead. (a) Uncompressed. (b) Compressed. (c) Shared.

secondary part (containing both level-1 and level-2 fields) for this object. To do this, we allocate a memory block large enough to hold both level-1 and level-2 fields of this object, and then initialize this block by setting all the level-2 fields to zero and copying the values of the level-1 fields from the original secondary part of the object. Of course, the pointer SPtr and the flags U and C in the primary part of this object should be updated. After the uncompressed secondary part is created, we write the new value into the field specified by the instruction.

In our implementation, each object is created in the compressed format (Figure 4(b)). During execution, it may be expanded into "uncompressed" format or further compressed into the "shared" format. A Mark-Sweep-Compact-Compress garbage collector is invoked when the free space in the heap is insufficient for creating a new object. This collector not only collects dead objects, but also compresses objects by eliminating level-2 fields. Specifically, when marking the primary part of an uncompressed live object during the mark phase, the collector also checks the values of the level-2 fields in the secondary part of this object. If all the level-2 fields of this object contain zero values, the collector splits the secondary part of this object into two blocks: the block containing the level-2 fields and the block containing the level-1 fields. The former block is not marked so that it can be swept in the following sweep phase. The latter block is marked as live since it is the compressed (but not shared) secondary part of a live object.

If the Mark-Sweep-Compact-Compress garbage collector cannot collect sufficient space for the new object, we scan the heap using an additional pass to find the compressed objects that can share their secondary parts. In our approach, only the objects of the same class can share a compressed secondary part with each other. To identify the objects that can share their compressed secondary parts, we maintain n frequent value pointers $(p_i, i = 1, 2, ..., n)$ for each class. Further, each frequent value pointer (p_i) is associated with a counter (c_i). A frequent value pointer p_i of class C_x either is null or points to the secondary part of an object of class C_x. We scan the heap, and, for each object O of class C_x in the heap, we compare its secondary part field-by-field against each secondary part that is pointed by a frequent value pointer of class C_x. The counters associated with the

frequent value pointers that point to a secondary part not identical to that of O are decreased by one, and, if counter c_i is less than a threshold N, we set the corresponding frequent value pointer p_i to null. On the other hand, if there is a frequent value pointer p_i pointing to a secondary part that is identical to that of O, we increase c_i by one, and let O share the secondary part pointed to by p_i. If we cannot find any matches for O and there is a frequent value pointer p_i whose value is null, we let p_i point to the secondary part of O, and initialize the counter associated with p_i to zero. In our experiments, we assumed that $n = 3$ and $N = -3$.

4. EXPERIMENTAL RESULTS

In this section, we present the results from our experimental evaluation. Our presentation is in two parts. First, in Section 4.1, we present our baseline results. Then, in Section 4.2, we conduct a sensitivity analysis by modifying the parameters/strategies used in the baseline experiments. As mentioned in Section 2.1, we use a trace-based simulator to evaluate the memory behavior of a JVM. Our simulator maintains a heap and allocates objects in this heap as a JVM does. It invokes garbage collector/compressor when the heap space is used up. It also reads and writes the contents of object fields as captured by the trace file. Therefore, the heap memory access behavior of our simulator is very close to that of JVM.

4.1 Baseline Results

We present the maximum heap occupancy results in Figure 5. Recall that the heap occupancy is the sum of the sizes of all the live objects at any given moment, and the maximum heap occupancy gives the minimum heap size needed to run the application without giving an out-of-memory exception.[2] The y-axis in this figure represents the values *normalized* with respect to the maximum heap occupancy of the original JVM without any object compression. *All space overheads incurred by each scheme are included in these results.* User-level optimization reflects the memory saving potential of the static analysis based space optimization scheme. This scheme can reduce the sizes of some class files by removing the fields that are not used by the application. Consequently, the memory space for storing the loaded classes can be reduced. Further, if these classes are instantiated, the size of each instance can also be reduced. It should be noted that, in order to ensure the correctness of the optimized program, the static analysis based optimizations must be conservative. In Figure 5, we observe that user-level optimization, Scheme-1, and Scheme-2 reduce the space for storing object instances by 7%, 26%, and 38% on

[2]Depending on the garbage collection algorithm used, a JVM may need larger heap memory than the maximum heap occupancy to execute a Java application without an out-of-memory exception. However, no JVM can execute a Java application without out-of-memory exception when the heap size is smaller than the maximum heap occupancy of the application. Since our schemes compress objects during garbage collection, the frequency of garbage collection invocations affects the value of the maximum heap occupancy. To find the lower bound of maximum heap occupancy for each scheme, we execute each benchmark many times with different heap sizes. The minimum heap size that allows the benchmark to execute without out-of-memory exception gives the maximum heap occupancy values presented in Figure 5.

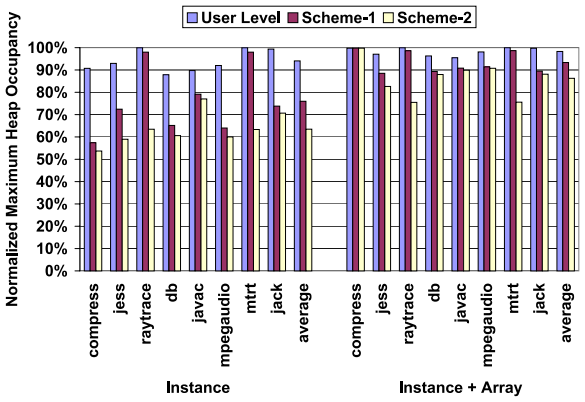

Figure 5: Maximum heap occupancy. The y-axis represent the values, normalized with respect to the original maximum heap occupancy without any object compression.

average, respectively. Although we only present the characterization of the heap occupancy of object instances and arrays, we can still conclude based on these results that our schemes can be applied to the applications that have already been statically optimized using user-level optimizations to further reduce heap memory requirements. When we consider both arrays and instances, we observe that there is a drop in savings as compared to just instance size reduction. This is because of the large size of the arrays involved in some of these applications. In particular, none of the approaches achieve any significant savings in the benchmark compress, which is dominated by a few large arrays. Still, the average maximum heap occupancy savings achieved by the user-level optimization, Scheme-1, and Scheme-2 are 2%, 7%, and 14%, respectively.

While the savings in maximum heap occupancy are important, there are also cases where the average heap occupancy can be critical to consider. For example, this could provide more opportunities for energy savings in a multi-banked memory based system [15], by increasing the number of memory banks that can be turned off at a given time. Therefore, it is also important to consider the heap usage profile over the course of execution. Figure 6 gives this profile for two representative benchmark codes, jess and raytrace, with the original JVM and the JVM with our Scheme-2. In obtaining these results, each scheme was executed using the minimum size heap with which it could complete execution. We observe that the JVM with Scheme-2 consistently utilizes a smaller heap space as compared to the original JVM. The bar-chart in Figure 7 shows the normalized average heap occupancy for all the benchmarks when using Scheme-2. We see that the average heap occupancy saving is about 10%, even when considering both object instances and arrays. That is, the proposed scheme reduces both maximum heap occupancy and average heap occupancy.

Because our experiments are based on simulation, it is not possible for us to obtain 100% accurate information on the execution time overheads incurred by our schemes. However, we estimate the performance overheads of our schemes by counting the number of the executed instructions and the number of memory accesses for performing the extra operations that are due to our schemes. For example, in

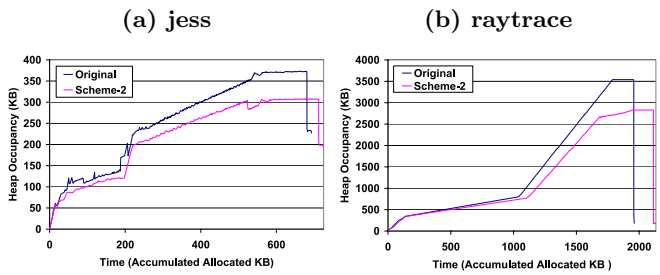

Figure 6: Heap occupancy profiles for two benchmarks with s1 input: jess and raytrace.

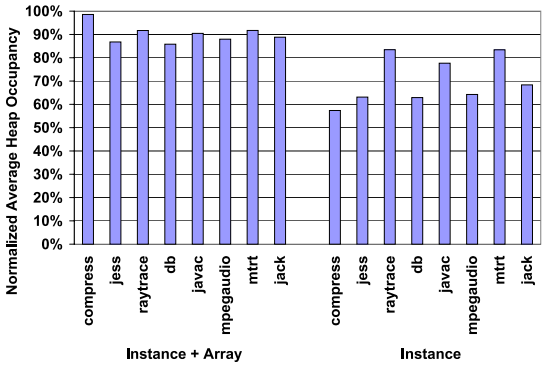

Figure 7: Average heap occupancy for Scheme-2. The y-axis are normalized with respect to the original average heap occupancy without any object compression.

Scheme-1, to access a level-2 field of an object, we need three extra operations: loading flag C and pointer SPtr into a register, executing a conditional branch based on the value of flag C, and loading the value from the secondary part of the object (if $C = 1$) or returning a zero (if $C = 0$). We assume that each instruction incurred by our schemes is executed in one cycle and each extra memory access requires an extra cycle. Figure 8 shows estimated performance overheads incurred by Scheme-1 and Scheme-2. The y-axis in this bar-chart gives the numbers of extra execution cycles introduced by our schemes, which are normalized with respect to the baseline execution cycles shown in the sixth column of Table 1 (i.e., the execution cycles obtained by running the benchmarks using JDK 1.4 with Hotspot engine client version). Each bar in this figure is broken down into four parts. The first part (denoted COMPRESS) gives the time spent in compressing objects (such as checking if each level-2 field contains zero, and finding the objects that can share the same secondary part in Scheme-2), the second part (denoted EXPAND) gives the time spent in expanding objects (such as allocating memory for the secondary part of the compressed objects). The last two parts capture the extra overheads due to putfield and getfield operations. It should be noted that, the extra execution cycles for COMPRESS and EXPAND are affected by the heap size – the larger the heap size, the less frequent compressions and decompressions. To estimate the maximum overheads, we simulate each benchmark with the minimum heap size that allows the benchmark to complete its execution with the specific compression scheme. From these results, we observe that the

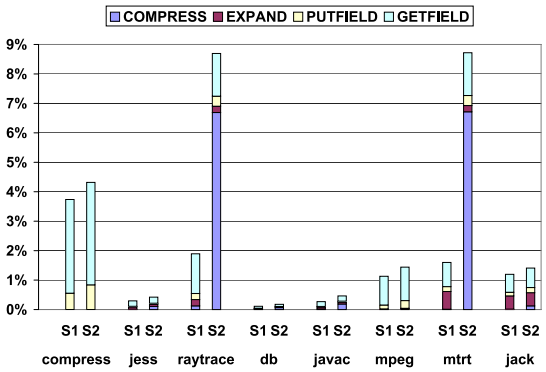

Figure 8: Percentage increases due to Scheme-1 (S1) and Scheme-2 (S2) in execution cycles over the base results. Each scheme is simulated with the minimum heap size that allows the benchmark to complete its execution.

extra execution cycles due to Scheme-1 is marginal (less than 3.7% for all the benchmarks), and the numbers of the extra execution cycles introduced by Scheme-2 are slightly greater than those of Scheme-1 for most of the applications. For benchmarks raytrace and mtrt, however, Scheme-2 incurs a performance overhead of about 8.6%. This is mainly due to the fact that Scheme-2 compares the level-1 fields of a large number of objects to find the objects that can share level-2 fields (Figure 1 confirms this observation). If we assume that each extra memory access due to our schemes costs two cycles, the average overheads for Scheme-1 and Scheme-2 are 1.8% and 4.2%, respectively. This estimation may not be accurate for high performance systems with deep-pipelined processor core and multiple-level caches. However, for low-end embedded environments at which our heap compression techniques are targeting, counting the number of memory accesses and instructions can give a reliable estimation of the performance overheads incurred by our schemes with a reasonable accuracy. For example, a widely used processor for today's mobile phones is ARM7TDMI [1], which has a 3-stage pipeline, and no cache. The maximum frequency of this processor is 100MHz at 0.13um technology (133MHz at 0.13um). Note that the length of a cycle at this frequency is close to the access delay of typical SDRAM today. Further, for 3-stage pipeline, the branch penalty would normally be small. As a result, estimating the performance impact by counting the number of instructions and memory operations is expected to be reasonably accurate in practice.

4.2 Sensitivity Analysis

In this subsection, we vary some of the parameters and strategies used for obtaining the baseline results. The objective is to test the robustness of the proposed approach.

4.2.1 Impact of the Field Selection Scheme

Recall that our default method for classifying the fields of class C_i is to select $F^* \subseteq F$ (where F is the set of the fields of class C_i) such that the value of:

$$|F^*| \min_{f_j \in F^*} q(i,j)$$

is maximized. Let us refer to this method as the "minimum-based" scheme. In this subsection, we experiment with a

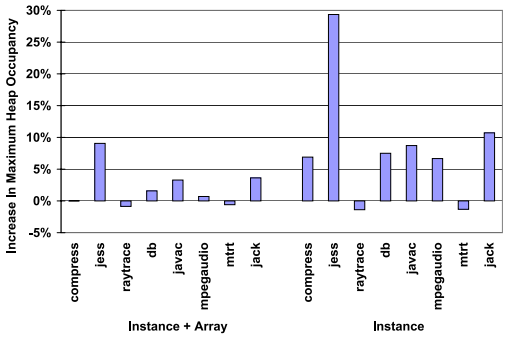

Figure 9: Increase in the maximum heap occupancy of Scheme-2 when a product-based potential field selection scheme is used. The percentage increases are with respect to those obtained with the minimum-based (default) scheme.

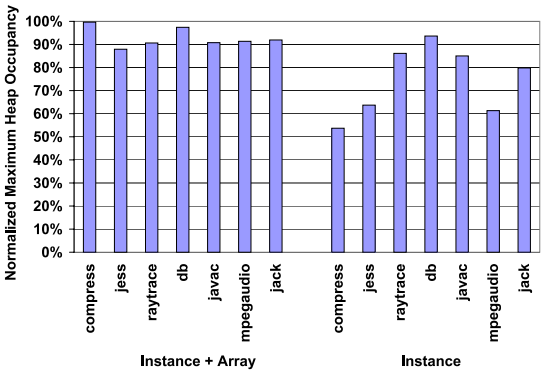

Figure 10: Maximum heap occupancy of Scheme-2 with the s10 input. The y-axis represent the values, normalized with respect to the maximum heap occupancy of the original applications without any object compression. The level of each field is determined using the profile information obtained from the s1 input.

"product-based" field classification scheme, in which the fields of class C_i are classified by maximizing:

$$|F^*| \prod_{f_j \in F^*} \frac{q(i,j)}{N}, \text{ where } N = \sum_{f_j \in F} q(i,j).$$

The minimum-based selection scheme is based on the optimistic assumption that the most frequent values of the different fields tend to co-exist in the same object. In other words, we assume that, for class C_i, the set of objects where $f_1 = v_1$ is the subset of the set of objects where $f_2 = v_2$ if $q(i,1) \leq q(i,2)$, where v_1 and v_2 are the most frequent values for f_1 and f_2, respectively. In comparison, the product-based scheme is based on the assumption that each field assumes its most frequent value independently from the other fields of the same object. Figure 9 presents the percentage increase in the maximum heap occupancy due to the product-based Scheme-2 over the heap occupancy of the minimum-based Scheme-2. One can observe that, for most of the benchmarks, the product-based scheme increases the maximum heap occupancy. This indicates that the most frequent values of the different fields *do* co-exist in the same object. Therefore, these results suggest that our default method seems to work better. In fact, although not presented here in detail, we also found during our experiments that the minimum-based scheme incurs less performance overhead than the product-based scheme.

4.2.2 Robustness of the Profiling-Based Approach

In Section 3, we mentioned that, while the values themselves may change from one input set (execution) to another, the fields with frequent values do not change significantly. To demonstrate this, we run our benchmarks with the s10 input; the level of each field, however, is determined using the profile information obtained from s1 input. Figure 10 presents the maximum heap occupancy of each benchmark using Scheme-2. On an average, with s10, we achieve 35% reduction in the maximum heap occupancy for object instances. With arrays included, we still achieve 8% reduction in the maximum heap occupancy on the average. Recall that the corresponding values with s1 were 38% and 14%. Therefore, we can conclude that our profiling-based approach performs well across the different input sets.

5. RELATED WORK

Embedded virtual machines are being increasingly used in many embedded and mobile environments, and commercial implementations [5, 2, 11] have now been around for some time. In addition to these, McDowell el al. [25] presented a Java environment that supports the complete Java language and all the core Java packages except AWT using as little as 1MB of RAM. TinyVM [8] is an open source Java platform for the Lego Mindstorms RCX microcontroller. TinyVM's footprint is about 10KB in the RCX. Shaylor [29] implemented a Java JIT compiler for memory-constrained low-power devices. Their implementation requires 60KB of the ARM machine code. An example non-Java small-footprint virtual machine is Maté [23], a tiny communication-centric virtual machine designed for sensor networks. While memory footprint reduction has been one of the objectives of many of these efforts, to our knowledge, none of them has considered compressing heap objects by exploiting the frequent field values.

Many embedded systems employ data/code compression to reduce memory space requirements, power consumption, and the overall cost of the system. In the domain of code compression, instruction compression has been an active area of research for the last decade or so [21, 22]. Clausen el al. [16] proposed compressing bytecodes by factoring out common sequences. A different line of work [26] tried to reduce the space occupied by bytecode sequences using a new format to represent files. Debray and Evans [17] used profiling information to guide code compression. The main focus of the work in [19] is to reduce leakage energy consumption by turning off memory banks saved by compressing class libraries. In addition to these studies, memory compression has also been adopted in high-end systems. For example, Rizzo [27] presented a fast algorithm for RAM compression. Similarly, Franaszek el al. [18] developed a set of algorithms and data structures for compressed-memory machines, effectively doubling the available memory capacity. The study discussed in this paper is different from these prior studies in that it targets reducing heap space by exploiting frequent field values in Java objects.

In [14], Chen el al. proposed heap compression techniques to reduce the size of the heap memory. Their compression scheme treats each object as a structureless byte stream. Therefore, a compressed object must be decompressed before its contents can be accessed. The work described in this paper is different from that in [14] in that our new object compression schemes are aware of the structures of objects, which allows the contents of a compressed object to be accessed without decompressing the entire object.

Marinov and O'Callahan [24] proposed Object Equality Profiling (OEP) for helping programmers discover optimization opportunities in programs. Based on profiling of objects, they partition the objects into equality sets. The objects of the same equality set can be replaced with a single representative object by rewriting the program. However, there are several limitations to applying this optimization. For example, to merge multiple objects into the representative one, their technique requires that the objects should not be mutated, and that the program should not perform any operation that depends on the object's identity. In addition, their approach is meant to applied by the programmer. Our work differs from [24] in three main aspects. First, we do not have the limitations mentioned above. Second, our schemes are meant to be used within the virtual machine in a programmer-transparent fashion. Third, we can reduce the space in cases where there are some fields with the same value, but no objects are equal to each other.

Tip et al. [30] present an application extraction tool, JAX, that reduces the size of class files, as well as memory footprint of Java programs by removal of redundant methods and fields, transformation of the class hierarchy, and renaming of packages, classes, methods, and fields. As discussed in Section 4.1, our work can be complementary to these efforts. Specifically, our schemes can be applied to the programs that are already optimized by static optimization tools such as JAX to further reduce heap memory requirements. Ananian et al. [12] present a set of techniques for reducing memory consumption of Java programs. Their optimizations include field reduction, unread and constant field elimination, static specialization, field externalization, class pointer compressions, and byte Packing. Except for field externalization, all these optimizations are compiler-based and are similar to the optimizations performed by JAX. Our schemes can be applied to the programs that are already optimized by these optimizations (except for filed externalization). Field externalization uses profiling to find the fields that almost always have the same default value, and removes these fields from their enclosing class. A hash table stores the values of these fields that differ from the default value. Write accesses to these field are replaced with an insertion into the hash table (if the written value is not the default value) or a removal from the hash table (if the written value is the default value). Read accesses, on the other hand, are replaced with hash table lookups; if the object is not present in the hash table, the lookup simply returns the default value. Our scheme-1 achieves similar memory occupancy reduction effects of field externalization; however, as compared their hash-table-based approach, our scheme incurs less performance overhead.

6. CONCLUDING REMARKS

The market for mobile devices and phones is continuing to increase at a rapid rate. However, there exist several challenges in supporting applications in mobile/embedded devices. For example, the memory space and energy supply of mobile devices impose an entirely different set of constraints as compared to high-end computing environments. In particular, memory management related issues are becoming increasingly pressing as the rate at which applications are growing in complexity exceeds the memory capacity growth rate. Consequently, optimization techniques that help make better use of a given memory capacity are extremely important. Based on this motivation, this paper has presented two memory footprint reduction schemes for Java applications based on object compression. The proposed schemes take advantage of the frequent field value locality, which says that the fields of multiple objects hold the same value for a large fraction of their lifetimes. Our first scheme focuses on eliminating the space allocated for holding zeroes. Our second scheme enhances the first one by letting multiple object instances share the same copy of the fields that contain frequent values. In addition, we also quantified the benefits that could come from a pure user-level compression strategy. The performance overhead imposed by these schemes is below 2% for most of the cases.

7. REFERENCES

[1] ARM7TDMI 32-bit RISC core with 16-bit system costs. http://www.arm.com/ products/ CPUs/ ARM7TDMI.html.

[2] Connected device configuration (CDC) specification. http://java.sun.com/j2me/.

[3] Connected limited device configuration (CLDC) reference implementation. http://java.sun.com/j2me/.

[4] Kaffe virtual machine. http://www.kaffe.org.

[5] Kvm: Cldc reference implementation. http://java.sun.com/products/cldc/.

[6] Mobile device games. http://www.zingy.com/gameFaq.php.

[7] SPEC JVM98 benchmarks. http://www.specbench.org/benchmarks.html#java.

[8] TinyVM. http://tinyvm.sourceforge.net/index.html.

[9] White pape: The java hotspot virtual machine, v1.4.1. http://java.sun.com/products/hotspot/, 2002.

[10] Java technology is everywhere, surpasses 1.5 million devices worldwide. http://www.sun.com/smi/Press/ sunflash/ 2004-02/ sunflash.20040219.1.html, 2004.

[11] WebSphere micro environment. http://www-306.ibm.com/software/wireless/wme/, 2004.

[12] C. S. Ananian and M. Rinard. Data size optimizations for Java programs. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 59–68. ACM Press, 2003.

[13] A. W. Appel and M. J. R. Goncalves. Hash-consing garbage collection. Technical Report TR-412-93, Princeton University, Computer Science Department, Feb. 1993.

[14] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, California, Oct. 2003.

[15] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection in an embedded Java environment. In *the 8th International Symposium on High-Performance Computer Architecture*, Cambridge, MA, USA, Feb. 2002.

[16] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, May 2000.

[17] S. Debray and W. Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105. ACM Press, 2002.

[18] P. A. Franaszek, P. Heidelberger, D. E. Poff, and J. T. Robison. Algorithms and data structures for compressed-memory machines. *IBM Journal of Research and Development*, 45(2):245–258, Mar. 2001.

[19] G.Chen, M.Kandemir, N.Vijaykrishnan, and W.Wolf. Energy savings through compression in embedded Java environments. In *the 10th International Symposium on Hardware/Software Codesign*, Colorado, USA, May 2002.

[20] M. Kanellos. Intel crams more memory into cell phones. CNET News.com, Oct. 2003.

[21] H. Lekatsas, J. Henkal, and W. Wolf. Code compression for low power embedded system design. In *the 37th Conference on Design Automation*, pages 294–299, 2000.

[22] H. Lekatsas and W. Wolf. SAMC: a code compression algorithm for embedded processors. *IEEE Transactions on CAD*, 18(12):1689–1701, Dec. 1999.

[23] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, Oct. 2002.

[24] D. Marinov and R. O'Callahan. Object equality profiling. In *18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, Anaheim, California, Oct. 2003.

[25] C. E. McDowell, B. R. Montague, M. R. Allen, E. A. Baldwin, and M. E. Montoreano. Javacam: Trimming Java down to size. *IEEE Internet Computing*, 2(3), may/jun 1998.

[26] W. Pugh. Compressing Java class files. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 247–258, 1999.

[27] L. Rizzo. A very fast algorithm for RAM compression. *ACM SIGOPS Operating Systems Review*, 31(2):36–45, Apr. 1997.

[28] K. Sayood. *Introduction to Data Compression (Second Edition)*. Morgan Kaufmann, 2000.

[29] N. Shaylor. A just-in-time compiler for memory constrained low-power devices. In *USENIX Java Virtual Machine Research and Technology Symposium*, San Francisco, CA, USA, Aug. 2002.

[30] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 292–305. ACM Press, 1999.