The Runabout*

Christian Grothoff

Department of Computer Sciences, University of Denver
christian@grothoff.org
http://grothoff.org/christian/runabout/

Abstract. This paper presents a variation of the visitor pattern which allows programmers to write visitor-like code in a concise way. The Runabout is a library extension that adds a limited form of multi-dispatch to Java. While the Runabout is not as expressive as a general multiple dispatching facility, the Runabout can be significantly faster than existing implementations of multiple dispatch for Java, such as MultiJava. Unlike MultiJava, the Runabout does not require changes to the syntax and the compiler.
This paper illustrates how to use the Runabout, details its implementation, and provides benchmarks comparing its performance with other approaches. Furthermore, the effect of an automatic static program transformation tool that translates bytecode using the Runabout to equivalent bytecode is evaluated. The tool uses double dispatch and runtime type checks to achieve the same semantics that the Runabout has. The performance comparisons on large benchmarks that make extensive use multiple dispatch show that using the Runabout does not result in a significant loss of performance for realistic applications and that, depending on the application and platform, small performance gains are also possible.

1 Introduction

A fundamental problem in programming language design is to make software extensible while avoiding changes to existing code and still retaining static type safety [30]. For example, a programmer may want to add functionality that operates on a number of existing classes, or he may want to introduce a new class to existing code. For such purposes, one of the strengths of object-oriented programming is that it is easy to introduce new classes. Adding functionality to existing classes is more difficult, particularly because this typically requires access to the source code. It may also be undesirable to add the functionality to all subclasses.

> **Extensibility Problem:** Devise a mechanism for adding functionality and classes to existing code while avoiding recompilation and still retaining efficiency and static type safety.

* This paper is an extended and revised version of the paper "Walkabout revisited: the Runabout" in ECOOP 2003 – Object-Oriented Programming, pages 103-125, Darmstadt, Germany, 2003.

One traditional solution to this problem is to use the visitor pattern [20]. The visitor pattern allows the addition of functionality in the form of `visit` methods that are invoked from an `accept` method which is defined in each visitee class. The `accept` is only specific with respect to the type of an abstract visitor. Visitors do not completely solve the extensibility problem. If the set of visitee classes changes, the type of the abstract visitor changes. Using visitors, it becomes more difficult to change the set of visitees since all visitors must be adjusted to provide a `visit` method matching the visitee types. Another solution to the extensibility problem is to use multi-methods which allow both new functionality and new classes to be added in a flexible and concise manner. The Runabout is a step toward achieving many of the benefits of multi-methods without requiring a new language.

This paper addresses the extensibility problem for Java, giving a solution that does support changing sets of visitee types, and provides both acceptable performance (only 2-10 times slower than visitors) and the minimum amount of programming effort. The solution is based on an approach proposed by Palsberg and Jay [39] called Walkabout. Their approach takes advantage of Java's reflection mechanism to implement double-dispatch (Figure 1). The Runabout presented in this paper is an extension of the Java libraries that adds two-argument dispatch to Java. The Runabout is itself implemented in Java (without any native methods). The code for the Runabout is about 1,000 lines of code, which is freely available on our web-page. Like the Walkabout [39], the Runabout uses reflection to *find* visit methods. But instead of invoking the visit methods with reflection, the Runabout uses dynamic code generation to create ordinary (verifying) Java bytecode that will invoke the appropriate visit method (Figure 2). The dynamically generated bytecode is type-safe and can be analyzed and optimized by the just-in-time compiler just like the rest of the application.

Generating bytecode for multi-dispatching is also what the MultiJava compiler [9] does. MultiJava compiles Java with multi-methods to ordinary Java bytecode. Unlike MultiJava, the Runabout generates the invocation code when the application is executed, not at compile time. Thus the Runabout does not require changes to the compiler or the virtual machine. Contrary to previous assumptions [39], the approach using reflection to determine visit targets does not automatically imply an extraordinary runtime overhead.

The Runabout Optimizer, a preprocessor for Java bytecode that converts programs which use the Runabout to the equivalent visitor code, was used to evaluate the performance impact of the Runabout for real-world applications. The rather astonishing result is that the Runabout can sometimes be faster than the equivalent visitor code. The worst-case performance loss for the applications is well within the estimates from the micro-benchmarks; these show that for 100 million visit invocations on 2,000 visitee classes, the Runabout is slower by less than a factor of two compared to visitors (217s vs. 137s).

The remainder of the paper is structured as follows. First, an example for programming with runabouts is given and the semantics of the Runabout are described in detail. In section 3 the implementation of the Runabout is pre-

sented. The bytecode transformation which translates Runabout client code to visitors is presented in section 4. Performance evaluations are detailed in section 5. Section 6 discusses some related work.

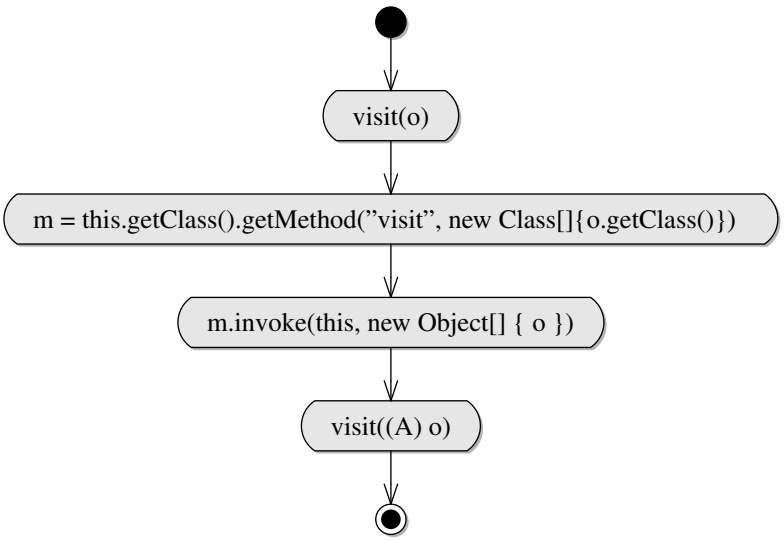

Fig. 1. Control flow in the Walkabout. The client calls the `visit` method of the Walkabout which then uses reflection to find the method that best matches the receiver. That method is then invoked using reflection. The Walkabout does not use caching or dynamic code generation.

2 Using the Runabout

Writing runabouts is similar to writing visitors or using multi methods. In order to demonstrate how to write code with runabouts, an an example is first implemented using a variety of approaches to achieve the same functionality. This example is illustrated using dedicated methods, visitors, MultiJava, and finally the Runabout. Next, the semantics of the `visitAppropriate` method of the Runabout are described. Then the specific benefits and drawbacks of each of the implementations in terms of expressiveness and restrictions imposed on the programmer are discussed.

2.1 A simple example

The example uses a set of visitee classes $\texttt{A}_i$ that implement the common interface `A`. Given an array `a` of instances of type `A`, the goal is to compute $\sum_{a \in A} I(a)$ where $I(a) = i$ if a is of type $\texttt{A}_i$.

Dedicated methods Dedicated methods can be used to solve the problem efficiently. The problem with dedicated methods is that for every operation that is to be performed on the visitee classes, a method must be added to each of the visitee classes. This spreads the code used by a particular operation over many classes and often makes it hard to maintain. Fig. 3 shows the solution using a dedicated method.

Cascading conditionals Another possibility would be to use a sequence of `instanceof` tests, which is certainly impractical for larger numbers of visitee types and requires modification each time a visitee is added (Fig. 4).

Visitors Fig. 5 details the code for expressing a solution with visitors. The example uses overloading for the `visit` methods. Overloading is not needed for visitors and is used here to emphasize the similarities with MultiJava and the Runabout. For simplification, the example uses only one visitor and there is therefore no need for a visitor interface for the `accept` methods to dispatch upon.

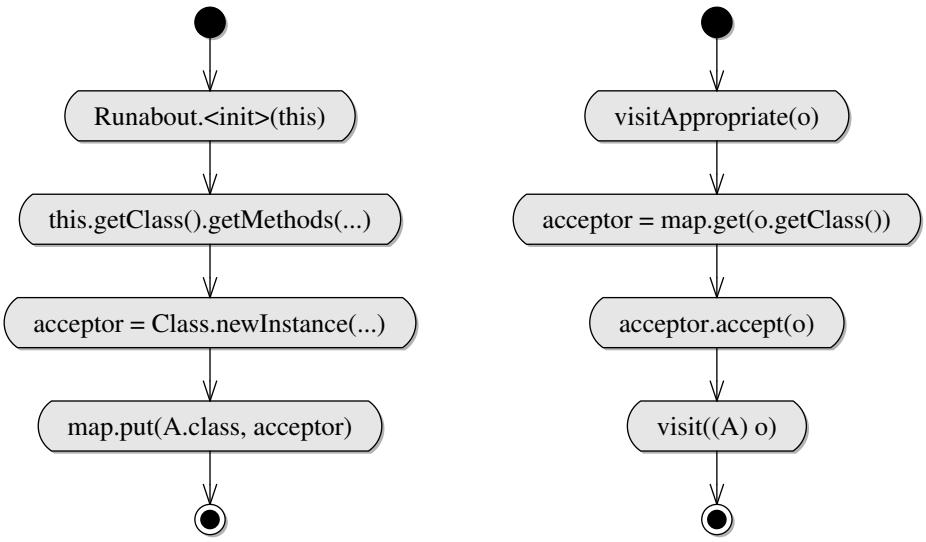

Fig. 2. Overview of the control flow in the Runabout. The constructor of the Runabout (`<init>`) uses reflection to find all `visit` methods. For each such method it creates an entry in a *lookup map* using dynamic code generation. The resulting lookup map contains for each class an acceptor instance with a synthetic accept method. Later, the client calls `visitAppropriate` which obtains the acceptor from the map. The acceptor is then used to invoke the appropriate `visit` method. Compared to the Walkabout, the Runabout shifts most work into the constructor and replaces the reflective invocation with synthetic code.

```
interface A {
  int dedicated();
}
class A0 implements A {
  int dedicated() { return 0; }
}
class A1 implements A {
  int dedicated() { return 1; }
}
class A2 implements A {
  int dedicated() { return 2; }
}
long run(A[] a) {
  long sum = 0;
  for (int j=0;j<a.length;j++)
    sum += a[j].dedicated();
  return sum;
}
```

Fig. 3. The visitee classes with a dedicated method (`dedicated`).

```
interface A {}
class A0 implements A {}
class A1 implements A {}
class A2 implements A {}
long run(A[] a) {
  long sum = 0;
  for (int j=0;j<a.length;j++) {
    A aj = a[j];
    if (aj instanceof A2)
      sumInstanceof += 2;
    else if (aj instanceof A1)
      sumInstanceof += 1;
    else if (aj instanceof A0)
      sumInstanceof += 0;
    else
      throw new Error("Illegal call");
  }
  return sum;
}
```

Fig. 4. No changes to the visitees are required with cascading conditionals.

In practice, the code would consist of multiple visitors for multiple computations that would be performed over the visitee objects.

Multi-methods An implementation using MultiJava (Fig. 6) does not require the accept methods. Instead, the compiler can see that multi-dispatch is declared (@) and generates code to invoke the appropriate visit method.

Runabouts The Runabout code (Fig. 7) lies somewhere between visitors and MultiJava. The visit methods do not require any additional syntax; all that is required is that the class extends `Runabout` and that `visitAppropriate` (a method provided by the parent class) is invoked instead of `visit`. As with MultiJava, no `accept` method is required in the visitees. A UML class diagram for the Runabout is given in Fig. 8.

2.2 Semantics

In order to create a Runabout, the client code must create a public subclass of `Runabout`. The `Runabout` class provides the method `visitAppropriate` which can be used for two-argument dispatch. The two arguments of the two-argument dispatch are the receiver of `visitAppropriate` and the first and only argument of `visitAppropriate`. The callee of the dispatch is determined by the `lookup` method.

```
interface A {
  void accept(Visitor v);
}
class A0 implements A {
  void accept(Visitor v) { v.visit(this);  }
}
class A1 implements A {
  void accept(Visitor v) { v.visit(this);  }
}
class A2 implements A {
  void accept(Visitor v) { v.visit(this);  }
}
class Visitor {
  long sum = 0;
  public void visit(A0 a) { sum += 0; }
  public void visit(A1 a) { sum += 1; }
  public void visit(A2 a) { sum += 2; }
}
long run(A[] a) {
  Visitor v = new Visitor();
  for (int j=0;j<a.length;j++)
     a[j].accept(v);
  return v.sum;
}
```

Fig. 5. Visitors require `accept` methods in the visitees.

```
interface A {}
class A0 implements A {}
class A1 implements A {}
class A2 implements A {}
class MultiJavaSum {
  long sum = 0;
  public void visit(A a)    { throw new Error(); }
  public void visit(A@A0 a) { sum += 0; }
  public void visit(A@A1 a) { sum += 1; }
  public void visit(A@A2 a) { sum += 2; }
}
long run(A[] a) {
  MultiJavaSum v = new MultiJavaSum();
  for (int j=0;j<a.length;j++)
    v.visit(a[j]);
  return v.sum;
}
```

Fig. 6. MultiJava indicates multi-dispatch using minimal changes to the syntax.

```
public class A0 {}
public class A1 {}
public class A2 {}
public class RunaboutSum extends Runabout {
  long sum = 0;
  public void visit(A0 a) { sum += 0; }
  public void visit(A1 a) { sum += 1; }
  public void visit(A2 a) { sum += 2; }
}
long run(Object[] a) {
  RunaboutSum v = new RunaboutSum();
  for (int j=0;j<a.length;j++)
    v.visitAppropriate(a[j]);
  return v.sum;
}
```

Fig. 7. Runabouts extend the `Runabout` class to inherit `visitAppropriate`.

visitAppropriate The callee in the dispatch performed by `visitAppropriate` is either `visitDefault` or exactly one of the `visit` methods defined in or inherited by the class of the receiver. The concrete selection of the `visit` method is performed by the `lookup` function, which, given a `Class`, returns `Code` to invoke one of the `visit` methods. `lookup(T)` may only select non-static `visit` methods that have a return type of `void` and take only a single argument of public type `S` where `S` must be a supertype of `T`. `lookup` may return `null` in which case `visitDefault` is invoked. If not overridden, `visitDefault` throws

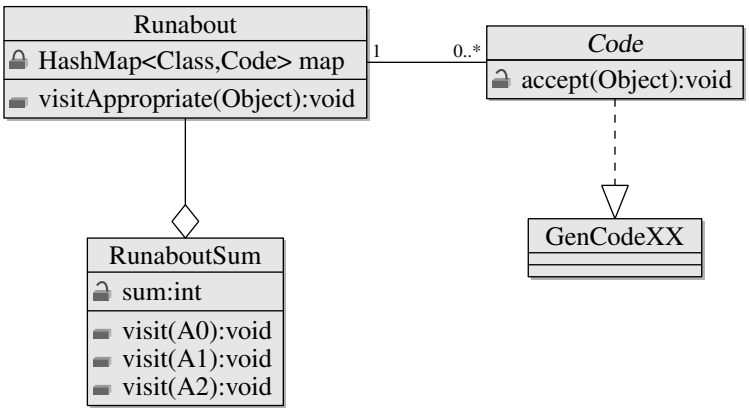

Fig. 8. UML class diagram for the Runabout.

a run-time exception to indicate that no `visit` method was found. `lookup` also throws runtime exceptions to indicate ambiguities in the method resolution.

Note that `visitAppropriate` does *not* require that all `visit` methods have a common base class other than `Object`. Thus, the Runabout does not require the interface `A` that most of the other implementations use to declare the dedicated method, to declare the accept method, or to assist the type system in the form of the `A@`.

The fact that the Runabout does not require `accept` methods or a common interface in the visitees is often beneficial when dealing with code where adding an `accept` method is not possible, as with, for example, `String`. A simple example for this is given in Fig. 9.

```
public static void main(String[] arg) {
  MyRunabout mr = new MyRunabout();
  mr.visitAppropriate("Hello");
  mr.visitAppropriate(new Integer(1));
  assertTrue(mr.cnt == 3);
}
public class MyRunabout extends Runabout {
  int cnt = 0;
  public void visit(String s)  { cnt += 2; }
  public void visit(Integer i) { cnt += i.intValue(); }
}
```

Fig. 9. Using the Runabout on any kind of visitee.

lookup Which `visit` method is invoked by `visitAppropriate` is specified by the *lookup strategy* that is implemented by `lookup`. Defining a lookup strategy is similar to defining how a compiler (like `javac`) resolves method invocations for overloaded methods [23, section 15.11.2]. The main difference is that instead of the static type, the dynamic type of the argument object is used. As with overloading, multiple methods may be applicable. In the case of `javac`, the method with the closest matching signature is chosen, and a compile error is generated in the case of ambiguities.

As an input, the `lookup` function is passed the dynamic type of the object on which the dispatch takes place. The dynamic type is a node in the inheritance hierarchy (a directional acyclic graph), which can then be traversed by the function to find a matching type for which a visit method exists. A simple example for an implementation of `lookup` that does not consider interfaces is given in Fig. 10. The helper method `getCodeForClass(c)` tests if a `visit` method for the type `c` exists and if so returns the `Code` instance for that `visit` method.

```
protected Code lookup(Class c) {
  while (c != null) {
    Code co = getCodeForClass(c);
    if (co != null)
      return co;
    c = c.getSuperclass();
  }
  return null;
}
```

Fig. 10. Example of a `lookup` method.

What follows is an informal description of the default lookup strategy used by the Runabout. If `visit` methods for both classes and interfaces are applicable to the given dynamic type, the `visit` method for the *class* closest to the dynamic type is chosen. If no `visit` method for a superclass of the dynamic type exists and if there is only one `visit` method matching any of the interfaces implemented by the dynamic type, then that `visit` method is selected. If `visit` methods for multiple interfaces implemented by the dynamic type (but none for its parent classes) exist, a runtime exception indicating the ambiguity is thrown. If no applicable `visit` method exists at all, `null` is returned, causing the invocation of `visitDefault`.

2.3 Discussion

The Runabout as described so far is more expressive than typical visitors and has fewer requirements for the visitees. Primarily, the Runabout does not require `accept` methods in the visitees. On the other hand, additional restrictions

imposed by the Runabout are that all the visitee classes and all subclasses of `Runabout` must be public (the Runabout must internally cast to these types) and that all `visit` methods must be public. These restrictions are minor since if the visitees are legacy code, the classes are probably already public; adding `accept` methods (or even dedicated methods) would typically be much harder. Making the subclass of `Runabout` or its `visit` methods public is even less likely to be a problem. A slightly more limiting constraint is that the Runabout requires `visit` methods to return `void` and take just one argument. A more sophisticated implementation should be able to relax this requirement.

MultiJava does not impose restrictions on the access modifiers, the specialized compiler takes care of these problems. The advantage of extending the language is that MultiJava is more expressive than any other solution; for example, it is possible to dispatch on more than one argument. MultiJava is also not constrained by the requirement that the methods that are multi-dispatched are named `visit`, which allows MultiJava to support many multiply-dispatched method families in the same class.

While the Runabout could be extended to allow names other than `visit`, we feel that in practice this limitation will hardly ever be a problem; in fact, it is likely that several multi-dispatch method families in the same class without any syntax to mark these methods would instead confuse programmers. MultiJava's approach of extending the Java syntax solves this problem but prevents users from deploying other language extensions like GJ [4] or AspectJ [28] in the same code. The current implementation of MultiJava uses linear sequences of `instanceof` tests, making the tool impractical for large numbers of visitee classes. A better implementation of MultiJava might take care of this major performance issue.

One of the drawbacks of visitors is that they often require writing excessive amounts of trivial code. All `visit` methods must be declared in a base class (or interface) which is used by the accept method. The accept methods themselves can be tedious if the code has many visitees. Also, the visitor pattern is less expressive than the Runabout since it requires the programmer to occasionally add additional code to perform the intended dispatch. For example, suppose some of the visitee types form a hierarchy where A, B and C represent similar visitees and thus extend the common parent P. In this case, the visit methods for P, A, B and C are sometimes identical. In the case of the Runabout, only one visit method for P needs to be implemented; the `lookup` for A, B and C will automatically result in the invocation of `visit(P)`. For visitors, either the code is replicated or the default visitor pattern [27] where `visit(A a)` calls `this.visit((P)a);` must be used, forcing the user to write additional methods which merely indirect the control flow.

3 Implementation

This section describes the implementation of `Runabout`. In particular, it describes how the constructor builds the *dynamic code map* and how the `visitAppropriate`

method uses that map to invoke the appropriate `visit` method. The section also discusses extensions to the core functionality, such as handling of primitive visitees and addition of `visit` methods that are not declared in the subclass of `Runabout`.

3.1 The dynamic code map

Central to the implementation of the Runabout is the dynamic code map. This hash table maps the dynamic type of the argument to an implementation of `Code` (see Fig. 11), an abstract class. Instances of `Code` are stateless and can be seen as the Java equivalent of C function pointers. The virtual method table of the code objects refers to a piece of code that is to be invoked for arguments of the corresponding dynamic type.

The constructor of `Runabout` scans the `Runabout` instance (using reflection) and creates a specialized object of type `Code` for every visit method that is found. Note that the constructor does not inspect the `Runabout` class but the specific subtype that is being instantiated. This the primary reason why this cannot be done inside of the static initializer of the `Ruanbout` – the possible dynamic subtypes are not known at that time.

The constructor creates a class for each instance of `Code` (Fig. 11) on-the-fly using Java's dynamic class-loading mechanism. The code generation only substitutes constant pool entries in a pre-compiled `.class` file – which itself stored as a byte array inside of the `Runabout`. As a result, the generation code is extremely small and fast, consisting only of a single allocation and a few `System.arraycopy` operations. The generated code is best illustrated with an example. If the concrete instance of `Runabout` is of type `RunaboutExample` and the visit method takes `String` as the argument, the dynamically generated code will correspond to the Java code in Fig. 12. The `X` is replaced with a unique number to avoid name clashes. An instance of `GenCodeX` is instantiated and installed in the dynamic code map.

```
public static abstract class Code {
  public abstract void visit(Runabout r, Object o);
}
```

Fig. 11. The `Code` class is an inner class of the `Runabout` that defines the interface for the dynamically generated and loaded code.

3.2 Lookup

The implementation of `Runabout.visitAppropriate` is now simple (Fig. 13). `visitAppropriate` does a `get` on the dynamic code `map` to find an object of

```
class GenCodeX
  extends Runabout.Code {
  public void visit(Runabout r, Object o) {
    ((RunaboutExample) r).visit((String)o);
  }
}
```

Fig. 12. Source equivalent of the code that is dynamically generated code when reflection finds the method `RunaboutExample.visit(String)`. An instance of this type is returned by `map.get(String.class)` in `visitAppropriate`.

type `Code`. If no matching code is found, the `lookup` procedure is invoked to find a matching piece of code and the dynamic code map is updated. Finally, the code found in the code map is invoked. Note that lookup returns a code instance `nocode` with an implementation of `visit` that throws a `RunaboutException` if no matching visit method was found in the lookup. Note that lookup runs at most once for every dynamic type passed to `visitAppropriate` per `Runabout` class. Lookup also never needs to perform dynamic loading; the initial population of the dynamic code table in the constructor has created all the `Code` instances that are needed.

```
public final void visitAppropriate(Object o) {
  Class cl = o.getClass();
  Code co = map.get(cl);
  if (co == null) {
    co = lookup(cl);
    if (co == null)
      co = visitDefaultCode;
    map.put(c, co);
  }
  co.visit(this, o);
}
```

Fig. 13. visitAppropriate finds the dynamically generated code for an object in the hash table `map`.

3.3 Caching generated code

As in [5], the Runabout uses caching to improve the performance. The dynamic code map as described above caches the results of the lookup. While this is effective to improve the time of running `visitAppropriate`, creating a `Runabout` instance is also a performance concern. Creating runabouts involves the use of

reflection to find the declared `visit` methods and dynamic code generation, class loading and reflective instantiation of `Code` objects. The performance of runabout creation can be improved by sharing the dynamic code map between instances of the same runabouts. For this, the implementation uses a second `Cache` that is basically a thread-local hash table that maps subclasses of `Runabout` to instances of the dynamic code map. The `Cache` is thread-local to eliminate the need for synchronization on the maps. Every new instance of `Runabout` is checked against the cache, limiting the use of reflection and dynamic code generation to once per `Runabout` class. Since the code maps are shared, this also further limits the use of the lookup function to only once for each combination of thread, `Runabout` class, and dynamic type that is used in the dispatch. The Runabout uses the same class loader for all instances that share the same `Cache`.

3.4 Extensions

In order to support primitive visitees, our Runabout implementation provides a second `visitAppropriate` method, which takes an additional argument of type `java.lang.Class`. This second argument is used to distinguish between primitive types and their wrapper classes. The Runabout provides empty `visit` methods for the 8 primitive types that can be overridden by subclasses.

A typical use of this facility would be the iteration over an object graph using reflection. Fig. 14 shows the code of a simple iterator that counts the number of primitive `int`s that are reachable from any argument passed to `visitAppropriate`. Note that the example code does not handle cycles in the object graph.

Another simple extension is adding methods to the code map that are not `visit` methods in the subclass of `Runabout`. The Runabout interface provides the method `addExternalVisit(Class cl, Code co)` to allow adding an *external* visit method to the Runabout. If the Runabout encounters an object of the specified class, it calls the visit method defined in `Code`. Note that the `visit` method declared in `Runabout.Code` is declared to take `Object` as the type of the visitee argument, and while the Runabout guarantees that the object passed will be a subtype of `cl`, it cannot verify that casts in `Code` are safe. It is important to mention here that the cost of the dispatch in the Runabout is not changed at all by this extension.

Adding external visit methods to an instance of `Runabout` should not modify the behavior of other instances. Thus, the shared dynamic code that was obtained from the `Cache` is copied when the first *private* extension is added.

4 The Runabout Optimizer

The Runabout Optimizer is a static program transformation tool that performs a semantics-preserving translation of code that uses the Runabout into code that does not require reflection or dynamic code generation. The generated code instead uses a combination of `instanceof` tests and double dispatch to achieve the

```
public class CountInt extends Runabout {
  int count = 0;
  public void visit(int i) {
    count++;
  }
  public void visit(Object o) {
    Field[] fields = o.getClass().getDeclaredFields();
    for (int i=0;i<fields.length;i++)
     if (! Modifier.isStatic(fields[i].getModifiers()))
        visitAppropriate(fields[i].get(o),
                         fields[i].getType());
  }
}
void run() {
  CountInt ci = new CountInt();
  ci.visitAppropriate("Example");
  System.out.println(ci.count + " int fields reachable");
}
```

Fig. 14. Using the Runabout with primitives: counting the number of reachable fields of type `int` in an object graph without cycles.

same dispatching semantics. The Optimizer performs a static data-flow analysis on a closed-world application to replace the `visitAppropriate` call with semantically equivalent and possibly more efficient code. Essentially, the Optimizer converts code that uses the `Runabout` into equivalent code using the visitor pattern.

There are several motivations for using the Runabout Optimizer. First, there exist JVMs in which using reflection or dynamic code generation is either not supported at all or results in severe performance penalties. An example of the first category is a VM for embedded devices [1]; the second category includes VMs that lack a just-in-time compiler and instead rely on ahead-of-time compilation for real-time performance [36].

These exotic uses aside, the primary motivation why the Runabout Optimizer was developed was to allow for a direct performance comparison of Runabout and visitor-based code through real-world examples without the need to manually rewrite the code to support both variants.

Knowing the semantics of the `visitAppropriate` call, the key idea behind the Runabout Optimizer is simple: to replace such calls with double dispatch. Given all runabouts and all of the visitee classes, the Optimizer can replace `visitAppropriate` calls with new dispatching code. While the idea is rather simple, there are quite a few corner cases that need to be handled.

The Optimizer needs to distinguish between three different categories of visitee types. First, there are the types where the Optimizer is allowed to add `accept` methods. This category includes the code from the application itself

and, if applicable, that of the framework which the application is using. This first set typically *excludes* types from the JDK, since the JVM may rely on a specific implementation of these types. Even if this is not the case, it is typically not desirable to ship a copy of the runtime libraries with the application. Hence, the second category of visitee types are the types where the Optimizer is not allowed to insert synthetic `accept` methods. The third category consists of the primitive types. These types cannot be handled with `instanceof` tests and thus require special treatment. Naturally, dispatch on primitives is rare in practice. The details of this special case are uninteresting, however, and are thus not discussed further.

For each `visitAppropriate` call site the Optimizer performs an intraprocedual data-flow analysis to conservatively determine the possible visitor and visitee types that could be used at that callsite. In many cases, the specific subtype of `Runabout` is statically known and all possible visitee types are defined by the application, allowing the Optimizer to extended those classes with `accept` methods. In those cases, the `visitAppropriate` call is replaced with a call to those synthetic `accept` methods, which then perform the appropriate double dispatch. If visitee types from the system library flow into the callsite, a sequence of `instanceof` tests is used to invoke the correct visitor method, possibly followed by an `accept` call for the remaining application types (see Figure 15). Visitees of primitive type are also handled by appropriate tests if necessary. Since the data-flow analysis is conservative and purely intraprocedual, it may drag in otherwise dead classes. It is therefore advisable to run a dead code elimination tool (such as [43] or [24]) on the code before running the Runabout Optimizer.

If multiple different subtypes of `Runabout` are determined as possible visitors, the call to `visitAppropriate` is preserved. This virtual call is used to dynamically determine the runtime type of the Runabout. The `visitAppropriate` method is overridden with a synthetic `visitAppropriate` method in each subclass. This synthetic method then contains the `accept` call and, if applicable, the sequence of `instanceof` tests that are necessary for this particular visitor.

Both cases are complicated by the possibility of the user having already overridden `visitAppropriate`. In that case, it is not acceptable to inline the call to `accept` or to override `visitAppropriate` in a subclass. The Optimizer detects this case and instead inserts a general version of `visitAppropriate` into the method that a call to `super.visitAppropriate` inside of the user-defined `visitAppropriate` method would invoke. This general version then determines the dynamic type of the visitor and triggers the appropriate visit call. Naturally, the resulting general dispatch sequence is more expensive and thus clients that are intended to be optimized with the Optimizer are discouraged from overriding `visitAppropriate`.

Table 1 gives an overview of the different costs associated with each approach for the different types of arguments. The table differentiates between callee types from the application and from the framework. The reason for this is that in practice there is a significant difference between these two categories for both the visitor pattern and the Runabout. For the visitor pattern, it is *problematic*

```
public final class Visitor extends Runabout {
  public void visit(A a) { }
  public void visit(B a) { }
  public void visit(String s) { }
}
public interface Visitor_acceptor { // synthetic
  public void accept__Visitor(Visitor v);
}

public class A implements Visitor_acceptor {
  public void accept__Visitor(Visitor v) { // synthetic
    v.visit(this);
  }
}
public class B implements Visitor_acceptor {
  public void accept__Visitor(Visitor v) { // synthetic
    v.visit(this);
  }
}
static void callVisit(Visitor v, int i) {
  Object o = new Object();
  switch (i) {
    case 1: o = new A(); break;
    case 2: o = new B(); break;
    case 3: o = "string"; break;
  }
  // replaced ``v.visitAppropriate(o);'' with synthetic code:
  if (o instanceof Visitor_acceptor)
   ((Visitor_acceptor)o).accept__Visitor(v);
  else if (o instanceof String)
    v.visit((String)o);
  else
    v.visitDefault(o);
}
```

Fig. 15. Simple example for code generated by the Runabout Optimizer for a call to `visitAppropriate`. Here the visitor type is statically known and both double-dispatch and `instanceof` tests are necessary. Note that the actual tool operates on bytecode.

to extend the types of the framework with `accept` methods. This is because it is generally desirable from a design perspective for a client application to not modify any framework code; such modifications make maintenance more difficult. For the Runabout, the callee types from the framework are actually *less* problematic than dispatching on types from the application: the constraint that the types need to be public is trivially true, since an application using a

given framework can obviously only get direct access to the public types. This makes the restriction that the visitee types must be public redundant.

	Visitor Pattern	Runabout	Runabout Optimizer
Application	inconvenient	requires `public`	double-dispatch
Framework	problematic	%	double-dispatch
Core library	infeasible	%	`instanceof`
Primitives	infeasible	wrapper overhead	additional tests

Table 1. Overview of the costs of using the different approaches on types in different categories.

While the Runabout Optimizer needs to know all of the runabouts and their visitee classes in order to operate, it does not need a completely closed-world setting. Clearly, adding unrelated classes - that is, classes that neither extend `Runabout` nor any of the visitees - to a system transformed by the Runabout Optimizer is unproblematic. Furthermore, it is possible to add subclasses to the visitees' classes. However, if runabouts use an interface as one of the visitee types, adding new subtypes of that interface which do not extend existing classes that implement the interface after running the Optimizer will result in runtime errors. The reason for this is that the Runabout Optimizer may have added synthetic `accept` methods to those interfaces which would then not be implemented by those new subtypes. It is possible to add new runabouts to a program that was already transformed by the Runabout Optimizer as long as those extend `Runabout` directly. This is possible since the Runabout Optimizer preserves the original Runabout code. The synthetic non-reflective version generated by the Optimizer is invoked through a synthetic constructor that switches `visitAppropriate` to the non-reflective variant that is used for transformed code.

5 Experimental Results

This section presents various experimental results. Unless otherwise noted, the results are for the plain Runabout code (without transformations from the Runabout Optimizer). To begin with, micro-benchmarking results are used to demonstrate that the Runabout is comparable in performance with the other designs. Then, using the static bytecode analysis tools Kacheck/J and Hitsuji as examples, we describe our experience with refactoring actual applications to use the Runabout instead of visitors. Our experience confirms an opinion also expressed in [33]: multi-methods simplify the task of building extensible compiler frameworks. Finally, the performance impact of using the Runabout in these tools is evaluated. This is achieved by measuring the runtime for both the code based on the Runabout and the result of the bytecode transformation by the Runabout Optimizer which yields the equivalent visitor code.

5.1 Synthetic Micro-benchmarks

In order to evaluate the performance of the Runabout, we have run variations of the example presented in section 2.1. The four major designs were run on IBM JDK 1.4.0 and Sun JDK 1.4.1 on a PIII-1000 running Linux 2.4.18. The time measured corresponds to a total of 10 million invocations. The reported numbers are the average over 10 runs in single user mode. All methods are invoked with equal frequency.

Two parameters have a significant impact on the benchmark. First of all, the number of visit methods (and visitee classes) is important to see if the design scales to complex visitee structures. The graphs contain the results for one to 20 visitee types; appendix A shows the results for up to 200 visitee types. For 2,000 types, MultiJava is currently unable to compile the test-case. Appendix B shows results for Sun JDK 5.0, IBM JDK 5.0 and GNU GCJ 4.xxx.

The second important parameter is the hierarchy of the visitee types. The graphs show the numbers for a totally flat hierarchy (every visitee extends Object, as illustrated in Figure 16) and for a hierarchy of maximum depth (visitee class n extends $n-1$, as illustrated in Figure 17). Note that in the case of the deep hierarchy, the number of visit methods is equal to the depth of the hierarchy. The choice of hierarchy impacts the runtime of subtype tests, performed frequently by the MultiJava and the Runabout implementation. The flat hierarchy is an important data point since it characterizes both the best-case (in terms of performance) as well as a common case – parser generators like the Java Tree Builder [26] often generate a completely flat hierarchy. The deep hierarchy is the worst case, especially for virtual machines that do not use a constant-time subtype test [38]. For real-world benchmarks, the actual performance should fall between these two extremes – with a strong bias towards the case of the flat hierarchy.

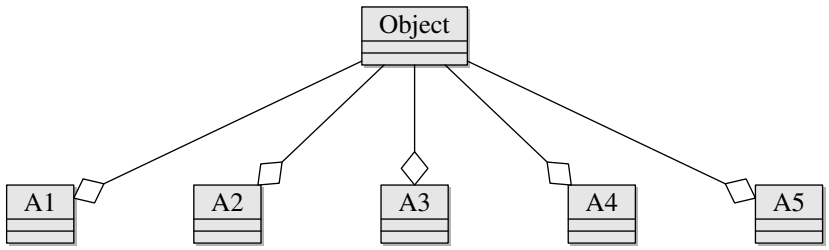

Fig. 16. Illustration of the flat visitee hierarchy used for the micro-benchmarks (for four visitee classes).

As the benchmarks in Fig. 18 and 20 show, the differences in execution speed between the approaches are small. The Runabout has the highest cost for just one visit method; MultiJava degrades with higher numbers of visitee types.

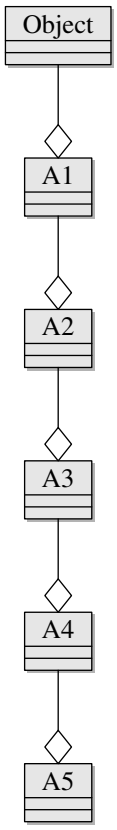

Fig. 17. Illustration of the deep visitee hierarchy used for the micro-benchmarks (for four visitee classes).

If the hierarchy is deep (Fig. 19 and 21), the `instanceof` tests in MultiJava become more expensive. The performance of dedicated methods, visitors and the Runabout is not changed significantly.

In practice, the differences between all five approaches (with the potential exception of MultiJava for large hierarchies) are minor. In particular, the fact that the double dispatch of the visitor turns out to be faster than the dedicated method cannot be explained and is presumably just an artifact of the optimizing compiler. Overall, the numbers from the micro benchmark are too close to rule out any of the variants: the runtime of any application using any of the variants of the visitor pattern is typically not determined by the tiny cost of the dispatch but rather by the actions performed in the visit methods. Also, hierarchies in real applications are typically not very deep; thus, Fig. 18 is more realistic than Fig. 19.

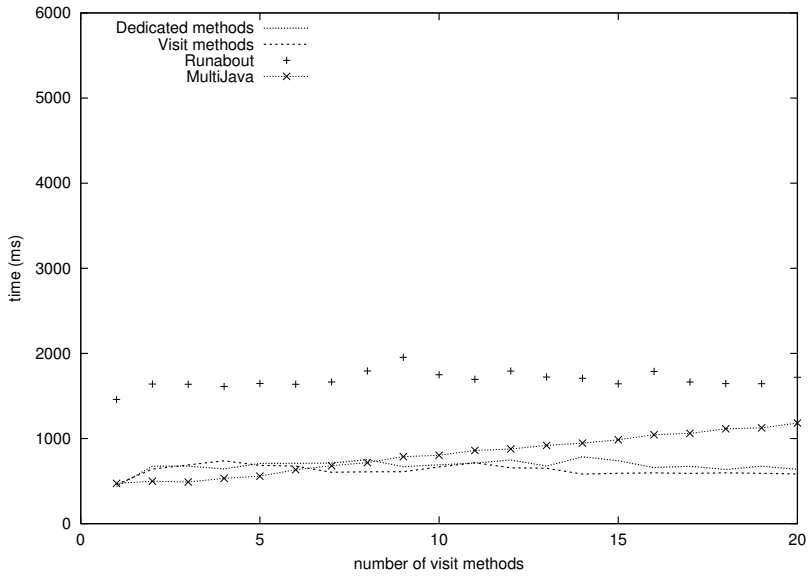

Fig. 18. Sun JDK 1.4.1, flat hierarchy.

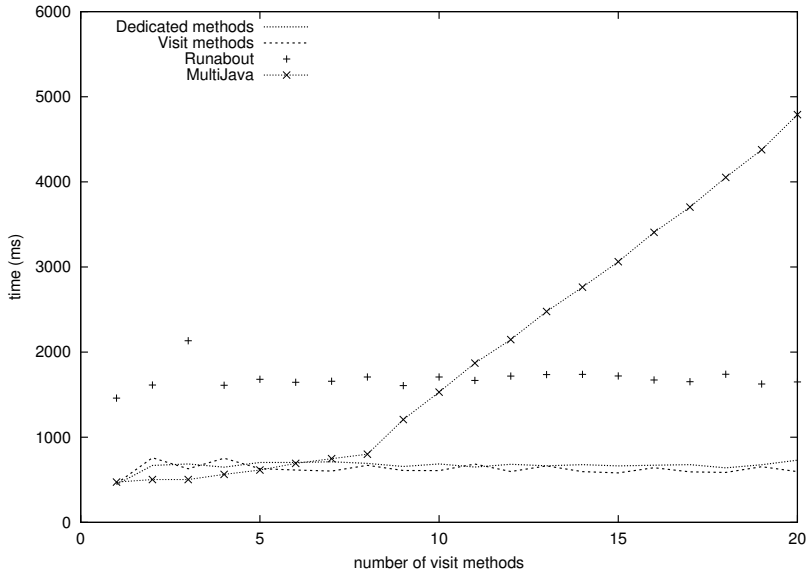

Fig. 19. Sun JDK 1.4.1, deep hierarchy.

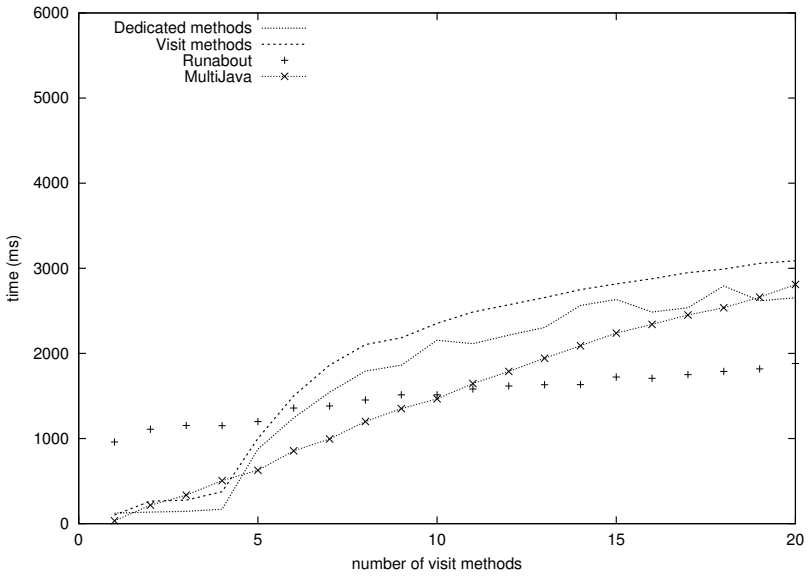

Fig. 20. IBM JDK 1.4.1, flat hierarchy.

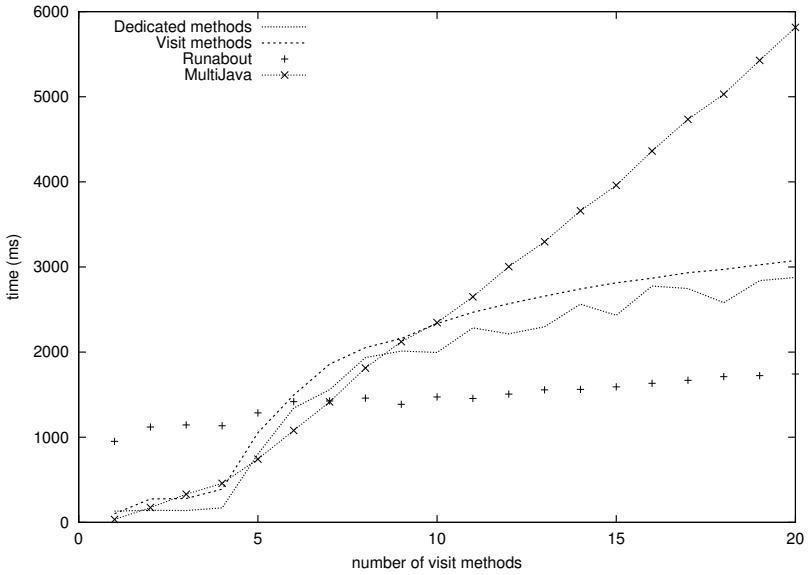

Fig. 21. IBM JDK 1.4.1, deep hierarchy.

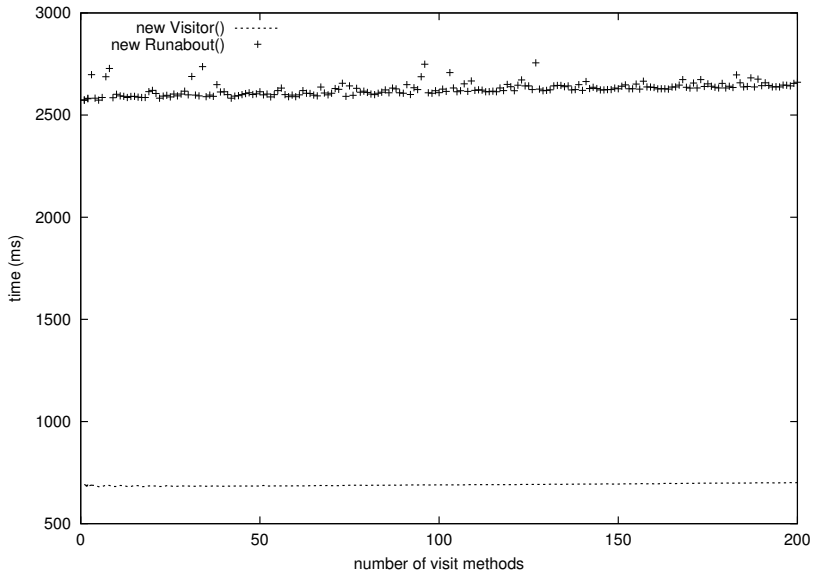

Fig. 22. This graph compares the time to create 10,000,000 instances of the same Runabout with the time to create 10,000,000 visitors on Sun JDK 1.4.1. The creation of runabouts with the cache is about a factor of 3.8 slower.

The measurements above only reflect the time required for the invocation. The Runabout is also a bit more costly to create compared to instances of ordinary visitors. Fig. 22 shows the cost of creating 10,000,000 instances of `Runabout` (with caching enabled) compared with the creation of 10,000,000 instances of an equivalent visitor. As the numbers imply, creating runabouts can be as little as roughly a factor of two slower than creating visitors. This is, however, only half-true. The Runabout implementation caches reflective information, in particular instances of the dynamically generated and loaded classes, in a thread-local cache. The creation of the first instance of a given `Runabout` type in a new thread is more expensive. Fig. 23 shows the cost of creating 1,000 runabouts without the cache. The numbers show that caching the reflective information per-thread reduces the overhead of creating runabouts by a factor of up to 810,000 for 200 visit methods. Note that while it would be possible to cache the information globally and not just per thread, this would introduce synchronization operations in `visitAppropriate`, which would probably be worse in most applications. Profiling Runabout creation shows that dynamic class loading and reflective instantiation are each responsible for over 30% of the time of Runabout creation.

5.2 Importance of Code Generation

The approach of caching objects used for dispatching as described in the previous section was previously suggested by Bravenboer and Visser [5]. However, Bravenboer and Visser cached instances of `java.lang.reflect.Method` instead of instances tied to dynamically generated code. Using instances of `Method` for dispatching introduces a significant performance penalty compared with code generation. Figure 24 compares the performance of Bravenboer's approach with the Runabout. Note that the generic object traversal, which is also part of [5] was not used – the only differences in the implementations are the use of dynamic code generation vs. reflection.

5.3 Refactoring Kacheck/J

Kacheck/J is a bytecode analysis tool used to infer confined types [25] and is written using the Ovm bytecode framework [37]. Ovm [36] is a customizable Java Virtual Machine whose framework contains a bytecode analysis and manipulation framework which was previously based on visitors. In particular, the framework provides an abstract interpretation engine that uses flyweight instruction objects as visitees to support abstract execution. Kacheck/J uses this abstract execution framework to analyze code.

In the previous version of the framework, every instruction object had an `accept` method. In addition to this, every instruction had an `execute` method

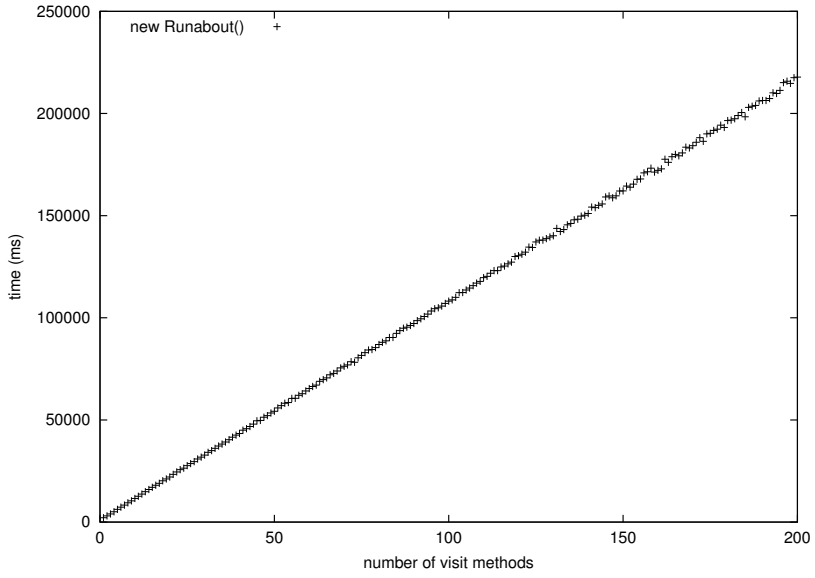

Fig. 23. This graph shows the time it takes to create 1,000 runabouts *without* the cache on Sun JDK 1.4.1.

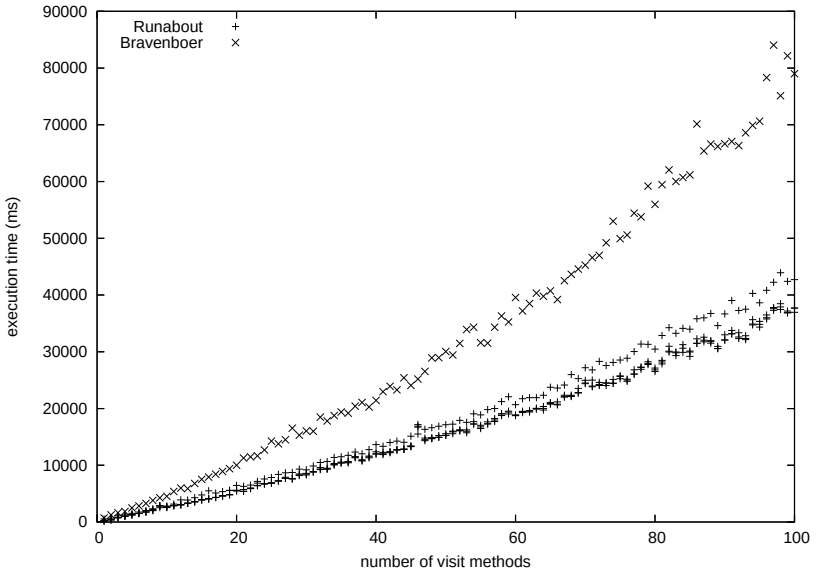

Fig. 24. Using dynamic code generation (Runabout) vs. using reflection (Bravenboer). Using Sun JDK 1.5.0 with a flat hierarchy with the same setup as used in Appendix B.

which would perform the state manipulation on the abstract interpreter for this instruction during abstract interpretation. In its main loop, the abstract interpreter would call the `execute` method to simulate the instruction. These `execute` calls are interleaved with accepting on a visitor which executes code specific to the analysis. Additional code in the main loop took care of control flow handling and merging of abstract states. The `execute` method can be seen as a dedicated method for abstract execution.

The analysis framework was refactored to use runabouts instead of the dedicated `execute` method and the `accept` method. An immediate benefit of this change was that adding new instruction types for analysis on VM specific bytecodes (for example, quick opcodes) no longer required adding visit methods to parts of the framework that were not concerned with these types of instructions. Factoring out the `execute` method into a runabout has made it easier to change its behavior. In order to ease the selective manipulation of the abstract execution, each abstract interpretation step was split into two runabouts: one that manipulates the local variables and the stack of the abstract machine, and one that is responsible for control flow, exception handling, and merging of states.

In addition to getting rid of hundreds of `accept` methods and hundreds of `visit` methods in the matching abstract visitor, the introduction of the Runabout also allowed other code reductions. Many analyses were grouping `visit` methods for closely related instructions, such as *invokevirtual*, *invokeinterface*, *invokespecial* and *invokestatic*. This was achieved with the default visitor pattern [27]. Since the hierarchy of instructions is fairly complex, multiple default visitors (where one visit method was just dispatching to another visit method) existed in the old framework. Maintaining these default visitors, especially with changes in the hierarchy of the instructions, has always been a problem. With the Runabout, most of what we were trying to achieve with the default visitors was covered by the lookup algorithm, making all of these classes obsolete.

The new framework also has some additional features unrelated to the Runabout that impact Kacheck/J's performance (for example, an extensible set of abstract values for application-specific abstract execution domains). Additionally, Kacheck/J's implementation is slightly more powerful; for example, it records and reports much more detailed information about the constraint system. These and other changes make the two versions of the code not entirely comparable. The original Kacheck/J tool takes about 57 seconds to analyze the entire Sun JDK 1.4.0 from the Purdue Benchmark Suite (JDK5) running on top of Sun JDK 1.4.1 on a PIII-800. The performance of the redesigned Kacheck/J is about 77s to analyze the entire Sun JDK 1.4.0.[1] During the analysis, the innermost loop performs slightly more than 10 million invocations of `visitAppropriate`.

5.4 Refactoring Hitsuji

Hitsuji [40] is an implementation of 0-CFA [41] for Java. It uses a demand-driven approach to compute receiver sets, walking over the data-flow graph build by the abstract interpreter. The computation of the receiver sets dominates the runtime of the analyzer. This iteration over the data-flow graph initially used visitors, but the use of visitors resulted in some awkward code. The reason was that the data-flow graph could be built naturally using the abstract values from the data-flow analysis. However, the canonical abstract values lacked the proper `accept` methods. Since the abstract values were logically part of the bytecode framework, adding the tool-specific visit methods was not a good option. An initial implementation used the adaptor pattern [20] to wrap the abstract values to provide the desired `accept` methods.

Using runabouts instead of visitors avoids creating these wrappers, resulting in a slight reduction of memory consumption and garbage collection times at the expense of a more expensive call in the central loop of the analysis.

[1] On a dual-processor system, the difference between the original tool (44s real time, 60s CPU time) and the new implementation (62s real time, 95s CPU time) is slightly different.

5.5 Using the Runabout Optimizer on Kacheck/J and Hitsuji

The Runabout Optimizer was used on both Kacheck/J and Hitsuji to obtain a fair comparison between code using visitors and code using the Runabout. Note that neither Kacheck/J nor Hitsuji make use of the more exotic Runabout features. The `visitAppropriate` call is always performed on application types[2] and not on primitives or JDK classes. For Hitsuji, the Optimizer can generate visitors with what is in practice straightforward double-dispatch code. For Kacheck/J, the generated code requires an additional `instanceof` test, since `visitAppropriate` is overridden in one place.

Table 2 shows the performance of Kacheck/J and Hitsuji analyzing GNU classpath 0.07 [22] for Hotspot 1.4.1 and the IBM JDK 1.4.2 on a PIII-1000 before and after applying the Runabout Optimizer. The Optimizer itself converts the code of both applications in a few seconds. Note that the resulting code after applying the Runabout Optimizer is essentially equivalent to the original code using visitors – only this time, the `accept` methods are machine-generated.

	Sun JDK 1.4.1	IBM JDK 1.4.2
Kacheck/J 3.1.0	10.840s	23.770s
Kacheck/J 3.1.0-RO	12.410s	21.340s
Hitsuji 0.9.1	51.040s	64.300s
Hitsuji 0.9.1-RO	48.860s	96.780s

Table 2. Performance data for Kacheck/J and Hitsuji on Sun JDK 1.4.1 and IBM JDK 1.4.2 before and after conversion (of the static analysis tool) with the Runabout Optimizer. The numbers show the CPU user time as reported by `time` of the fastest of three consecutive runs.

The benchmarks primarily illustrate two properties of dispatches in Java. First, there is no clear winner in terms of implementation. It depends on both the benchmark and the virtual machine. Only given both it is even meaningful to try to optimize performance by picking either approach. Secondly, the benchmarks show that, in general, using the (unoptimized) Runabout does not necessarily lead to a loss in overall performance, since the performance for these real-life benchmarks is roughly the same using the (unoptimized) Runabout or visitors.

How useful the Runabout Optimizer is as a tool clearly depends on both the benchmark and the virtual machine. A virtual machine using ahead-of-time compilation is likely to require or at least benefit significantly more from the Runabout Optimizer than a just-in-time compiled or even interpreted system. The benchmarks also show that the Runabout itself is clearly useful. In Hitsuji, the use of the Runabout avoids adding wrappers and polluting the *source* of the framework with accept methods at no cost. In cases where it would be beneficial

[2] In this case, the Ovm framework is specified to be a part of the application and can thus be modified by the Optimizer.

for performance, the Runabout Optimizer can be used to add accept methods to the *binary* `class` files of the framework, resulting in the same performance that a visitor would have provided.

6 Related Work

6.1 Walkabout Variations

The Walkabout pattern as described in [39] allows the traversal of an arbitrary object graph without double dispatch. Instead of double dispatch, reflection is used to find the matching visit method. The authors noted that their implementation was impractically slow. The implementation given in [39] also requires that the type of the argument to the visit method exactly matches the type of the object that is being visited. Bravenboer [5] improved the performance of the Walkabout by caching reflective results, but their performance is still poor; the authors report results that are about 100 times slower than visitors. While modern virtual machines have reduced the cost of reflection somewhat, Bravenboer's approach is still significantly slower than the Runabout (even when limited to dispatching without reflective traversal) as shown earlier in Section 5.2.

The Sprintabout [18] is a variation of the Runabout where the programmer implements an abstract base class with `visit` methods and an abstract `visitAppropriate` method. In the design of the Sprintabout, dynamic code generation is then used to generate a subclass of the abstract base class where `visitAppropriate` is instantiated with code that essentially equivalent to that of the Runabout. The main difference is that by putting `visitAppropriate` into a subclass, the generated dispatching code can be simplified by a single cast – the Runabout needs to cast to the specific subclass (since only the subclasses have the necessary `visit` methods) whereas the Sprintabout can avoid this cast. The author of the Sprintabout claims performance benefits over the Runabout of up to 30%. However, the Sprintabout is significantly more complex – the implementation is significantly larger (see Table 3) and the requirement of changing the instantiation of abstract base-classes by the Sprintabout framework requires changes in allocation code which maybe in unrelated parts of the code. Finally, the author of this paper was unable to get the Sprintabout to work – the Sprintabout only links with one particular version of the ASM framework that I could find, and that particular version fails to generate verifying bytecode.

The Sprintabout's approach of using subclasses for the dispatching code instead of the parent is shared by PolyD [10], another multi-dispatching facility inspired by the Runabout. PolyD enables true multi-dispatch, supports arbitrary return values, custom dispatch configuration including mixins [16] and even features a runabout-compatibility mode. For simple single-dispatch, we expect the Sprintabout to have roughly the same performance as PolyD which is slightly better than that of the Runabout. PolyD shares the requirement of programmers using an abstract base-class which is instantiated by the PolyD framework with the Sprintabout. Appendix B provides results from micro-benchmarks com-

Approach	Size
Dedicated Methods	0
Visitor Pattern [21]	0
Bravenboer [5]	4566
Runabout	8655
Sprintabout [18]	35461
PolyD [10]	92975
JMMF [17]	120341
MultiJava compiler [9]	2024608

Table 3. Code size for the different multi-dispatching techniques. The sizes listed are the size of the compressed JAR files. The goal of this table is simply to give a rough idea of the complexity of the projects, minor differences in size are not significant.

paring the performance of visitors, runabouts and PolyD. In terms of size and complexity, PolyD dwarfs the Sprintabout (Table 3).

6.2 Multiple-Dispatch

Implementations of virtual method dispatch which are efficient in terms of both space and time have been the subject of a large body of research [8,12,32,42]. In a statically-typed single inheritance setting, the most common approach is to use virtual function tables. More advanced languages, such as Java, require more complicated designs [2] to handle features like multiple inheritance and dynamic class loading. A good overview of recent research is provided in [46]. The Runabout is not concerned with these low-level compilation techniques and instead uses a large hashtable which puts it close to dynamic perfect hashing [11], a design that is not suitable for general dispatch techniques since it comes with a large space penalty. For the Runabout, space is not really a concern since there are typically few subtypes of `Runabout` in any given application. Furthermore, the type-safe high-level implementation of the dispatch in the Runabout cannot use some of the low-level techniques that compilers would use for dispatching.

A technique to provide multi-dispatch in the JVM is described in [13]. The authors have extended the virtual machine to use multi-dispatch for classes that were marked for multi-dispatch. While their approach is compatible with existing java compilers and libraries, it requires modifications to the VM. The authors also implemented MDLint, a tool to statically analyze code and warn programmers about ambiguities in the multi-dispatch. The Runabout Optimizer also statically detects such ambiguities and does not defer them until runtime.

Closely related to the Runabout Optimizer is doublecpp [3], an extension of C++ that supports multi-dispatch. doublecpp transforms a variant of C++ with multi-dispatch into C++ code that uses visitors. doublecpp supports a modular transformation (with is equivalent to the split into application and

runtime library classes in the Runabout Optimizer) which uses runtime type information instead of double-dispatch to determine the target method.

6.3 Language-based Approaches

Another language-based approach that can be used to address the extensibility problem is the use of aspect oriented programming [29]. With aspects, it is possible to introduce `accept` methods into existing classes. These `accept` methods can then be used with visitor-aspects that traverse those classes. In terms of performance, the resulting object code would correspond precisely to that of using visitors. The main problem with this approach is that it pretty much requires whole-world compilation. Since `accept` methods cannot be introduced (without significant overhead) after instances of the particular classes have been created, the aspect compiler must be aware of all aspects at link time (for Aspect/J [28], the latest possible link time is the stage where the Aspect/J runtime performs load-time weaving of aspects). With the runabout, it is possible to dynamically load subtypes of `Runabout` that will visit instances that existed long before the particular extension was made available. In other words, in contrast to aspect-oriented programming, the runabout fully supports modular compilation.

Various techniques to extend traditional programming languages in order to allow programmers to write more extensible, reusable code have been proposed in the past [14,15,16,28,33,34,45]. These implementations are often incompatible with each other and require specialized compilers. While these techniques are more general than the Runabout, the Runabout is a more lightweight solution for Java. Other examples for lightweight extensions of the Java language that also use dynamic code generation and reflection are the dynamic generation of helper classes for structural conformance, automatic delegation, and mixins [6].

6.4 Related Patterns

While the Runabout and other multi-dispatch related research has focused on the dispatch element in visitors, other researchers [5,35,44] have made suggestions on how to specify the visit strategy — that is, the order in which objects are visited. In the same way that the Runabout allows for a concise and dynamic specification of how to find the target of the dispatch, the Demeter Java project has focused on designing specifications for the visit order, allowing programmers to specify the order in which objects in a graph should be visited [35]. A guiding visitor [5] can be used to specify the order of the traversal, allowing programmers to make the actual code independent from the specification of the traversal order. An entire language for composing visitors and specifying traversal orders is described in [44]. Note that both research areas (dispatch strategy and visit strategy) are orthogonal and thus most solutions can be easily composed. The Runabout satisfies the single-responsibility principle [31] by being only concerned with dispatch and leaving questions about the visit order to other classes.

A pattern that also addresses the extensibility problem is the extension objects pattern [19]. This pattern is useful if different subsets of the subtypes of

a particular class will need to be extended with unanticipated operations. The extension objects pattern prescribes that the parent class provides a generic operation to lookup an extension (for example, by name). Individual instances can then be modified to support individual extensions. The pattern is similar to the Runabout in that it makes it easier to add functionality to an existing object structure. The extension object pattern is also similar in terms of performance – an operation requires a hash table lookup and a cast. However, the cast performed by the Runabout is guaranteed to succeed, programming with extension object patterns requires disciplined matching of keys and values in the extension map from the programmer. With the Runabout, the key for the lookup is the type of the object structure, the operation is already known. In contrast, with the extension object pattern, the key is the operation (name of the extension), the specific object is known since it is used to perform the lookup. Runabouts (and visitors) are better at centralizing state associated with the operation (in the visiting instance), whereas the extension objects pattern is better at preserving encapsulation within the visitees.

7 Conclusion

The Runabout is a viable alternative solution to the extensibility problem. Unlike other designs, the Runabout does not require extensions to the Java language. The Runabout can exhibit speeds comparable to other solutions, including MultiJava, visitors, and dedicated methods. Experience in implementing projects using the Runabout shows that it helps significantly in decoupling client code from the framework by allowing the applications to use visitor-like dispatching on objects provided by the framework without the need for changes to the framework itself. In cases where a loss of performance can be attributed to the Runabout, a static program transformation can be used to generate the equivalent visitor code.

Acknowledgments

My thanks go to Krista Bennett for editing and Jens Palsberg for advice.

References

1. J2ME building blocks for mobile devices. Technical report, Sun Microsystems, May 2000.
2. Bowen Alpern, Anthony Cocchi, Stephen J. Fink, David Grove, and Derek Lieber. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *OOPSLA 2001 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tampa, Florida*, pages 108–124, 2001.
3. Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Double Dispatch in C++. *Software - Practice and Experience*, 36(6):581 – 613, 2006.

4. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA Proceedings*. ACM Press, Vancouver, BC, October 1998.
5. M. Bravenboer and E. Visser. Guiding visitors: Separating navigation from computation. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2001.
6. T.M. Breuel. Implementing dynamic language features in java using dynamic code generation. In *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*, page 143, Washington, DC, USA, 2001.
7. Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.
8. Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 238–255, Denver, CO, November 1999. ACM.
9. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.
10. Antonio Cunei and Jan Vitek. Polyd: a flexible dispatching framework. In Ralph Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 487–503. ACM, 2005.
11. Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *IEEE Symposium on Foundations of Computer Science*, pages 524–531, Dortmund University, West Germany, 1988.
12. Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on pipelined processors. *Lecture Notes in Computer Science*, 952:253–283, 1995.
13. C. Dutchyn. Multi-dispatch in the Java Virtual Machine: Design and implementation. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 2001.
14. Matthew Flatt. Programming Languages for Reusable Software Components. Technical Report TR99-345, 20, 1999.
15. Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. 33(5):236–248, 1998.
16. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.
17. Rémi Forax, Étienne Duris, and Gilles Roussel. Java multi-method framework. In *TOOLS (37)*, pages 45–56. IEEE Computer Society, 2000.
18. Rémi Forax, Etienne Duris, and Gilles Roussel. Reflection-based implementation of java extensions: the double-dispatch use-case. *Journal of Object Technology*, 4(10):49–69, December 2005.
19. Erich Gamma. *Extension object*, pages 79–88. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
20. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

21. Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.
22. GNU Project. http://www.classpath.org/. 2004.
23. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.
24. Christian Grothoff. Jamit. http://ovmj.org/jamit/, 2003.
25. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating Objects with Confined Types. In *OOPSLA 2001 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tampa, Florida*, pages 241–253, 2001.
26. UCLA Compilers Group. The java tree builder. http://compilers.cs.ucla.edu/jtb/.
27. Martin E. Nordberg III. Default and extrinsic visitor. In *Pattern languages of program design 3*, pages 105–123. Addison-Wesley, 1997.
28. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
29. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
30. Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing Object-Oriented and Functional Design to Promote Re-use. *Lecture Notes in Computer Science*, 1445:91–113, 1998.
31. Robert C. Martin. *Agile Software Development, Principles, Patterns and Practices*. Prentice Hall, 2003.
32. M. Naik and R. Kumar. Efficient message dispatch in object-oriented systems. volume 35(3) of *ACM SIGPLAN Notices*, pages 49–58. ACM, March 2000.
33. Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Compiler Construction 2003, LNCS 2622, Warsaw, Poland*, pages 138–152, 2003.
34. M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.
35. Johan Ovlinger and Mitchell Wand. A Language for Specifying Traversals of Object Structures. Technical report, College of Computer Science, Northeastern University, Boston, MA, November 1998.
36. OVM Consortium. http://ovmj.org/. 2004.
37. Krzysztof Palacz, Jason Baker, Chapman Flack, Christian Grothoff, Hiroshi Yamauchi, and Jan Vitek. Engineering a Customizable Intermediate Representation. In *ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME 2003)*, San Diego, California, 2003. ACM SIGPLAN.
38. Krzysztof Palacz and Jan Vitek. Java subtype tests in real-time. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 378–404. Springer, 2003.
39. Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC*, pages 9–15. IEEE Computer Society, 1998.
40. Vidjut Samanta and Christian Grothoff. Hitsuji. http://ovmj.org/hitsuji/, 2004.
41. Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

42. B. Stroustrup. Multiple Inheritance for C++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987.
43. Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. *ACM SIGPLAN Notices*, 34(10):292–305, 1999.
44. Joost Visser. Visitor combination and traversal control. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 270–282, New York, NY, USA, 2001. ACM Press.
45. Matthias Zenger and Martin Odersky. Implementing Extensible Compilers. In *Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, Hungary, June 2001.
46. Yoav Zibin and Joseph (Yossi) Gil. Fast Algorithm for Creating Space Efficient Dispatching Tables with Application to Multi-Dispatching. In *OOPSLA 2002 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Seattle, Washington*, pages 142–160, 2002.

A Micro-benchmarks: one to 200 visit methods

In this appendix, the results for the micro-benchmarks that were shown for one to twenty visit methods in Figures 18, 19, 20 and 21 are repeated, this time for one to 200 visit methods.

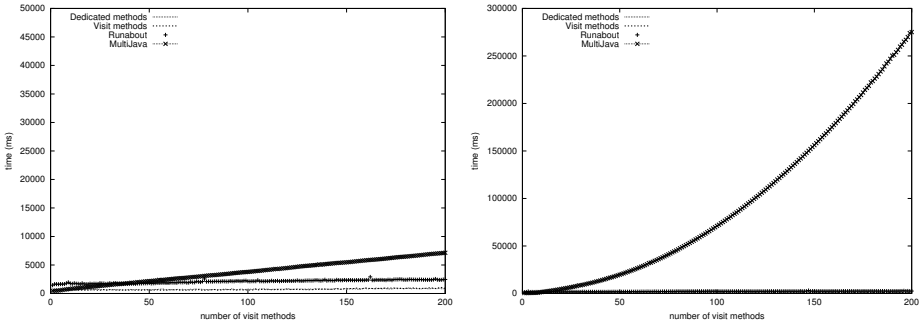

Fig. 25. Sun JDK 1.4.1 with flat (left) and deep (right) hierarchy.

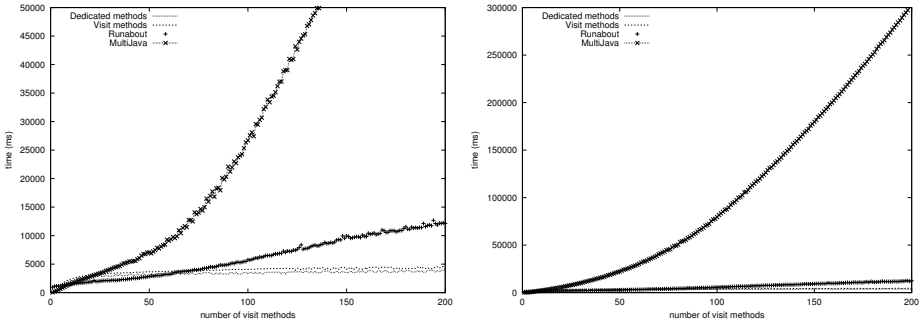

Fig. 26. IBM JDK 1.4.1 with flat (left) and deep (right) hierarchy.

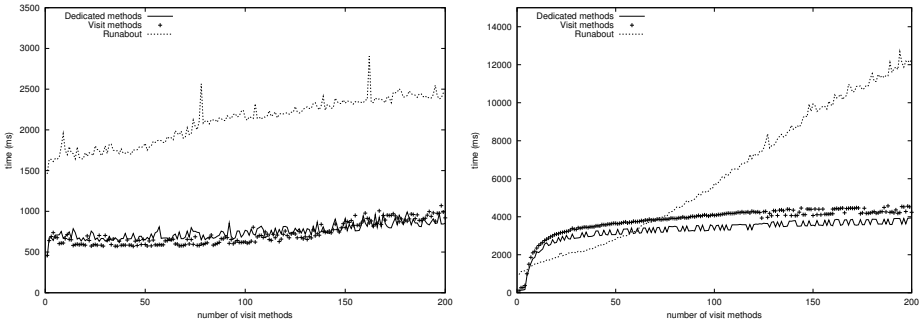

Fig. 27. SUN (left) and IBM (right), flat hierarchy without MultiJava.

B Micro-benchmarks with Java 5

This appendix gives results for the same benchmarks that were used in appendix A but uses the Java 5.0 versions from Sun and IBM as well as GNU GCJ. Furthermore, the benchmarks were run on a 64-bit AMD Sempron 2600+ at 1608 MHz. The benchmark suite was extended to include PolyD (20050403-0031) [10] a multi-dispatching variant of the Runabout. JMMF [17] was also measured; the numbers are not shown since the performance was generally more than a factor of ten worse than any of the other approaches.

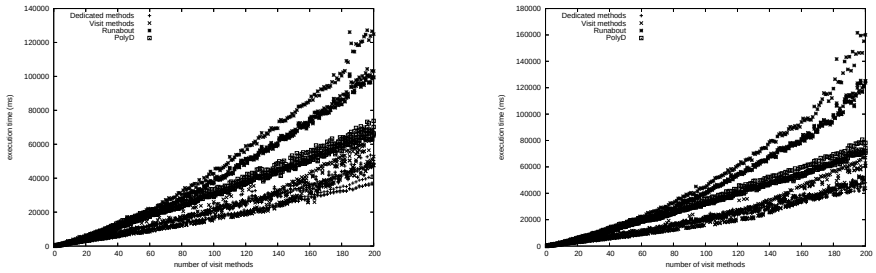

Fig. 28. Sun JDK 1.5.0 with flat (left) and deep (right) hierarchy.

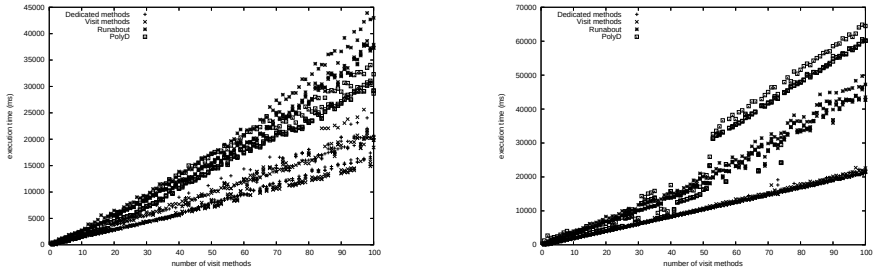

Fig. 29. IBM JDK 1.5.0 with flat (left) and deep (right) hierarchy.

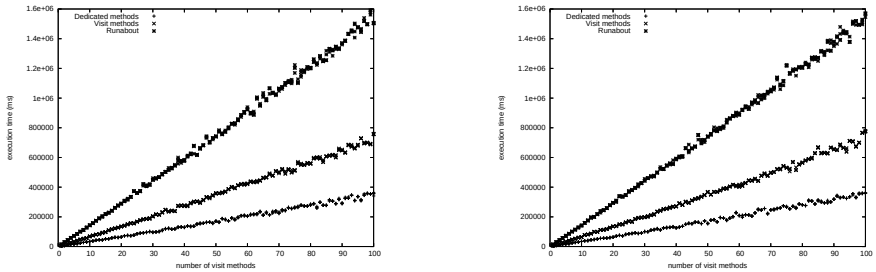

Fig. 30. GNU GCJ with flat (left) and deep (right) hierarchy.