A Java™*Bytecode Optimizer Using Side-effect Analysis

Lars R. Clausen
elascurn@daimi.aau.dk

Department of Computer Science
Aarhus University

Abstract

This paper describes Cream, an optimizer for Java bytecode using side-effect analysis to improve the optimizations. Dead code elimination and loop invariant removal is implemented and tested, as well as several variations of the side-effect analysis. The optimizer is tested on real-world applications such as itself and JavaSoft's Java compiler. Results show that the optimizations benefit a lot from the side-effect analysis. The best side-effect analysis gives five to ten times as many optimizations as without an analysis, and, in one case, makes a sped increase of 25 % possible.

1 Introduction

Java, being a relatively young language, doesn't yet have as sophisticated optimizing compilers as other, more mature languages like C and Fortran. When designing the optimizers needed to improve Java's speed towards a more acceptable level, one must carefully consider the special traits of the language. In contrast with C programs, Java programs have many small method invocations, many structure references, extensive exception handling and dynamically-loaded classes. A successful optimizer must deal correctly and effectively with such programs.

Normally, optimization of Java code is done in two places: (1) In the compiler, and (2) in the runtime system. The compiler from Java source to bytecode can perform some large-scale optimizations, but access to the source code is a necessity for doing this. The runtime system, whether it does JIT compilation or interpreting, is limited by having little time to spare for optimizing, and thus can only do some local optimizations.

A third possibility is to optimize on the Java bytecode, as shown in Figure 1. This has several advantages:

- Access to source code is not required. In fact, it will be possible to optimize programs written in other languages, if they can be compiled to bytecode.
- Machine independency is maintained.
- Inter-procedural analysis is possible for the user, so library variations are recognised. The user will be able to optimize with respect to his own libraries and still get valid results.

In any object-oriented language, massive use is made of object fields. Since objects generally live through several method invocations, it is difficult to know much about their state locally. On the other hand, interprocedural analysis is complicated by the existence of virtual methods and interfaces, which are very common in Java. Good methods for optimizing field accesses interprocedurally are essential when working on object-oriented programs.

To address this problem, we have implemented a simple side-effect analysis. It does inter-procedural analysis where possible, determining if and how side-effects are made. Using this information, many more optimizations of field accesses and method invocation become possible.

In this paper, we describe a working implementation of Cream, an optimizer for Java bytecode, which performs dead-code elimination and loop-invariant removal using standard optimization techniques from [1], as well as the side-effect analysis. It handles native code, exceptions and monitors, and optimizes large class collections successfully.

In Section 2 we describe how we analyze Java bytecode. Section 3 describes the side-effect analysis and its effectiveness in our implementation, and Section 4 shows how it is used in Cream, and the resulting optimizations. Section 5 briefly describes related work, and Section 6 gives conclusions and future work.

2 Analyzing Java bytecode

Java bytecode [7] is a simple, stack-based language with about 200 instructions, about 50 of which are special cases of other instructions. It is strongly typed, and contains information about the class structure. Each method has a fixed maximum stack depth, and the stack depth can easily be calculated at any point in the program. This makes it possible to regard stack variables as short-lived local variables and use well-known optimization techniques on them.

As illustrated in Figure 1, Cream consists of 9 phases. The first two phases are interprocedural, while the remaining phases are performed for one method at a time. A brief description of each phase is given below.

Class graph construction: The class graph is made in order to make it possible to analyze virtual and in-

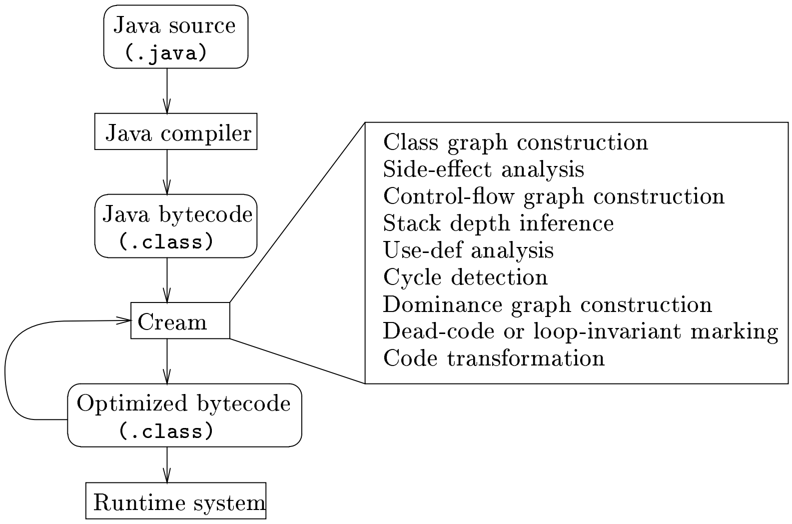

Figure 1: Lifetime of a Java program

terface methods. It contains all the classes that may be used by the program, as well as their superclasses. Those classes that may have instances allocated during program execution are marked as such.

Side-effect analysis: Before starting the normal analysis, side-effect analysis for virtual methods (see Section 3.3) is done, starting at a given `main()` method. Methods that are not reached by this will be analyzed when needed.

Control-flow graph construction: The instructions of each method are divided into blocks at places where the control flow may change, such as after `if` or `goto`, or before instructions that are the target of such. The possible flow of control is then represented as edges between these blocks. If an exception handler is present, instructions that may throw an exception[1] also end a block, and cause an edge to be added from the block to the exception handler. Exception and monitors are also considered when doing the actual optimization.

During this stage we also do the one abstraction that has proven necessary: The control flow is separated from the order of the instructions, making it much easier to move instructions around. Once this abstraction was in use, the problems encountered changed to be of a nature that any control-flow graph would give.

Stack depth inference: In order to reason about stack variables, we have to know how high the stack is at any given point. This is found by a simple traversal of the control-flow graph. After this, stack variables are identified by their offset from the bottom of the stack, not by their position relative to the stack pointer.

[1] It is necessary for good optimizations to be able to optimize on field accesses. For this reason, we use the option where null pointer and class cast exceptions do not cause branches. Programming to catch such exceptions isn't very nice, anyway.

Use-def analysis: With the control-flow graph and stack depths ready, we build use-def chains for local variables and stack variables. This uses a standard analysis with kill- and gen-sets. The only difference is that stack variables are always considered killed when they are used.

Cycle detection and dominance graph construction: Before doing loop invariant elimination, a cycle detector as in [1] is run, and a dominance graph is built.

Dead code marking: The dead code marker considers all impure instructions, return instructions, and instructions that may alter the control flow, to be live. Those that are not live or used to construct anything live will be eliminated in the transformation step.

Loop-invariant detection: Loop-invariant detection detects, fairly conservatively, instructions that are invariant during a loop. This may include field accesses and invocations of method without side effects, if the loop itself is found to be side-effect free.

Code transformations: Finally, the possible transformations are performed. Care must be taken to keep the stack consistent — if the result of a method invocation is no longer used, because the use turned out to be dead code, the result must still be popped off the stack. Exceptions and monitors present special problems for moving code outside loops. Code that may reasonably throw exceptions should not be moved outside an area that has a handler for the exceptions they may throw. Similarly, in monitored areas, code that might involve the monitored objects should not be moved away. We take a conservative approach by simply disallowing movement of any code that may throw exceptions outside areas with an exception handler, and object field accesses or updates may not be moved outside a monitored area.

3 Side-effect analysis

The side-effect analysis is an inter-procedural analysis to determine when computations may alter or inspect of object fields or array elements. We have called it 'purity analysis' after the usage in functional languages, where 'purely' functional languages are those without the possibility of side-effects.

An instruction or collection of instructions is said to have a purity for all object fields and the five basic types of array elements[2]. Purity can be inferred for methods, loops and instructions. The inferred purity of an instruction for an object field or array element can be one of four:

- pure: An instruction is pure for a field or element if it neither alters nor inspects the field or elements. These include allocations and array length.
- read-only: An instruction is read-only for a field or element if it reads that field or element, but does not write to it.
- write-only: An instruction is write-only for a field or element if it changes the field or element.
- read/write: These instructions both alter and inspect the object field or array element. The only single instruction that can do this is a method invocation.

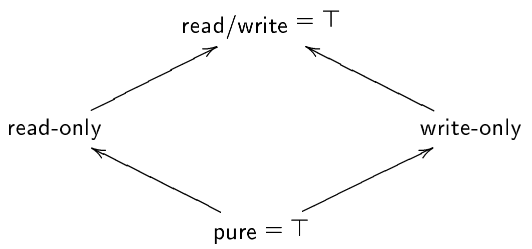

Figure 2: Partial ordering of purity information

These three types of purity form a partial ordering as shown in Figure 2. With this, finding the purity for a field or an element of blocks, loops or methods becomes a matter of finding the least upper bound of the corresponding purities of the instructions involved. Figure 3 shows examples of purities of methods. This ordering can be extended to allow for more complex information, as demonstrated in section 3.2.

3.1 Simple purity

The purity of most instructions can immediately be found. The only directly write-only instructions are array- and field-store, while array- and field-loads are read-only.

The local purity of a method or a loop is found for each field or array element by taking the least upper bound of the purities of the instructions involved.

[2] At bytecode level, determining the exact type of the elements of an array involves a great deal of analysis.

```
class Purities {
  int a;
  int b[];
  int pure1(int x) { return(x+2); }
  int pure2(int x) { return(x+b.length); }
  int readonly1(int x) { return(x+a); }
  int readonly2(int x) { return(x+b[0]); }
  void writeonly1(int x) { a = x; }
  void writeonly2(int x) { b[x] = 0; }
}
```

Figure 3: Examples of pure, read-only, and write-only methods

To find the purity of a method invocation, it is necessary to recursively find the purity of that method. This would be easy, where it not for recursive methods. Because of them, more thought is needed. Figure 4 shows such a recursive method. Since `qux()` is write-only, `bar()` will also be write-only, but `baz()` may already have been visited and will wrongly be considered non-write-only.

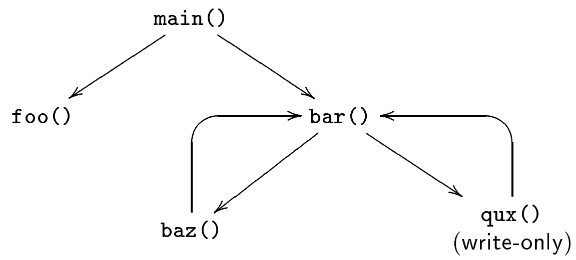

Figure 4: Call graph with recursion

The fastest way to find the real purity of the recursive methods is by finding the strongly connected components of the call graph [5] and treat them as one virtual node. Methods included in the same component will have the same purity, as they may at some time call any of the others. Finding the strongly-connected components has a well-known linear-time solution [11]. By using these virtual nodes, we turn the call graph into a directed acyclic graph, where a simple depth-first traversal can be used to find the purities.

The example in Figure 4 will then have `bar()`, `baz()` and `qux()` as one virtual node in the callgraph, called from `main()`. All the methods involved in that node will be write-only, since one of them is.

Once the strongly-connected components are found and replaced by virtual nodes, a depth-first traversal will take only $O(ek)$ time, where e is the number of edges in the reduced call graph, and k is the time needed to find the least upper bound of two purities. This holds because each edge in the graph is only visited once, and thus cannot be involved in more than one LUB. Updating the m nodes represented by virtual nodes takes no more than $O(m)$ time. Thus the whole purity analysis is done in linear time times the height of the purity ordering.

Native methods can of course not be analyzed, so they must be taken as being both read-only and write-only. The naive approach also considers invocations of interfaces and virtual, non-final methods to be both, too, as it is undetermined what actual code they execute. A more reasonable analysis of virtual methods and interface methods is given in Section 3.3.

3.2 Advanced purity

While the above purity gives some useful information (see Section 3.4), it is a very rough approximation. If a field has been altered in any object or array, it is considered write-only for all fields and arrays. In a loop like in Figure 5, the field accesses to `foo` and `bar` are invariant, but because the array is being changed, the whole loop is considered write-only for `foo` and `bar`. Thus the optimizer can't assume that the field accesses are invariant and could be moved outside the loop.

```
for (i = 0; i < 10; i++) {
  this.foo.bar[i] = 0;
}
```

Figure 5: A loop-invariant field access not optimizable with simple purity analysis

In order to find the interesting loop invariants in such a loop, more information is needed about which objects have which fields manipulated.

Our first approximation was to find the least object of which all objects with affected fields are all subclasses and store that. Arrays are kept separately, with purity information being kept for each kind of array known in the bytecode (Integer, Long, Float, Double and Object). This does not take much space, but each LUB operation can take time proportional to the height of the inheritance tree. This we call subclassing purity.

A better approximation is to store the classes that are affected. This solves the problem of subclassing purity that two simple reads in different classes may cause a total loss of information, but takes up a lot more space. We call this class-based purity.

The optimal representation is of course to store exactly which fields have been altered or inspected. This will take even more space than the classbased purity, but gives as good information as can be hoped for without using other kinds of analysis to improve the code first. This is what we call field-based purity.

3.3 Virtual methods

The analyses discussed above have the unfortunate drawbacks that they cannot analyze virtual methods or interface methods, since they don't know what actual method is being called. These methods are the most common in Java, since a method is virtual unless explicitly declared otherwise.

There are two ways to solve this problem: One is to reduce the number of virtual methods by inferring the actual type of the objects. Though this would not get rid of all virtual calls, it should still give a vast improvement. Type inference of this kind is well studied ([9], [6]), but outside the current scope of this project. Instead we have chosen the other, somewhat simpler solution, to make our analysis over all possible methods called. This costs precision, as we may include some methods in the analysis though they couldn't possibly be called, but in many cases only one method will be examined this way.

To find the possible methods called, one has to make certain assumptions. The most important is that the code being optimized must not change after optimization or be used in other programs than the one currently being optimized. If that assumption is violated, we can not know what possible new subclasses can be introduced and possibly should be included in the analysis. Another assumption is that all the classes used in the program must be available, for similar reasons. These make up the 'closed-world' assumption, that the classes we optimize are part of one and only one application[3].

Additionally, we must know the entry point to the program. This is needed because we must be able to deduce which actual classes can possibly be allocated during program execution. We only need concern ourselves with these, instead of trying to find all existing classes. This entry point will typically be the `main` method in one class.

When analyzing the entry point, which should be done before analyzing any other method, the optimizer simultaneously traverses the full call graph and creates a class graph. The class graph will consist of all classes that are allocated during the call, as well as their superclasses. The classes that are known to be allocated are marked as such.

Whenever a virtual method call is analyzed during the purity analysis, the class graph will be consulted to find where that method is defined. This is in the defining class, and possibly also in any of the subclasses. These methods are individually analyzed as if they were non-virtual methods, and the result is the least upper bound of the resulting purities. In Figure 6, the purity of a call to `A.g()` will be the least upper bound of the individual purities of `A.g()`, `E.g()`, `F.g()`, and `G.g()`.

If an interface method is called, we have to find all allocated classes that implement that interface and subinterfaces, and analyze the corresponding method therein. These can be virtual methods, too, in which case they must be analyzed as such. Thus, in Figure 6, a call to `Interface1.h()` would combine purities of the `h()` methods in the following classes: `B` and `H`, because they implement `Interface1`, `H`, `I`, and `G`, because these are subclasses of classes implementing `Interface1`, and `J`, because it implements a subinterface if `Interface1`.

3.4 Experimental purity results

To see the effects of the purity analysis on realistic programs, we have selected four programs and run the various variants of the purity analysis on them. As can be seen in Figure 7, these programs are of a non-trivial size. The programs used are:

Kem: A bytecode disassembler.

BYTEmarks: BYTE's benchmark program adapted for Java. To make the measurements comparable,

[3] A class being used in several programs could of course be optimized separately for each program.

	Classfiles size (in kilobytes)	Classes	Methods	Virtual methods
Kem	61	43	199	187
BYTEmarks	58	19	149	18
Javac	485	167	1142	883
Cream	439	380	2033	1892

Figure 7: Size of test programs

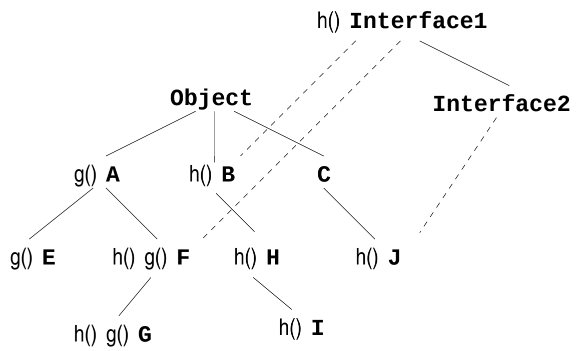

Figure 6: A class graph with virtual and interface methods

they are modified to do a fixed number of iterations instead of adjusting it by measuring their speed. Since this is directly translated from the C version, the style is more imperative than object-oriented.

Javac: Javasoft's Java compiler, taken from JDK 1.02.

Cream: the optimizer itself, excluding a few support classes. This contains a lot of very small classes, including one for each bytecode instruction.

First, we consider the non-virtual purity analyses from Section 3.1 and 3.2. To compare the quality of the variants, we define a common measurement, the average number of read-only or write-only fields per method. This is found for a method by counting through all fields in all known classes, and collecting their purity. We then take the average of these numbers over all the methods that we want to optimize. The results are found in Figure 8.

It may seem odd that the number of write-only or read-only fields can rise when going to a better analysis. This is because some methods in the better analysis are found to only read or write that field, but not both. The actual average number of fields that are being written to is the sum of the fields that are only written to and those that are both written to and read from. As can be seen, this sum never rises.

There is a big difference in the purity of these programs. Javac, for some reason, gives poor results, while Cream, having a large subclass structure, has many pure and read-only methods. The analysis of virtual methods does not give any substantial improvement here, but it is worth noting that the improvement is in the amount of read-only methods. This can be the effect of small encapsulation methods that only read a single field.

It turned out that `Object.<init>`, the initialization method for `Object`, does not contain any code. This means that any call to it will be pure. Since methods that do no explicitly make any `<init>` procedure automatically gets one that calls `super.<init>`, there are many pure `<init>` methods. These account for up to one-fourth of the pure fields in some cases.

Native methods are also an important consideration. Since we do no know anything about what they do, they should be considered to be read/write for all fields. Unfortunately, this makes quite a lot of methods read/write, for instance any that may eventually call `System.out.println`. If we could annotate the native code with information about its purity, we could avoid this totally. As an intermeditate solution, we have used the option of letting native methods be read/write on all fields in their own object, and pure on everything else, which gave a tremendous improvement in the purity analysis

Figure 9 shows the purity when using virtual purity. We see that the virtual purity analysis gives better results than the non-virtual case. This was not the case when native methods were considered to be read/write for all fields, where the amount of information levelled out when reaching subclassing purity. This effect is still visible in the non-virtual purity.

4 Using purity

When the purity information has been found, what is it good for? It tells us how safe it is to transform the code. Most common optimizations can benefit from it.

Dead code removal: If the result of a fully pure or read-only instruction is never used, it is safe to remove that instruction. Only in the case of an write-only instruction must we keep it, as other computations may depend on the changes made there. In the example in Figure 10, the results of line 9 is never used, and since it is not write-onlyfor anything, it can be safely removed. We have implemented this, and the results are listed in Section 4.1.

It may even be possible to determine when an object only can be referenced locally. If a method creates an object, but none of its instructions are write-only for fields that could contain it, and it is not returned, it cannot possibly be known after the method has executed. It may then become possible to remove the object entirely.

Loop invariant removal: A loop-invariant pure instruction can be moved outside the loop safely. An instruction which is read-only for some field can be moved

		Simple	Subclassing	Class-based	Field-based
Kem	pure fields:	21.6 %	31.5 %	31.6 %	31.7 %
	read-only fields:	3.5 %	0.5 %	0.3 %	0.3 %
	write-only fields:	5.5 %	0.2 %	0.2 %	0.2 %
	read/write fields:	69.3 %	67.8 %	67.9 %	67.9 %
BYTEmarks	pure fields:	8.1 %	49.3 %	56.0 %	57.5 %
	read-only fields:	3.4 %	9.6 %	4.6 %	3.4 %
	write-only fields:	2.7 %	2.5 %	1.8 %	1.7 %
	read/write fields:	85.9 %	38.5 %	37.6 %	37.4 %
Javac	pure fields:	6.0 %	33.8 %	38.1 %	38.2 %
	read-only fields:	9.4 %	3.3 %	0.2 %	0.1 %
	write-only fields:	5.1 %	0.5 %	0.1 %	0.1 %
	read/write fields:	79.6 %	62.4 %	61.6 %	61.6 %
Cream	pure fields:	35.5 %	75.9 %	77.7 %	78.1 %
	read-only fields:	23.4 %	1.6 %	0.5 %	0.3 %
	write-only fields:	14.9 %	0.9 %	0.3 %	0.2 %
	read/write fields:	26.3 %	21.5 %	21.4 %	21.4 %

Figure 8: Non-virtual purity analysis

		Simple	Subclassing	Class-based	Field-based
Kem	pure fields:	21.6 %	64.7 %	80.9 %	82.4 %
	read-only fields:	4.0 %	8.1 %	3.0 %	1.7 %
	write-only fields:	5.5 %	0.8 %	0.6 %	0.7 %
	read/write fields:	68.8 %	26.4 %	15.5 %	15.1 %
BYTEmarks	pure fields:	8.1 %	64.2 %	77.2 %	81.6 %
	read-only fields:	3.4 %	10.8 %	8.2 %	4.6 %
	write-only fields:	2.7 %	5.0 %	1.9 %	1.9 %
	read/write fields:	85.9 %	20.0 %	12.6 %	11.9 %
Javac	pure fields:	6.0 %	55.5 %	63.9 %	77.5 %
	read-only fields:	13.1 %	7.6 %	18.3 %	5.9 %
	write-only fields:	5.1 %	0.7 %	0.2 %	0.4 %
	read/write fields:	75.8 %	36.1 %	17.6 %	16.3 %
Cream	pure fields:	35.5 %	85.6 %	89.9 %	92.2 %
	read-only fields:	23.7 %	2.1 %	1.9 %	0.9 %
	write-only fields:	14.9 %	1.3 %	0.6 %	0.4 %
	read/write fields:	26.0 %	10.9 %	7.7 %	6.5 %

Figure 9: Virtual purity analysis

out only if the loop contains no instructions that are write-only for the same field. Line 6 in Figure 10 does not depend on i in any way, and can be moved outside the loop. We have also implemented this optimization, the results of which are in Section 4.1. This optimization benefits the most from the subclass purity analysis, as it is common practice to make several field-accesses in one assignment, as shown in Figure 5.

Constant propagation: If a field in an object or an array element is known to be constant, a method that is not write-only for that field or element will never change its value, and a method that is pure for it will not even need the value. In Figure 10, the value of `baz.qux.quod` is not changed since the assignment in line 4, and the assignment in line 11 can be reduced to an increment.

Common subexpression elimination: If common subexpressions involve only pure instructions, the optimizer only need to consider any local variables involved to ascertain whether it is safe to evaluate them only once. If there are read-only instructions, there may not be any instructions that are write-only for the relevant fields between the subexpressions. In Figure 10, `bar[i].qux` is evaluated in lines 7 and 8, though it does not change within one loop iteration. Thus, that calculation need be done only once.

```
1    public int foo(SomeClass bar[]) {
2      int i,a,b,c;
3      SomeClass baz = new SomeClass();
4      baz.qux.quod = 1;
5      for (i = 0; i < 10; i++) {
6        a = bar[0].qux.quod;
7        b = bar[i].qux.quis;
8        c = bar[i].qux.quam;
9        b += a + bar[i].qux.quam;
10     }
11     b += baz.qux.quod;
12     return(b);
13   }
```

Figure 10: Examples of possible optimizations

```
public int foo(SomeClass bar[]) {
  int i,a,b;
  a = bar[0].qux.quod;
  for (i = 0; i < 10; i++) {
    OtherClass tmp = bar[i].qux;
    b = tmp.quis;
    b += tmp.quam + a;
  }
  return(b++);
}
```

Figure 11: The same example after all optimizations.

4.1 Experimental results

Figure 13 shows how many instructions were removed by the deadcode analysis, while Figure 14 shows the number of instructions moved as invariant. It is obvious that the purity analysis is essential for the optimizer, and that the more precise analysis does in fact allow for more optimizations.

The dead-code optimizations do not benefit at all from the better kinds of analysis. This is because it does in fact no use more information than whether an instruction is write-only or not. For loop-invariant removal it makes a lot more of a difference, because a field access can be moved outside a loop only if there are no instructions in the loop that write exactly that field.

In order to get maximum benefit from loop invariant removal, it is necessary to know is the loop body is executed at least once. As most loops appear to be while loops at the bytecode level, it was not possible to move instructions whose values may be used outside the loop.

In order to overcome this problem, we added a partial loop unroller. It finds all exits from the loop above the instruction in question and puts a copy of those before the loop. Instructions that are moved outside the loop are then placed after the copied part, but before the entry to the loop. This is shown in Figure 12, where the assignment to `k` can be moved after unrolling. The assignment to `l` can never be moved, as we can not be sure that it will be executed at all. Unrolling code turns out to give a factor five to ten increase in the number of instructions that can be moved.

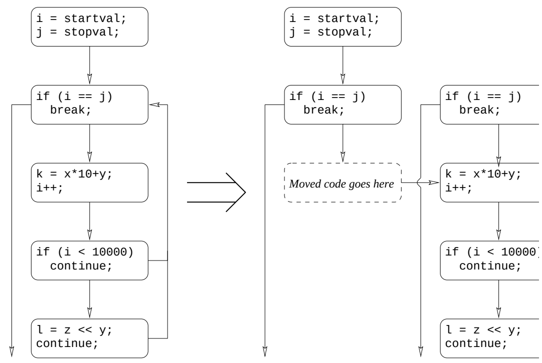

Figure 12: Partial unrolling of a loop

On the other hand, this could decrease performance if large parts of code is being duplicated in order to perform relatively small optimizations. To avoid this, it is necessary to add some heuristics to judge how much of a loop it is worthwhile to unroll. Gripping hand is, any invariant code in the unrolled part has automatically been moved outside the loop when unrolling, so those are even cheaper to optimize. Exactly how to get the best results out of this is a separate subject; in our implementation we have just unrolled all necessary code.

In Figure 15 is shown the improvement in execution speed for the four programs described in Section 3.4. The figure shows raw execution times in seconds and relative exection times for unoptimized code, optimization without purity analysis, optimization with simple purity analysis, and optimization with virtual and subclassing purity analysis. The programs have been optimized

		Purity type				
	Total instructions	None	Non-virtual simple	Virtual subclassing	Virtual class-based	Virtual field-based
Kem:	3476	0	165	165	165	165
BYTEMarks:	15037	21	43	43	43	43
Javac:	40281	20	116	122	122	122
Cream:	32487	274	376	376	376	376

Figure 13: Number of instructions removed as dead code

		Purity type				
	Total instructions	None	Non-virtual simple	Virtual subclassing	Virtual class-based	Virtual field-based
Kem:	3471	8	8	94	179	209
BYTEMarks:	15015	10	10	965	1150	1307
Javac:	40209	91	91	200	485	497
Cream:	33709	159	159	388	653	873

Figure 14: Number of instructions moved as invariant

with both dead-code elimination and loop-invariant removal, and using full unrolling, treating native methods as being read/write only for their own object. The test programs were run on a Pentium Pro with Linux 2.0.28 using JDK1.02.

The greatest speed improvements are in the BYTEMarks, caused by the many tight, imperative-style loops in that program. The other programs, though they have been quite heavily optimized, too, do not show any interesting increase in speed. This seems to indicate that other kinds of optimizations are necessary for object-oriented programs. The *decreased* speed seen for Cream is caused by a poor reconstruction of the abstracted instruction order.

5 Related work

Little optimization has been done on Java so far, but with the popularity of the language, we will surely see more work in this area. Suns compiler has some inlining, though in versions before 1.1 it could break privacy of classes [3]. The only optimization they suggest is for the runtime system to calculate certain constants ahead of time.

Cierniak and Li [4] have made a flexible compiler/optimizer using an intermediate representation. It is also able to optimize bytecode to bytecode, but their article does not mention any interprocedural analysis or actual results. This is the only optimizer we know of that are akin to Cream.

There are several bytecode-to-C compilers [8] [10], which then use the C compiler for standard optimizations. While this gives many of the well-known optimizations, it does not take the special structure of an object-oriented language into account.

Side-effect analysis was first thoroughly covered by Banning in [2], and the methods improved upon by Kennedy and Cooper in [5]. These papers considered optimizing imperative languages, and thus did not consider classes and virtual functions.

6 Conclusions and future work

We have defined and implemented an optimizer using inter-procedural side-effect analysis. The analysis has been shown to significantly improve the amount of optimizations possible, even with a rather conservative optimizer. Without the purity analysis, few optimizations could be done, but with it, we see several times as many optimizations, and, in one case, a performance improvement of up to 25 % with the most advanced analysis.

We have also seen that Java bytecode, because of its simplicity and type information, is an easy and worthwhile place to perform optimizations. The only abstraction needed is to separate control flow and instruction ordering.

It is noteworthy how optimization techniques that work well on imperative programs have little effect on programs using an object-oriented design. Other methods should be developed for these, that take into account the amount of field references and small, virtual methods.

Implementing common subexpression elimination would likely be worthwhile, as there are often long chains of fields accesses occuring more than once in a loop without being invariant.

The results obtained here should be compared to what can be done using the Harissa bytecode-to-C translator [8], both alone and together. Using an optimizing C compiler as backend gives local optimizations such as strength reduction and register allocation for free, and the large-scale optimizations suggested and partially in this paper will still be available.

7 Acknowledgements

The optimizer is based on Clark Verbrugge's Coffi program [12]. Though some rewriting was necessary, it has turned out to be a good starting point for the program. Ulrik Pagh Schultz co-implemented the first version of the optimizer, primarily the virtual purity analysis and the class graph, but also acted as a sounding board for

	No optimization	No purity		Non-virtual simple purity		Virtual subclassing purity		Virtual class-based purity		Virtual field-based purity	
Program	Sec.	Sec.	Rel.	Sec.	Rel.	Sec.	Rel.	Sec.	Rel.	Sec.	Rel.
Kem	2.83	2.80	0.99	2.74	0.97	2.78	0.98	2.77	0.98	2.78	0.98
BYTEMarks:	30.2	30.7	1.02	30.7	1.02	24.8	0.82	24.6	0.82	24.1	0.80
Javac:	17.75	17.56	0.99	17.45	0.98	17.49	0.99	17.55	0.99	17.45	0.98
Cream:	3.8	4.2	1.09	4.2	1.09	4.1	1.07	4.1	1.07	4.2	1.08

Figure 15: Improvement in execution time

ideas. Laurie Hendren has been helpful with support, advice and lots of corrections.

References

[1] Alfred V. Aho, Ravi Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1985.

[2] John Banning. An efficient way to find side effects of procedure calls and aliases of variables. *POPL*, pages 29–41, 1979.

[3] Doug Bell. Make Java fast: Optimize. *JavaWorld Magazine*, April 1996.

[4] Michael Cierniak and Wei Li. Briki: A flexible Java compiler. Technical Report TR 621, University of Rochester, Computer Science Department, May 1996.

[5] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. *PLDI*, pages 57–66, 1988.

[6] Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. In Benjamin Pierce, editor, *Proc. 4th Int'l Workshop on Foundations of Object-Oriented Languages (FOOL), Paris, France*, January 1997.

[7] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[8] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: a flexible and efficient Java environment mixing bytecode and compiled code. To appear in Proceedings of COOTS'97, June 1997.

[9] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1993.

[10] Todd Proebsting, John Hartman, Gregg Townsend, Patrick Bridges, Tim Newsham, and Scott Watterson. Toba: A Java-to-C translator. Technical report, University of Arizona, 1997.

[11] Robert Endre Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[12] Clark Verbrugge. Using Coffi. Technical report, McGill University, October 1996.