Adding Tuples to Java: a Study in Lightweight Data Structures

C. van Reeuwijk

C.vanReeuwijk@cs.tudelft.nl

H.J. Sips

sips@cs.tudelft.nl

Delft University of Technology
Mekelweg 4
2628 CD Delft, The Netherlands

ABSTRACT

Java classes are very flexible, but this comes at a price. The main cost is that every class instance must be dynamically allocated. Their access by reference introduces pointer dereferences and complicates program analysis. These costs are particularly burdensome for small, ubiquitous data structures such as coordinates and state vectors. For such data structures a *lightweight* representation is desirable, allowing such data to be handled directly, similar to primitive types. A number of proposals introduce restricted or mutated variants of standard Java classes that could serve as lightweight representation, but the impact of these proposals has never been studied.

Since we have implemented a Java compiler with lightweight data structures we are in a good position to do this evaluation. Our lightweight data structures are *tuples*. As we will show, their use can result in significant performance gains: for a number of existing benchmark programs using tuples we gain more than 50% in performance relative to our own compiler, and more than 20% relative to Sun's Hotspot 1.4 compiler. We expect similar performance gains for other implementations of lightweight data structures.

With respect to the expressiveness of Java, lightweight variants of standard Java classes have little impact. In contrast, tuples add a different language construct that, as we will show, can lead to substantially more concise program code.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Data types and structures

General Terms

Languages, Design, Performance

Keywords

tuple, Java, lightweight data structures

1. INTRODUCTION

Java classes are flexible enough to represent arbitrary data, but this flexibility comes at a price. The main cost is that every class instance must be dynamically allocated, incurring allocation and garbage-collection overhead. Moreover, the fact that classes are accessed by reference introduces extra pointer dereferences and aliases, which complicates program analysis, in particular because Java supports threads. Finally, Java classes incur memory overhead: at a minimum, a pointer to the class instance and some memory for allocation administration are required. For class instances with a single reference this overhead is at least 8 bytes (4 bytes for the pointer, and 4 bytes for the memory block header). This means, for example, that for an array of two-dimensional `float` coordinates the overhead *doubles* the required memory.

All these costs are especially burdensome for simple, ubiquitous data structures such as coordinates and state vectors. For such data structures a lightweight representation is much more desirable, allowing direct manipulation of the data instead of through references, similar to the manipulation of primitive types.

In recognition of this problem, one of the designers of Java, James Gosling, has proposed [5] a new class modifier, `immutable`, which modifies the class as follows: the class and all instance variables are final, `a==b` maps to `a.equals(b)`, such classes must be initialized, and they cannot be compared to `null` or assigned `null`. These restrictions ensure that the programmer cannot distinguish between a class instance and a clone of it, allowing a compiler to manipulate class instances directly, similar to primitive types. However, as far as we know the impact of such a modification on performance and expressiveness has never been evaluated.

In our superset of Java, called Spar/Java, we provide support for *tuples*, which are also very suitable for representing lightweight data. Tuples allow groups of values to be aggregated into a single compound value, and decomposed into separate values, without requiring explicit class declarations. Tuple instances can be manipulated as a single value, but most operations (assignment, parameter passing, etc.) can be directly translated to a series of corresponding operations on the fields of the tuple. This allows a very efficient implementation of all tuple operations.

In this paper we evaluate the impact of tuples on performance and expressiveness of Java. We do this using three programs from the Java Grande benchmark suite [3, 4, 7]. The impact on expressiveness is specific to tuples. With respect to performance, we expect that the results accurately predict those of other approaches to lightweight data structures.

In Section 2 we give an informal description of tuples and their uses. In Section 3 the exact extensions to the Java Language Specification are described. In Section 4 we describe our static Java compiler. In Section 6 we describe how we rewrote the Java Grande benchmarks to use tuples, and the impact this had on the readability of the program. In Section 7 we evaluate the impact of using tuples on the performance of the benchmark programs. In Section 8 we discuss the merits of tuples and immutable classes as an addition to Java. In Section 5 we describe a method to translate programs containing tuples to standard JVM bytecode. In Section 9 we describe related work, and in Section 10 we draw some conclusions.

2. INFORMAL DESCRIPTION

A tuple can be seen as 'bundle' of values that can be treated as a single entity. A tuple is constructed with an expression such as [45,'a']. The type of this tuple is [int,char]. Individual components of a tuple can be accessed through subscripts. Thus, after x = [45,'a'], the expression x[0] is 45, and x[1] is 'a'. It is also possible to assign to the individual components.

Elements can also be accessed by *pattern matching*. For example, assuming a tuple x as above, its elements can also be accessed as follows:

```
int n;
char c;
[n,c] = x;
```

Tuples are manipulated by value, similar to Java's primitive types. Thus, after execution of the following code:

```
[int,char] x = [45,'a'];
[int,char] y = x;
y[0] = 42;
```

variables x and y have different values.

When all the elements of the tuple have the same type, it is called a *vector* tuple. The type of a vector tuple can be abbreviated. For example, the type of [1,2] can be written as [int^2].

Java's standard unary and binary operators have been generalized to work on tuples. Unary operators apply to each element, binary operators on two tuples apply to corresponding elements, binary operators on a scalar and a tuple apply the scalar value to each element. Thus, -[3,4] evaluates to [-3,-4], 2*[1,2] evaluates to [2,4], and [1,1]+[-1,1] evaluates to [0,2]. This allows many operations on vectors to be expressed clearly and concisely.

Tuples can be used in a number of idioms that are otherwise much more awkward to express. For example, the following statement exchanges the values of a and b:

```
[a,b] = [b,a];
```

This works because first the right-hand side of the statement is evaluated, resulting in a tuple containing the original values of b and a. This tuple is then decomposed into a and b again.

Tuples also allow functions with multiple return values, such as a function that searches a list, and returns both a success flag, and the position of the match:

```
static [boolean,int] search(int a[], int val)
{
  for( int i=0; i<a.length; i++ )
    if( a[i] == val ) return [true,i];
  return [false,0];
}
```

This function would be used as follows:

```
int ix;
boolean found;
[found,ix] = search( a, 42 );
```

Finally, tuples can be used as array subscripts. Our extended Java dialect, Spar/Java, also contains support for multi-dimensional arrays, and tuples can be a very convenient accessing an array. For example:

```
int a[*,*] = new int[9,9];
[int^2] v = [1,1];
a@v = 12;
```

causes 12 to be assigned to array element a[1,1]. For more information on this aspect of our language extensions see [13].

3. FORMAL DEFINITION

In this section we define the syntax and semantics of tuples as a set of extensions to the second edition of the Java Language Specification [6], called JLS2 from now on. All non-terminals not defined in this paper refer to non-terminals in JLS2.

A tuple is a list of elements. The list is of fixed size, and each element can be of any type. Tuples can be constructed by surrounding a list of expressions with square brackets. Such an expression is called an *explicit tuple*. Explicit tuples have the following syntax:

> *Expression*:
>   [*Expression$_{list}$*]

The type of a tuple has the following syntax:

> *type*:
>   [*VerboseType$_{list}$*]
>
> *VerboseType*:
>   *PrimitiveType*
>   *TupleType*
>   type *Type*

The types in a tuple specification must be preceded by the keyword type to distinguish them from variable names. In cases where no ambiguity is possible (primitive types and tuples), this keyword can be left out. For example, the following is a valid declaration and initialization of a variable of a tuple type:

```
[type int, type int, type Object] a = [1,1,null];
```

Since for primitive types the type keyword can be omitted, the following is equivalent:

```
[int, int, type Object] a = [1,1,null];
```

Since a tuple usually contains elements of primitive types, this allows for a compact notation.

Array access expressions (JLS2 §15.13) have been generalized to also access elements of a tuple. For tuple access, the indexing expression must evaluate to a compile-time constant. This restriction is necessary to ensure that the type of a tuple access expression is always known at compile-time[1]. Attempts to access elements beyond the tuple cause a compile-time error.

Similar to arrays, tuples have a field length that represents the length (the number of elements) of the tuple. For tuples this expression is a compile-time constant.

A *vector tuple* is a tuple where all elements are of the same type. For the type of such a vector tuple there is a special notation:

> *type*:
>   [*VerboseType* ^ *expression*]

---

[1]This restriction could be lifted for vector tuples, but that would necessitate a run-time bounds check which we consider too costly for this lightweight data structure.

An explicit tuple can be used at the left-hand side of an assignment.

LeftHandSide:
    [*LeftHandSide$_{list}$*]

Such an expression can only be assigned a tuple of the same length. The left-hand side tuple should only contain assignable expressions. They are assigned the values of the element at the same place in the right-hand side tuple expression. These assignments should follow the usual rules for assignment.

Furthermore, the addition of tuples modifies the language specification as follows:

Keywords. Next to the keywords listed in JLS2 §3.9, there is a new keyword `type`.

Initial values. As described in JLS2 §4.4.5, certain kinds of variables, such as class variables, are initialized with a *default value*. For tuple variables this is the tuple consisting of the default values of the fields of the tuple.

Conversions. Next to the conversions described in JLS2 §5.1, there is also *tuple widening conversion*, *tuple narrowing conversion* and *tuple identity conversion*. All conversions are only valid on tuples of the same length, and convert elements at the same position in the two tuples. There is an identity conversion between tuples if there is an identity conversion between the elements of the tuple. There is a widening conversion if there is a identity or widening conversion between all elements, and at least one element has a widening conversion. There is a narrowing conversion if there is a narrowing, widening or identity conversion between all elements, and at least one element has a narrowing conversion.

Evaluation order. Similar to argument lists of method or constructor invocations (JLS2 §15.7.4), explicit tuples are evaluated left-to-right.

Class Literals. Currently, tuples do not have class literals (JLS2 §15.8.2). Should this prove to be a problem, they can be added easily. At that point it will also necessary to decide what the type descriptor of a tuple (i.e. the string returned by the `getName()` method of its class literal) will be.

Operators. All binary operators (JLS2 §15.17 and further), except the `&&` and `||` operators, also work on tuples. If both operands are tuples, they must be of the same length, and the operator is applied to all elements at the same position in the two tuples. If one of the operands is a scalar, it is evaluated once, and the result is applied to all elements in the tuple.

All unary operators except `++` and `--` also work on tuples (JLS2 §15.15). The operator is applied to all elements in the tuple.

String conversion. The generalization of binary operators to tuples as described above also applies to string concatenation (JLS2 §15.18.1). As a consequence, an expression such as `"tuple: "+[1,2]` evaluates to `["tuple: 1","tuple: 2"]`.

Assignment operators. All assignment operators (JLS2 §15.26), including compound assignment operators, can be applied to tuples. For compound assignment operators the left-hand operand must be a *primitive tuple* (defined below), and the right-hand operand must be of primitive type or must be a primitive tuple.

A primitive tuple consists entirely of elements that are either of primitive type, or primitive tuples.

For the `+=` operator the allowed types are generalized in the obvious way to also allow string concatenation.

Constants. Next to the ones listed in JLS2 §15.28, there are four additional compile-time constant expressions:

- An empty tuple is a compile-time constant.

- A tuple containing only compile-time constant expressions, is a compile-time constant.

- An index expression applied to a compile-time constant tuple expression is a compile-time constant. (Remember that the index expression of a tuple subscript expression must be a compile-time constant.)

- For any tuple `t`, the expression `t.length` is a compile-time constant.

4. IMPLEMENTATION

We have implemented tuples in our compiler, Timber. The compiler supports a superset of Java, called Spar/Java. The extensions not only comprise tuples, but also complex numbers, multidimensional arrays, and generic types. The set of extensions was chosen to substantially improve the support of Java for scientific computations. For further information see [13]. The Timber compiler is available for downloading at [12].

Timber is a static compiler that translates the source code of an entire program to a C++ program. The generated code has been tested with the Gnu C++ compiler (versions 2.95 through 3.1), and with Intel's C++ compiler for Linux (version 6.0). Porting to other compilers should be simple.

The compiler implements full Java except for dynamic class loading and threads.

Garbage collection is implemented as a simple mark/sweep allocator on top of the standard `malloc` library. The Timber compiler generates explicit code to maintain a strict root set. This approach makes the garbage collector portable and simple to implement.

The compiler contains optimizations that avoid compiling unused classes and methods, eliminate and optimize array bound and null pointer checks, eliminate static class initialization administration, explicitly free allocated memory, and optimize garbage collection administration code.

It is important to remember that since the Timber compiler has been developed by a very small team, the compiler has some restrictions. In particular, the compiler only does simple program analysis and optimization, it generates C++ code whereas other Java compilers usually generate machine code. Although we think that the compiler gives a good indication of what a Java compiler with tuples can achieve, the reader should keep these limitations in mind.

Tuples are represented by C++ classes. C++ allows classes to be passed by value, returned from a function and directly stored in arrays. Therefore, no special effort is necessary to generate correct C++ code for tuples.

5. TUPLES AND THE JVM

Since our compiler does not use the Java Virtual Machine (JVM), we did not have the problem of constructing a suitable representation of tuples for the JVM. For traditional Java implementations this problem must be addressed, however.

For a large part the problem can be solved by 'unbundling' tuples into scalar components. This allows tuples to be used within methods, and allows them to be passed as parameters. By defining standard expansion rules binary compatibility is also ensured.

This leaves two problems: how to return a tuple from a function, and how to represent arrays of tuples. Philippsen and Günthner [11] have already described a method to solve a subset of this problem: how to translate all uses of complex numbers to traditional Java, and hence to bytecode. In their approach complex numbers are 'unbundled' into the real and imaginary part in a similar way to what we propose. They handle the problematic cases using wrapper classes or arrays.

The same approach could be used to represent tuples. It would be tempting to represent tuples using classes, since the mapping is natural and fairly efficient. However, for each length of tuple and for each combination of types, a separate class would be necessary, resulting in a potentially infinite set of wrapper classes. Pragmatically, classes could be generated by the frontend compiler on demand, but the administrative problems would be great.

A better solution is to use `Object` arrays. That way, arbitrary tuples can be wrapped without requiring additional class definitions. A disadvantage is that elements of primitive types in the tuples must also be wrapped individually. For vector tuples the wrapper could be refined to be an array of the appropriate type.

Of course, a JVM implementation that is aware of tuples could undo all the wrapping, and generate more efficient code. Similar 'extension-aware' JVM implementations have been proposed for multi-dimensional arrays [10] and generic types [2].

6. EXPRESSIVENESS OF TUPLES

To evaluate the impact of using tuples, we have taken three benchmarks from Section 3 of the Java Grande forum benchmark suite [3, 4, 7], and rewrote them to use tuples. Two of these programs create large numbers of instances of small classes and are therefore good candidates for applying tuples. The third program represents its central data as scalar variables (e.g. `velocityx`, `velocityy` and `velocityz`), and is therefore a good program to compare an approach using tuples to one where groups of scalars are used.

In the remainder of this section we discuss for each benchmark the modifications that we made, and the impact of these modifications on the readability of the programs. The performance of the resulting programs is discussed in the next section.

The Euler benchmark

The Euler benchmark solves the time-dependent Euler equations for flow in a channel with an obstacle on one of the walls.

For the tuple version, two transformations were done: the class `Statevector`, representing the state of a cell, and the class `Vector2`, representing a 2-dimensional vector, were replaced by tuples. One method in the original program, `svect()`, calculates the difference between two state vectors, and returns the result as a new `Statevector`, causing a considerable amount of garbage. In the rewritten version, this method call is replaced by tuple subtraction, resulting in much more efficient and readable code.

There were other places where binary operators on tuples could be used effectively. For example, the following fragment from the original program (in the function `calculateR`, the 'hot spot' of the program):

```
r[i][j].a += temp*(f[i][j].a + f[i+1][j].a);
r[i][j].b += temp*(f[i][j].b + f[i+1][j].b);
r[i][j].c += temp*(f[i][j].c + f[i+1][j].c);
r[i][j].d += temp*(f[i][j].d + f[i+1][j].d);
```

updates the individual elements of the state vector of array element `r[i][j]` one by one.

When the state vector is represented as a tuple, this fragment is rewritten to:

```
r[i][j][0] += temp*(f[i][j][0] + f[i+1][j][0]);
r[i][j][1] += temp*(f[i][j][1] + f[i+1][j][1]);
r[i][j][2] += temp*(f[i][j][2] + f[i+1][j][2]);
r[i][j][3] += temp*(f[i][j][3] + f[i+1][j][3]);
```

Code like this is a prime candidate for using binary operators on tuples, as described in Section 3. Therefore, the code shown above can be rewritten to:

```
r[i][j] += temp*(f[i][j] + f[i+1][j]);
```

A similar transformation was done on 7 other sections of the same function. In total such a transformation was done on 22 sections of the program.

The RayTracer benchmark

The RayTracer benchmark does ray tracing on a scene of spheres.

When rewriting to use tuples, two transformations were done: The class `Vec`, representing a 3-dimensional vector, and the class `Isect`, representing the result of a ray/object intersection, were replaced by tuples.

The `Vec` class also contains vector operations. Many of these operations could be rewritten to use binary operators on tuples. For example, the original program contains the following method:

```
static Vec comb(double a, Vec A,
  double b, Vec B)
{
  return new Vec(
    a * A.x + b * B.x,
    a * A.y + b * B.y,
    a * A.z + b * B.z);
}
```

This is then used as follows (original code fragment):

```
Vec D = comb(xlen, Vleft, ylen, Vup);
D.add(viewVec);
D.normalize();
```

By using tuples the methods `comb()` and `add()` (not shown) are not necessary. Instead the code fragment above can be implemented as:

```
[double^3] D =
  Vec.normalize(xlen*Vleft + ylen*Vup + viewVec);
```

This results in more readable code, and a lot of memory allocation is also avoided.

Another important transformation is that the result of an intersection calculation is not returned as a `Isect` instance, but as a tuple. Since in the original program such a class instance is constructed for every ray/sphere intersection calculation, the reduction in the amount of memory allocation (and garbage) is significant.

The MolDyn benchmark

The MolDyn benchmark performs a simple N-body calculation.

In contrast to the other two benchmarks, the original program does not use classes, but groups of similarly named variables (e.g. `xvelocity`, `yvelocity`, and `zvelocity`) to represent vectors. In the rewritten program we replace such groups by tuples. We cannot expect a performance gain from this transformation; the introduction of tuples may in fact cause a *slowdown* of the program. This makes the program a useful addition to the benchmark set, since it indicates the overhead of using tuples.
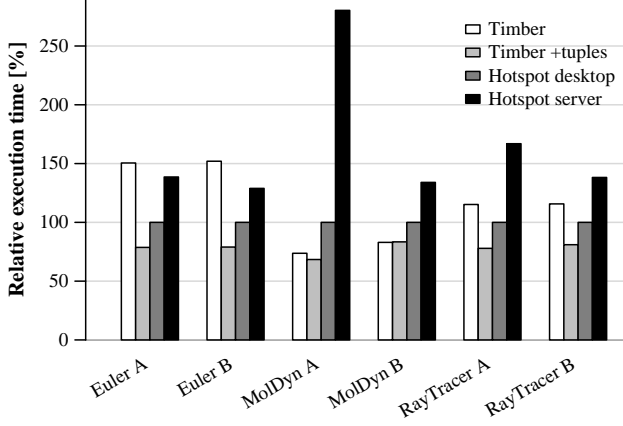
In rewriting the program we modified the central class of the program, called `particle`. Each class instance represents the state of an individual particle. We rewrote the class to represent the position, velocity, and force of a particle as 3-dimensional tuples instead of a set of scalar variables.

For example, in the original program the following fragment occurs, which updates the velocity of the particle of array element: `one[i]`:

```
one[i].xvelocity = one[i].xvelocity * sc;
one[i].yvelocity = one[i].yvelocity * sc;
one[i].zvelocity = one[i].zvelocity * sc;
```

Table 1: Results for the Intel system.

Benchmark	size	Timber		HotSpot	
		Java	+ tuples	desktop	server
Euler	A	41.7	21.9	27.7	38.4
	B	93.2	50.4	61.3	79.0
MolDyn	A	5.6	5.1	7.6	21.3
	B	260.1	254.9	313.5	419.9
RayTracer	A	16.7	11.3	14.5	24.2
	B	180.2	126.1	155.8	215.3

Table 2: Results for the AMD system.

Benchmark	size	Timber		HotSpot	
		Java	+ tuples	desktop	server
Euler	A	31.6	9.3	13.8	17.8
	B	69.6	20.7	29.4	32.1
MolDyn	A	3.4	4.0	7.8	8.4
	B	381.6	371.7	309.1	329.1
RayTracer	A	12.1	5.7	9.2	8.0
	B	131.1	63.0	99.4	83.0

Figure 1: Relative performance for the Intel system.

Figure 2: Relative performance for the AMD system.

In the rewritten program this fragment is replaced by an operation on the velocity vector of the particle:

```
one[i].velocity = one[i].velocity * sc;
```

In total, nine such transformations could be done.

7. PROGRAM PERFORMANCE USING TUPLES

To compare the performance of the original benchmark programs and the rewritten versions described in the previous section, we measured their execution times on two systems: a system with a 1GHz Intel Pentium III and a system with a 1.5GHz AMD Athlon-MP. In both cases the system contained 512MB of memory, and the heap size for all runs was explicitly set to 400MB. Version 2.0.1 of the Timber compiler was used, and version 3.1 of the Gnu C++ compiler. All shown results are the sum of the user and system execution times in seconds of the programs, as measured with the Unix `time` command.
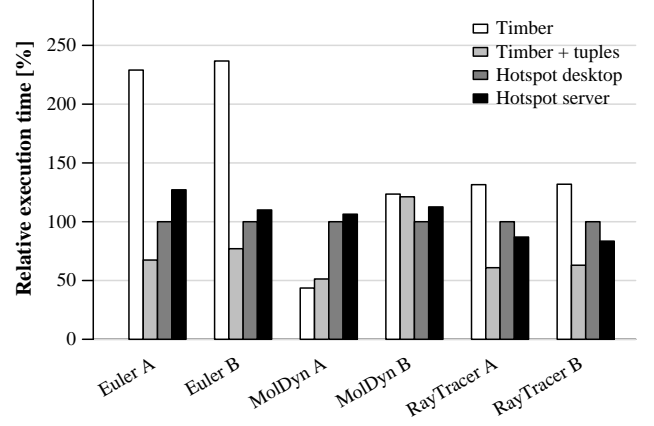
For comparison, the execution times of the Sun Hotspot 1.4 compiler were measured on the same systems. Both the 'desktop' version and the 'server' version (enabled with the `-server` flag) were measured. The server version is designed for use on long-running programs such as web servers. It spends more compilation time in an effort to generate more efficient program code.

Table 1 shows the resulting execution times for the Intel system, and Fig. 1 plots these results as percentages relative to the execution time of the Hotspot desktop version. Similarly, Table 2 and Fig. 2 show the results for the AMD system.

Comparing the two sets of Timber results, the results for the Mol-Dyn benchmark are nearly the same for the two versions, indicating that there is little overhead in using tuples instead of separate variables. From the results for the other two benchmarks we conclude that using tuples does indeed make a difference. Both benchmarks on both systems show an improvement in execution time of 40% to 50%. The Euler benchmarks on the AMD system improve even more, but in this case the results for the original benchmark are poor.

As we explained earlier, there are two major factors for the performance improvement: reduced dynamic memory use and better opportunities to generate efficient code. From the benchmark results it is difficult to distinguish between these effects. The results for the MolDyn benchmark suggest that the Timber compiler is able to generate very efficient code for the manipulation of sets of scalars, whether they are wrapped in tuples or not. To identify the sources of execution overhead of the various benchmarks, we have also measured the execution times with null pointer checking[2] and array bound checking disabled. As Table 3 shows, for the Euler benchmark bounds checking and null pointer checking is a significant overhead.

To study the influence of dynamic memory use, we have instrumented our memory system to print statistics; Table 4 shows the results for the benchmark programs. As expected, for the MolDyn benchmark the impact of using tuples on memory use is insignificant. However, for the other two benchmarks using tuples eliminates virtually all dynamic memory use. We are confident that this is a significant factor in the improved execution times of the Timber compiler, in particular because of Timber's simple memory allocator/garbage collector. We expect that the Hotspot compiler uses a more sophisticated memory allocator/garbage collector, but even

---

[2]These tests ensure that a `NullPointerException` is thrown when a null pointer is dereferenced.

Table 3: Results for JGF Section 3 benchmarks for the Intel system. To study the overhead of null pointer checks and array bound checks, four variants were measured: normal (b1n1), bounds checking disabled (b0n1), null pointer checking disabled (b1n0), and both disabled (b0n0).

Benchmark	size	Timber				Timber + tuples				HotSpot	
		b1n1	b0n1	b1n0	b0n0	b1n1	b0n1	b1n0	b0n0	desktop	server
Euler	A	41.7	32.1	36.2	31.0	21.9	17.6	19.6	16.6	27.7	38.4
	B	93.2	71.1	79.8	70.7	50.4	39.2	43.9	37.2	61.3	79.0
MolDyn	A	5.6	4.5	5.0	4.4	5.1	4.6	4.8	4.4	7.6	21.3
	B	260.1	249.7	250.6	243.9	254.9	254.0	262.4	253.2	313.5	419.9
RayTracer	A	16.7	16.6	15.6	15.7	11.3	11.0	10.8	10.5	14.5	24.2
	B	180.2	180.4	168.4	168.9	126.1	121.9	119.5	117.0	155.8	215.3

Table 4: Allocation and garbage collection statistics for the various benchmark programs. Shown are the total number of allocation requests, the total amount of allocated memory (bytes), and the number of garbage collections done during execution of the program.

Benchmark	size	tuples	allocs	memory	GCs
Euler	A	no	6,541,891	447,346,399	1
		yes	10,619	6,993,326	0
	B	no	14,798,596	1,012,047,251	2
		yes	13,562	14,968,593	0
MolDyn	A	no	2,744	268,759	0
		yes	2,743	317,717	0
	B	no	9,483	1,023,415	0
		yes	9,481	1,233,983	0
RayTracer	A	no	5,903,577	354,306,910	0
		yes	501	124,947	0
	B	no	65,730,443	3,944,828,784	9
		yes	503	1,035,256	0

there the huge difference in memory consumption should make a difference in execution time.

8. TUPLES VERSUS IMMUTABLE CLASSES

Since both tuples and immutable classes are intended to provide support for lightweight data structures, it is useful to compare the merits of these approaches.

Both immutable classes and tuples have only a minimal impact on the existing syntax. Both require one new keyword (`immutable` and `type` respectively). Immutable classes require no further syntax, and tuples use language constructs that are illegal in current Java.

The greatest strength of immutable classes is that they allow the construction of encapsulated lightweight objects with user-defined methods. In contrast, tuples are always open for inspection, and only a limited set of predefined operations is available.

The greatest strength of tuples is that they need not be declared, and that the tuple pattern matching and unary and binary operators on tuples allows for a very clear and compact notation of common tuple operations. In contrast, immutable classes and all operations on them must be declared explicitly.

Another advantage of tuples is that single elements can be modified easily. This is not so simple for immutable classes, since fields of an immutable class are `final`. This restriction is not necessary for performance reasons, but to remain compatible with the existing semantics of classes. In the face of this restriction, the simplest approach to modifying single fields of an immutable class is to provide special modification methods. Thus, for a 2-dimensional coordinate class we could define:

```
immutable class Coord2D {
```

```
    int x, y;
    Coord2D(int a, int b){ x = a; y = b; }
    Coord2D setX(int v){ return new Coord2(v,y); }
    Coord2D setY(int v){ return new Coord2(x,v); }
}
```

Now we can set single elements as follows:

```
Coord pos = new Coord(5,5);
pos = pos.setX(3);
pos = pos.setY(9);
```

This is quite cumbersome, and the compiler must eliminate the redundant operations on unchanged elements of the class.

In contrast, when using tuples one can simply write:

```
[int^2] pos = [5,5];
pos[0] = 3;
pos[1] = 9;
```

9. RELATED WORK

Instead of explicitly using lightweight classes in one form or the other, a compiler could also try to recognize the required circumstances, and label classes (or class instances) as lightweight. This is called *object inlining* or *unboxing*. Although some success has been reported on this [8, 9], the required analysis is complicated. Also, the transformation does not always result in more efficient code, so the compiler must consider the tradeoffs carefully. Alternatively, the decision can be left to the programmer by providing explicit lightweight classes.

As far as we know, the most faithful implementation of James Gosling's immutable classes is in Titanium [15]. However, there are no reports on the impact of Titanium's lightweight classes on performance or expressiveness.

David Bacon [1] describes a dialect of Java, called Kava, that provides extensive support for lightweight classes. Conceptually, even primitive types such as integers are implemented as lightweight classes. These `value` classes are similar to immutable classes, but the semantics of these classes explicitly state that they are manipulated by value, and they do not have some of the restrictions on immutable classes. Kava also adds operator overloading to implement binary and unary operators. Their paper describes a possible translation scheme: the use of standard primitive types is recognized by the frontend, and in this case standard JVM bytecodes are generated for efficiency. Other classes are translated to a potentially inefficient implementation in standard Java classes. A special JVM implementation must recognize the value classes, and generate efficient code for them. This is a complicated operation, which was not completed at the time of writing of their paper. Therefore the paper does not show any performance results.

The idea of adding tuples to a programming language is very old. The oldest occurrence is probably in Lisp or SETL, but the parameter and subscript lists occurring in almost any programming language can also be considered as tuples. There is also a close link with the notion of tuples in mathematics. The idea of treating strictly typed tuples as first-class citizens originated in functional languages. We were directly inspired by Miranda [14], but most functional languages have a similar concept. Support for tuples in imperative languages is less common, but programming languages such as Perl, Python and BETA do support them.

10. CONCLUSIONS

We have demonstrated that tuples can improve both the performance and the readability of Java programs that contain small ubiquitous data structures such as coordinates and state vectors.

By rewriting a number of existing benchmark programs to use tuples, we have been able to gain more than 50% in performance relative to our own compiler, and more than 20% relative to the Hotspot 1.4 compiler. There are two major factors for this performance improvement: reduced dynamic memory use, and better opportunities to generate efficient code. Although we have not been able to fully distinguish between these effects, memory allocation statistics show a huge reduction in memory use, indicating that this is an important factor. For this reason alone the use of tuples should also have an impact in other Java implementations. We expect that similar performance gains are possible for other implementations of lightweight data structures.

Using binary operations on tuples notably improved conciseness and readability in all benchmark programs.

Compared to immutable classes, tuples have the advantage that their declaration is also lightweight. It is not necessary to explicitly declare them, so they provide a simple notation for ad-hoc construction of data-structures.

Although we have not implemented this, we do not expect any major obstacles in generating Java bytecode for tuple manipulation. In most cases the tuples can simply be expanded into individual variables. For two bottlenecks (returning tuples from methods, and storing tuples in arrays), standard Java arrays can be used as wrappers. This incurs a considerable overhead, but a tuple-aware JVM could recognize these wrappers and generate efficient code for them.

11. REFERENCES

[1] David F. Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. In *ACM Java Grande – ISCOPE Conference*, pages 68–77, 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*, October 1998.

[3] J.M. Bull, L.A. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for scientific applications. In *ACM Java Grande – ISCOPE Conference*, pages 97–105, 2001.

[4] J.M. Bull, L.A. Smith, M.D. Westhead, D.S. Henty, and R.A. Davey. A benchmark suite for high performance Java. *Concurrency – Practice and Experience*, 12(6):375–388, May 2000.

[5] James Gosling. The evolution of numerical computing in Java. webpage.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Reading, Massachusetts, June 2000.

[7] Java Grande forum benchmarks website. www.epcc.ed.ac.uk/javagrande.

[8] Peeter Laud. Analysis for object inlining in Java. In *Proceedings of the Joses Workshop*, 2001.

[9] X. Leroy. The effectiveness of type-based unboxing. In *Proceedings of the Workshop "Types in Compilation"*, Amsterdam, June 1997.

[10] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.

[11] Michael Philippsen and Edwin Günthner. Complex numbers for Java. *Concurrency: Practice and Experience*, 12(6):477–491, May 2000.

[12] C. van Reeuwijk. Timber download site. www.pds.twi.tudelft.nl/timber/downloading.html.

[13] C. van Reeuwijk, F. Kuijlman, and H.J. Sips. Spar: an extension of Java for scientific computation. In *ACM Java Grande – ISCOPE Conference*, pages 58–67, June 2001.

[14] David Turner. Miranda: A non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Functional programming languages and computer architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[15] K. Yelick, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, Phil C., and A. Aiken. Titanium: a high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, pages 1–13, February 1998.