Data Size Optimizations for Java Programs

C. Scott Ananian and Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{cananian, rinard}@lcs.mit.edu

Abstract. We present a set of techniques for reducing the memory consumption of object-oriented programs. These techniques include optimizations that eliminate fields with constant values, reduce the sizes of fields based on the range of values that can appear in each field, and eliminate fields with common default values or usage patterns. We apply these optimizations both to fields declared by the programmer and to implicit fields in the runtime object header. We describe analysis algorithms to extract the information required to apply these optimizations. We have implemented these techniques in the MIT FLEX compiler system and applied them to the programs in the SPECjvm98 benchmark suite. Our experimental results show that our combined techniques can reduce the maximum live heap size required for the programs in our benchmark suite by as much as 40%. Some of the optimizations reduce the overall execution time; others may impose modest performance penalties.

1 Introduction

This paper presents a set of techniques for reducing the amount of data space required to represent objects in object-oriented programs. Our techniques optimize the representation of both the programmer-defined fields within each object and the header information used by the run-time system:

- **Field Reduction:** Our flow-sensitive, interprocedural bitwidth analysis analysis computes the range of values that the program may assign to each field. The compiler then transforms the program to reduce the size of the field to the smallest type capable of storing that range of values.
- **Unread and Constant Field Elimination:** If the bitwidth analysis finds that a field always holds the same constant value, the compiler eliminates the field. It removes each write to the field, and replaces each read with the constant value. Fields without executable reads are also removed.
- **Static Specialization:** Our analysis finds classes with fields whose values do not change after initialization, even though different instances of the object may have different values for these fields. It then generates specialized versions of each class which omit these fields, substituting accessor methods which return constant values.

- **Field Externalization:** Our analysis uses profiling to find fields that almost always have the same default value. It then removes these fields from their enclosing class, using a hash table to store only values of the field that differ from the default value. It replaces writes to the field 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). It replaces reads with hash table lookups; if the object is not present in the hash table, the lookup simply returns the default value.
- **Class Pointer Compression:** Our rapid type analysis computes an upper bound on the number of classes that the program may instantiate. Objects in standard Java implementations have a header field, commonly called `claz`, which contains a pointer to the class data for that object, such as inheritance information and method dispatch tables. Our compiler uses the results of the analysis to replace the reference with a smaller offset into a table of pointers to the class data.
- **Byte Packing:** All of the above transformations may reduce or eliminate the amount of space required to store each field in the object or object header. Our byte packing algorithm arranges the fields in the object to minimize the object size.

All of these transformations reduce the space required to store objects, but some potentially increase the running time of the program. Our experimental results show that, for our set of benchmark programs, all of our techniques combined can reduce the peak amount of memory required to run the program by as much as 40% and never increase the running time by more than 60%. In many scenarios, a 10% speedup occurs.

1.1 Contributions

This paper makes the following contributions:

- **Space Reduction Transformations:** It presents a set of novel transformations for reducing the memory required to represent objects in object-oriented programs.
- **Analysis Algorithms:** It presents a set of analysis algorithms that automatically extract the information required to apply the space reduction transformations.
- **Implementation:** We have fully implemented all of the analyses and techniques presented in the paper. Our experience with this implementation enables us to discuss the pragmatic details necessary for an effective implementation of our techniques.
- **Experimental Results:** This paper presents a set of experimental results that characterize the impact of our transformations, revealing the extent of the savings available and the performance cost of attaining them.

2 Example

We next present a pair of examples that illustrate the kinds of analyses and transformations that our compiler performs.

```
public class JValue {
 int integerType = 0;
 int floatType = 1;
 int type, positive;
 Object value;
 void setInteger(Integer i) {
  type = integerType; value = i;
  positive =
   (i.intValue() > 0) ? 1 : 0;
 }
 void setFloat(Float f) {
  type = floatType; value = f;
  positive =
   (f.floatValue() > 0) ? 1 : 0;
 }
}
```

(a)

```
public final class String {
 private final char value[];
 private final int offset;
 private final int count;
 ...
 public char charAt(int i) {
  return value[offset+i];
 }
 public String substring(int start)
 {
  int noff = offset + start;
  int ncnt = count - start;
  return new String
        (noff, ncnt, value);
 }
}
```

(b)

Fig. 1. (a) The `JValue` class. (b) Portions of the `java.lang.String` class.

2.1 Field Reduction and Constant Field Elimination

Figure 1a presents the `JValue` class, which is a wrapper around either an `Integer` object or a `Float` object. The `type` field indicates which kind of object is stored in the `value` field of the class, essentially implementing a tagged union.[1] The class also maintains the `positive` field, which is `1` if the wrapped number is positive and `0` otherwise.

Our bitwidth analysis uses an interprocedural value-flow algorithm to compute upper and lower bounds for the values that can appear in each variable. This analysis tracks the flow of values across procedure boundaries via parameters, into and out of the heap via instance variables of classes, and through intermediate temporaries and local variables in the program. It also reasons about the semantics of arithmetic operators such as `+` and `*` to obtain bounds for the values computed by arithmetic expressions. This analysis discovers the following facts about how the program uses this class: a) the `integerType` field always has the value `0`, b) the `floatType` field always has the value `1`, c) the `type` field always has a value between `0` and `1` (inclusive), and d) the `positive` field always has a value between `0` and `1` (also inclusive).

Our compiler uses this information to remove all occurrences of the `integerType` and `floatType` fields from the program. It replaces each read of the `integerType` field with the constant `0`, and each read of the `floatType` field with the constant `1`. It also uses the bounds on the values of the `type` and `positive` variables to reduce the size of the corresponding fields. Our currently implemented compiler rounds field sizes to the nearest byte required to hold the range of values that can occur. Our byte packing algorithm then generates a dense packing of the values, attempting to preserve the alignment of the variables if possible. In this case, the algorithm can reduce the field sizes by six bytes and the overall size of the object by one four-byte word. If the runtime can support unaligned objects

[1] This class is a simplfied version of similar classes that appear in some of our benchmarks. See for example the `jess.Value` class in SPECjvm98 benchmark `jess`.

```
public final class SmallString {
 private final char value[];
 private final int count;
 int getOffset() { return 0; }
 ...
 public char charAt(int i) {
  return value[getOffset()+i];
 }
}
public final class BigString
  extends SmallString {
 private final int offset;
 int getOffset() { return offset; }
}
```

(a)

```
public SmallString substring(int start)
{
 int noff = offset + start;
 int ncnt = count - start;
 if (noff==0)
  return new SmallString
              (value, noff, ncnt);
 else
  return new BigString
              (value, noff, ncnt);
}
```

(b)

Fig. 2. (a) Static specialization of `java.lang.String`. (b) Dynamic selection among specialized classes in a method from `java.lang.String`.

without external fragmentation, we can reduce the object size by the full six bytes.

2.2 Static Specialization

Figure 1b presents portions of the implementation of the `java.lang.String` class from the Java standard class library. The `value` field in this class refers to a character array that holds the characters in the string; the `count` field holds the length of the string. In some cases, instances of the `String` class are derived substrings of other instances (see the `substring` method in Figure 1b), in which case the `offset` field provides the offset of the starting point of the string within a shared `value` character array. Note that the `value`, `offset`, and `count` fields are all initialized when the string is constructed and do not change during the lifetime of the string.

In practice, most strings are not created as explicit substrings of other strings, so the `offset` field in most strings is zero. In fact, all of the public `String` constructors create strings with `offset` zero; only the `substring` method creates strings with a non-zero offset. And even at calls to the private `String(int, int, char[])` constructor inside the `substring` method, it is possible to dynamically test the values of the parameters at the allocation site to determine if the newly constructed string will have a zero or non-zero offset.

Our analysis exploits this fact by splitting the `String` class into two classes: a superclass `SmallString` that omits the `offset` field, and a subclass `BigString` that extends `SmallString` and includes the `offset` field. Each of these two new classes implements a `getOffset()` method to replace the field: the `getOffset()` method in the `SmallString` class simply returns zero; but the `getOffset()` method in the `BigString` class returns the value of the `offset` field in `BigString`. Figure 2a illustrates this transformation.

At every allocation site except the one inside the `substring` method, the transformed program allocates a `SmallString` object. Inside the `substring` method, the program generates code that dynamically tests if the offset in the substring will be zero. If so, it allocates a SmallString object; if not, it allocates a BigString object. (See Figure 2b.) This transformation therefore eliminates the offset field in the majority of strings.

The analysis required to support this transformation takes place in two phases. The first phase scans the program to identify fields that are amenable to transformation.[2] In our example, the analysis determines that the offset field is never written after it is initialized. The next phase determines if the value of this field is determined either by the constructor that initialized it or if it is a simple function of the parameters of the constructor. In our example, the analysis determines that the offset field is zero for all constructors except the private constructor invoked within the substring method. It also determines that, for objects created within substring, the value of the offset field is simply the value of the noff parameter to this constructor.

This analysis identifies a set of candidate fields. The analysis chooses one of the candidate fields, then splits the class along the possible values that can appear in the field. Our current implementation uses profiling to select the field that will provide the largest space savings; our policy takes both the size of the field and the percentage of objects that have the same value for that field. In our example, the analysis identifies the offset field as the best candidate and splits the class on that field. We can apply this idea recursively to the new program to obtain the benefits of splitting on multiple fields.

2.3 Field Externalization

In the string example discussed above, it was possible to determine which version of the specialized class to use at object allocation time. In some cases, however, a given field may almost always have a given value, even though it is not possible to statically determine when the value might be changed or which objects will contain fields of that value. In such cases we apply another optimization, *field externalization*. This optimization removes the field from the class, replacing fields whose values differ from the default value with hash table entries that map objects to values. If an object/value mapping is present in the hash table, that entry provides the value of the removed field. If there is no mapping for a given object, the field is assumed to have the default value. In our current implementation, we use profiling to identify the default value.

In this scheme, writes to the field are converted into a check to see if the new value of the field is the default value. If so, the generated code simply removes any old mappings for that object from the hash table. If not, the generated code replaces any old mapping with a new mapping recording the new value.

2.4 Hash/Lock Externalization

Our currently implemented system applies field externalization in a general way to any field in the object. We would, however, like to highlight an especially useful extension of the basic technique. Java implementations typically store an

[2] See Section 3.5 for a more precise definition.

object hash code and lock information in the object header. For many objects, however, the program never actually uses the hash code or lock information. Our implemented system therefore uses a variant of field externalization called *hash/lock externalization*. This variant allocates all objects without the hash code and lock information fields in the header, then lazily creates the fields when necessary. Specifically, if the program ever uses the hash code or lock information, the generated code creates the hash code or lock information for the object, then stores this information in a table mapping objects to their hash code or lock information.[3]

Note that, in general, this transformation (as well as field externalization) may actually increase space usage. But in practice, we have found that our set of benchmark programs rarely uses these fields. The overall result is a substantial space savings. The combination of class pointer compression and hash/lock elimination can produce a common-case object header size of one byte — one byte for a class index and no space at all for hash code or lock.

3 Analysis Algorithms

In this section we will present details of the analyses that enable our transformations.

3.1 Rapid Type Analysis

We start with a rapid type analysis [6] to collect the set of instantiated classes and callable methods. This analysis allows us to generate a conservative call graph for the program, using the known receiver type at the call-site and its set of instantiated subclasses in the hierarchy. Based on the class hierarchy, we can also tag all leaf classes as `final`, regardless of whether the source code contained this modifier. Methods which are not overridden, based on the hierarchy, are also marked `final`, and calls with a single receiver method are devirtualized. We also remove uncallable methods and assign non-conflicting slots to interface methods using a graph-coloring algorithm. The results of some class casts and `instanceof` operations can also be determined statically using these results.

Our analysis keeps separate the set of *mentioned* and *instantiated* classes. Although type-checks can be made and methods invoked on abstract, interface, or otherwise uninstantiated classes, every object in the heap must belong to one of the instantiated class types. The size of the set of instantiated classes is quite small for a typical Java program, and all but two of the benchmarks in SPECjvm98 have less than 256 instantiated class types.[4] We use this information to replace the class pointer in the object header, which identifies the type of the object, with a one-byte *index* into a small lookup table. The `jess` and `javac` benchmarks require more than one byte of index, but a two byte index amply suffices in these two cases.

[3] The object's address is used as its key when field externalization is done. The garbage-collector is responsible for updating the field entries if it moves objects, by rehashing on the new address.

[4] *All* have more than 256 *total* class types.

$$\begin{aligned}
-\langle m,p\rangle &= \langle p,m\rangle \\
\langle m_l,p_l\rangle + \langle m_r,p_r\rangle &= \langle 1+\max(m_l,m_r), 1+\max(p_l,p_r)\rangle \\
\langle m_l,p_l\rangle \times \langle m_r,p_r\rangle &= \left\langle \begin{matrix} \max(m_l+p_r, p_l+m_r), \\ \max(m_l+m_r, p_l+p_r) \end{matrix} \right\rangle \\
\langle 0,p_l\rangle \wedge \langle 0,p_r\rangle &= \langle 0, \min(p_l,p_r)\rangle \\
\langle m_l,p_l\rangle \wedge \langle m_r,p_r\rangle &= \langle \max(m_l,m_r), \max(p_l,p_r)\rangle
\end{aligned}$$

Fig. 3. Some combination rules for bitwidth analysis. Note that the penultimate entry is a special-case rule that only applies if the neither of the arguments can be negative.

3.2 Bitwidth Analysis

We use a flow-sensitive interprocedural bitwidth analysis to find constant values, unused and constant fields, and to reduce field sizes where possible. Our dataflow framework uses Wegman and Zadeck's Sparse Conditional Constant (SCC) propagation algorithm [19] as a basis. We then extend their analysis interprocedurally, add coverage of Java language features, and extend the value lattice to handle bitwidths. Since almost all types in Java are signed (with the exception of the 16-bit `char`), we must be able to describe bitwidths of both negative and positive numbers, which we do by splitting the set of values into negative, zero, and positive parts, and describing the bitwidth of each individually.

We abstract sets of non-singleton integer values into a tuple $\langle m, p\rangle$ where $m \geq 1 + \lfloor \log_2 N \rfloor$ for all negative N in the set, and $p \geq 1 + \lfloor \log_2 N \rfloor$ for positive N. We use $m = p = 0$ to represent the constant zero. Some combination rules for arithmetic operations are shown in Figure 3. The rules for simple arithmetic operators should be self-evident upon examination (adding two N bit integers yields at most an $N + 1$-bit integer, for example) although care must be taken to ensure that combinations of negative and positive integers are handled correctly. Our implementation contains additional rules giving it greater precision for common special cases, such as multiplication by a one-bit quantity, division by a constant, or (as the figure shows) bitfield operations on positive numbers.

Treatment of Fields Dataflow on this bitwidth lattice is performed on the entire Java program interprocedurally. The analysis is what Heintze and Tardieu [11] would call *field-based*; that is, given a field f defined in class X, and an instance of X named x, we consider an assignment to $x.f$ to be an assignment to the field $X.f$ and ignore the base object x.[5] The result of the analysis is a bitwidth specification for each variable and field in the program. As the analysis is based on SCC, we also identify constant variables and fields; we replace reads of constant fields with their constant value and the field is eliminated.

[5] An obvious extension is to use pointer analysis to discriminate between fields allocated at different sites in the program.

Other Details Our analysis handles method calls by merging the lattice values of the method parameters at the call site with the formal parameters of the method. Similarly, the return value of the method is propagated back to all call-sites. Our compiler's intermediate representation handles thrown exceptions by treating the method return value as a tuple, and the call site as a conditional branch. The "normal return value" is assigned and the first branch taken on a normal method return, and the "exceptional return value" is assigned and the second branch taken when an exception is thrown from the method.

Our implementation of this analysis is actually context-sensitive, with a user-defined context length. All results presented here were obtained with the context set to zero; we saw no clear benefit from 1- or 2-deep calling contexts, and the increase in analysis execution time was considerable.

Space does not permit us to describe the remaining details of the full analysis, including the extension of the value lattice to handle the full range of Java types, the class hierarchy, `null` and `String` constants, and fixed-length arrays. We refer the interested reader to [4] for an exhaustive description.

3.3 Definite Initialization Analysis

Java field semantics dictate that uninitialized fields must have the value zero (or `null`, for pointer fields). It may seem, then, that the starting lattice value for every integer field should be $\mathbf{0}$. This, however, prevents us from finding non-zero field constants in the program: a simple initialization statement like `x=5` will assign `x` the value $\mathbf{0} \sqcap \mathbf{5}$, which is not equal to $\mathbf{5}$![6]

We perform a *definite initialization* analysis to remedy this problem and restore precision to our analysis. For example, with only constructor $\mathtt{A}_1$ in the following code, field `f` will get the lattice value $\mathbf{5}$:

```
public class A {
    int f;
    A1(...) { f = 5; }
    A2(...) { /* no assignment to f */ }
}
```

Without constructor $\mathtt{A}_2$ in the class, we say that field `f` is *definitely initialized* because every constructor of `A` assigns a value to `f` before returning or calling an unsafe method. Adding constructor $\mathtt{A}_2$ allows the default $\mathbf{0}$ value of `f` to be seen; `f` is then no longer definitely-initialized.

We actually allow the constructor great flexibility in regard to definite initialization; it is free to call any method which does not read `A.f` before finally executing a definite initializer. We construct a mapping from methods to all fields which they may read, in a flow-insensitive manner, and compute a transitive closure of this map over the call graph to determine a "safe set" of methods which the constructor may call before a definite initialization of `f`. As long as control flow may not pass to a method not in the safe set before `f` is written, then `f` is definitely initialized.

[6] On the SCC lattice of [19], $\mathbf{0} \sqcap \mathbf{5} = \top$ (but see footnote 7).

When performing bitwidth analysis, definitely-initialized fields are allowed to start at $\perp$ in the dataflow lattice.[7] All other fields must start at value $\mathbf{0}$, which will make it impossible for the field to represent a non-zero constant value. The results of the definite initialization analysis are also used when profiling mostly-constant fields, as described in the next section.

3.4 Profiling Mostly-Constant Fields

To inform the static specialization and field externalization transformations, we instrument a profiling build of the code to determine which fields are *mostly-constant*. Our implementation builds one binary per examined constant, that is, one binary to look for "mostly-zero" fields, a separate binary to look for fields which are usually "one", a third binary to look for fields commonly "two", and so forth. We built ten binaries for each benchmark, looking for field default values in the interval $[-5, 5]$. For pointer fields, we only look for `null` as a default value. Although our use of multiple binaries is by no means necessary, for ease of exposition we will discuss our profiling technique as if there is a single default value N which we are looking for.

Our instrumentation pass starts by adding a counter per class to record the number of times each exact class type is instantiated. We also add per-field counters which are incremented the first time a non-N value is stored into a certain field.[8] By comparing the number of times the class (thus field) is instantiated and the number of times the field is set to a non-N value, we can determine the amount of memory recoverable by applying a "mostly-N" transformation to the field, whether static specialization or field externalization. We use this potential savings to guide our selection of fields for static specialization, using the field and default value which the profile indicates will yield the largest gain. If static specialization isn't an option, the proportion of non-N fields helps indicate whether externalization is likely to result in a net savings; see Section 4.2 for further discussion.

There is one last detail to attend to: when looking for non-zero N values, the default zero value of uninitialized fields becomes a problem. For these cases, we use the definite-initialization analysis described in the previous section to increment the "non-N" counter on any path where the field in question is not definitely initialized.

3.5 Finding Subclass-Final Fields

Our static specialization transformation can only be applied to what we call *subclass-final* fields. Subclass-finality is a less strict but similar constraint to

[7] We use $\perp$ for "nothing known" and $\top$ for "under-constrained"; another segment of the compiler community commonly reverses these definitions.

[8] Note that this requires storing an additional bit per field during profiling to record whether a non-N value has been seen previously.

Java's `final` modifier. We do a single-pass analysis to determine subclass-finality, using the results from the bitwidth analysis to improve our precision.[9]

A *subclass-final* field `f` of a class `A` can be written to from any method of a *subclass* of `A`, as well as in any constructor of `A`. In each write, the receiver's type must be a subtype of `A`, except inside `A`'s constructors, where the receiver may also be the method's `this` parameter. Multiple writes to `f` are permitted, unlike the Java-`final` fields.

Subclass-finality matches the requirements of the static specialization transformation. Since we always insert a "big" version of the original split class as parent to any subclasses, subclasses can write to the field in objects of the "big" type without restriction. We need only restrict writes which occur in the class proper.

Our analysis constructs the set of subclass-final fields by finding its dual, the set of *non*-subclass-final fields. We scan every method and collect all fields with illegal writes; all fields found are added to the set of non-subclass-final fields.

3.6 Constructor Classification

The final requirement to enable static specialization is to identify constructors which always initialize certain fields in a given way. In particular, we wish to find constructors which always give fields statically-known constant values, as well as constructors which initialize fields with simple functions of their input parameters. The first case enables us to unconditionally replace an instantiated class with a smaller split version; the second case allows us to wrap the constructor in an appropriate conditional to enable the creation of the small version when possible.

This analysis is simple, because it builds upon our previous results. In a single pass over the constructor, we merge the values written to a selected subclass-final field, treating `Param`N as an abstract value for the Nth constructor parameter. We treat any call to a `this()` constructor as if it were inlined. By the properties of subclass-final fields, we know that all writes to the field are to the `this` object and that there are no bad writes to the field outside of the constructor. If the merged value at the end of the pass is a `Param` value or a constant equal to the desired "default" value of the selected field, then we can statically specialize on the field at this constructor site. Further, we rule out specialization on any otherwise-suitable fields for which there is not at least one callable constructor amenable to static specialization.

4 Implementation Issues

In this section we will talk briefly about some of the practical issues arising in an implementation of our space-saving techniques.

[9] By using analysis rather than relying on programmer specification, the author need not restrict *all* users of their code in order to obtain maximum efficiency for *some* constrained uses of it.

4.1 Byte Packing

A typical Java implementation may waste large amounts of space by aligning fields for the most efficient memory access. Fields are often aligned to their widths (a 4-byte field will be placed at an address which is an even multiple of 4, for example), and the object as a whole is often placed on a double-word boundary. Our implementation places object fields at the nearest byte boundary, although the information provided by our bitwidth analysis is sufficient to *bit*-pack the fields in the object when space is truly at a premium. Preliminary investigation indicated that the amount of additional space gained by bit-packing is typically only a few percent, because there aren't enough sub-byte fields to fill the space "wasted" by byte alignment.

Some architectures penalize unaligned accesses to fields. It is worthwhile to *attempt* to align fields to their preferred alignment while not allowing this to cause the object size to grow. Further, there are often *forced* alignment constraints on (for example) pointers. Our Java runtime uses a conservative garbage collector; its efficiency decreases markedly if pointers are not word-aligned.[10]

Our "byte-packing" heuristic achieves tight packing of fields while respecting forced alignments. Packing proceeds recursively through superclasses, and returns a list of free-space intervals available between the fields of the superclass. The algorithm first places all *forced-alignment* fields in the class, from largest to smallest. The aim is for the alignment-induced spaces left by the large fields to be fillable by the following smaller fields.

When there are no more forced-alignment fields, we attempt to allocate fields on their "preferred" alignment boundaries, largest first. At this stage fields are not allowed to introduce an alignment gap at the end of the object. If their preferred alignment does not allow them to be placed flush against the last field of the object, they are skipped.

Finally, when there are no more fields satisfying preferred-alignments, we allocate the *smallest* available field at the lowest possible byte boundary. The aim is that the small fields will fill space and nudge the end of the object out so that a larger field may be allocated on its preferred alignment. After each field is placed, we begin again by attempting to place fields on preferred boundaries.

This heuristic strategy has been observed to work well in practice, and the penalties for occasionally placing an unaligned non-pointer field have not been observed to have a material adverse effect on performance (see Section 5.3).

4.2 External Hashtable Implementation

Close attention to the implementation of the hashtable used for field and hash/lock externalization is necessary to realize the identified possible gains. To maximize space savings, we need to use as little space per field stored in the table as possible. The overhead of dynamically allocated buckets and the required *next* pointers makes separate chaining impractical as a hashtable implementation technique. Open-addressing implementations are preferable: in addition to the value being stored, all that is necessary is a key value and the empty space

[10] This means that objects have to be word-aligned as well.

required to limit the load-factor. A load factor of two-thirds and one-word keys and values yield an average space consumption of three words per field. This implementation breaks even when the mostly-zero fields identified are zero over 66% of the time. This break-even point is compared to the profiling data to allow our field externalization transformation to intelligently choose targeted fields.

Key-size reduction is an important component of the implementation: a naïve approach would combine a one-word reference to the virtual-container object and a one-word field identifier for a two-word key. The large key will shift the break-even point up so that only fields which are 82% zero will profit. Instead, we can offset the object reference (up to the limit of its size) by small integers to discriminate the externalized fields of the object, yielding a single-word key.

Our implementation leverages the garbage-collector to remove unneeded entries from the hashtable.

4.3 Class Loading and Reflection

This research was conducted using the MIT FLEX compiler infrastructure,[11] which is a whole-program static compiler. Although the analyses as described reflect this, it would be straightforward to use *extent analysis* [16] to apply transformations to only the closed-world portions of a program which used dynamic class loading. The space allocated to the class index could be updated during garbage collection as new classes are discovered. Concurrent profiling could actually expose more opportunities for space compression in a JIT environment. And our various transformations need not be user-visible if the reflection implementation is carefully written.

5 Experimental Results

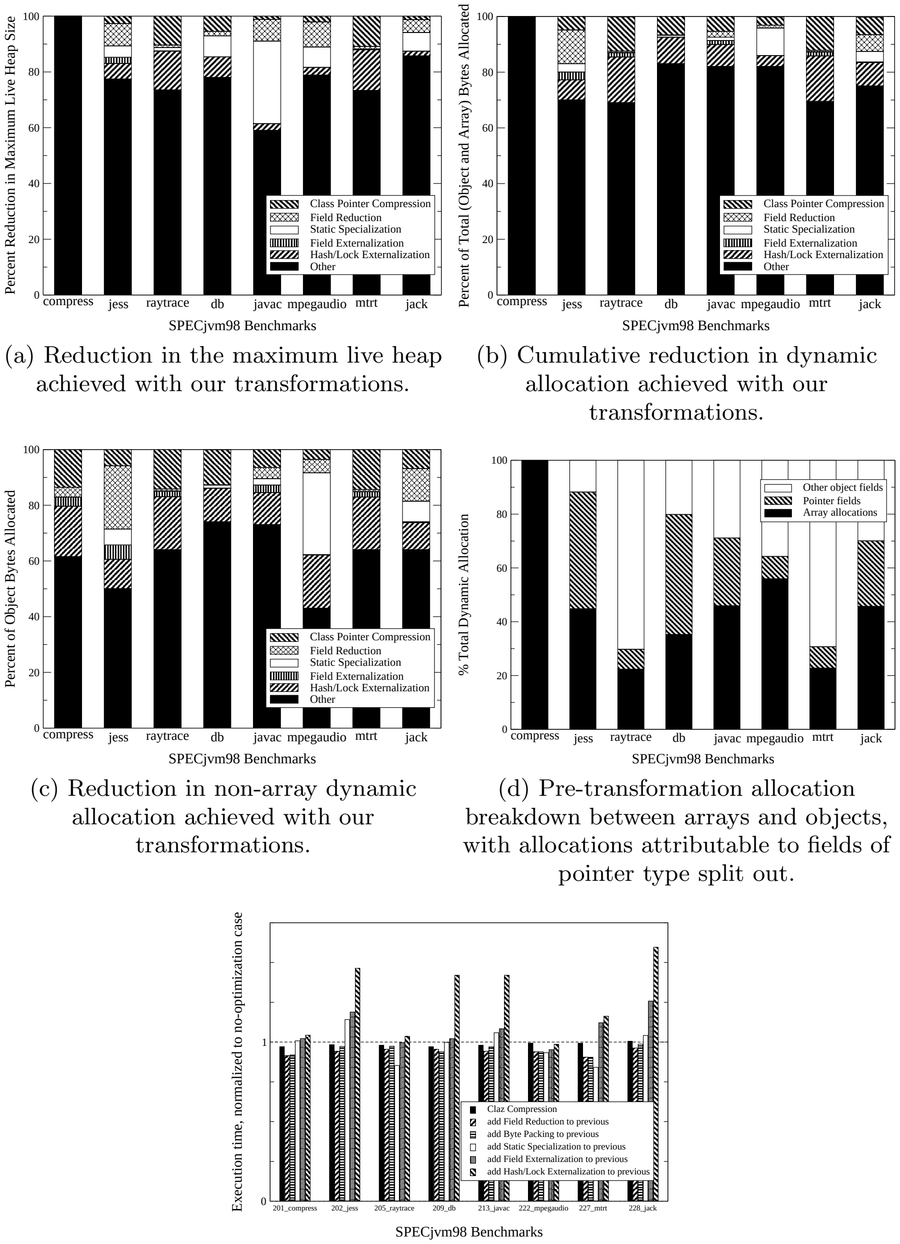
We have implemented all of the analyses and transformations described in this paper in FLEX. We measure the effectiveness of our optimizations by using FLEX to analyze the SPECjvm98 benchmarks and apply our transformations, then measuring the resulting space savings and performance. All benchmarks were run on a dual-processor 900 MHz Pentium III running Debian Linux.

5.1 Memory Savings

To evaluate the effectiveness of our technique at reducing the amount of memory required to execute the program, we first ran an instrumented version of each application with no space optimizations. We used this instrumented version to compute the maximum amount of live data on the heap at any point during the execution. We then ran an instrumented version of our program after each stage of optimization. These versions enabled us to calculate the amount by which each technique reduced the size of the live heap data.

Figure 4a presents the total space savings numbers. This figure contains a bar for each application, with the bar broken down into categories that indicate

[11] Available from `http://flexc.lcs.mit.edu/`.



(a) Reduction in the maximum live heap achieved with our transformations.

(b) Cumulative reduction in dynamic allocation achieved with our transformations.

(c) Reduction in non-array dynamic allocation achieved with our transformations.

(d) Pre-transformation allocation breakdown between arrays and objects, with allocations attributable to fields of pointer type split out.

Execution time, normalized to no-optimization case
0
1
201_compress
202_jess
205_raytrace
209_db
213_javac
222_mpegaudio
227_mtrt
228_jack
SPECjvm98 Benchmarks
Claz Compression
add Field Reduction to previous
add Byte Packing to previous
add Static Specialization to previous
add Field Externalization to previous
add Hash/Lock Externalization to previous

(e) Runtime performance of space optimizations.

Fig. 4. Experimental results of space optimization transformations.

the percentage of live data from the original unoptimized execution that we were able to eliminate with each optimization. The black section of each bar indicates the amount of live heap data remaining after all optimizations. We obtain as much as 40% reduction in live data on the `javac` benchmark, with almost all of this reduction coming from our bitwidth-driven field reductions and static specialization. In fact we obtain more than 15% reduction on all of the “object-oriented” benchmarks. The `compress` benchmark allocates a small number of very large arrays, limiting the optimization opportunities discoverable by our analysis. Likewise, the `raytrace` and `mtrt` benchmarks make heavy use of floating-point numbers, limiting the applicability of our integer bitwidth analysis. However, these raytracing benchmarks allocate a large number of small arrays to represent vectors and matrices, and so our header optimizations still allow us to reduce the maximum live data size by over 20%.

We also used an instrumented executable to determine the total amount of memory allocated during the entire execution of the program, in both the optimized and unoptimized versions. Reducing this total allocation decreases the load on the garbage collector. Figure 4b presents the space savings according to this metric. Comparison to the previous figure reveals that long-lived objects provide proportionally more opportunities for optimization.

5.2 Objects Versus Arrays

The majority of our optimizations are designed to optimize object fields rather than arrays. For context, we present numbers that characterize the reductions in total allocation for objects only, rather than for both objects and arrays. Figure 4c presents space savings numbers for objects alone, omitting any storage required for arrays. Figure 4d explains the difference by showing how the total program allocation for each benchmark is broken down into array and object allocations. The reason for our poor performance on `compress` is now obvious—a few large uncompressible integer arrays account for over 99% of the total space allocated.

5.3 Execution Times

We next evaluate the execution time impact of applying our space optimizations. Figure 4e presents the normalized execution times of each application after the application of our sequence of optimizations. These numbers show that the first several optimizations (class pointer compression, field reduction, and byte packing) typically reduce the execution times, while the remainder (static specialization, field externalization, and hash/lock externalization) generate modest increases in the execution times. The speedup is due to reduced GC times, despite the indirection and misalignment costs. Static specialization’s virtualization of fields is responsible for its slowdown; it is likely that an optimized speculatively-inlined implementation of the field accessors which it adds to the program would improve its performance. Field externalization (including hash/lock externalization) causes the expected penalty for hashtable lookup; note that synchronization elimination would greatly reduce the cost of hash/lock externalization in the four cases where the overhead is unreasonable.

6 Related Work

Many researchers have focused on the problem of reducing the amount of header space required to represent Java locks [5, 12, 1]. The vast majority of programs do not use the lock associated with every object in its full generality, so it is possible to develop improved algorithms optimized for the common case. The idea is to represent the lock with the minimum amount of state (typically a bit) required to support the common usage pattern of an acquire followed by a release, and to back off to a more elaborate scheme only when the thread exhibits a more complex pattern such as nested locking. The primary focus has been on improving performance rather than on reducing space; however, many of the algorithms also eliminate the need to store the complicated locking objects required to support the most general lock usage pattern possible in a Java program. These techniques typically reduce the lock space overhead to 24 header bits [5].

Research in escape analysis and related analyses can enable the compiler to find objects whose locks are never acquired [2, 7, 20, 9, 13, 15]. This information can enable the compiler to remove the space reserved for synchronization support in these objects. Our hash/lock removal algorithm uses a totally dynamic approach based on our field externalization mechanism.

Several researchers have used bitwidth analysis to reduce the size of the generated circuits for compilers that generate hardware implementations of programs written in C or similar programming languages [3, 4, 14, 17, 8].

Dieckmann and Hölzle have performed an in-depth analysis of the memory allocation behavior of Java programs [10]. Although space is not their primary focus, their study does quantify the space overhead associated with the use of a two-word header and of 8-byte alignment. In general, our measurements of the memory system behavior of Java programs broadly agree with their measurements.

Sweeney and Tip [18] did a study of dead members of C++ programs, which is similar to the unread field elimination done by our bitwidth analysis. However, they fail to identify *constant* members, which our SCC-based algorithm does easily. Further, our results show that unread and constant field elimination is very dependent on the coding style of a particular application. The collection of techniques we have presented here gives much more consistent savings over a wide range of benchmarks.

7 Conclusions

We have presented a set of techniques for reducing the memory consumption of object-oriented programs. Our techniques include program analyses to detect unused, constant, or overly-wide fields, and transformations to eliminate fields with common default values or usage patterns. These techniques apply equally well to both user-defined fields and fields implicit in the runtime's object header, and can reduce the maximum heap required for a program by as much as 40%. Our experimental results from our fully-implemented system validate the opportunity for space savings on typical object oriented programs.

References

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[2] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the 6th International Static Analysis Symposium*, September 1999.

[3] C. Scott Ananian. Silicon C: A hardware backend for SUIF. Available from `http://flex-compiler.lcs.mit.edu/SiliconC/paper.pdf`, May 1998.

[4] C. Scott Ananian. The static single information form. Technical Report MIT-LCS-TR-801, Massachusetts Institute of Technology, 1999.

[5] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268, Montreal, Canada, 1998.

[6] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 324–341, California, 1996.

[7] J. Bogda and U. Hoelzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[8] M. Budiu, S. Goldstein, M. Sakr, and K. Walker. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the EuroPar 2000 European Conference on Parallel Computing*. Munich, Germany, August 2000.

[9] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[10] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, August 1999.

[11] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)*, pages 254–263, Snowbird, Utah, June 2001.

[12] T. Onodera and K. Kawachiya. A study of locking objects with bimodal fields. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[13] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[14] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 182–195, Vancouver, Canada, June 2000.

[15] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, UT, June 2001.

[16] Vugranam C. Sreedhar, Michael Burke, and Jong-Deok Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 196–207. ACM Press, 2000.

[17] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[18] Peter F. Sweeney and Frank Tip. A study of dead data members in C++ applications. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, 1998.

[19] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[20] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.