Walkabout revisited: The Runabout

Christian Grothoff

S^3 lab, Department of Computer Sciences, Purdue University
grothoff@cs.purdue.edu
http://www.ovmj.org/runabout/

Abstract. We present 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.
In this paper we illustrate how to use the Runabout, detail its implementation and provide benchmarks comparing its performance with other approaches.

1 Introduction

A fundamental problem in programming language design is to make software extensible while avoiding changes to existing code and retaining static type safety [19]. For example, we may want to add functionality that operates on a number of existing objects, or we may want to introduce a new object to existing code. For such purposes, a strength of object-oriented programming is that it is easy to introduce a new class. Adding functionality to existing classes is more difficult, particularly because this typically requires access to the source code. It also may be undesireable to add the functionality to all subclasses.

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

One traditional solution to this problem is to use the visitor pattern [14]. The visitor pattern allows adding functionality in the form of `visit` methods that are invoked from an `accept` method which is defined in each visitee object. 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 towards achieving many of the benefits of multi-methods without requireing a new language.

In this paper we address 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. Our solution is based on an approach that was proposed by Palsberg and Jay [26] called Walkabout. Their approach takes advantage of Java's reflection mechanism to implement double-dispatch.

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 available on our webpage. Like the Walkabout [26], 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 verifying bytecode that will invoke the appropriate visit method. The dynamically generated bytecode is type-safe and can be analyzed and optimized by the compiler.

Generating bytecode for multi-dispatching is also what the MultiJava compiler [7] 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 beliefs [26], the approach using reflection to determine visit targets does not automatically imply an extraordinary run-time overhead. In fact, 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 presented. Performance evaluations are detailed in section 4. Section 5 discusses some related work.

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 example that implements the same functionality using dedicated methods, visitors, MultiJava and the Runabout is first presented. 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

For our example, we are going to use a set of visitee classes $\mathtt{A}_i$ 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 $\mathtt{A}_i$.

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 makes it often hard to maintain. Fig. 1 shows the solution using a dedicated method.

```
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. 1. The visitee classes with a dedicated method (`dedicated`).

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. 2).

Visitors Fig. 3 details the code for expressing a solution with visitors. The example uses overloading for the `visit` methods. Overloading is not needed for visitors and it is used here to emphasize the similarities with MultiJava and the Runabout. For simplification, we assume here that only one visitor is being used and that thus there is no need for a visitor interface for the `accept` methods to dispatch upon. In practice, the code would consist of multiple visitors for multiple computations that would be performed over the visitee objects.

```
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. 2. No changes to the visitees are required with cascading conditionals.

```
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. 3. Visitors require `accept` methods in the visitees.

Multi-methods An implementation using MultiJava (Fig. 4) 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.

```
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. 4. MultiJava indicates multi-dispatch using minimal changes to the syntax.

Runabouts The Runabout code (Fig. 5) is 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 in MultiJava, no `accept` method is required in the visitees.

```
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. 5. Runabouts extend the Runabout class to inherit `visitAppropriate`.

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.

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 a run-time exception to indicate that no `visit` method was found. `lookup` may also throw run-time exceptions (for example, 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 as a help for 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, like for `String`. A simple example for this is given in Fig. 6.

```
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. 6. 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 [15, 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.

Client code can define a specific lookup strategy by overriding the `lookup` function[1]. 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. 7. 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. 7. Example of a `lookup` method.

The Runabout has the following default lookup strategy. 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 run-time exception indicating the ambiguity is thrown. If no applicable `visit` method exists at all, `null` is returned, causing the invocation of `visitDefault`.

[1] Functions are methods that do not access any state except for the arguments and that have no side-effects. We restrict the lookup strategy to a function since this restriction allows a static checker to verify that the lookup strategy always succeeds when run in a closed-world setting. This is not a significant restriction since a lookup strategy that depends on the state of the application is likely to have an unexpected behavior for the programmer and thus such a design should in fact be made impossible for other reasons.

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. Extending the language has the advantage that MultiJava is more expressive than any other solution. For example, it is possible to dispatch on more than one argument. MultiJava also does not have the requirement that the methods that are multi-dispatched are named `visit`, which also 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 and that in fact several multi-dispatch method families in the same class without any syntax to mark these methods would instead likely confuse programmers. MultiJava's approach of extending the Java syntax solves this problem but prevents users from deploying other language extensions like GJ [2] or AspectJ [18] 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. We expect that a better implementation of MultiJava will take care of this major performance issue.

A drawback 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 [17] where `visit(A a)` calls `this.visit((P)a);` must be used, forcing the user to write additional methods that merely indirect the control flow.

3 Implementation

In this section, we describe our implementation of `Runabout`. In particular, we describe how the constructor builds the *dynamic code map* and how the `visitAppropriate` method uses that map to invoke the appropriate `visit` method. We then discuss 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. 8), 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 class (using reflection) and creates a specialized object of type `Code` for every visit method that is found. The class for each instance of `Code` (Fig. 8) is generated on-the-fly and dynamically loaded into the VM using Java's class-loading mechanism. The generated code is illustrated best 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. 9. 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. 8. The Code class is an inner class of the Runabout that defines the interface for the dynamically generated and loaded code tunks.

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

Fig. 9. 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`.

3.2 Lookup

The implementation of `Runabout.visitAppropriate` is now simple (Fig. 10). `visitAppropriate` does a `get` on the dynamic code `map`, to find an object of 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 object `nocode` with an implementation of `visit` that just returns 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. 10. visitAppropriate finds the dynamically generated code for an object in the hash table `map`.

3.3 Caching generated code

Like [3], 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 a Runabout 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 Runabout types. 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 a 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. 11 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.

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

Another simple extension is adding methods to the code map that are not `visit` methods in the subclass of the 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. Note 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 Experimental Results

In this section we present experimental results. We first present micro-benchmarking results which demonstrate that the Runabout is comparable in performance with the other designs. We then describe our experience with refactoring the Kacheck/J to use the Runabout instead of visitors.

4.1 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.

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) and for a hierarchy of maximum depth (visitee class n extends $n-1$). 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.

As the benchmarks in Fig. 12 and 14 show, the differences between the approaches in execution speed are small. The Runabout has the highest cost for just one visit method; MultiJava degrades with higher numbers of visitee types. If the hierarchy is deep (Fig. 13 and 15), 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 can not be explained and is presumabably just an artefact of the optimzing 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 that deep, thus Fig. 12 is more realistic than Fig. 13.

The measurements above just reflect the time required for the invocation. But the Runabout is also a bit more costly to create compared to instances of ordinary visitors. Fig. 16 shows the cost of creating 100,000,000 instances of Runabout (with caching enabled) compared with the creation of 100,000,000

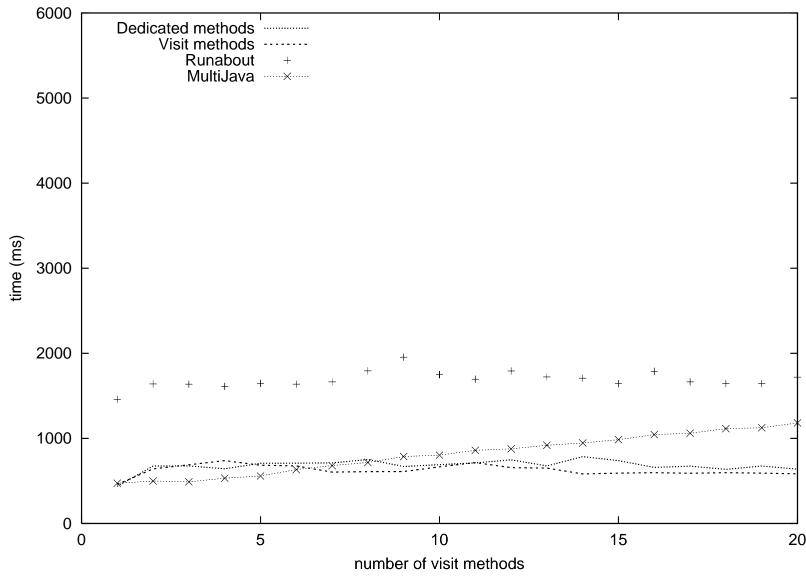

Fig. 12. Sun JDK 1.4.1, flat hierarchy.

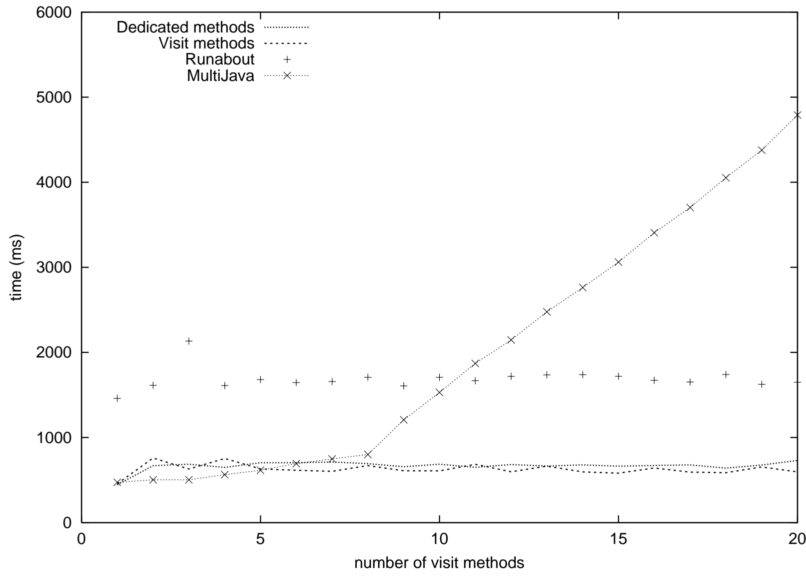

Fig. 13. Sun JDK 1.4.1, deep hierarchy.

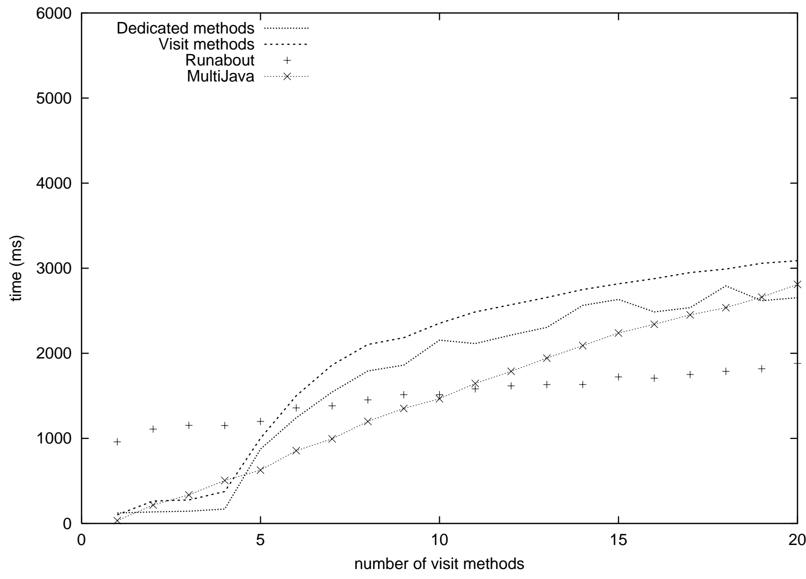

Fig. 14. IBM JDK 1.4.1, flat hierarchy.

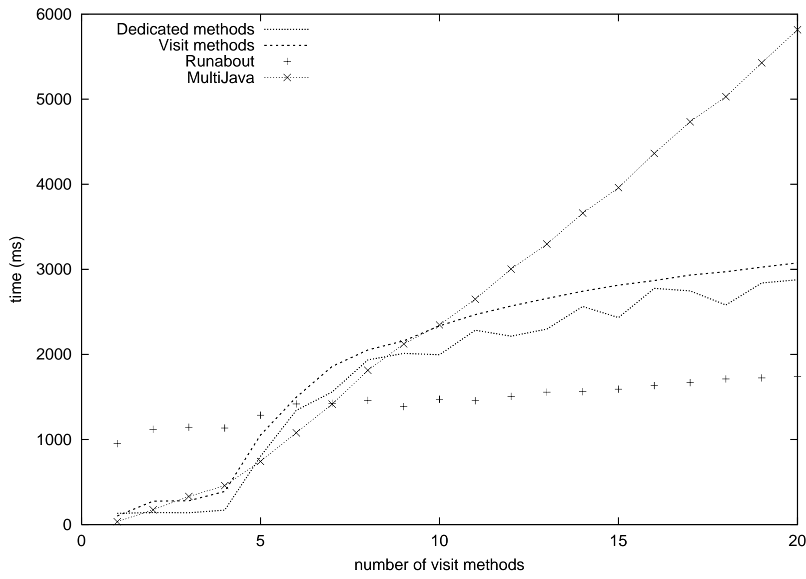

Fig. 15. IBM JDK 1.4.1, deep hierarchy.

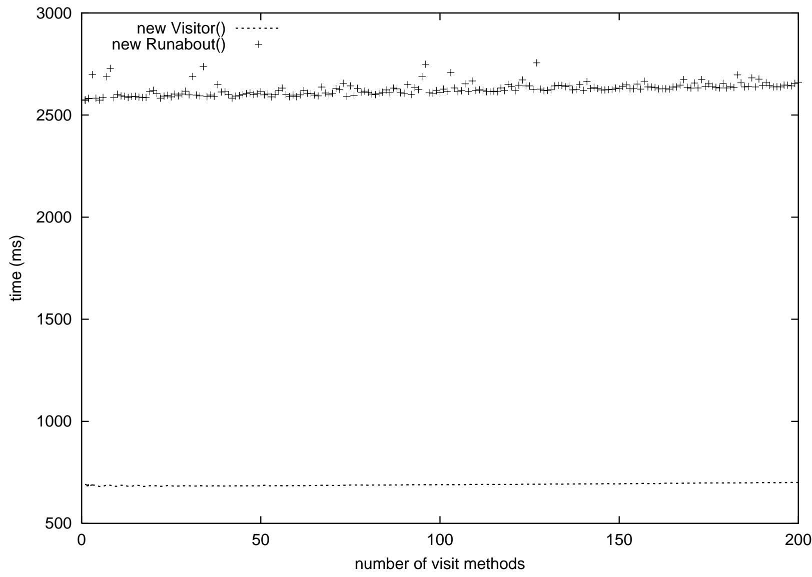

Fig. 16. This graph compares the time to create 10,000,000 instances of the same Runabout with the time to create 10,000,000 visitors on Sun JDK 1.4.1. The creation of a Runabout with the cache is about a factor of 3.8 slower.

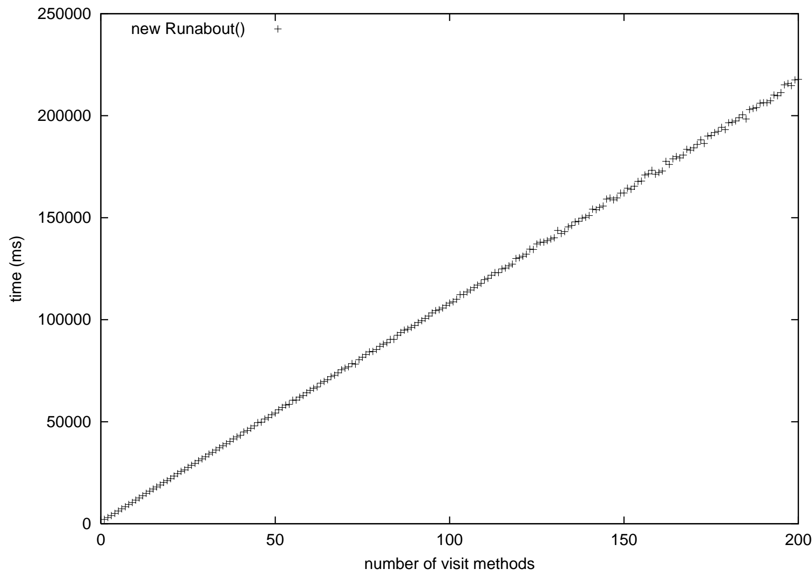

Fig. 17. This graph shows the time it takes to create 1,000 Runabouts *without* the cache on Sun JDK 1.4.1.

instances of an equivalent visitor. As the numbers show, creating a Runabout can be only about a factor of two slower than creating a visitor. But this is 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. 17 shows the cost of creating 100,000 Runabouts without the cache. The numbers show that caching the reflective information per-thread reduces the overhead of creating a Runabout 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.

4.2 Refactoring Kacheck/J

Kacheck/J is a bytecode analysis tool to infer confined types [16]. Kacheck/J is written using the Ovm bytecode framework [25]. Ovm [24] is a customizable Java Virtual Machine. The Ovm 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 that, every instruction had an `execute` method 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 interleaved with accepting on a visitor which would do the analysis. Some 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.

We have refactored the analysis framework 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 requires adding visit methods to parts of the framework that are not concerned with these types of instructions. Factoring out the `execute` method into a Runabout makes 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 saving 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 [17]. 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 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 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.[2] During the analysis, the innermost loop performs slightly more than 10 million invocations of `visitAppropriate`.

To evaluate the impact of the `visitAppropriate` calls on the overall performance of the application, additional calls to `visitAppropriate` were put into the inner loop. The additional calls invoke cheap but not entirely trivial `visit` methods that computes the sum of the opcodes of the instructions visited. The original inner loop contains three Runabout invocations, using additional invocations to these opcode-counting Runabouts the number of calls in the inner loop was increased to up to a total of 9 calls. Overall, the inner loop is run about 3.3 million times, resulting in 10 to 30 million runabout invocations for the profiling. For 30 million calls, the total runtime increases to about 95s. Fig. 18 shows the runtime of the abstract execution and the solving of the constraint system for three to nine Runabout invocations in the inner loop. The cost of parsing is nearly constant at 90s.

The cost of the introduction of the Runabout call in the innermost loop of Kacheck/J is thus about 12% of the time spent for abstract execution without constraint solving. While 12% might sound rather large, most steps in the abstract execution in practice consist of extremely cheap operations where even the traditional double-dispatch would take a fair share of the runtime. For example, for a `POP`, our implementation calls a visitor to do stack manipulation (which reduces the height of the stack by one) and a visitor to record Kacheck/J specific constraints (which only does something for 11 out of 200 Java opcodes), and finally calls a third visitor which most of the time just increments the program counter. That the three calls are taking a large share (and with the Runabout even 12%) of the runtime is thus more an effect of the way the code is written.

[2] 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.

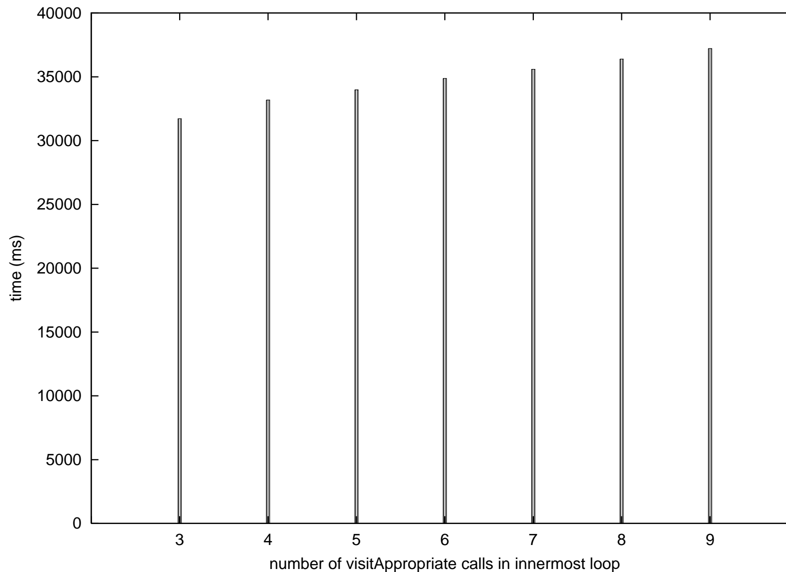

Fig. 18. Timings for the abstract execution phase of Kacheck/J with 0 to 6 additional Runabout invocations in the innermost loop.

For the overall application with parsing and constraint solving, the cost of the Runabout drops down to less than 3%. Considering that double dispatch and even dedicated methods also incur some cost, the actual cost of using the Runabout instead of traditional visitors is more around 2%.

5 Related Work

The walkabout pattern as described in [26] 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. Bravenboer [3] improved the performance of the walkabout by caching reflective results, but their performance is still poor; they report being about 100 times slower than visitors. The implementation given in [26] also requires that the type of the argument to the visit method matches exactly the type of the object that is being visited.

Space and time efficient implementations of virtual method dispatch have been the subject of a large body of research [6,9,20,27]. 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 [1] to handle features like multiple-inheritance and dynamic class loading. [29] provides a good overview of recent research. 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 [8], 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 Runabout classes 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 [10]. 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.

While the Runabout and other multi-dispatch related research has focused on the dispatch element in visitors, other researchers [3,23] 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 [23]. A guiding visitor [3] 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. Note that both research areas (dispatch strategy and visit strategy) are orthogonal and thus most solutions can be easily composed.

Various techniques to extend traditional programming languages in order to allow programmers to write more extensible, reusable code have been proposed in the past [11,12,13,18,21,22,28]. 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 extentions of the Java language that also use dynamic code generation and reflection are the dynamic generation of helper classes for structural conformance, automatic delegation or mixins [4].

6 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 perform about as fast as other solutions, including MultiJava, visitors or dedicated methods. While the current implementation of the Runabout only supports double-dispatch, future work may extend this approach to support full multi-dispatch.

Acknowledgements

My thanks go to the anonymous reviewers for helpful suggestions, Jan Vitek and Jens Palsberg for editing and to Tom VanDrunen, Suresh Jagannathan and James Noble for discussions.

References

1. 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.
2. 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.
3. M. Bravenboer and E. Visser. Guiding visitors: Separating navigation from computation. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2001.
4. 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*, pages 143–52, 2001.
5. 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.
6. 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.
7. 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.
8. 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, 1988.
9. Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on pipelined processors. *Lecture Notes in Computer Science*, 952:253–283, 1995.
10. 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.
11. Matthew Flatt. Programming Languages for Reusable Software Components. Technical Report TR99-345, 20, 1999.
12. Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.
13. 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.
14. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

15. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.
16. 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.
17. Martin E. Nordberg III. Variations of the Visitor Pattern. 1996.
18. 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.
19. 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.
20. 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.
21. 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.
22. 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.
23. 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.
24. OVM Consortium. http://www.ovmj.org/. 2002.
25. 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)*. ACM SIGPLAN, 2003.
26. Jens Palsberg and C. Barry Jay. The Essence of the Visitor Pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 1998.
27. B. Stroustrup. Multiple Inheritance for C++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987.
28. Matthias Zenger and Martin Odersky. Implementing Extensible Compilers. In *Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, Hungary, June 2001.
29. 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 12, 13, 14 and 15 are repeated, just this time for one to 200 visit methods. Note that a different y-scale is used for the flat and the deep hierarchy.

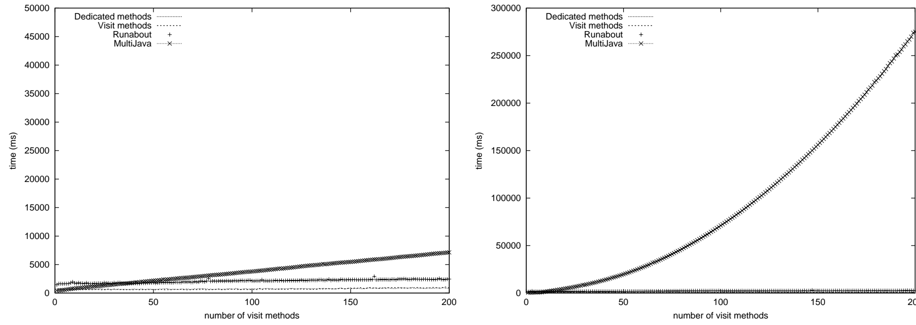

Fig. 19. Sun JDK 1.4.1 with flat (left) and deep (right) hierarchy.

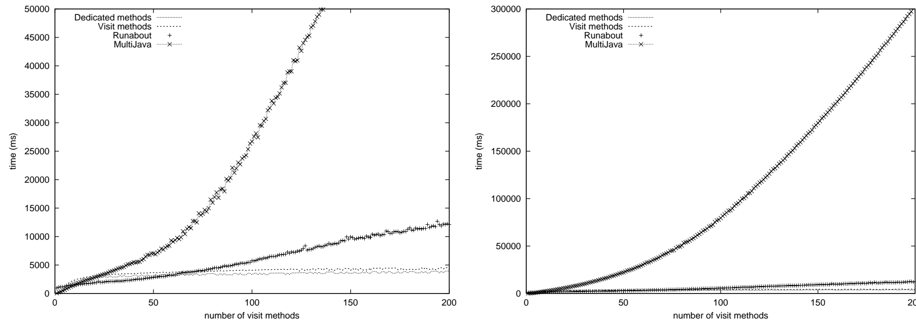

Fig. 20. IBM JDK 1.4.1 with flat (left) and deep (right) hierarchy.

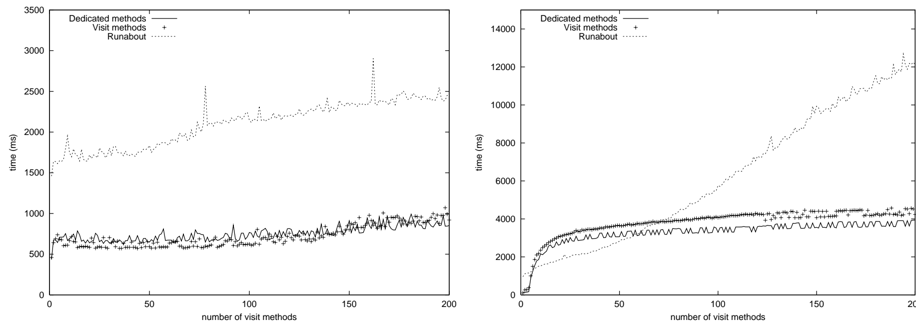

Fig. 21. SUN (left) and IBM (right), flat hierarchy without MultiJava.