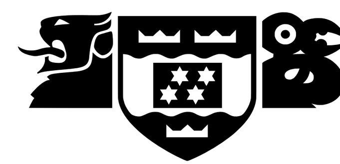
Optimising Java programs with pure functions



Andrew Walbran

supervised by Associate Professor Lindsay Groves and Doctor David Pearce, 2008

1. Motivation

- Advances in compiler technology have improved performance of Java programs.
- Better performance is still needed, especially for resource-limited devices like cellphones.
- Compilers need to produce better code with less work from the programmer.
- By moving method calls out of loops (when possible), we can make programs run faster.

If a method call in a loop always returns the same result:

```
TestCollection collection = ...
for (int i = 0; i < collection.size(); ++d) {
    System.out.println(collection.get(i));
}
```

we can make the program faster by moving the call out of the loop:

```
TestCollection collection = ...
int size = collection.size();
for (int i = 0; i < size; ++d) {
    System.out.println(collection.get(i));
}
```

While the programmer could make this transformation manually, the original form is easier to read and write. We would like the compiler to make this sort of transformation (and many others like it) automatically.

2. Background

- Loop invariant code motion is well-known optimisation technique of moving invariant expressions out of loops.
- Compilers are generally limited to moving simple expressions, not method calls.

e.g., if x and y are integer variables which are not changed in a loop:

```
for (int i = 0; i < x * y; ++i) {...}
```

```
int temp = x * y;
for (int i = 0; i < temp; ++i) {...}
```

- We want to do better, moving method calls and field accesses too.

Knowing which method calls can safely be moved is hard. Among other things, it is necessary to know that arguments to the method call will not be changed within the loop.

Aliasing makes method motion hard

- Easy to spot direct changes to an object which prevent code motion:

```
TestCollection list = new TestCollection();
for (int i = 0; i < list.size(); ++i) {
    list.add();
}
```

- More difficult to spot indirect changes via aliases:

```
TestCollection list = new TestCollection();
TestCollection alias = list;
for (int i = 0; i < list.size(); ++i) {
    alias.add();
}
```

This and more complicated situations involving references in different classes prevent the compiler from knowing, in general, whether such an optimisation would be safe.

3. Methodology

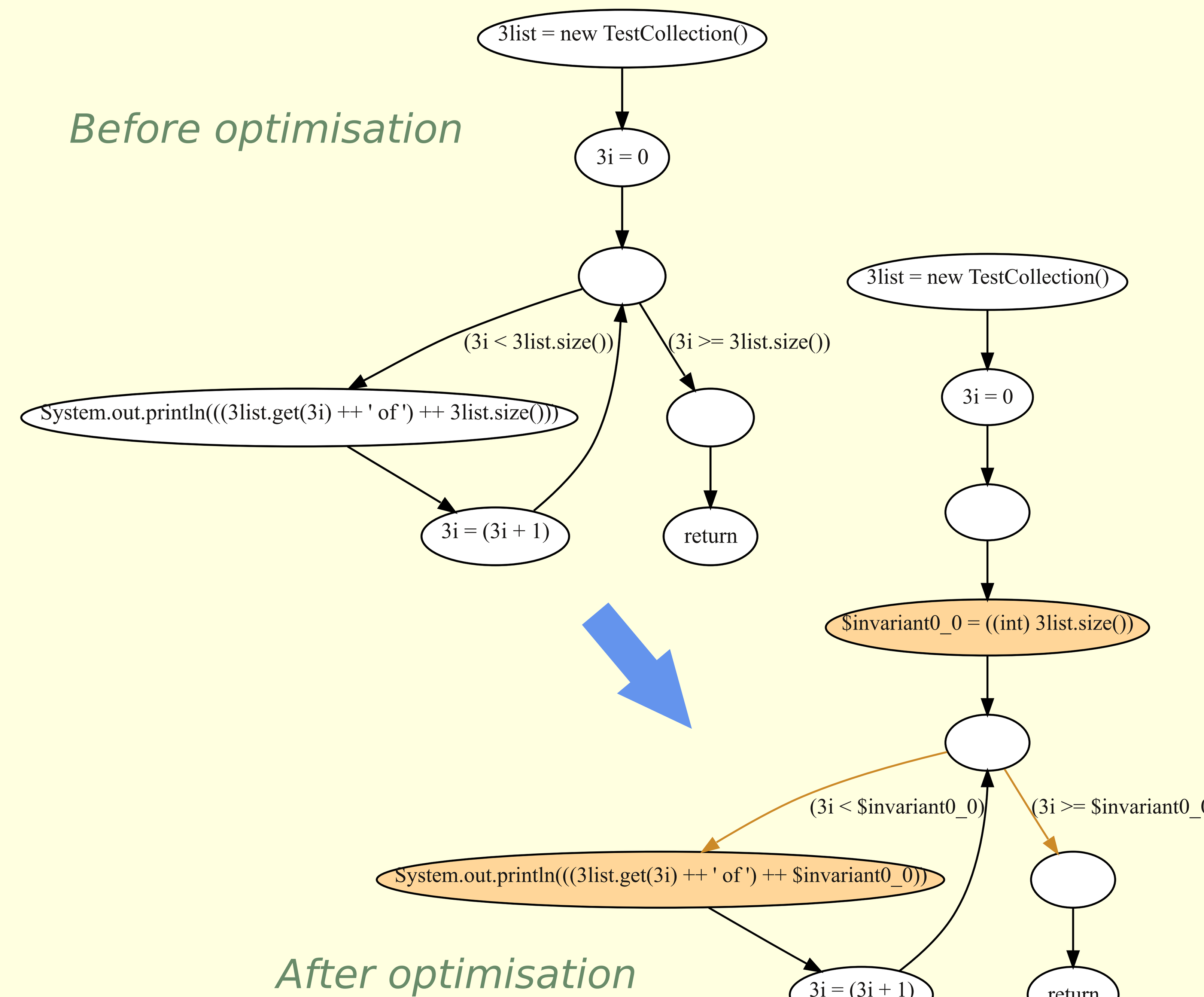
We require the programmer to express their knowledge of the program with annotations, so that the compiler has sufficient information to make a wider range of optimisations.

We introduce 5 annotations for methods, classes and local variables:

Annotation	Type	Semantics
@Pure	method	Does not alter any state, and return value depends only on parameters (including receiver).
@Const	method	Does not alter the state of its receiver object.
@Immutable	class	Objects never change once constructed.
@Encapsulated	class	Objects of the class are well-encapsulated, in that their state cannot change except by calling their methods (not by external references to internal objects, for example).
@Unique	variable	There will be no other references to the object pointed to by the local variable.

Using these annotations, we can detect cases where expressions involving method calls and field access can safely be moved out of loops. The following two flow graphs show the changes for a simple example.

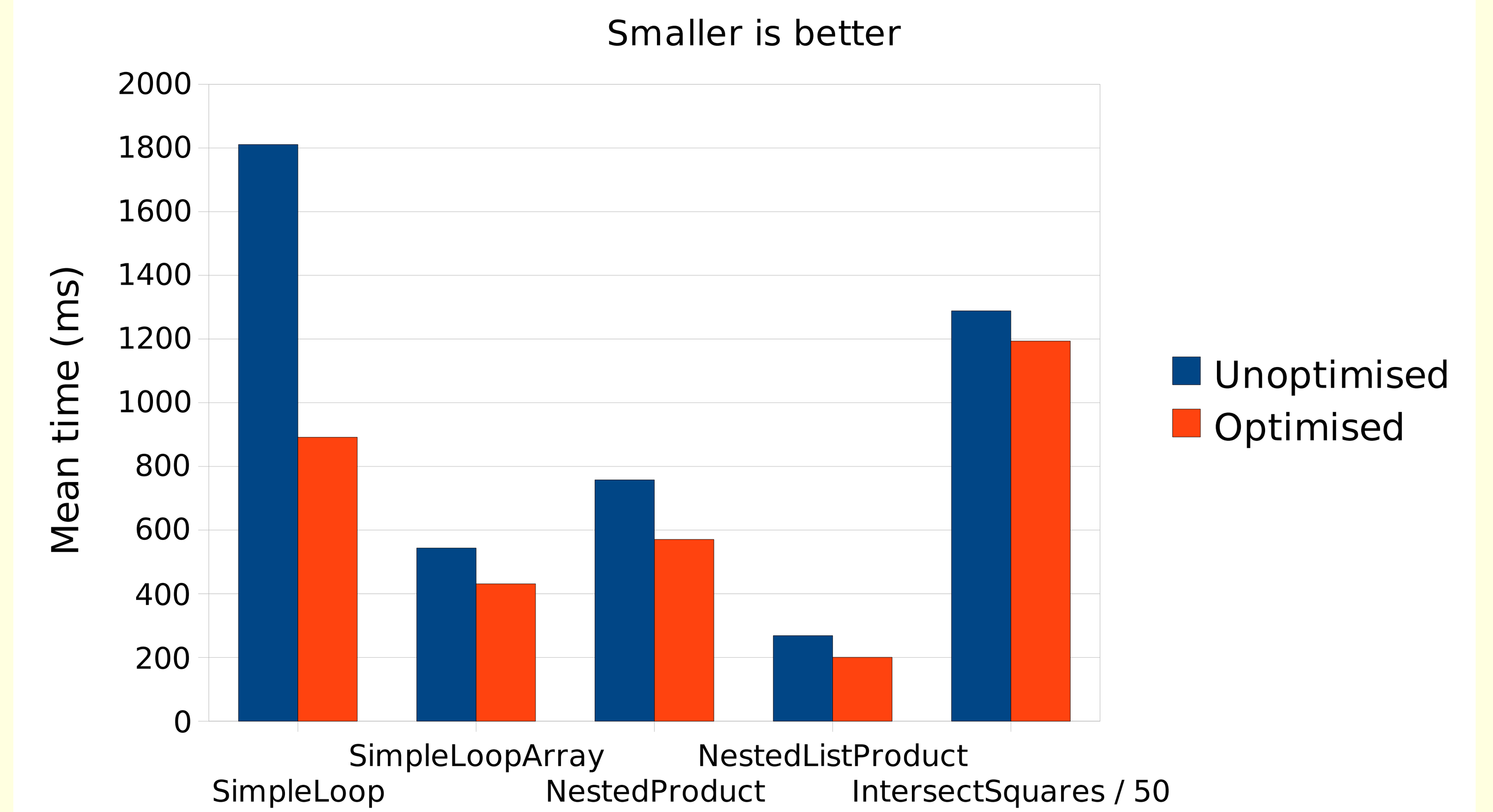
```
@Unique TestCollection list = new TestCollection();

for (int i = 0; i < list.size(); ++i) {
   System.out.println(list.get(i) + " of " + list.size());
}
```

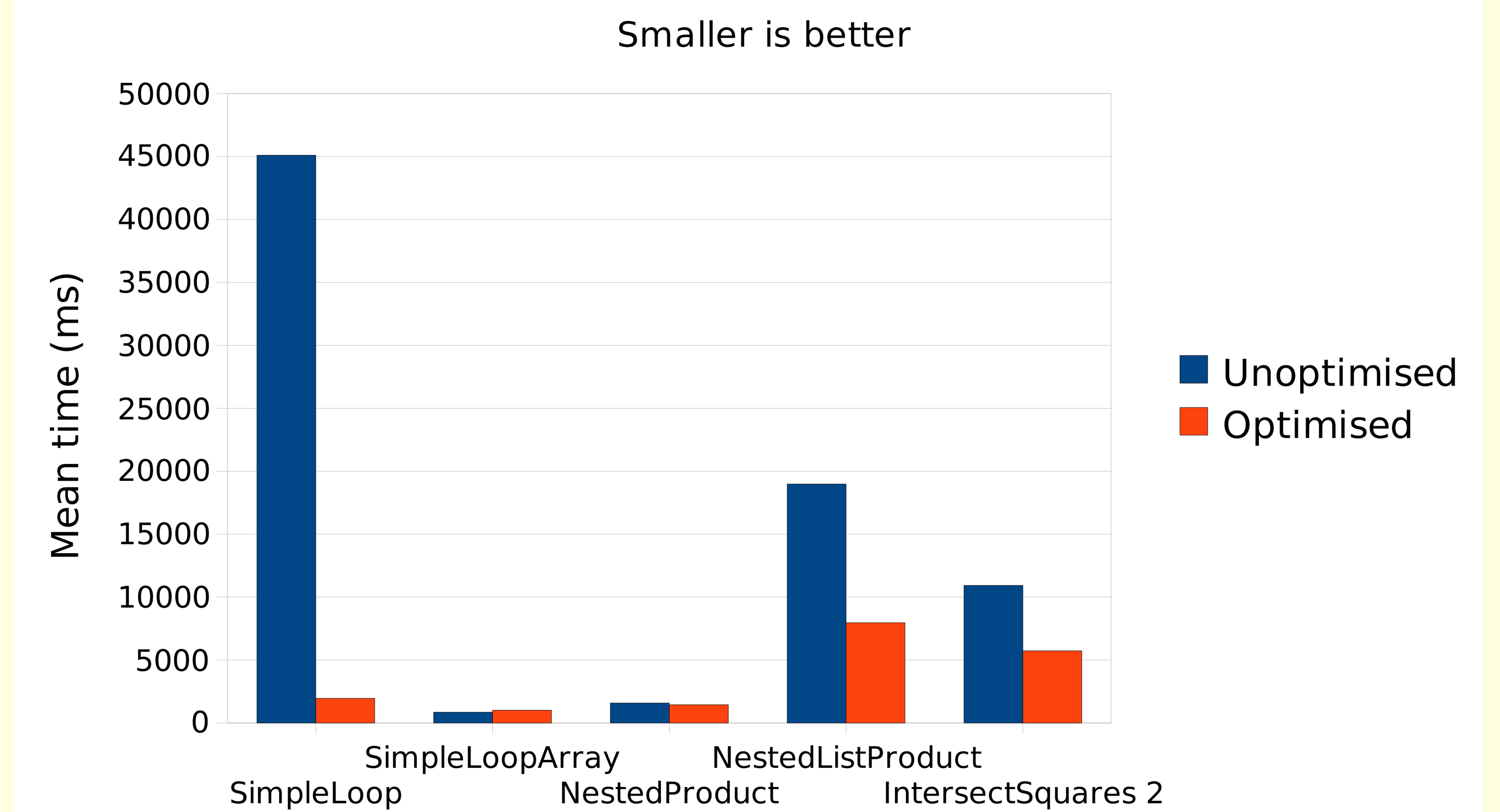


4. Evaluation / Results

Benchmarks

Some benchmark programs were constructed to demonstrate the optimisations possible. Some improvements are more noticeable with a simpler JVM.





(Kaffe tests were conducted on a different computer to HotSpot tests, so times are not directly comparable.)

Real-world programs

Testing has been conducted with a few real-world programs, but with limited success.

- SimpleLisp LISP interpreter: 30 field dereferences and 8 method calls factored out of loops, but no measurable speed improvement due to structure of code.
- GeoffTrace raytracer: No useful factorisations could be made.

5. Conclusions

We have developed a new optimisation technique for Java compilers, building on existing techniques to allow optimisation in a wider range of situations. While we have found that this technique does not apply as widely as we had originally hoped, it is still worthwhile because it can acheive significant performance gains in certain situations, especially when using a simple JVM such as might be found on mobile devices.