Utilizing Field Usage Patterns for Java Heap Space Optimization

Zhuang Guo José Nelson Amaral Duane Szafron Yang Wang

Department of Computing Science
University of Alberta, Edmonton, Canada
{zhuang,amaral,duane,yangwang}@cs.ualberta.ca

Abstract

This research studies the characteristics of field usage patterns in the SpecJVM98 benchmarks. It finds that multiple object instances of the same class often exhibit different field-usage patterns. Motivated by this observation, we designed a heap compression mechanism that classifies object instances at runtime based on their field-usage patterns and eliminates unused fields to save space. To achieve the maximum space savings while minimizing the space and time overhead, our design combines three interrelated techniques in a novel manner: runtime object instance classification, field virtualization, and bidirectional object layout. An experimental evaluation reveals that this mechanism can reduce the maximum heap occupancy of SpecJVM98 benchmarks by up to 18% and 14% on average while keeping the application execution overhead low.

1. Introduction

The growing adoption of Java as a software platform for embedded and mobile systems requires the memory footprint of embedded Java programs to be small. The rising popularity of Java in the mobile computing world is due to its many attractive features such as cross-platform portability, automatic memory management, and built-in security. It is estimated that more than 708 million cellphones and hand-held devices were Java-enabled in June 2005, and this number is expected to exceed 1.5 billion by 2008 [18].

Restrictions on device size, weight, price, and power consumption dictate that the memory capacity in mobile devices be more constrained than in servers or desktop computers. Improvements in technology have made more memory available to mobile devices, but memory demand increases faster than capacity increases. The emergence of feature-rich applications, such as multimedia streams and video games, increases the mobile devices' memory requirements. The time that it takes a cellphone to double its memory requirements has reduced from two years to 15 months [11].

Several researchers have attempted to reduce the data footprint of Java programs by eliminating unused fields [6, 19, 4, 20]. However, none of them have fully exploited the opportunities for space optimization that can be obtained by studying field usage patterns to identify unused fields. This paper presents the first extensive study of the characteristics of field usage patterns for the SpecJVM98 benchmark [2]. Based on the insight that we gained, a heap compression mechanism was created. The central idea is to classify object instances, at runtime, based on their field usage patterns and then eliminate unused fields to save space.

The main contributions of this paper are:

- An extensive study of the characteristics of field usage patterns of Java programs using the SpecJVM98 benchmark.
- New opportunities for space optimization associated with field usage patterns. These opportunities reveal the shortcomings of existing approaches.

- The design of a heap compression mechanism that reduces the heap space requirements of Java programs. This mechanism combines three techniques: runtime object instance classification, field virtualization, and bidirectional object layout. The resulting code improves space savings while keeping time overhead low.
- An extensive experimental evaluation that indicates that this new heap compression mechanism can reduce the maximum heap space consumption by an average of 14% for the seven programs in the SpecJVM98 benchmark, while keeping the application performance overhead within 4.4%, on average.

Section 2 studies the characteristics of field usage patterns. Section 3 shows how this field usage pattern characterization leads to the design of a new compression mechanism for Java heap spaces. Section 4 presents a performance evaluation of this mechanism, and reveals a trade off between heap compression ratio and execution time overhead. Section 5 surveys the related work and Section 6 presents our conclusion.

2. Field Usage Patterns

In Java, a class is a language construct that abstracts object instances. A class has a fixed data layout, in that each instance stores the same set of instance fields of the class. However, not all the fields are necessarily used in each instance of a class. Object instances of the same class may exhibit different field usage patterns.

DEFINITION 2.1. *The field usage pattern of an object is the set of fields used by the object.*

For example, in `jess`, a benchmark in the SpecJVM98 suite, the class `Value` has three fields: `intval`, `floatval`, and `Objectval`. For each instance of `Value`, the value in a field called `_type` determines which one of the three fields stores an integer, a float, or an object reference. The other two fields remain unused. A field is unused if it stores the frequent (default) value: zero or null. Information about field usage patterns can be used to save heap space. The potential for space optimization depends on the prevalence of dominant field usage patterns. To evaluate this potential, we studied the characteristics of field usage patterns in the SpecJVM98 benchmark [2].

An extension of the field usage definition captures fields that contain a non-zero frequent value. For instance, the field `_type` of class `Value` in `jess` often contains a non-zero frequent value that can be externalized to save space. Thus, the definition of field usage pattern can be refined as:

DEFINITION 2.2. *The field usage pattern of an object is the set of fields of the object that do not store frequent field values.*

In the rest of this paper the analysis, design, and experiments that only consider zero or null frequent values are referred to as scheme-1, and those that also consider non-zero frequent values are referred to as scheme-2.

2.1 Characteristics of Field Usage Patterns

This study used the SpecJVM98 benchmark suite because it covers a wide range of applications, including expert shells, video decoders, graphic renderers, and database querying systems. The benchmark `mtrt` is omitted, because it is a parallel implementation of the `raytrace` benchmark and exhibits the same characteristics as `raytrace` for the purpose of this study. All the experiments were conducted using the `s100` input, because `s100` more faithfully captures an application's characteristics than `s1` or `s10`.[1]

Field usage patterns are discovered with a profiling run of each benchmark program using an instrumented Kaffe Virtual Machine (VM) (version 1.1.5) [1]. The profiling run scans and outputs the heap image of each object in the heap, whenever 50KB of memory allocations is accumulated. The profiling output is used to gather the frequent value distribution of each class field and study the field usage patterns. A frequent-value field is a field that has at least one value that appears in more than 10% of the object instances. All the base class object instances and derived class instances that contain the class field are used to determine whether a field has a frequent value. Therefore, if a field is considered a frequent-value field in a base class, it is also a frequent-value field in derived classes, and vice versa. After identifying frequent-value fields, objects in the heap are classified, based on their field usage patterns. Let N_c^p be the number of observed

[1] The JVM98 benchmark suite has three input sizes: `s1`, `s10`, and `s100`. `s1` should be used to check correctness; `s10` needs one tenth of the execution time of `s100`; SPEC requires that performance be reported with `s100` inputs.

object instances of class c that exhibit pattern p and let Z_c be the size, in bytes, of an uncompressed object instance of class c. If the life time of an object is longer than the heap scan interval, the object may be counted multiple times. Consider class c with n patterns $p_0, p_1, \cdots, p_{n-1}$, where in pattern p_0 all fields are used. Patterns $p_1, p_2, \cdots, p_{n-1}$ are compressible and $N_c^{p_1} \geq N_c^{p_2} \geq \cdots \geq N_c^{p_{n-1}}$. S_k is the total number of objects in a benchmark that account for the k^{th} most dominant field usage pattern of each class, and T_k is the combined size of all the objects that have the k^{th} most dominant field usage patterns of each class in the benchmark.

$$\begin{aligned} S_k &= \sum_{\forall c} N_c^{p_k}, \\ T_k &= \sum_{\forall c} N_c^{p_k} \cdot Z_c \end{aligned}$$

S_0 is the total number of observed objects that are not compressible in each benchmark and T_0 is the total storage, measured in bytes, of non-compressible objects.

Figure 1(a) and 1(b) show the distribution of field usage patterns for scheme-1 and scheme-2. Incompressible refers to objects that cannot be compressed based on field usage patterns, 1st and 2nd correspond to the total number (T_1, T_2) and size (S_1, S_2) of objects that use the most dominant and the second most dominant field usage patterns, respectively. Others corresponds to the sums of the remaining objects, $\sum_{k\geq 3} S_k$, and the sum of their sizes, $\sum_{k\geq 3} T_k$.

Table 1 uses several metrics to characterize the field usage patterns in each benchmark program. Number of Classes is the number of classes whose instances are observed during the heap scan process; Number of Patterns records the number of field usage patterns observed for each scheme. The last two columns present the weighted average of the number of field usage patterns and frequent value fields per class using the following two formulas:

$$\frac{\sum_{\forall c} A(c) \cdot B(c)}{\sum_{\forall c} A(c)} \quad and \quad \frac{\sum_{\forall c} A(c) \cdot F(c)}{\sum_{\forall c} A(c)},$$

where $A(c)$ is the number of objects, $B(c)$ is the number of field usage patterns, and $F(c)$ is the number of frequent value fields observed during the heap scan process for class c. The weighted average is reported because a prolific class, *i.e.* a class that create a large number of instances at runtime, is more important than a non-prolific one for space optimization. Another interpretation for the last two columns in Table 1 is as follows: if an object is picked randomly from the heap, the object is expected to have the number of field usage patterns and the number of frequent value fields that appear in these columns.

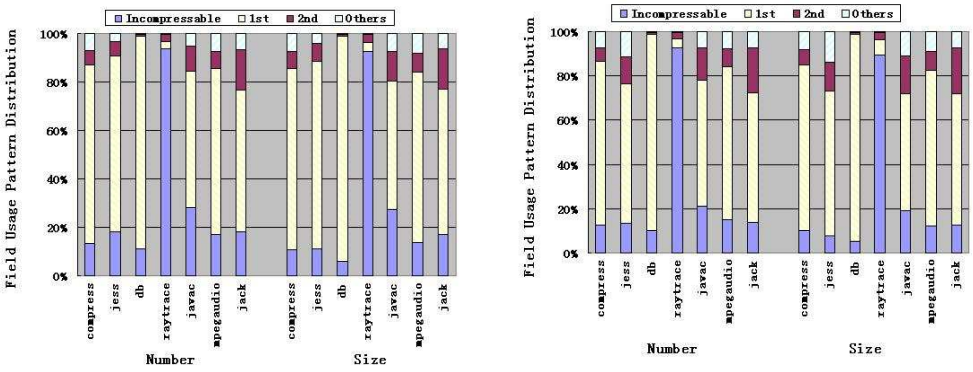
Figure 1 and Table 1 lead to the following observations:

1. In all the benchmarks, except `raytrace`, instances with compressible field usage patterns account for about 80% of all the instances.
2. Each class is expected to have 2 to 3 frequent value fields. Each object instance has 3 to 4 different field usage patterns.
3. The 1st and 2nd most frequent field usage patterns account for a significant portion of field usage pattern occurrences for most benchmarks (except `db`).
4. The number of field usage patterns increases dramatically for some benchmark programs including `jess` and `javac`. This implies that these benchmark programs should benefit the most from compressing non-zero frequent field values using scheme-2.

Thus, field usage patterns should reduce the requirements of heap allocation in most benchmark programs, except `raytrace`. Two thirds of objects observed during the heap scan in `raytrace` are instances of class `Point`, which has no frequent value field. Although a large number of other classes exhibit multiple field usage patterns — Table 1 shows that there are 262 different field usage patterns out of 123 classes — the significance of their existence is overwhelmingly shadowed by the prolific class `Point`.

2.2 Previous Use of Field Usage Patterns

Field usage patterns have been previously used for Java heap space optimization. JAX [20], an application extractor for Java, developed by Tip *et al.*, uses static analysis to remove unused fields that are either unreachable or unread. Their optimization only exploits inter-application field usage patterns. It does not make use of the observation that object instances of the same class may exhibit multiple field usage patterns within one application.



(a) Scheme-1

(b) Scheme-2

Figure 1. Distribution of Field usage patterns (a) for scheme 1 (b) for scheme 2. The left part of each figure shows the distribution of the number of objects; the right part shows the distribution of object size.

Benchmark	Number of Classes	Number of Patterns		Average Number of Patterns per Class		Average Number of Freq. Value Fields per Class	
		Scheme-1	Scheme-2	Scheme-1	Scheme-2	Scheme-1	Scheme-2
compress	107	181	213	3.8	3.9	2.9	3.0
jess	238	330	411	3.3	5.8	2.1	3.9
db	104	158	214	4.4	4.4	2.4	2.7
raytrace	123	204	262	1.5	1.5	0.3	0.4
javac	219	585	754	4.0	4.7	2.1	3.3
mpegaudio	129	204	252	3.7	3.8	2.7	2.9
jack	138	247	315	4.1	4.5	3.1	3.5

Table 1. Characteristics of field usage patterns of the SpecJVM98 benchmark

To the best of our knowledge, Ananian and Rinard were the first to use frequent value fields for space optimization [4]. Their technique removes these fields from a class and uses hash tables to store values of the field that differ from the frequently stored value. A hash table lookup is needed to access a frequent value field. Also, a hash table used in this context requires significant storage space because object ids must be stored to handle potential conflicts.

Chen, *et al.* [6] present a detailed characterization of frequent field values in the SpecJVM98 benchmark and propose two heap compression schemes. They used a heuristic formula to classify the fields of a class into three levels: At Level-0 the field does not have a dominant frequent field value; at Level-1 the field has a non-zero frequent field value; and at Level-2 the field has a frequent field value that is zero or null. In their first scheme, an object is divided into two parts: a primary part containing level-0 and level-1 fields, and a secondary part containing level-2 fields. Figure 2 shows an object in the uncompressed format. During object creation only the primary part is allocated. The secondary part is allocated lazily when the first non-zero value is written into a level-2 field. The secondary part can also be compressed at run time if it only contain zeros. Their second scheme builds upon the first one by allowing level-1 fields to be shared among objects if all level-2 fields are zero. Chen's scheme incurs a two-word space overhead for objects in uncompressed format.

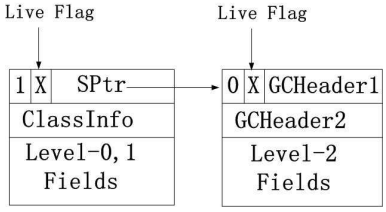


Figure 2. Chen's first compression scheme

Ananian and Rinard's and Chen *et al.*'s approaches may work well when all instances of a class use the same compressible field usage pattern. However, they will miss compression opportunities for classes that exhibit more than one compressible field usage pattern. In fact, in the SpecJVM98 benchmark around two thirds of the classes that create more than 100 instances at runtime exhibit more than one compressible field usage pattern. Furthermore, their compression mechanism incurs space overhead for objects in uncompressed format, which negates the space savings resulting from compressing objects.

In general, none of these techniques fully exploits the space optimization opportunities associated with field usage patterns. Of these techniques, Chen's approach is most comprehensive, so we compare our approach with his in Section 4.

3. A New Compression mechanism

The results presented in the previous section indicate that there are unexplored opportunities for field-usage-pattern-based space optimization. Even though the study presented in Section 2 is based on the SpecJVM98 benchmark, the results should apply to object-oriented systems. In the world of object-oriented programs, the design of a class is often general and oriented towards reusability. Thus, a class may exhibit different field usage patterns under different use contexts. We designed a heap compression mechanism to exploit the opportunities for space optimization associated with field usage patterns. This section first discusses the reason why heap compression is a favorable technique for space optimization, and then describes the design of our heap compression mechanism.

3.1 Why Heap Compression

There are a number of ways to exploit field usage patterns for space optimization. Feedback may be provided to guide programmers on the manual specialization of an application's class hierarchy. For instance, the class `Value` in Jess can be specialized into three subclasses, each corresponding to an `intval`, a `floatval`, or an `Objectval`. Manually rewriting an application can be tedious and impractical to application libraries whose source code is unavailable. Another approach is to specialize an application's class hierarchy automatically through compilers. This approach requires extensive static analysis such as whole-program point-to analysis and type inference [19]. Furthermore, static analysis cannot fully exploit space-optimization opportunities that can only be identified dynamically. A third approach, adopted in this paper, is to achieve space optimization through run-time heap compression. This approach works by compressing the heap objects if free space is still insufficient to satisfy an allocation request after garbage collection has occurred — it is used as a last resort.

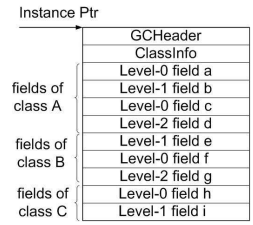
3.2 Implementation Details

The general idea of our heap compression mechanism is to eliminate unused fields from object instances to save space. This mechanism is very aggressive in that it tries to eliminate all unused fields from object instances in order to achieve the maximum space savings. After compression, all object instances with a given field usage pattern correspond to one object layout format. The experimental study in Section 2 revealed that the object instances of a class often exhibit a few different field usage patterns. This aggressive heap compression mechanism must deal with the following:

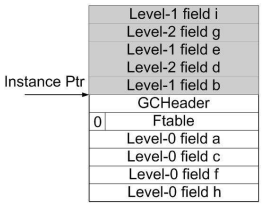
- A frequent-value field that is used must remain in a field while unused ones must be eliminated.
- The frequent value of an eliminated field whose frequent value is neither zero nor null must be recorded.
- Because of field elimination, the offset of the same field in different objects may be different.
- The space overhead for recording information about object layouts must be low since it offsets the compression gains.
- We must minimize the impact of the compression mechanism on run-time performance.

The design proposed in this paper adopts three closely related techniques to address these challenges: object instance classification, field virtualization, and bidirectional object layout.

The first technique is *object instance classification*. After compression, an object assumes a memory layout that differs from the layout of an uncompressed object. How should this layout information be stored? One possible way is to store this information within an object. However, such a solution incurs space overhead for each compressed object. Because object instances with an identical field-usage pattern share a common object layout for-



(a) Traditional layout



(b) Bidirectional layout

Figure 3. Object Layout (Class C inherits B, and B inherits A. In 3(b), any shaded field can be eliminated to save space.)

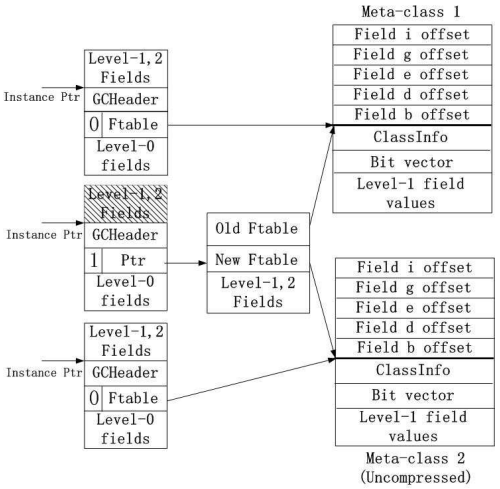


Figure 4. Field offset lookup mechanism

mat, they can share the layout information. In Java, each object has a two-word object header. The second word of this header points to the class information as illustrated in Figure 3(a). In our technique, this second word stores a pointer, `Ftable`, to reference a common data structure which is a meta-class shared by object instances with the same field usage pattern as illustrated in Figure 4.

After compression, the state of a field, eliminated or not, and its offset are the same for all object instances that share the same field usage pattern. The meta-class stores the related information — field offset and frequent field value — necessary to access each field. Since this technique resembles method virtualization, we name it *field virtualization*. Also, an object's meta-class can be determined quite efficiently. When scanning an object for compression, a bit vector with one bit corresponding to each field is calculated by scanning the object once. A bit in the vector is set 0 to indicate that the corresponding field is used, and 1, otherwise. A meta-class has a similar bit vector. If a meta-class's bit vector matches the object's bit vector, then the object belongs to the meta-class. Since most objects have only a few fields, one word (32 bits) is sufficient for storing the bit vector in the meta-class data structure.

Like Chen *et al.* [6], our field compression mechanism also classifies fields into three levels: level-0 fields have no frequent field values, level-1 fields have non-zero frequent field values and level-2 fields have zero (or null) frequent field values. However, instead of using an heuristic formula to predict the level of a field, we use profiling information. A field is considered to have a frequent value if its most frequent field occurs in more than T percent of the total field value occurrences. The choice of the threshold T has an impact on the compression performance and the application execution time. This discussion is deferred to Section 4.

Our method replaces the traditional Java object layout with a bidirectional object layout where level-0 fields are placed below the object header and level-1 and level-2 fields are placed above. Figure 3(b) shows the heap image of an object of class C in the bidirectional layout, where C inherits from class B, and B inherits from class A. When an object is compressed, some level-1 and level-2 fields are eliminated. With the bidirectional object layout, compacting an object by eliminating level-1 or level-2 fields may shift the position of other level-1

and level-2 fields, while the offset of level-0 fields remain the same. Also, since a level-0 field will never be eliminated in our compression scheme, its value can be loaded by simply using its offset and the object reference. Thus, there is no additional performance overhead to access level-0 fields. A JVM can mark each `getfield` or `putfield` bytecode based on the level of the field being accessed (denoted as getfield-n and putfield-n). Such a marking can be performed at the bytecode preparation phase, or the first time the bytecode is executed — similar to the `_quick` instruction [21].

The content of each meta-class is as follows. It has a reference to the class information structure and a bit vector as explained above. For each level-1 or level-2 field, there is an associated table entry to store the field offset. When executing an instruction, a level-2 field is loaded by a `getfield-2 index` instruction. The parameter `index` corresponds to the table entry of the level-2 field in the meta-class. This instruction first fetches the field offset by dereferencing the `Ftable` pointer. If the offset has a negative value, the field has not been compressed and the object reference and this negative offset are used to load the field value and put it onto the stack. With the bidirectional layout, a level-2 field is before the object reference address. If the offset is not negative, then this field was compressed and zero (or null) is pushed onto the stack. The handling of the `getfield-1` instruction is similar to that of `getfield-2`, except that a positive offset indicates that object instances of this meta-class take the frequent value, which is stored at the offset field in the meta-class data structure. The field offset table grows upwards with negative offsets and the non-zero frequent field value is stored at the positive offset. This aggressive heap compression mechanism is very space efficient because its space overhead is linear on the number of field usage patterns.

Instructions `putfield-n` and `getfield-n` have a similar implementation, except that a compressed object is fully expanded when a non-frequent value is written into a frequent-value field. Figure 4 shows an example of an expanded object. Because level-0 fields are not affected by this expansion, the expansion only reallocates a secondary portion of memory to store the level-1 fields, level-2 fields, and the old and new `Ftable` references. The second word in the object header is overwritten to store a pointer `Ptr` that references the secondary portion. If checking whether an object is expanded every time a level-1 or level-2 field is accessed becomes costly, an alternative design is to rely on the memory-protection mechanism already implemented in hardware. A designated address range can be set in the memory-protection table to cause a trap when it is accessed. An address within the designated range can be stored in the second word of the object header of expanded objects. When this word is dereferenced — in order to access a level-1 or level-2 field — an exception handler takes control and handles the field access transparently. This trap is expensive but it only occurs for objects that have been expanded and object expansion happens very rarely.

4. Experimental Study

This section presents an extensive simulation-based evaluation of the new compression mechanism described in Section 3. This performance evaluation reveals that:

- Compressing only unused fields reduces the heap space requirement, on average, by 12%. When frequent-value fields are also compressed the reduction increases to 14%. This reduction is 4% greater than that obtained by Chen *et al.*.
- When the benefit of running the application with a smaller memory footprint is not taken into consideration, the field compression mechanism has a small performance penalty (4.4% for scheme-2 on average).
- The selection of the threshold used to determine if a value in a field is frequent does not have a significant effect on the effectiveness of the compression. However, increasing this threshold may significantly lower the performance overhead of the mechanism.

All the experiments are based on trace-driven simulations. Trace files are generated using an instrumented Kaffe VM (version 1.1.5) [1]. To simulate the memory behavior of a JVM, the trace file contains a record for each of the following events: object creation, object use (via bytecodes `getfield`, `putfield`, `invokevirtual`, `instanceof`, and `checkcast`), garbage collection, and object finalization. A garbage collection followed by the finalization of all the dead objects is forced for every 50KB of object allocation. By reclaiming storage allocation for objects soon

Benchmark	compress	jess	db	raytrace	javac	mpegaudio	jack
Max Heap Occupancy without Array (KB)	91	529	3749	2509	6111	108	343
Max Heap Occupancy with Array (KB)	264	1228	7222	3824	9441	198	633
Execution Cycles (10^6)	22621	9715	23663	8258	13547	11289	7290

Table 2. Characteristics of SpecJVM98 benchmark programs

after their death, an application's maximum heap space occupancy (without applying compression) can be measured. However, unless garbage collection is triggered with a very small granularity, it is impossible to know the precise time of an object's death. Very frequent garbage collection is extremely expensive [9]. To avoid this imprecision, the maximum heap space occupancy is measured by sampling the heap size immediately after each garbage collection (shown in Table 2).[2] Similarly, by reclaiming storage allocation for unused fields through the application of heap compression immediately after each garbage collection, the maximum heap space occupancy with heap compression is measured.

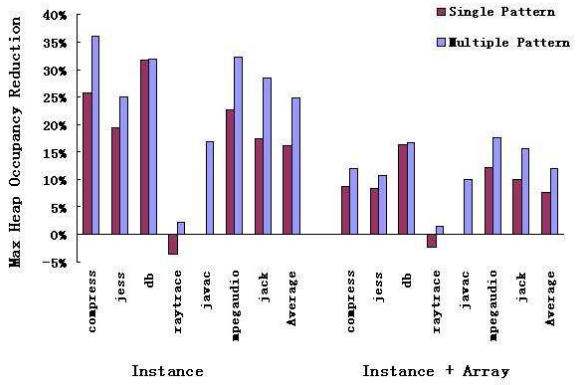
4.1 Compression Effectiveness

Filed Usage Pattern	Number of Instances	Space Savings	
		Multiple Pattern	Single Pattern
intval	376	4512	3008
_type, Objectval	415	3320	-3320
_type, intval	581	4648	4648
_type, intval, floatval	123	492	-984

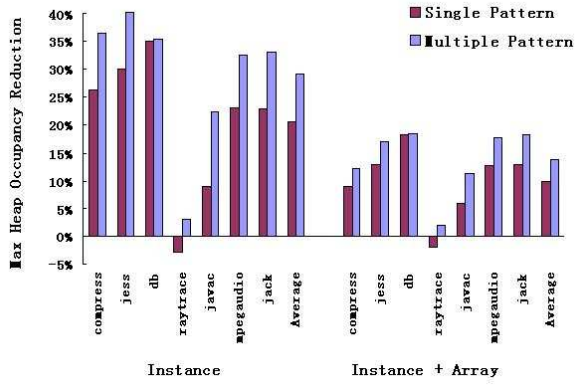
Table 3. Field usage patterns of class `Value` in `Jess`.

Figure 5 shows the reduction in maximum heap occupancy due to compression, in comparison with the baseline that does not do field-level compression. The Single Pattern bars are the results for Chen *et al.*'s compression scheme while the Multiple-Pattern bars are the results for our compression mechanism. These labels reflect the fact that our mechanism compresses objects based on multiple field-usage pattern while Chen *et al.*'s mechanism only works with a single pattern. Compression using our scheme-1 only compresses fields that are not used, while scheme-2 also compresses fields that store a frequent value. Both compression schemes significantly reduce the maximum heap space requirements for all benchmarks except `raytrace`. This result is consistent

[2] The heap occupancies reported in Table 2 account only for live objects. They do not account for other space costs such as fragmentation and object field alignment.



(a) Scheme-1



(b) Scheme-2

Figure 5. Reduction in maximum heap occupancy in comparison with no compression

with the field usage patterns shown in Figure 1. Our scheme-1 and scheme-2 mechanisms reduce the storage allocation for objects by 25% and 29% on average. Table 2 shows that arrays consume a large portion of the heap space in the SpecJVM98 benchmarks. Although our two schemes only compress object instances, they still reduce the heap space requirement by 12% and 14% on average. Compressing benchmarks `jess` and `javac` using scheme-2 is more efficient than compression using scheme-1, a result that is consistent with the benchmark characteristics in Figure 1 and Table 1. The 17% improvement of scheme-2 over scheme-1 for `jess` is explained by the fact that the number of frequent value fields in `jess` almost doubles when level-1 fields are considered.

Chen *et al.*'s first scheme fails to reduce the maximum heap occupancy for `raytrace` and `javac`. Their compression even slightly increases the heap space requirements of these two programs. This result indicates that space savings achieved by Chen *et al.*'s schemes may not always be sufficient to amortize the two-word overhead paid for each incompressible object. For instance, the class `Identifier` in benchmark `javac` has two fields: `name` and `value`. At a point of program execution, there are 811 instances of `Identifier` that use only the field `name`, and 422 instances that use both fields. Four bytes are saved for each of the 811 instances, but an 8-byte space overhead is paid for each of the 422 instances. The result is a space penalty of 92 bytes. There are many cases similar to this example in the SpecJVM98 benchmarks.

In addition, Chen *et al.*'s schemes only consider a single compressible field-usage pattern. This restriction may either turn other compressible field-usage patterns incompressible or it may miss space optimization opportunities. For class `Value` in `jess`, Chen *et al.*'s second scheme only recognizes one compressible field-usage pattern: (`_type`, `intval`). However, this class exhibits four different field-usage patterns at one point of the program execution. Table 3 shows the number of instances that exhibit each of the four patterns. By recognizing all four patterns, our scheme-2 outperforms Chen *et al.*'s by 9620 bytes.

The results in Figure 5 show that for these seven SpecJVM98 benchmarks, our techniques show improvements over Chen *et al.*'s by a significant margin for all benchmarks except `db` where we only obtain a slight improvement. In `db` the majority (around 96%) of compressible object instances in each class have a single field usage pattern (observed from Figure 1).

Through careful design, our mechanism is able to consider all field usage patterns of a class to save space by only paying a small space overhead for each field usage pattern. On average, our scheme-1 and scheme-2 outperform Chen *et al.*'s two schemes by 4% respectively, for application heap space reduction. Figure 6 shows that our schemes are also more effective than Chen *et al.*'s two techniques for reducing the average heap space size. Our scheme-2 achieves an average heap space reduction of 16%, which outperforms Chen *et al.*'s by 5%.

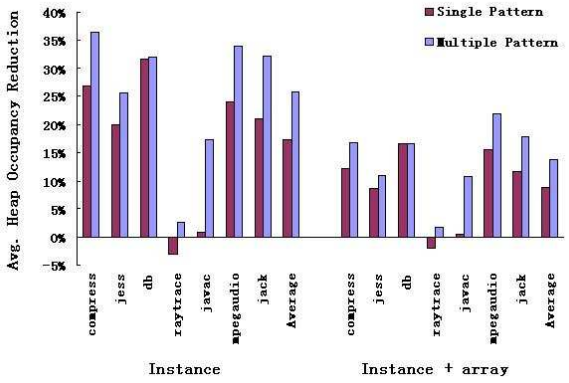
4.2 Performance Overhead

This subsection studies the performance overhead of our heap compression mechanism. Performance overheads caused by either object compression or by the extra operations used to access compressed fields. The performance overhead due to object compression varies with the heap size. When less memory is available, more time is spent in compressing objects to save space. However, after the compression, the frequency of garbage collection should be reduced [7]. The time saved by performing fewer garbage collection may be sufficient to offset the performance overhead reported in this section. In some systems, the performance gains due to less frequent garbage collection may even completely offset the performance overhead reported here.

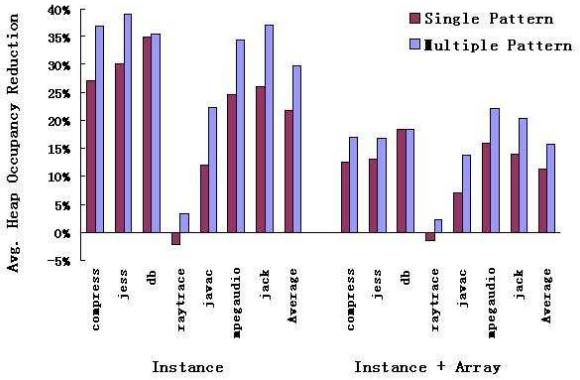
What is the impact of our compression mechanism on performance when no compression takes place? To obtain the answer we set the heap size high enough to prevent the compression from being invoked by the runtime system. In this case the run-time overhead is only caused by the extra operations used to access compressible fields:

- When accessing a level-1 field or a level-2 field, the compression mechanism executes three extra operations: a load of pointer `Ftable`, a fetch of the field offset, and a branch based on the value of the offset.
- One extra level of indirection is needed for executing bytecodes `invokevirtual`, `instanceof` and `checkcast`.

The run-time overhead to access compressed objects is estimated by counting the number of ex-



(a) Scheme-1



(b) Scheme-2

Figure 6. Reduction on average heap occupancy compared with no compression

tra instructions and memory accesses. Like Chen *et al.*, we assume that each instruction is executed in one machine cycle and that each memory access requires an extra cycle [6]. They argue that this assumption is reasonable for embedded microprocessors with low clock frequencies and short pipeline architectures. Moreover, the number of cache misses caused by loading a field offset is small because the meta-class data structures are used frequently and therefore often reside in the cache. The number of machine cycles needed to execute each benchmark in the ideal situation is measured using Sun JDK 1.4.05 with the default settings [3] on a 1.3 GHZ Intel Itanium-2 machine (shown in the fourth row of Table 2). Figure 7 shows that the performance impact of our compression mechanism is small. Most of the overhead is due to frequent-value-field accesses, rather than to the execution of three bytecode instructions. The performance overhead of scheme-2 is slightly higher than that of scheme-1, because of the increase of frequent value fields access, due to non-zero frequent value fields. The average performance overhead for scheme-2 is 4.4%.

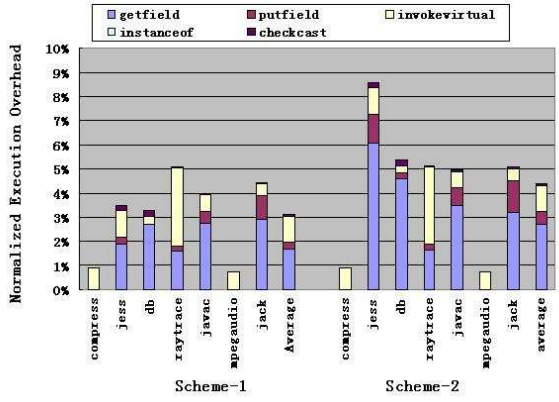


Figure 7. Performance overhead

4.3 Threshold Selection

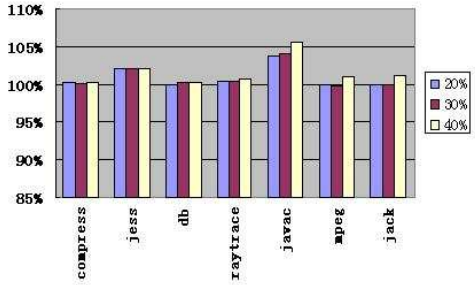


Figure 8. Maximum heap occupancy normalized with the 10% threshold (scheme-2)

For all the experiments described so far, a 10% threshold is used to determine frequent-value fields. Is the performance of the heap compression mechanism sensitive to the selection of this threshold value? To answer this question, Figure 8 shows the maximum heap occupancies of the eight SpecJVM98 benchmarks, measured using thresholds of 20%, 30%,

[3] The JIT compiler is enabled in the default settings.

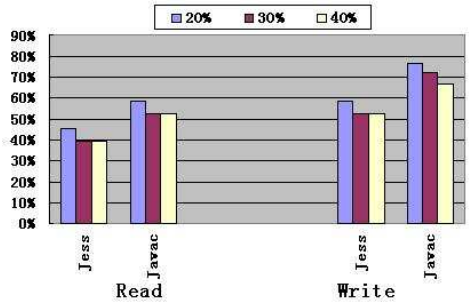


Figure 9. Number of frequent value field accesses normalized with the 10% threshold (scheme-2)

and 40%. Little impact on heap compression effectiveness is observed. However, for two benchmarks (shown in Figure 9), the number of frequent-value-field accesses drops significantly. For `jess`, with the 20% threshold, the maximum heap occupancy increases by only 3%; however, the number of frequent-value-field accesses drops by half. This result indicates that most space reductions are due to fields whose frequent values appear with a relatively high frequency. In `jess`, the most frequent values of two fields `_type` and `intval` in class `Value` account for 15.9% and 17.5% of the total value occurrences of the two fields. By increasing the threshold from 10% to 20%, neither `_type` nor `intval` is considered as a frequent value field. This increases the the space consumption by the `Value` objects at the point of program execution shown in Table 3 by 12.6%. However, with the 20% threshold, the number of frequent-value field read and field write reduces by 76% and 20% respectively. In general, this experiment shows that users can choose a relatively large threshold to reduce the performance overhead of the compression mechanism without worrying about an increase in the application heap space requirements.

5. Related Work

Various research prototypes and commercial products have been developed to allow Java programs to execute in memory-constrained environments. McDowell *et al.* designed an embedded Java environment that supports the complete Java programming language and all class libraries except AWT with as little as 1 MB of RAM [13]. Sun Microsystems provides KVM, a reference implementation of the Connected-Limited-Device Configuration (CLDC) for embedded Java [17]. KVM can operate with as little as 128 KB of memory. The MicroChaiVM supplied by HP has a minimum ROM requirements of 37 KB [10]. TinyVM, a Java-Runtime Environment (JRE) for RCX microcontrollers has a footprint of only 10 KB in RCX [3].

Besides reducing the memory footprint of the JRE, it is also important to reduce the code/data size of Java programs. JAX [20] uses a combination of extraction techniques — such as dead method elimination, dead field elimination, and class hierarchy transformation — to reduce the code size of Java class libraries. Pugh [14] presents a wire-code format to compress Java class files. Class files compressed in his format are typically $\frac{1}{2}$ to $\frac{1}{5}$ the size of the corresponding compressed Jar files. Clausen *et al.* develop a technique that replaces redundant bytecode sequences with new instructions to reduce the memory footprint [8].

To reduce the data size of Java programs, Shaham *et al.* propose a technique called object drag-time analysis that enables the reclamation of objects after their last-use to save space [16]. They demonstrate, through manual rewriting of the code, an 18% space reduction of the SpecJVM98 benchmark programs. Rizzo discovers that there are large number of redundant zeros in the RAM [15]. Based on this observation, an efficient RAM compression algorithm is developed. This algorithm was later adopted by Chen [7] to compress Java objects.

Object equality profiling is a technique that replaces identical objects with a single copy to save space [12]. This technique requires two objects to be immutable in order to be merged into an identical one. Eliminating unused fields in objects to save space has been studied by Ananian and Rinard and Chen *et al.* [4, 6]. Our mechanism improves upon theirs by better exploiting the space-saving opportunities associated with unused fields. Most JVM implementations use a two-word object header format. Bacon *et al.* propose a heuristic compression technique that allows most object to be instantiated with a single-word object header [5]. This optimization yields a space saving of 7% on average.

6. Conclusion

The extensive study of the field-usage-pattern characteristics in the SpecJVM98 benchmark suite reveals that classes often exhibit multiple field usage patterns. Thus we proposed a heap compression mechanism that allows the compression of object instances of such classes. A comprehensive per-

formance evaluation indicates that our mechanism can reduce the maximum Heap memory occupancy of Java programs by 14% while paying a small performance overhead.

Acknowledgements and Author Biographies

This research is supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), and from IBM Corporation.

Zhuang Guo is a PhD student at the department of Computing Science, University of Alberta. He received his bachelor degree in CS from Jilin University, China in 1997 and his master's degree in CS from the University of Alberta, Canada in 2003. His research interests include object-oriented programming languages, memory management, compiler optimization, and parallel processing.

José Nelson Amaral is a professor in the Department of Computing Science at the University of Alberta. He is the head of the Compiler Design and Optimization Laboratory. His areas of expertise include the design of optimizing compilers, high-performance computing and computer architecture. He has published extensively in these areas (see http://www.cs.ualberta.ca/~amaral/epublications.html). Prof. Amaral's extensive service to the scientific community includes membership in the program committee for many international conferences and proceedings chair for the International Parallel and Distributed Processing Symposium and for the International Conference on High Performance Computing. He is a co-chair for the Workshop on Compiler-Driven Performance. Prof. Amaral is an associate editor for the IEEE Transactions on Computers. He is a Faculty Fellow with the IBM Center for Advanced Studies in Toronto and he has received IBM Faculty Awards.

Duane Szafron is a Professor of Computing Science and Vice Dean of the Faculty of Science at the University of Alberta. He has been doing research in object-oriented computing since 1980, including language design, language implementation, programming environments and parallel computing. He teaches object-oriented computing courses to students at all levels, from fist year through graduate school.

Yang Wang is currently a PhD student in the department of Computing Science, University of Alberta. He received his bachelor degree in applied mathematics from Ocean University of China in 1989 and his master's degree in computer science from Carleton University, Canada in 2001. From 1989 to 1996, he was a research assistant at the Institute of Computing Technology (ICT), Chinese Academy of Science. His research interests include distributed and parallel systems and object-oriented programming languages.

References

[1] Kaffe Virtual Machine. http://www.kaffe.org.

[2] Spec jvm98 benchmarks. http://www.specbench.org.

[3] TinyVM. http://tiny.sourceforge.net/index.html.

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

[5] D. Bacon, S. Fink, and D. Grove. Space and time-efficient implementation of the Java object model. *Proceedings of the 6th European Conference on Object-oriented Programming*, pages 111–132, June 2002.

[6] G. Chen, M. Kandemir, and M. J. Irwin. Exploiting frequent field values in Java objects for reducing heap memory requirements. *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 68–78, June 2005.

[7] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 282-301, 2003.

[8] 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.

[9] M. Herz, N. Immerman, and J. E. B. Moss. Error free garbage collection traces: How to cheat and not get caught. *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems*, 30(1):140–151, June 2002.

[10] HP. HP MicroChaiVM: Doing more with less. HP Computer News, April 2001.

[11] M. Kanellos. Intel crams more memory into cellphones. CNET News, http://news.com.com, October 2003.

[12] M. Marinov and R.O'Callahan. Object equality profiling. *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 313–325, 2003.

[13] C. E. McDowell, B.R. Montague, M. R. Allen, and E. A. Baldwin. Javacam: Trimming Java down to size. *IEEE Internet Computing*, (3):53–59, 1999.

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

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

[16] R. Shaham, E. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. *Proceedings of ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 104–113, 2001.

[17] Sun. J2ME: Building blocks for mobile devices. white paper. http://java.sun.com/products/cldc/wp/KVMwp.pdf, May 2000.

[18] Sun. Sun microsystems celebrates first decade of Java. Sun press release, June 2005.

[19] F. Tip and P. F. Sweeney. Class hierarchy specialization. *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 271–285, 1997.

[20] F. Tip and P. F. Sweeney. Practical extraction techniques for Java. *ACM Transaction on Programming Languages and Systems*, 24(6):625–666, November 2002.

[21] B. Venners. *Inside the Java Virtual Machine*. MacGraw-Graw Hill, 1997.