Run-time Evaluation of Opportunities for Object Inlining in Java

Ondřej Lhoták Laurie Hendren
Sable Research Group
School of Computer Science
McGill University
Montreal, Canada
{olhotak,hendren}@sable.mcgill.ca

ABSTRACT

Object-oriented languages, such as Java, encourage the use of many small objects linked together by field references, instead of a few monolithic structures. While this practice is beneficial from a program design perspective, it can slow down program execution by incurring many pointer indirections. One solution to this problem is object inlining: when the compiler can safely do so, it fuses small objects together, thus removing the reads/writes to the removed field, saving the memory needed to store the field and object header, and reducing the number of object allocations.

The objective of this paper is to measure the potential for object inlining by studying the run-time behaviour of a comprehensive set of Java programs. We study the traces of program executions in order to determine which fields behave like inlinable fields. Since we are using dynamic information instead of a static analysis, our results give an upper bound on what could be achieved via a static compiler-based approach. Our experimental results measure the potential improvements attainable with object inlining, including reductions in the numbers of field reads and writes, and reduced memory usage.

Our study shows that some Java programs can benefit significantly from object inlining, with close to a 10% speedup. Somewhat to our surprise, our study found one case, the `db` benchmark, where the most important inlinable field was the result of unusual program design, and fixing this small flaw led to both better performance and clearer program design. However, the opportunities for object inlining are highly dependent on the individual program being considered, and are in many cases very limited. Furthermore, fields that are inlinable also have properties that make them potential candidates for other optimizations such as removing redundant memory accesses. The memory savings possible through object inlining are moderate.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JGI'02, November 3–5, 2002, Seattle, Washington, USA.
Copyright 2002 ACM 1-58113-599-8/02/0011 ...$5.00.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Object-oriented languages, Java*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization, Memory management (garbage collection)*

General Terms

Languages, Measurement, Performance

Keywords

Object inlining, Java, compilers, optimization

1. INTRODUCTION

Object-oriented programs organize data in objects linked together with pointers. While this enhances modularity, making programs easier to understand and maintain, it has a run-time cost due to the pointers which take up memory, and must be dereferenced to access the data.

Object inlining consists of finding sets of objects that can be efficiently fused into larger objects, and fusing them. This reduces the time and space overhead associated with pointers, because pointers between the fused objects are no longer needed; the fields storing the pointers are said to be *inlined*.

Figure 1(a) shows a trivial program that can benefit from object inlining. The `Complex` class is used to store complex numbers, and the `NormFinder` manipulates them. In its original form, this program requires four reads of the field `z` for every call to its `normSq()` function. Also, every `NormFinder` object allocates a `Complex` object, with possibly considerable memory overhead. Figure 1(b) shows the program after the field `z` has been inlined. The fields `re` and `im` that previously had to be accessed through `z` can now be accessed directly. Also, the allocation of the `Complex` object `z` is now no longer necessary, so the memory overhead is reduced.

In C++, the programmer may explicitly request that an object be inlined into another by including the contained object (rather than a pointer to it) as a field inside the container object. The contained object then becomes part of the container object. In Java, this is not allowed; fields can contain only references to non-primitive objects, not the objects themselves. Even in C++, the burden of specifying which objects are to be inlined is on the programmer, rather than on the compiler.

Existing work describes static analyses and transformations for finding inlinable fields and inlining them [3–7]. In

```
class Complex {
    double re, im;
}
class NormFinder {
    Complex z;
    NormFinder( double re, double im ) {
        z = new Complex();
        z.re = re;
        z.im = im;
    }
    double normSq() {
        return z.re*z.re + z.im*z.im;
    }
}
```

(a) Before inlining

```
class Complex {
    double re, im;
}
class NormFinder {
    double z_re, z_im;
    NormFinder( double re, double im ) {

        z_re = re;
        z_im = im;
    }
    double normSq() {
        return z_re*z_re + z_im*z_im;
    }
}
```

(b) After inlining field `z`

Figure 1: Example of object inlining

contrast, this study examines run-time data about fields that could be inlined in a comprehensive collection of benchmark programs. It therefore gives an upper bound on the amount of inlining possible in typical programs, even if a perfectly precise compiler analysis were used. That is, our aim is to determine the potential usefulness of object inlining optimizations for typical Java programs.

In order to handle the existing approaches to object inlining, we propose three categories for inlinable fields: *simply one-to-one*, *field-specific one-to-one*, and *unique-store*. The inlining approach suggested by Dolby and Chien [3–5] corresponds to the first two categories, whereas the approach suggested by Laud [7] handles the first and third categories. We define these categories in terms of four predicates, contains-unique, unique-container-same-field, unique-container-different-field, not-globally-reachable, in order to make the definitions clear and comparable.

Our experimental approach consists of collecting dynamic traces of Java programs and categorizing each field as one of the three kinds of inlinable fields, or as non-inlinable. We then study the dynamic behaviour of the program in order to determine how many reads/writes could be eliminated if the inlinable fields were inlined, and how much memory could be saved. Based on these results, we determine which benchmarks are most likely to benefit from inlining, and we perform the inlining of the most important fields by hand. This allows us to determine the actual speedup that could be obtained for those benchmarks.

The following are our main findings:

- On some benchmarks, most notably `compress`, object inlining could eliminate up to 90% of field reads, and produce significant speedups (close to 10%).
- Our dynamic numbers indicate that one inlinable field in the `db` benchmark is very important. To our surprise, when we studied the source code for this field, we found that it was the result of unusual program design, and when this design flaw was fixed by effectively inlining the field, both the program design and the program performance improved.
- However, in general, opportunities for object inlining are highly dependent on the benchmark, and are often very limited.
- The main benefit from object inlining is the reduction in the number of field reads; the potential reduction in the number of field writes is relatively minor.
- Object inlining potentially saves moderate amounts of memory, reducing garbage collector pressure. Up to 6 MB of allocations could be saved during the execution of most benchmarks, accounting for up to 7% of all space allocated, and up to 21.6% of all objects allocated.

This paper is organized as follows. First, we describe other work related to object inlining in Section 2. In Section 3, we present the criteria used to determine whether a given field could be inlined. In Section 4, we explain our experimental setup. We report our results in Section 5. Finally, we conclude in Section 6.

2. RELATED WORK

The most well-known work on object inlining is by Dolby and Chien [3–5]. They formulate a definition of dynamic one-to-one fields, and show that such fields can be inlined in a way that preserves semantics. They also describe an implementation of their transformation in ICC, a compiler for a language similar to both Java and C++, and evaluate its cost and effectiveness on several C++ benchmarks ported to their compiler. They do not provide experimental results for Java benchmarks, so one goal of our study is to measure the potential impact of their approach when applied to Java.

Laud [7] gives a slightly different definition of inlinable fields, and explains the details that would be required in a Java compiler implementing the optimization. He does not provide an implementation or experimental results. Our study measures the opportunities for inlining that are possible via Laud's approach, but not possible via Dolby and Chien's approach and vice versa.

Ghemawat, Randall, and Scales [6] describe object inlining as one application of their field analysis. Their analysis is context-insensitive, and takes advantage of access modifiers to limit the amount of code that needs to be analyzed for each field. This makes their analysis less aggressive, but much faster, than Dolby and Chien's.

Shuf et al. [9] measure run-time data behaviour in typical Java programs, as does our study. However, they do not address the specific issue of object inlining.

3. DEFINITIONS

In this section, we describe ways in which a field could be inlined, and the behaviours that the field must exhibit to be inlinable in these ways. We start with a simple definition, and then extend it in two different ways. Although other definitions of fields suitable for inlining are conceivable, our focus is on determining the effectiveness of the inlining approaches that have already been proposed [3–5, 7].

Before proceeding, we will make explicit some of the terminology used throughout this paper. By the term *object*, we mean a single specific instance, at run time, of some class. By the term *field*, we mean a single declaration of a field in some class. As such, a field is uniquely identified by its name and the class in which it is declared.

3.1 Simply one-to-one field

Definition 1 (Simply one-to-one field). *A* simply one-to-one field *f is a field for which:*

- [contains-unique] *every container object having the field f refers to only one contained object through f throughout its lifetime,*
- [unique-container-same-field] *none of these contained objects are ever referred to by this same field f of any other object,*
- [unique-container-different-field] *none of these contained objects are ever referred to by any field other than f of any object, and*
- [not-globally-reachable] *none of these contained objects are ever referred to by any static field, or by elements of any array.*

Simply one-to-one fields can be inlined both by the transformation suggested by Dolby and Chien [3–5], and by the one suggested by Laud [7].

Figure 2 depicts such a field. The container objects p and q, throughout their lifetimes, each correspond to exactly one contained object through the field f: c and d, respectively. References to these contained objects are not stored in any other instance fields, static fields, or array elements.

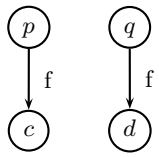


Figure 2: Simply one-to-one field

In order to inline a simply one-to-one field, a compiler must gather additional information in addition to proving that the field is simply one-to-one. First, it must determine the flow of the container and contained objects from their allocation sites to the point where the reference to the contained object is written to the field. Second, it must find all uses of the field in the program. The allocation sites of the two objects are replaced by a single site allocating a combined object. This combined object contains the fields and methods of both the container and contained objects, but not the inlined field. Each load of the field (`y = x.f`) in the program is converted to a copy (`y = x`), since the object `x` now has all the fields of `x.f`. Issues related to the methods of the objects and their types are discussed in detail in Dolby and Chien's papers [3–5]; we do not discuss them here.

In Figure 3, we see the effect of inlining the field from Figure 2. The contents of the contained objects, c and d, have been merged into the container objects, p and q, respectively. For each inlined field, we save, at run time, all the loads and stores that would be performed through that field. Also, for each pair of container and contained object allocated, we save one object allocation (because the objects are allocated together), and an amount of memory equal to the size of a pointer, plus the object header overhead of the contained object.

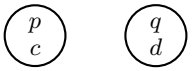


Figure 3: Simply one-to-one field after inlining

3.2 Field-specific extension

The definition of simply one-to-one fields just presented does not capture relationships like the one in Figure 4, in which more than two objects are created together, and connected by pointers. In order to inline such cases, we extend the definition by removing the unique-container-different-field and not-globally-reachable conditions.

Definition 2 (Field-specific one-to-one field). *A* field-specific one-to-one field *f is a field for which:*

- [contains-unique] *every container object having the field f refers to only one contained object through f throughout its lifetime, and*
- [unique-container-same-field] *none of these contained objects are ever referred to by this same field f of any other object.*

Field-specific one-to-one fields can be inlined by the transformation suggested by Dolby and Chien [3–5], but **not** by the one suggested by Laud [7].

The fields f and g in Figure 4 are both field-specific one-to-one. Each container object (p and q) refers to only one object (c) throughout its lifetime through the field. The contained object is not referred to by the field f of any object other than p, and it is not referred to by the field g of any object other than q. The fields therefore satisfy the conditions for being field-specific one-to-one.

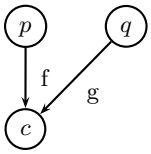


Figure 4: Field-specific one-to-one field

Field-specific one-to-one fields can be inlined one by one in the same way as simply one-to-one fields, by merging the allocation sites of the container and contained object, and by replacing references to the contained object with references to the container object. Figure 5 shows the effect of inlining field f from Figure 4. Notice how the field g, which was

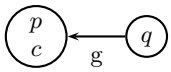


Figure 5: Field-specific one-to-one field after inlining f

referring to the object c contained in f, now refers to the container object, p.

Since pc has become a single object, the field g is now simply one-to-one, and can therefore be inlined in a second inlining step, as shown in Figure 6.

Figure 6: Field-specific one-to-one field after inlining f and g

Because field-specific one-to-one fields are inlined in the same way as simply one-to-one fields, the benefit from inlining them is the same. All loads and stores through them are eliminated, and for each pair of container and contained object, one allocation and the memory for one pointer and one object header is saved.

3.3 Unique-store extension

The definition of simply one-to-one fields can be extended in a different way, by simply dropping the contains-unique condition.

Definition 3 (Unique-store field). *A* unique-store field *f is a field such that*

- [unique-container-same-field] *none of the objects contained in f are ever referred to by this same field f of any other object,*
- [unique-container-different-field] *none of these contained objects are ever referred to by any field other than f of any object, and*
- [not-globally-reachable] *none of these contained objects are ever referred to by any static field, or by elements of any array.*

We call these fields unique-store because a reference to the contained object is only ever stored in a unique field of a unique object, and may not be stored in any other fields or array elements. Unique-store fields can be inlined by the transformation suggested by Laud [7], but **not** by the one suggested by Dolby and Chien [3–5].

Because a unique-store field of a container object may refer to multiple contained objects throughout its lifetime, it must be inlined differently from a one-to-one field. The allocation sites of container and contained object are not merged. The inlined field is deleted from the container object, and the contents of the contained object are inserted. The field store (`y.f = x`) is replaced by a copy of the contents of `x` into the corresponding fields that have been added to `y`. As before, all references to `y.f` are replaced with references to `y`. Because container objects with unique-store fields are not created together with their contained objects, the compiler must prove that at the program point where the reference to the contained object is stored into the field, there are no live aliases to it, or to the contained object that is being overwritten. By the definition of unique-store fields, such aliases could only be in local variables. Because the allocation sites are not merged, and updates are replaced with copies, this transformation might increase the space needed and add copying overhead.

Figure 7 shows an example of a unique-store field. The field f is unique-store because the objects referred to by the field f of p during p's lifetime, c and d, are not referred to anywhere else. The field would be inlined as in Figure 8. The contained objects c and d still exist independently before they are stored into the container. The crossed-out c reflects the fact that the contents of c are first stored into the container, and later overwritten with the contents of d.

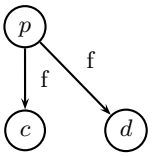


Figure 7: Unique-store field

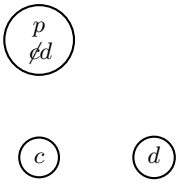


Figure 8: Unique-store field after inlining

3.4 Non-inlinable fields

Figure 9 shows an example that does not fit any of these definitions of inlinable fields. The field f is not one-to-one (neither simply nor field-specific) because a reference to the contained object c is stored into *this same* field f of two container objects p and q. It is also not unique-store, because the object c referred to by p through it is also referred to by q, and vice-versa. Because such a field cannot be inlined according to any of the procedures described, we consider such a field to be non-inlinable for our purposes.

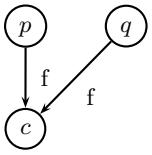


Figure 9: Non-inlinable field

4. EXPERIMENTAL SETUP

We used the Soot framework [10–12] to instrument the bytecode of several benchmarks to generate traces of instance field stores and loads, and of array element and static field stores. We then analyzed the traces produced by running the benchmarks, and identified fields that satisfied the definitions of simply one-to-one, field-specific one-to-one, or

Benchmark	Description	Classes	Statements	Fields
`compress`	Lempel-Ziv compressor/decompressor	39	7322	23
`db`	Memory resident database	30	7293	11
`jack`	Parser generator	80	16792	66
`javac`	Java compiler	206	31069	170
`jess`	Expert system	175	17488	65
`mpegaudio`	MP3 decompressor	78	19585	108
`mtrt`	Multi-threaded ray-tracer	53	10067	45
`raytrace`	Single-threaded ray-tracer	52	10037	45
`javasrc-p`	Java source HTML pretty-printer	161	40667	82
`kawa-c`	Scheme to bytecode compiler	437	33228	163
`rhino-a`	Javascript interpreter	80	26328	99
`sablecc-j`	Parser generator, Jimple grammar	326	26911	280
`sablecc-w`	Parser generator, Wig grammar	326	26911	297
`schroeder-m`	Audio editor, medium-length input	102	9713	27
`schroeder-s`	Audio editor, short input	102	9713	27
`soot-c`	Bytecode optimizer	621	42107	241
`toba-s`	Bytecode to C compiler	41	17504	56

Table 1: Benchmarks

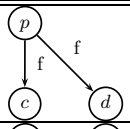
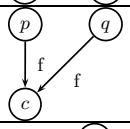
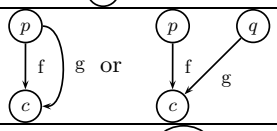
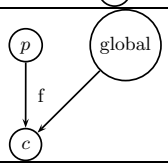
$store_i$	$store_j$	Relationship	Predicates
`p.f = c`	`p.f = d`		contains-unique(f) $\leftarrow$ false
`p.f = c`	`q.f = c`		unique-container-same-field(f) $\leftarrow$ false
`p.f = c`	`p.g = c` or `q.g = c`		unique-container-different-field(f) $\leftarrow$ false unique-container-different-field(g) $\leftarrow$ false
`p.f = c`	`Class.g = c` or `a[i] = c`		not-globally-reachable(f) $\leftarrow$ false

Table 2: Rules for computing predicates

unique-store. For each such field, we counted the number of stores and loads of the field. These loads and stores would be eliminated if the field were inlined (except for unique-store fields, for which only loads would be eliminated, and stores would be turned into copies).

4.1 Benchmarks

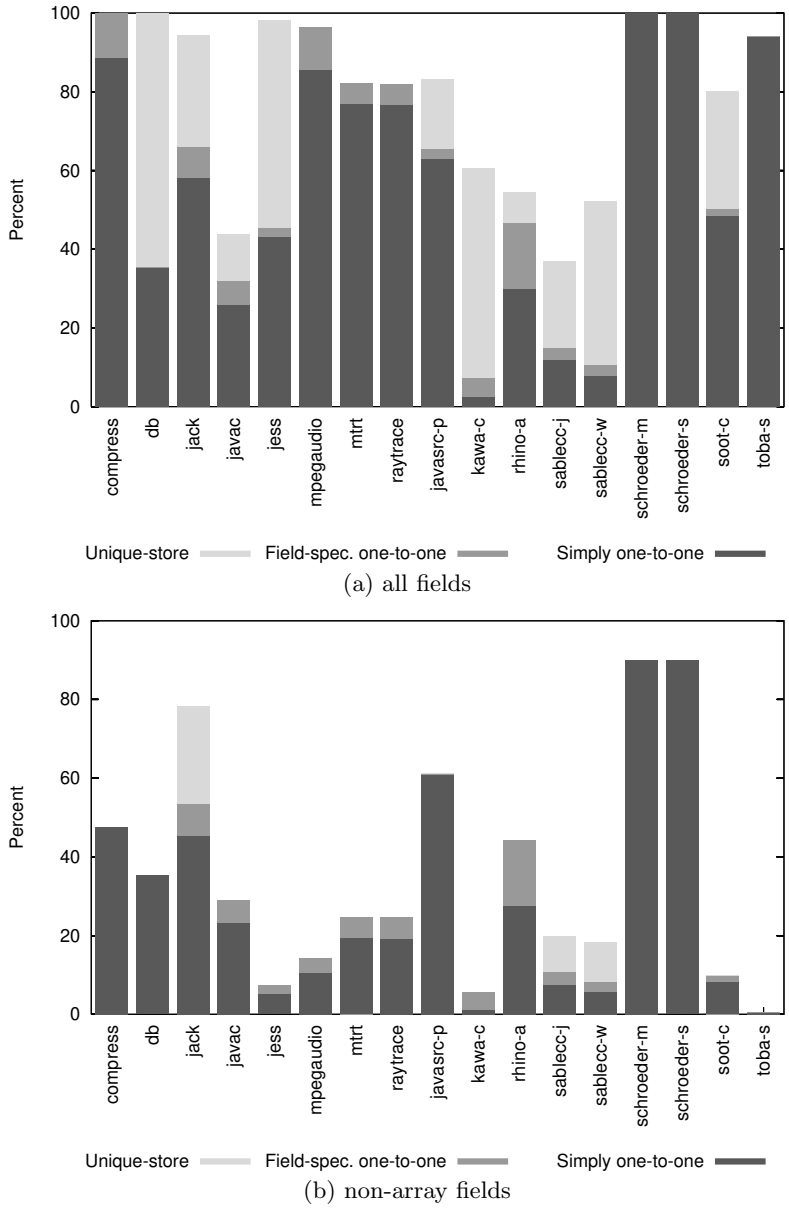
Our benchmarks were taken from the commonly-used SPECjvm [2] benchmark suite, and from the `ashesJSuite` portion of the Ashes [1] benchmark suite. The latter consists of real-world programs of non-trivial size. Brief descriptions of the benchmarks are given in Table 1, along with the number of classes, Jimple[1] statements, and instance fields of reference type (including arrays) in each benchmark. We count only fields that are read or written at least once during execution of the benchmark. For two of the benchmarks (`sablecc` and `schroeder`), we run the same code on two different input sets, and observe similar behaviour.

[1]Jimple is the three-address representation used by our Soot [10–12] bytecode manipulation framework.

4.2 Trace Analysis

Given program traces containing all instance field stores (instructions of the form `p.f = c`), static field stores (instructions of the form `ClassName.f = c`) and array stores (instructions of the form `a[i] = c`), we analyze the program traces using a three-step process.

First, we compute the predicates for each field. We begin by eliminating duplicate entries of instance field stores to reduce the trace. We then compute the predicates contains-unique, unique-container-same-field, unique-container-different-field, and not-globally-reachable for each field. To do this, we start by initializing each predicate to true for every field. This corresponds to initially estimating that all fields are inlinable. Then, for all pairs of statements $store_i$ and $store_j$ from the reduced trace, we check to see if any of the four patterns given in the first two columns of Table 2 apply. Note that in all the patterns given in the table, `p` and `q` are



(a) all fields

(b) non-array fields

Figure 10: Field reads through inlinable fields

distinct objects, as are `c` and `d`, and `f` and `g` are distinct fields. Each such pattern of stores creates the relationship depicted in the third column of the table, causing the violation of one of the four predicates. Therefore, for each pair of stores matching one of the patterns, we set the appropriate predicate(s) (given in the fourth column) to false.

Second, based on the values of these predicates, we classify each field as *simply one-to-one*, *field-specific one-to-one*, *unique-store*, or *non-inlinable*, according to the definitions given in Section 3. For example, for a field f, using Definition 1, we would classify f as *simply one-to-one* if contains-unique(f) and unique-container-same-field(f) and unique-container-different-field(f) and not-globally-reachable(f).

Note that according to our definitions, if a field is *simply one-to-one* it is also *field-specific one-to-one* and *unique-store*. However, for our categorization, we only count a field as *field-specific one-to-one* when it is *field-specific one-to-one*, but **not** *simply one-to-one*. Similarly, we categorize a field as *unique-store* only when it is *unique-store*, but **not** *simply one-to-one*.

Third, for each field, we count the number of loads and stores through that field, and the number of unique contained objects that are referred to by that field. We summarize these counts for the four different categories of fields.

5. EXPERIMENTAL RESULTS

5.1 Inlinable fields

Figure 10(a) shows the proportion of field reads executed (`getfield` instructions) that read from fields that were determined to be inlinable. For most of the benchmarks, this is the majority of all field reads. We can see that most of

the field reads occur through fields that are either simply one-to-one, or unique-store.

This measurement may be overly optimistic, because it includes all fields, including fields of array type. Fields of array type account for a large proportion of field reads, which is not surprising, because in Java, the only way to include arrays in objects is as fields. The Java language does not make it possible, in bytecode, to inline the elements of an array into the object containing the array. Even if this were possible (if an object inlining optimization were performed by the virtual machine, for example), fields of array type could not be inlined unless the size of the array were constant and known at compile time.

Figure 10(b) shows the proportion of all field reads executed that read from fields *of non-array type* that were determined to be inlinable. This proportion is significantly smaller for many of the benchmarks, and is much more dependent on the individual benchmark, ranging from 0.5% for `toba-s` to 90.0% for `schroeder-m`.

Most of the fields inlinable according to the unique-store extension are arrays in structures such as hash tables, which are resized (replaced with larger arrays) when they become full. The previously mentioned difficulties with inlining fields of array type therefore apply. Very few unique-store fields are of non-array type.

When fields of array type are ignored, field-specific one-to-one fields become relatively more significant, especially in `rhino-a`, where they account for 16.6% of all field reads. The most significant such field is a `java.lang.String` that is stored in several related objects. However, in general, most inlinable fields could be found by an analysis recognizing only simply one-to-one fields, without the field-specific and unique-store extensions.

For many of the benchmarks, a small number of inlinable fields account for a majority of the reads via inlinable fields at run time. Table 3 shows the number of inlinable fields accounting for 50%, 75%, 90%, 99% and 100% of the run-time reads via inlinable fields. For example, for the `compress` benchmark, the two most important inlinable fields account for at least 50% of all reads of inlinable fields, whereas the seven most important fields account for 99% of the reads and nine inlinable fields account for 100% of the reads. The column labeled Total gives all inlinable fields in the benchmark and the small difference between the columns 100% and total is due to some inlinable fields that are written but never read. In the case of `compress`, there are four fields that are inlinable, but never read.

Note that 50% of the reads are due to a very small number of fields (usually ≤ 3) and even 90% of the reads can usually be accounted for by fewer than 10 fields. The exceptions (`kawa-c`, `sablecc-j`, `sablecc-w`, and `soot`), where more than 10 fields are needed, correspond to those benchmarks where there are relatively few reads via inlined fields, and these benchmarks are unlikely to show much improvement in any case.

In many benchmarks (`compress`, `jack`, `javac`, `javasrc-p`, `schroeder-m`, and `schroeder-s`), some of the most important inlinable fields refer to objects used for input and output, and the fields are read for every unit of data input or output. Many of these field reads could probably be eliminated by methods other than field inlining, such as hoisting invariant loads out of loops.

In many other benchmarks (notably `db`, `javac`, `jess`, `kawa-c`, `mtrt`, `raytrace`, `rhino-a`, `sablecc-j`, `sablecc-w`, and `soot-c`), the most important inlinable fields are used to store data being manipulated by the benchmark. Across almost all benchmarks, fields referring to objects from the Java standard class library account for a majority of the non-array inlinable field reads. In this study, fields *contained in* objects defined in the standard class library were not considered, but in an earlier study [8], we found nearly all non-array fields declared in the standard class library to be accessed a negligible number of times.

Benchmark	50%	75%	90%	99%	100%	Total
`compress`	2	5	6	7	9	13
`db`	1	1	1	1	7	8
`jack`	2	5	7	12	41	45
`javac`	1	3	8	25	61	64
`javasrc-p`	2	4	6	15	35	38
`jess`	2	4	5	11	37	41
`kawa-c`	6	12	20	37	49	60
`mpegaudio`	2	3	4	22	66	72
`mtrt`	1	3	5	10	27	29
`raytrace`	1	3	5	10	28	30
`rhino-a`	2	2	3	5	37	43
`sablecc-j`	2	6	12	42	220	221
`sablecc-w`	2	5	11	34	233	235
`schroeder-m`	2	3	4	4	13	17
`schroeder-s`	2	3	4	4	13	17
`soot-c`	3	8	20	52	122	144
`toba-s`	3	4	6	10	12	15

Table 3: Number of fields accounting for a percentage of reads of inlinable fields

In many cases, objects referred to by inlinable fields are either created in the constructor of the container object, or are passed as arguments to the constructor. This should make it relatively easy for a static analysis to find these fields. In some of the benchmarks, some of these fields are even declared `private`.

Table 4 shows the actual numbers of fields found to be inlinable, as well as the number of reads, writes, and unique writes (that is, the number of unique contained objects) through them, and the number and total size of allocations that could be eliminated, as a percentage of all allocations.

Some of the benchmarks store references to slightly over half a million unique contained objects in inlinable fields. For each one-to-one field inlined, the memory for the object header of the contained object is saved, as well as the memory used by the field. Assuming 32-bit words, and two-word object headers, this amounts to a savings of 6MB of allocations for half a million contained objects inlined. In most of the benchmarks, the allocations that could be saved account for less than 3% of all space allocated, and often less than 1%. However, in the `soot-c`, `sablecc-j`, and `sablecc-w` benchmarks, up to 4.9%, 5.8%, and 7.0%, respectively, of the space that is allocated might be saved by inlining, reducing the pressure on the garbage collector. A much larger fraction of allocated objects (rather than bytes) could be eliminated by object inlining: often over 5%, and up to 21.6% in `sablecc-w`. The benefit could be significant to a garbage collector with a high per-object (rather than per-byte) cost.

		Static	Dynamic				
		# of	Memory Accesses			Allocation Reduction	
Benchmark	Definition	Fields	Reads	Writes	Unique Writes	Objects	Bytes
compress	Simply one-to-one	9	544584500	300	275	4.5%	0.0%
	Field-specific one-to-one	4	65	130	80	1.3%	0.0%
	Unique-store	0	0	0	0		
	Non-inlinable	0	0	0	0		
db	Simply one-to-one	4	50983106	15639	15639	0.5%	0.2%
	Field-specific one-to-one	3	19060	7	7	0.0%	0.0%
	Unique-store	1	114	99	99		
	Non-inlinable	1	28	0	0		
jack	Simply one-to-one	25	7828037	39749	39749	0.7%	0.3%
	Field-specific one-to-one	16	1376969	8185	8185	0.1%	0.1%
	Unique-store	4	4321910	314160	125936		
	Non-inlinable	11	605268	545955	163768		
javac	Simply one-to-one	28	15819858	58801	47368	0.8%	0.3%
	Field-specific one-to-one	34	4025510	130249	103995	1.7%	0.6%
	Unique-store	2	33997	20655	6784		
	Non-inlinable	87	37643581	8613307	2542767		
jess	Simply one-to-one	29	4761562	417	417	0.0%	0.0%
	Field-specific one-to-one	11	2100116	638926	638907	8.1%	2.6%
	Unique-store	1	7620	2565	1520		
	Non-inlinable	15	1684460	729387	3237		
mpegaudio	Simply one-to-one	54	51658467	156	156	1.3%	0.2%
	Field-specific one-to-one	17	18451690	7852	42	0.3%	0.1%
	Unique-store	1	226	9	9		
	Non-inlinable	9	7467032	62554	29		
mtrt	Simply one-to-one	18	24725274	170000	169999	2.6%	1.2%
	Field-specific one-to-one	11	7003891	239659	235221	3.5%	1.7%
	Unique-store	0	0	0	0		
	Non-inlinable	9	22719376	489819	90508		
raytrace	Simply one-to-one	19	24210712	160155	160155	2.5%	1.2%
	Field-specific one-to-one	11	6987341	207721	205502	3.2%	1.5%
	Unique-store	0	0	0	0		
	Non-inlinable	8	22671967	424077	58527		
javasrc-p	Simply one-to-one	29	26794633	101918	89904	0.9%	0.2%
	Field-specific one-to-one	7	136601	344	344	0.0%	0.0%
	Unique-store	2	89774	9105	6775		
	Non-inlinable	36	4807725	1304184	315275		
kawa-c	Simply one-to-one	13	613145	11770	11770	0.5%	0.1%
	Field-specific one-to-one	47	2385489	381507	179634	8.0%	2.1%
	Unique-store	0	0	0	0		
	Non-inlinable	75	20761054	2343506	682739		
rhino-a	Simply one-to-one	25	4727125	415	388	0.0%	0.0%
	Field-specific one-to-one	16	2865628	66	66	0.0%	0.0%
	Unique-store	2	20	379	10		
	Non-inlinable	34	4953524	1506019	235425		
sablecc-j	Simply one-to-one	51	3010974	336058	336058	9.7%	3.0%
	Field-specific one-to-one	166	1293812	517558	319338	9.2%	2.8%
	Unique-store	4	3620447	1714	1714		
	Non-inlinable	47	25217192	8397133	1349424		
sablecc-w	Simply one-to-one	52	4759580	338800	338800	8.0%	2.6%
	Field-specific one-to-one	178	2224428	846338	574006	13.6%	4.4%
	Unique-store	5	8529493	1508	1411		
	Non-inlinable	50	40507880	12715587	2287516		
schroeder-m	Simply one-to-one	17	78942573	146	81	0.0%	0.0%
	Field-specific one-to-one	0	0	0	0	0.0%	0.0%
	Unique-store	0	0	0	0		
	Non-inlinable	6	16	72	6		
schroeder-s	Simply one-to-one	17	10936548	148	83	0.0%	0.0%
	Field-specific one-to-one	0	0	0	0	0.0%	0.0%
	Unique-store	0	0	0	0		
	Non-inlinable	6	16	72	6		
soot-c	Simply one-to-one	64	10708588	494247	493873	8.8%	3.2%
	Field-specific one-to-one	76	2267035	310096	268113	4.8%	1.7%
	Unique-store	4	276019	64479	64479		
	Non-inlinable	66	26151342	7580746	2341603		
toba-s	Simply one-to-one	11	108146	90353	89216	1.2%	0.4%
	Field-specific one-to-one	4	119823	20661	5934	1.2%	0.4%
	Unique-store	0	0	0	0		
	Non-inlinable	22	2527623	619301	209975		

Table 4: Absolute numbers of reads and writes through inlinable non-array fields

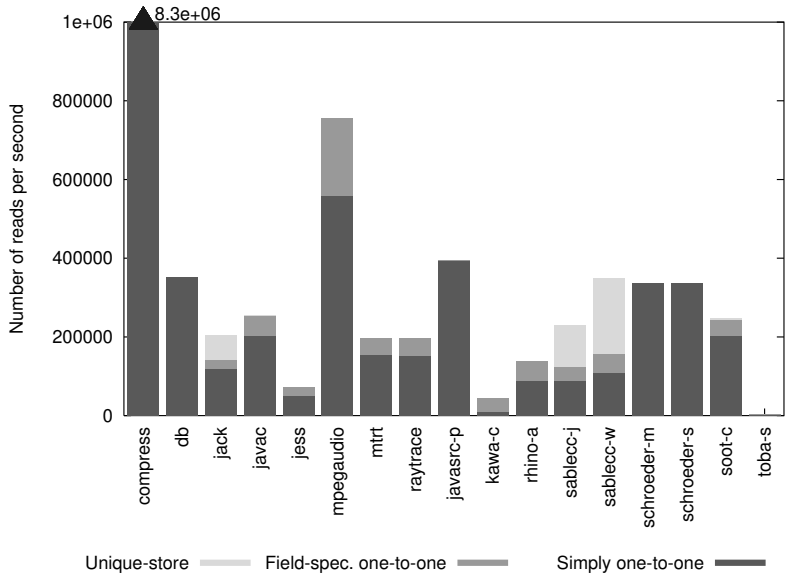


Figure 11: Number of field reads per second through inlinable non-array fields

When unique-store fields are inlined, the word of memory storing the field is saved, but the object header is not, because the contained object is still allocated separately. In addition, memory for the contents of the contained object must be allocated in the container object. We therefore expect an inlined unique-store field to use *more* memory than it would if it were not inlined. However, since the number of objects stored in unique-store fields is small, this overhead should be small.

5.2 Speedup

The relative number of field reads through inlinable fields is not necessarily predictive of the speedups attainable by field inlining, because different programs may perform very different numbers of field reads. Figure 11 shows the absolute numbers of field reads through inlinable non-array fields per second of execution of each benchmark. Note that the value for `compress` is much higher than for the other benchmarks, and did not fit on the graph; the actual value is 8.3×10^6. As this graph shows, the number of inlinable field reads, and therefore the potential speedup, is highly variable between benchmarks.

Compared to the number of reads, the number of writes is relatively small in most of the benchmarks. Any speedup due to eliminated field writes is therefore likely to be small. Aside from `jack`, the number of writes to unique-store fields is negligible, so the overhead of having to replace writes to unique-store fields with copies should be small.

In order to study the performance improvements attainable by inlining fields, we manually inlined the inlinable fields of non-array type in `compress`, `db`, and `javasrc-p`. We chose these benchmarks because they have relatively high numbers of field reads removable by inlining, and because unobfuscated source code for them was available. We compared the resulting benchmarks to the originals on an Intel Pentium II at 333 MHz with 384MB of memory running Linux 2.2.20, and a Sun UltraSPARC-III at 750 MHz with 1GB of memory running SunOS 5.8, using the latest released versions of Java virtual machines from Sun and IBM. The exact versions used were build 1.4.0-b92 of the Sun Client VM and build cx130-20020124 of the IBM Classic VM. Table 5 shows the speedups due to inlining, averaged over five runs of the benchmark on each platform.

Benchmark	Speedup		
	Pentium		SPARC
	IBM 1.3.0	Sun 1.4.0	Sun 1.4.0
`compress`			
object inlining	7.8%	8.4%	10.8%
loop invariant	3.5%	1.2%	5.0%
`db`			
object inlining	9.5%	4.0%	10.6%
`javasrc-p`			
object inlining	-0.4%	-0.4%	-1.7%

Table 5: Speedup from hand-optimizing fields

`Compress` has a very simple object hierarchy, with classes being used to group data, but with no complex relationships between objects. Seven fields account for over 99.9995% of the reads of inlinable fields, so we only inlined these seven fields.

In `db`, a single field accounts for 99.5% of reads of inlinable fields. This field is the only field in the class `Entry`, which is a wrapper class for `java.util.Vector`. Inlining the field is equivalent to making the `Entry` class extend `Vector`, rather than containing it. This simple transformation enables a significant speedup. In fact, given that `db` has many fewer reads per second than `compress` (see Figure 11), we were expecting that reducing the reads would lead to relatively little speedup. However, apparently inlining this one field has an enabling effect on other optimizations and a speedup of around 10% is exhibited.

The structure of `javasrc-p` is much more complex than `compress` and `db`. Fields that were easy to inline by hand accounted for 64% of the reads of inlinable fields; we only inlined these fields. Repeated experiments showed very little speedup or slowdown and we concluded that the effect of

inlining fields in this benchmark is not significant.

Unsurprisingly, we noticed that in `compress`, the inlinable fields accounting for most of the field reads were read in loops with large numbers of iterations. Furthermore, all these field reads were also loop invariant. This is also not surprising, and in fact follows from the definition of one-to-one fields (specifically from the `contains-unique` condition). We factored the loop invariant field reads out of the loops by hand. The resulting code eliminates almost all the field reads that could be eliminated by object inlining. The speedup, though smaller than the speedup from object inlining, is significant. A possible reason for the smaller speedup is that some of the loops that we optimized call other methods, so the loop invariant object that is read must be passed into these other methods, increasing the overhead of the method calls. Another possible reason is that object inlining may enable the virtual machine to perform more method inlining optimizations.

We emphasize that the direct benefit of object inlining, the reduction in the number of field reads, can sometimes be obtained by other optimizations such as loop invariant load hoisting, because one-to-one fields are by definition loop invariant, and because fields accounting for large numbers of reads are likely to be inside loops.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a categorization of inlinable fields, and we have reported results of a study of the potential usefulness of object inlining for typical Java programs. Our analysis of execution traces of Java programs allows us to quantify the potential effectiveness of object inlining as applied to Java.

Our study shows that on some Java programs, object inlining could be very effective, eliminating up to 90% of field reads and producing speedups of close to 10%. However, the opportunities for inlining are highly dependent on the individual benchmark, and for many benchmarks are quite limited. Furthermore, there may be complex interactions between object inlining and other compiler optimizations. For example, we showed that most of the field reads removed by object inlining could also be avoided by hoisting invariant reads out of loops. On the other side, the `db` benchmark showed that a single object inlining led to positive effects on other compiler optimizations, and a greater than expected speedup.

In addition, object inlining could save moderate amounts of memory (up to 6MB of allocations throughout the execution of most benchmarks, accounting for up to 7% of all space and 21.6% of all objects allocated), possibly reducing the workload of the garbage collector.

Although our study began as a way to estimate the potential benefit for static compiler techniques, we were pleasantly surprised about how useful the information was for the programmer. Relatively few fields are important for inlining, and in some cases, they are very easy for the programmer to modify. In the case of `db`, the modification of the program also led to a clearer program structure. Thus, we believe that our results should help compiler writers evaluate the usefulness of implementing object inlining in compilers. However, these sorts of profiles might also be useful as part of program development tools in order to help the programmer find the hot inlinable fields.

7. ACKNOWLEDGMENTS

This work was supported by NSERC and a Richard H. Tomlinson fellowship. We are grateful to Karel Driesen, Feng Qian, Navindra Umanee, Bruno Dufour, Patrick Lam, Rhodes Brown, and John Jorgensen for helpful discussions.

8. REFERENCES

[1] Ashes suite collection `http://www.sable.mcgill.ca/software/`.

[2] SPEC JVM98 benchmarks `http://www.spec.org/osg/jvm98/`.

[3] J. Dolby. Automatic inline allocation of objects. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 7–17, 1997.

[4] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 345–357, 2000.

[5] J. Dolby and A. A. Chien. An evaluation of automatic object inline allocation techniques. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–20, 1998.

[6] S. Ghemawat, K. H. Randall, and D. J. Scales. Field analysis: getting useful and low-cost interprocedural information. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 334–344, 2000.

[7] P. Laud. Analysis for object inlining in Java. In *JOSES workshop, a satellite event of ETAPS 2001*. `http://i44w3.info.uni-karlsruhe.de/~josesworkshop/01-laud.ps`.

[8] O. Lhoták. Run-time evaluation of object inlining opportunities in Java. Technical Report 2002-3, McGill University, School of Computer Science, `http://www.cs.mcgill.ca/resrchpages/reports/02/SOCS-02.3.ps.gz`, 2002.

[9] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Joint International Conference on Measurement and Modeling of Computer Systems*, pages 194–205, 2001.

[10] R. Vallée-Rai. Soot - a Java optimization framework. Master's thesis, McGill University, School of Computer Science, `http://www.sable.mcgill.ca/publications`, July 2000.

[11] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[12] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.