# Data Size Optimizations for Java Programs

C. Scott Ananian
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
cananian@lcs.mit.edu

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
rinard@lcs.mit.edu

## ABSTRACT

We present a set of techniques for reducing the memory consumption of object-oriented programs. These techniques include analysis algorithms and optimizations that use the results of these analyses to 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. Although it is possible to apply these techniques to any object-oriented program, we expect they will be particularly appropriate for memory-limited embedded systems.

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.

## Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages, Java*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; D.3.4 [**Programming Languages**]: Processors—*Optimization*; E.2 [**Data Storage Representations**]: Object representation

## General Terms

Languages, Performance, Experimentation, Algorithms

This research was supported by DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR-0086154, NSF Grant CCR-0073513, NSF Grant CCR-0209075, and the Singapore-MIT Alliance.

*LCTES'03,* June 11–13, 2003, San Diego, California, USA.

## Keywords

Embedded systems, size optimizations, static specialization, field externalization, field packing, bitwidth analysis

## 1. INTRODUCTION

We present 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 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:** We use rapid type analysis to compute 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%, although the running time may increase. In a memory-limited embedded system where performance is not critical, cost savings may directly result from the reduced minimum heap size.

## 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. EXAMPLES

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

## 2.1 Field Reduction and Constant Field Elimination

Figure 1 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. Assume that the analysis examines the rest of the program (not shown) and 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).

```
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;
   }
}
```

**Figure 1: The JValue class.**

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 without external fragmentation, we can reduce the size of all allocated JValue objects by the full six bytes.

## 2.2 Static Specialization

Figure 2 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 2), 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 nonzero offset. And even at calls to the private String(int, int, char[]) constructor inside the

[1] This class is a simplified 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 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);
   }
}
```

**Figure 2: Portions of the `java.lang.String` class.**

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 nonzero 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 3 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 4.) 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. In the next phase, we determine if the initialized value of the field can be determined before the object is created, by examining the specific constructor invoked and its parameters. 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.

In this example all of the relevant fields are `private`, which would, in principle, enable an implementation to apply the optimization with an analysis of only the `String` class. Our analysis, however, is powerful enough to examine the rest of the program and discover the facts required to apply the optimization in the absence of `private` or `final` declarations and even for fields accessed outside their declaring class.

```
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; }
}
```

**Figure 3: Static specialization of `java.lang.String`.**

```
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);
}
```

**Figure 4: Dynamic selection among specialized classes in a method from `java.lang.String`.**

## 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 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

[2] See Section 3.5 for a precise definition.

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 [8] 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 the program can contain type-checks on and method-invocations of 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 over half 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`, `javac`, and `jack` benchmarks require more than one byte of index, but a two byte index amply suffices in these three cases.

## 3.2 Bitwidth Analysis

We use a flow-sensitive interprocedural combined value-propagation and bitwidth analysis to find constant values, unread and constant fields, and to reduce field sizes where possible. 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.

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

**Figure 5: Some combination rules for bitwidth analysis of arithmetic and bitwise-logical operators. Note that the penultimate entry is a special-case rule that only applies if the neither of the arguments can be negative.**

We abstract non-singleton sets of 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 5. 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, and (as the figure shows) bitwise operations on positive numbers.

### 3.2.1 Treatment of Fields

Dataflow on this bitwidth lattice is performed on the entire Java program interprocedurally. The analysis is *field-based* [13]: for each field $f$ in class $X$, the analysis uses the abstract analysis value $X.f$ to represent all of the values in the $f$ field of instances of $X$. The analysis therefore models an assignment to $f$ in any instance of $X$ as an assignment to the corresponding analysis value $X.f$.[5] The result of the analysis is a bitwidth specification for each variable and field in the program. We also identify constant variables and fields; we replace reads of constant fields with their constant value and eliminate the field. Fields for which no reads are found (even if writes are present) are also eliminated.[6]

### 3.2.2 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

[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] Note that *all* have more than 256 *total* class types.

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

[6] Note that checks which may throw exceptions on reads and writes are preserved.

| Benchmark | total<br>fields | unread | constant | % alloc'ed<br>space saved |
|---|---|---|---|---|
| compress | 298 | 75 | 31 | 2.5% |
| jess | 485 | 91 | 43 | 9.9% |
| raytrace | 341 | 75 | 30 | 0.0% |
| db | 286 | 75 | 35 | 0.0% |
| javac | 531 | 85 | 34 | 0.6% |
| mpegaudio | 286 | 75 | 35 | 1.4% |
| mtrt | 341 | 75 | 30 | 0.0% |
| jack | 378 | 77 | 31 | 10.2% |

**Table 1: Number of unused and constant fields in SPEC benchmarks, and the savings realized (in % of total dynamic allocated bytes) by removing them.**

| | static field bits | | % alloc'ed |
|---|---|---|---|
| Benchmark | before | after | space saved |
| compress | 7591 | 5430 | 3.0% |
| jess | 13349 | 10634 | 30.1% |
| raytrace | 7467 | 5296 | 0.9% |
| db | 6777 | 4983 | 0.3% |
| javac | 11560 | 8161 | 5.4% |
| mpegaudio | 6777 | 4983 | 1.5% |
| mtrt | 7467 | 5296 | 0.9% |
| jack | 8356 | 6037 | 17.2% |

**Table 2: Number of field bits in SPEC benchmarks statically removed due to bitwidth analysis, and the dynamic savings (in % of total allocated bytes) of field bitwidth reduction using byte packing.**

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 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 [5] for an exhaustive description of the intraprocedural analysis.

In Table 1 we show the number of unread and constant fields found by this analysis in our benchmark set. Table 2 shows the space reductions due to bitwidth analysis and field reduction using our byte packing strategy.

## 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 starting value, however, prevents us from finding nonzero 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 $\mathbf{5}$![7]

We perform a *definite initialization* analysis to remedy this problem and restore precision to our analysis. For example, with only constructor 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 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 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 with 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.

When performing bitwidth analysis, definitely-initialized fields are allowed to start at $\bot$ in the dataflow lattice.[8] All other fields must start at value $\mathbf{0}$, which will make it impossible for the field to represent a nonzero 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 eleven 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. It should be stressed that our use of multiple separate binaries was solely for ease of implementation, and is not an inherent limitation of the technique.

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.[9] 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 nonzero $N$ 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

[7] On the SCC lattice of [22], $\mathbf{0} \sqcap \mathbf{5} = \top$ (but see footnote 8).

[8] We use $\bot$ for "nothing known" and $\top$ for "underconstrained"; another segment of the compiler community commonly reverses these definitions.

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

| Benchmark | Field | always-zero<br>bytes | | field bytes<br>dyn. alloc'd | zero<br>% | benchmark<br>total dyn. alloc'n |
|---|---|---:|---|---:|---:|---:|
| compress | Hashtable$Entry.next | 3,552 | / | 7,148 | 49.7% | 105MB |
| | String.offset | 3,180 | / | 3,500 | 90.9% | |
| jess | jess.Token.negcnt | 7,573,616 | / | 7,573,616 | 100.0% | 252MB |
| | jess.Value.floatval | 5,688,080 | / | 10,170,640 | 55.9% | |
| raytrace | Point.z | 4,101,328 | / | 17,464,188 | 23.5% | 126MB |
| | Point.x | 3,291,076 | / | 17,464,188 | 18.8% | |
| db | String.offset | 508,204 | / | 508,524 | 99.9% | 73MB |
| | Vector.capacityIncrement | 62,548 | / | 62,548 | 100.0% | |
| javac | String.offset | 3,735,388 | / | 3,847,816 | 97.1% | 161MB |
| | Statement.labels | 578,608 | / | 578,688 | 100.0% | |
| mpegaudio | Hashtable$Entry.next | 3,616 | / | 7,336 | 49.3% | 666kB |
| | String.offset | 2,352 | / | 2,672 | 88.0% | |
| jack | String.offset | 7,442,956 | / | 7,443,276 | 100.0% | 178MB |
| | Hashtable$Enumerator.type | 5,288,364 | / | 5,288,364 | 100.0% | |

**Table 3: Representative "mostly-zero" fields found in SPEC benchmarks.**

increment the "non-$N$" counter on any path where the field in question is not definitely initialized.

Table 3 presents some representative "mostly-zero" fields which our profiling technique identifies in the SPEC benchmarks.

### 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 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.[10]

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. Other writes are disallowed. Unlike fields marked with Java's `final` modifier, multiple writes to `f` are permitted, as long as each write satisfies the above constraints.

Subclass-finality matches the requirements of the static specialization transformation. Since we always insert a "big" version of the class between the specialized class and its children, subclasses can write to the field present 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 dynamically possible.

This analysis 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 $N$th 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 for calls to this particular constructor. 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.

### 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.[11]

Some architectures penalize unaligned accesses to fields. It is worthwhile to *attempt* to align fields to their preferred alignment while not allowing this alignment to cause the object size to grow. Further, there are often *forced* alignment

[10] 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.

[11] Note also that "bit-packing" may lead to the loss of atomicity on concurrent writes to adjacent fields packed within a byte, typically the processor's smallest atomic write size. An escape analysis would be sufficient to ensure that fields accessed from differing threads are not packed within the same atomic unit.

constraints on (for example) pointers. Our Java runtime uses a conservative garbage collector; its efficiency decreases markedly if pointers are not word-aligned.[12]

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.

We have observed that this heuristic strategy works well in practice, and the penalties for occasionally placing an unaligned non-pointer field were not seen to have a material adverse effect on performance (see Section 5.3).

## 4.2 External Hashtable Implementation

The implementation of the hashtable used for field and hash/lock externalization can dramatically affect the space savings possible with these transformations. 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 stored data, all that is necessary is a key value and the empty space 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 puts a weak reference to the object in the hashtable, enabling the garbage collector to remove unneeded entries.

## 4.3 Class Loading and Reflection

We conducted this research using the MIT FLEX compiler infrastructure,[13] which is a whole-program static compiler. Although the analyses as described reflect this compilation model, it would be straightforward to use *extant analysis* [18] 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. Finally, our various transformations need not be exposed to the program 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 with the full input size 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.[14]

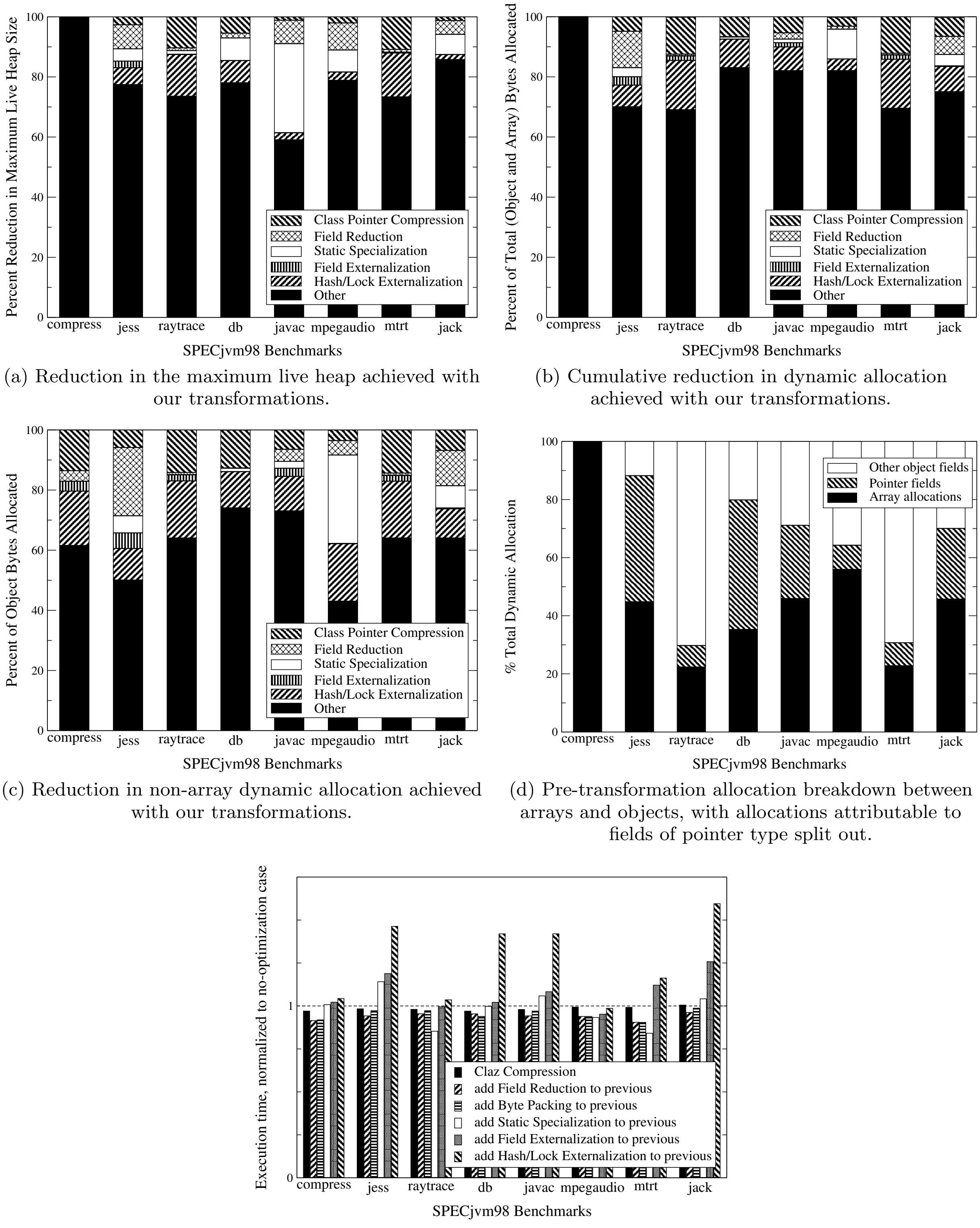
Figure 6a presents the total space savings. This figure contains a bar for each application, with the bar broken down into categories that indicate the percentage of live data from the original unoptimized execution that we were able to eliminate with each optimization. The black section of each bar 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 6b presents the space savings according to this metric. Comparison to the previous figure reveals that long-lived objects provide proportionally more opportunities for optimization.

[12] This pointer alignment restriction means that objects have to be word-aligned as well.

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

[14] The instrumented versions collect all non-live data before each allocation, so that our computed maximum heap sizes are accurate.



(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.

(e) Runtime performance of space optimizations.

**Figure 6: Experimental results of space optimization transformations.**

## 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 6c presents space savings numbers for objects alone, omitting any storage required for arrays. Figure 6d 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 6e presents the normalized execution times of each benchmark 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 [7, 15, 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 [7]; Bacon et al. in [6] show speed improvements from header-size reduction, in agreement with the results presented here.

Research on escape analysis and related analyses can enable the compiler to find objects whose locks are never acquired [3, 9, 23, 11, 16, 20]. 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 [4, 5, 17, 19, 10].

Dieckmann and Hölzle have performed an in-depth analysis of the memory allocation behavior of Java programs [12]. 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 [21] 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, as our analysis algorithm can. 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.

Aggarwal and Randall [2] described an array bounds check removal method using *related fields*. This work attempted to discover fields, such as `Vector.size`, which are guaranteed to be less than or equal to the length of some array, for example, the backing array stored in `Vector.data`. Tests against the related field could then provide information about bounds checks on accesses to the array. This technique could be used to infer additional bitwidth information on related fields from our analysis.

Marinov and O'Callahan have presented Object Equality Profiling [14], a technique which identifies when several instances of an object may be safely merged to a single representative instance. The merging which is suggested is an orthogonal memory-saving measure which could be used in addition to the ones described here.

Zhang and Gupta describe a runtime technique that recognizes two special cases when an integer or a pointer field in a designated C data structure may be compressed [24]. For all but two of their benchmarks, their heap savings (on these benchmarks, an average of 27%) are entirely due to a pointer compression techique which is orthogonal to the transformations described in this paper. The techniques could be combined for greater savings.

# 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.

# 8. 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*, pages 207–222, Denver, Colorado, November 1999.

[2] Aneesh Aggarwal and Keith H. Randall. Related field analysis. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)*, pages 214–220, Snowbird, Utah, June 2001.

[3] 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*, pages 19–38, September 1999.

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

[5] C. Scott Ananian. The static single information form. Technical Report MIT-LCS-TR-801, Massachusetts Institute of Technology, 1999. Available from `http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-801.pdf`.

[6] David F. Bacon, Stephen J. Fink, and David Grove. Space- and time-efficient implementation of the Java object model. In B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 111–132, Málaga, Spain, June 2002.

[7] 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.

[8] 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.

[9] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 35–46, Denver, Colorado, November 1999.

[10] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the 2000 Europar Conference*, volume 1900 of *Lecture Notes in Computer Science*, Munich, Germany, August 2000. Springer Verlag.

[11] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–19, Denver, Colorado, November 1999.

[12] 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.

[13] 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.

[14] Darko Marinov and Robert O'Callahan. Object equality profiling. Submitted to OOPSLA '03, 2003.

[15] Tamiya Onodera and Kiyokuni Kawachiya. A study of locking objects with bimodal fields. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 223–237, Denver, Colorado, November 1999.

[16] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 208–218, Vancouver, Canada, June 2000.

[17] Radu Rugină 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.

[18] 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.

[19] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 108–120, Vancouver, Canada, June 2000.

[20] Alexandru Sălcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 12–23, Snowbird, Utah, June 2001.

[21] 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)*, pages 324–332, Montreal, Canada, 1998.

[22] 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.

[23] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 187–206, Denver, Colorado, November 1999.

[24] Youtao Zhang and Rajiv Gupta. Data compression transformations for dynamically allocated data structures. In *International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 14–28, Grenoble, France, April 2002. Springer Verlag.