Immutability Specification and its Applications

Igor Pechtchanski and Vivek Sarkar

1 IBM T. J. Watson Research Center

SUMMARY

A location is said to be immutable if its value and the values of selected locations reachable from it are guaranteed to remain unchanged during a specified time interval. We introduce a framework for immutability specification, and discuss its application to code optimization. Compared to a final declaration, an immutability assertion in our framework can express a richer set of immutability properties along three dimensions — lifetime, reachability and context. We present a framework for processing and verifying immutability annotations in Java, as well as extending optimizations so as to exploit immutability information. Preliminary experimental results show that a significant number (61%) of read accesses could potentially be classified as immutable in our framework. Further, use of immutability information yields substantial reductions (33% to 99%) in the number of dynamic read accesses, and also measurable speedups in the range of 5% to 10% for certain benchmark programs.

KEY WORDS: Immutability, Annotations, Optimization

1. Introduction

An object or datum is said to be immutable if its value is guaranteed to remain unchanged. Immutable data values are encountered more frequently in functional and object-oriented applications with system support for memory management and garbage collection than in procedural applications where memory management is performed by the user. In object-oriented applications, the knowledge that an object is immutable can be very valuable in improving program documentation and understanding, reducing programming errors, and enabling performance optimizations including parallelization. Many programming languages allow the programmer to identify a variable as immutable by annotating its declaration with an attribute such as const or final. However, such declarations are limited in scope and unable to...
express many common occurrences of immutability in real programs. The goal of our research is to explore how a richer set of immutability declarations can be used to improve program documentation and optimization of programs written in object-oriented languages such as Java\textsuperscript{1}.

In this paper, we introduce a framework for \textit{immutability specification}, and discuss its application to \textit{code optimization}. An immutability specification consists of a set of \textit{immutability assertions}. Compared to a \textit{final} declaration, an immutability assertion can express a richer set of immutability properties along three dimensions — \textit{lifetime, reachability} and \textit{context}. In the lifetime dimension, an immutability assertion can specify the dynamic duration for which a datum is immutable as a set of temporal points in the program execution. In the reachability dimension, an immutability assertion can specify the set of locations declared to be immutable as a subset of locations reachable from a specific object reference. In the context dimension, an immutability assertion can identify subsets of dynamic types and allocation sites that restrict the set of object instances to which the assertion applies. The application of the framework to code optimization shows how standard optimizations for object-oriented programs can benefit from immutability assertions. We also present preliminary experimental results to support our approach. A study was performed to determine the maximum potential benefit that might be achievable by immutability optimizations. This \textit{limit study} of read and write accesses shows that a significant number (61\%) of read accesses could potentially be classified as immutable in our framework. Further, use of immutability information in automatic optimization yields substantial reductions (33\% to 99\%) in the number of dynamic read accesses, and also measurable speedups in the range of 5\% to 10\% for certain benchmark programs.

The rest of the paper is organized as follows. Section 2 discusses the use of immutability annotations in four examples, as motivation. Section 3 provides the details of our framework — immutability specification, immutability annotations, and immutability verification. Section 4 describes how immutability information can be used in code optimization. Section 5 contains our preliminary experimental results. Finally, Section 6 discusses related work, and Section 7 contains our conclusions.

2. Examples

In this section, we use four examples to introduce the richer set of immutability properties that can be expressed in our framework. The examples are written in the Java programming language, though the ideas would be equally applicable to other programming languages. The examples are not intended to represent meaningful computations; instead, they are designed to communicate the ideas behind the immutability properties as simply as possible.

As mentioned earlier, an immutability assertion in our framework can express a rich set of immutability properties along three important dimensions — \textit{lifetime, reachability} and \textit{context}.

\textsuperscript{1}Java is a registered trademark of Sun Microsystems.
Let us first understand how restricted these dimensions are for a final declaration in Java. Recall that a final declaration for a static or instance variable in Java indicates that the variable may only be written in its initializer or constructor. This means that the immutability lifetime for a final variable always starts at the point the initializer/constructor has completed execution and ends when program execution terminates. Next, the final declaration applies only to the declared variable, not to objects that may be referenced by the declared variable. This means that the immutability reachability for a final variable is shallow. Finally, the declaration applies to all instances of the variable; this means that the immutability context is empty.

Figure 1 contains a simple example of immutability assertions. In this example, we propose the use of new Javadoc tags of the form, @immutableField and @deepImmutableField, that can be applied to both static and instance fields (an alternative syntactic convention can be easily accommodated, if so desired). The @immutableField tag on field b2 asserts that the following values are immutable for the remainder of the program execution after initialization — the field b2 itself, and all locations directly contained in the object referenced by b2. Therefore, the values of pre.b2.a and post.b2.a in method foo1 are guaranteed to be identical, regardless of what may occur in the call to method bar1(). The @deepImmutableField tag on field b3 makes the stronger assertion that all locations reachable from b3 are also immutable. Finally, the @immutableField tag on field S shows how the same assertion can be applied to arrays. Using the immutability specifications, it is easy to see that method foo1() must always return true. This example also illustrates how immutability assertions are used in conjunction with final declarations. The final declaration ensures that field S is only written once, but the @immutableField assertion is necessary to ensure that the contents of the char[] array referenced by S are also immutable. A similar approach can be used to optimize implementations of the standard java.lang.String class by adding an immutability assertion for its char[] value.

In Figure 2, an @immutableParam tag (a proposed variant of the @param Javadoc tag) is used to specify that the object that parameter p points to is immutable for the duration of the method call. Therefore, the values of t1 and t2 in method foo2 will be the same. This is an example of a limited lifetime immutability assertion, since this assertion is only in effect during the call to method foo2. A @deepImmutableParam tag would make the stronger assertion that all objects reachable from fields in p are also immutable for the duration of the method call. Finally, the @immutableField and @deepImmutableField tags discussed earlier can also be used in a method scope, in which case the assertions are limited to the lifetime of the method call.

Figure 3 contains an example of an array immutability assertion. The use of an @arrayImmutableField tag before the method tmult indicates that the shape (rows and columns) of the array referenced by field M and the elements of this array are immutable within this method. Therefore, it is guaranteed that matrices this.M and A.M cannot be modified by the call to dotproduct or any other statement in the program while the call to tmult is in progress. This information can serve as useful documentation to the user, and can also be used to optimize the loop nest in tmult by recognizing that vectors this.M[i] and A[i][j] are invariant in the j and i loops respectively. However, if M contained a multi-dimensional array of references to objects, those objects would not be immutable with this assertion in effect.
class A { int value; ... }

class B { A a; ... }

class C {
    final B b1; // b1.a and b1.a.value can be modified, but not b1
    /**
     * @immutableField b2
     */
    final B b2; // b2.a.value can be modified, but not b2.a or b2
    /**
     * @deepImmutableField b3
     */
    final B b3; // none of b3, b3.a nor b3.a.value can be modified
    /**
     * @immutableField S
     */
    final static char[] S; // neither S nor S[*] can be modified
    ...

    boolean foo1() {
        B pre_b1, pre_b2, pre_b3, post_b1, post_b2, post_b3;
        char pre_S0; post_S0;
        pre_b1 = b1; pre_b2 = b2; pre_b3 = b3; pre_S0 = S[0];
        bar1(); // call with unknown side effects
        post_b1 = b1; post_b2 = b2; post_b3 = b3; post_S0 = S[0];
        return (pre_b1==post_b1) && (pre_b2.a==post_b2.a) &&
               (pre_b3.a.value==post_b3.value) && (pre_S0==post_S0); // must = true
    }
}

Figure 1. Example of shallow and deep immutability assertions on fields

In this regard, the @ArrayImmutableField tag can be viewed as specifying a reachability that is between shallow and deep by only traversing subarray objects in the multidimensional array.

Figure 4 shows a more realistic use of the @DeepImmutableField tag, on a static field called strings of type Vector. In this example, all objects reachable from strings are guaranteed not to be modified after the static initializer is executed. This assertion enables, for example, constant folding of Vector accesses after an inline expansion of a call to desc with a constant parameter.

Finally, Figure 5 contains an example of a refined immutability context assertion. In this example, a @DeepImmutableLocal Javadoc tag is used to indicate that all objects reachable from local variable h1 are immutable after the first assignment to h1. Both h1 and h2 are
/**
 * @ImmutableParam p
 * @return sum of pre- and post- values of p.value (before and after call to bar2)
 */
static int foo2(A p) {
    int t1, t2;
    t1 = p.value;
    bar2(p);
    t2 = p.value;  // Must be same as t1
    return t1 + t2;
}

Figure 2. Example of lifetime assertions on parameters

class ImmutableSquareMatrix {
    final int size;
    final double M[][];
    ...
    /**
     * @ArrayImmutableField M
     * @return this.M * transpose(A.M)
     */
    double[][] tmult(ImmutableSquareMatrix A) {
        double[][] D = new double[size][size];
        for (i=0 ; i<size ; i++)
            for (j=0 ; j<size ; j++)
                // this.M[*] and A.M[*] are guaranteed
                // to be immutable in this method
                D[i][j] = dotproduct(this.M[i], A.M[j]);
        return D;
    }
}

Figure 3. Example of array immutability assertion

---

import java.util.Vector;
class MyError {
    /**
     * @deepImmutableField strings
     */
    static final Vector strings = new Vector();
    static {
        strings.add("Error 0");
        strings.add("Error 1");
        strings.add("Error 2");
        ... 
    }
    ... 

    /**
     * @param code error code
     */
    String desc(int code) {
        return strings.get(code);
    }
}

Figure 4. Another example of a deep reachability assertion

/**
 * @deepImmutableLocal h1
 */
static int foo4(Map m, Object key) {
    final HashMap h1 = new HashMap(m);
    HashMap h2 = new HashMap();
    Object o1, o2;
    ...
    o1 = get(h1, key);
    bar4(h1, h2); // h1 will be unchanged
    o2 = get(h1, key); // must be same as o1
    ...
}

Figure 5. Example of refined context assertion
assigned references to *HashMap* objects, but the context assertion applies only to the *HashMap* object assigned to *hl*. Note that this immutability assertion really belongs at the point the local variable *hl* is declared. However, for consistency with current Javadoc conventions, we have specified this property as a method comment. If the Javadoc conventions are extended in the future to allow comments inside the body of a method, the immutability annotations should be moved closer to their associated statements. An alternative syntax can also be used.

3. The Framework

In this section, we describe the overall framework assumed in our approach. Section 3.1 contains definitions for specifying immutability properties. Section 3.2 summarizes the mechanism used for declaring immutability annotations. Section 3.3 outlines how immutability annotations can be verified.

3.1. Definitions

In this section, we develop formal definitions for immutability, culminating in item (9) that contains the definition of *partially immutable over a time interval*. A more complete set of definitions is contained in [22]. Intuitively, a location *ℓ* is said to be partially immutable over a time interval *τ* if some subset of locations reachable from *ℓ* are read, but not written, during the time interval *τ*. This sequence of definitions captures the general dimensions of *lifetime* (time interval), *reachability*, and *context* (location sets) discussed earlier.

For simplicity, we assume a single thread of execution and terminating programs in these definitions. Dealing with multi-threaded programs will require two extensions that are beyond the scope of this paper — assuming that temporal points are partially ordered, and specifying the memory model semantics for multi-threaded programs [26]. Dealing with non-terminating programs will involve modifying the definitions below to allow for infinite time intervals.

1. **Temporal Points and Time Intervals.**

   Define a *temporal point* to be a particular point in time in program execution. Since we assume a single thread of execution, all temporal points are totally ordered *i.e.*, for any two distinct temporal points *t₁* and *t₂*, either *t₁ < t₂* or *t₂ < t₁*. For convenience, we will use the terminology “temporal point *t*” and “time *t*” interchangeably.

   A *time interval* is a set of temporal points, that are not necessarily contiguous. Time interval *τ₁* is said to be a *sub-interval* of time interval *τ₂* if *τ₁ ⊆ τ₂*. A time interval *τ* is said to be *continuous* if any time point that lies between two points in *τ* also belongs to *τ*:

   \[ \forall t₁, t₂ \in \tau, t₁ ≤ t₃ ≤ t₂ \land t₂ ≤ t₃ \Rightarrow t₃ \in \tau \]

   Let *π* denote the continuous time interval from the start of program execution to its completion, *i.e.*, the universe of all temporal points for a given program execution.

2. **Span, pre-interval, post-interval.**
Define the span of time interval $\tau$ to be the minimal continuous interval that includes all the points in $\tau$, i.e., $\text{Span}(\tau)$ is the one-dimensional convex hull of $\tau$.

The pre-interval (post-interval) of a temporal point $t$ in time interval $\tau$ is defined to be the continuous sub-interval of $\tau$ that includes all temporal points in $\tau$ that precede (succeed) $t$:

$$\text{Pre}(t, \tau) \equiv \text{Span}\{t_1 | t_1 \in \tau \land t_1 < t\}$$

$$\text{Post}(t, \tau) \equiv \text{Span}\{t_2 | t_2 \in \tau \land t_2 > t\}$$

3 Abstract Locations and Location Instances.

A location instance is a dynamic instance of a field (static or instance) or array element allocated during program execution.

An abstract location is a static representative for a set of dynamic location instances created during program execution. A common approach is to associate a distinct abstract location for all location instances of a specific field or array element type in the program [6]. However, alternative approaches are possible, e.g., distinct abstract locations may be created for location instances originating from distinct allocation sites (contexts).

Given a location instance $i$, we define $V_i(t)$ as the value of $i$ at time $t$. Given an abstract location $\ell$, we define $\mathcal{H}_\ell$ to be the set of all instances of $\ell$. We also define $V_\ell(t)$ to be the set of values stored in all instances of $\ell$ at time $t$, and $V_\ell(\tau)$ as the set of values for a location $\ell$ over a time interval $\tau$:

$$V_\ell(t) = \bigcup_{i \in \mathcal{H}_\ell} V_i(t)$$

$$V_\ell(\tau) = \bigcup_{t \in \tau} V_\ell(t)$$

4 Accesses to Locations.

Define $\mathcal{R}_\ell(\tau)$ ($\mathcal{W}_\ell(\tau)$) to be the set of all temporal points from time interval $\tau$ at which an instance of abstract location $\ell$ is read (written):

$$\mathcal{R}_\ell(\tau) = \{t \in \tau : \exists i \in \mathcal{H}_\ell : i \text{ is read at time } t\}$$

$$\mathcal{W}_\ell(\tau) = \{t \in \tau : \exists i \in \mathcal{H}_\ell : i \text{ is written at time } t\}$$

$\mathcal{W}_\ell(\pi)$ is assumed to include all initialization write operations as well.

Define $\mathcal{A}_\ell(\tau)$ as the set of all accesses (read or write) for a location $\ell$ over a time interval $\tau$:

$$\mathcal{A}_\ell(\tau) = \mathcal{R}_\ell(\tau) \cup \mathcal{W}_\ell(\tau)$$

Note that $\mathcal{R}_\ell(\tau)$, $\mathcal{W}_\ell(\tau)$, and $\mathcal{A}_\ell(\tau)$ are non-continuous time intervals.

5 Method Execution.

A method $m$ is said to be executing at time $t$ if an invocation of $m$ is present anywhere on the program call stack at time $t$. A method $m$ is said to be exclusively executing at time $t$ if an invocation of $m$ is present at the top of the program call stack at time $t$. 
Define an *execution interval* for a method \( m \) over a time interval \( \tau \), \( C_m(\tau) \), as the union of all time points \( t \) in \( \tau \) at which \( m \) is executing. Define an *exclusive execution interval* for a method \( m \) over a time interval \( \tau \), \( C_m(\tau) \), as the union of all time points \( t \) in \( \tau \) at which \( m \) is exclusively executing.

**6 Lifetime.**

Define \( \mathcal{L}(\ell) \), the *lifetime* for a location \( \ell \), as the continuous time interval between the first access (write) to \( \ell \) and the last access to \( \ell \):

\[
\mathcal{L}(\ell) = \text{Span}(\mathcal{A}(\pi))
\]

**7 Reachability.**

Define \( \rightarrow_t \), the *points-to relation* between two locations: a location \( \ell \) points to \( p \) at time \( t \) (\( \ell \rightarrow_t p \)) if the value of \( \ell \) at time \( t \) is a reference to \( p \) (or an object containing \( p \)). Define \( \rightarrow^*_t \), the *reachability relation* between two locations as the [Kleene closure of the points-to relation]; a location \( p \) is reachable from \( \ell \) at time \( t \) (\( \ell \rightarrow^*_t p \)) if there exists a path at time \( t \) from \( \ell \) to \( p \) via reference locations. Note that the reachability relation is reflexive, i.e., \( \ell \) always reaches itself.

Define \( \rightarrow^k_t \), the *k-reachability relation*, as the subset of the reachability relation with paths of length at most \( k \): a location \( p \) is \( k \)-reachable from \( \ell \) at time \( t \) (\( \ell \rightarrow^k_t p \)) if there exists a path of length at most \( k \) at time \( t \) from \( \ell \) to \( p \) via reference locations.

Define the *k-reachability region* of a location \( \ell \) at time \( t \) as the set of locations that are \( k \)-reachable from \( \ell \) at time \( t \):

\[
\text{Reach}^k_t(\ell) = \{ x : \ell \rightarrow^k_t x \}
\]

Define the *deep reachability region* (or *reachability region* for short) of a location \( \ell \) at time \( t \) as the set of locations that are reachable from \( \ell \) at time \( t \):

\[
\text{Reach}_t(\ell) = \{ x : \ell \rightarrow^*_t x \}
\]

Define the *k-reachability region over a time interval* \( \tau \) of a location \( \ell \) as the set of locations that are \( k \)-reachable from \( \ell \) during that time interval:

\[
\text{REACH}^k_\ell(\tau) = \bigcup_{t \in \tau} \text{Reach}^k_t(\ell)
\]

Define the *deep reachability region over a time interval* \( \tau \) of a location \( \ell \) as the set of locations that are reachable from \( \ell \) during that time interval:

\[
\text{REACH}_\ell(\tau) = \bigcup_{t \in \tau} \text{Reach}_t(\ell)
\]

Define the *global reachability region* of a location \( \ell \) as the reachability region of \( \ell \) over time interval \( \pi \):

\[
\text{REACH}_\ell = \text{REACH}_\ell(\pi)
\]

**8 Finality.**

A location \( \ell \) is *final over a time interval* \( \tau \) if the set of its values is a singleton, and is unchanged over that time interval.

\[
\text{FINAL}(\ell, \tau) \equiv \forall t_1, t_2 \in \tau \, \forall V_\ell(t_1) = V_\ell(t_2) \land |V_\ell(t_1)| = |V_\ell(t_2)| = 1
\]
9 Immutability and Partial Immutability

Define the **immutability region over a time interval** $\tau$ for a location $\ell$ as the set of all locations that are reachable from $\ell$ and final during that time interval.

$$\text{IMMUT}_\ell(\tau) = \{ x : x \in \text{REACH}_\ell(\tau) \land \text{FINAL}(x, \tau) \}$$

A location $\ell$ is **immutable** if all locations reachable from it are final, **partially immutable** if any final location is reachable from it, and **mutable** otherwise.

Formally, a location $\ell$ is **immutable over a time interval** $\tau$ if its immutability region over $\tau$ coincides with its reachability region over that time interval, **partially immutable over a time interval** $\tau$ if its immutability region over $\tau$ is non-empty, and **mutable over a time interval** $\tau$ otherwise.

$$\text{IMMUTABLE}(\ell, \tau) \equiv \text{IMMUT}_\ell(\tau) = \text{REACH}_\ell(\tau)$$

$$\text{IMMUTABLE}^\phi(\ell, \tau) \equiv \text{IMMUT}_\ell(\tau) \neq \emptyset$$

$$\text{Mutable}(\ell, \tau) \equiv \text{IMMUT}_\ell(\tau) = \emptyset$$

3.2. Immutability Annotations

As discussed in Section 2, immutability assertions are expressed as *source code annotations* in our framework. Annotations are “meaningful comments”. As the name implies, they are used to annotate program source with information that is either not expressible in the language, or isn’t easy to infer from what the language can express.

We propose using Javadoc tags to express immutability annotations in Java programs. This approach is well-suited for highlighting immutability properties in Javadoc documentation. Currently, our framework supports seven immutability annotations, listed below with the corresponding dimensions for various contexts.
The semantics of these annotations should be clear from the descriptions in Section 2. These source code annotations are translated into bytecode attributes in the class files generated from the Java source code, thus making them accessible to the virtual machine for verification and for use in code optimization.

An alternative specification that we have considered is to use a parameterized notation for indicating the reachability of immutability annotations. So, an @immutable[1] Javadoc tag could be used to specify a fixed depth (1-level) reachability (equivalent to @immutable above). Then @immutable[*] could be used to signify deep reachability, and @immutable[@], for example, to signify the reachability depth of the declared array type. This is a more versatile notation, because it would be able to express rectangular arrays with immutable shape but mutable elements, which would be useful for optimizing various numeric applications. However, the complexity of specifying parameters may restrict the degree to which this convention may be adopted, compared to the non-parameterized Javadoc tags discussed earlier.

3.3. Immutability Verification

A major benefit of immutability assertions is that they can be checked automatically for violations, analogous to the checking of other safety properties in Java programs such as null pointer and index out-of-bounds accesses. Our framework includes a three-stage approach for verification of immutability properties. The focus of verification is on write operations, since immutability properties can only be invalidated by write operations:

Stage 1. Ahead-of-time static analysis: an approach such as [25] is used to identify fields that are guaranteed to be immutable, i.e., fields that can be considered to be final after being initialized. The static analysis can be performed with either closed-world or open-
world assumptions. All write operations on immutable fields can be ignored in Stages 2 and 3, since they are guaranteed to be part of the class initializer or constructor (for static and instance fields respectively). Despite the conservativeness of static analysis assumptions, this pre-pass is useful in reducing the overhead of dynamic checking in Stages 2 and 3.

**Stage 2. Just-in-time verification**: this stage uses the property verification framework introduced in [23], and summarized in Figure 6. This framework allows verification of arbitrary properties in dynamically loaded code, and maintains a mapping from methods to verification actions to be performed. A verification request is inserted for each method that contains a write operation that may conflict with an immutability assertion (after consulting the results of static analysis). The verification is performed dynamically for each such method on its first invocation (which may be interpreted or compiled), analogous to bytecode verification at dynamic link time [20]. If the verification passes, then no further action is needed. If the verification fails, then run-time checks are introduced for the non-verifiable write operations as described below in Stage 3.

The verification phase may also insert new verification requests for methods that have not been executed as yet, e.g., targets of unexecuted call sites. This ability allows the verification properties to make optimistic assumptions about the properties of these unknown methods. However, if one of these verification requests fails, it also triggers an invalidation of the verification of the method that made the request.

**Stage 3. Runtime checking**: this stage dynamically examines each execution instance of a write operation that did not pass verification, and checks if it violates any of the immutability assertions. Runtime checking can be expensive, which is why it is important to filter out as many write operations in Stages 1 and 2 as possible. Past experience with runtime checking using write barriers [18], suggests that runtime checking of write operations need not incur a large overhead if appropriate optimizations are performed. It is most convenient to implement run-time checking via code instrumentation; the instrumentation can be performed on the JIT-compiled code or on the bytecode. As a final note, we expect the relative frequency of write operations to be lower in applications
that are good candidates for immutability assertions. Later in Section 5.1, we will see that the ratio of write operations to read operations can be quite small for some applications, e.g., the ratio is only 2.7% for the jess benchmark in the SPECjvm98 benchmark suite, which bodes well for the usefulness of immutability annotations.

4. Use of Immutability Assertions in Code Optimization

In this section, we outline how immutability assertions can be used to improve the effectiveness of code optimization. One of the biggest impediments to optimization of object-oriented applications arises from potential aliasing of object references. For example, two getfield accesses of the same location, p.x, are not guaranteed to return the same value in general, if there is an intervening operation that may cause the value of the location to change, e.g., a call instruction, a synchronization instruction or a putfield operation on q.x such that p and q may potentially be aliased. However, the knowledge that the target of a read access is an immutable object can enable more aggressive code optimization. Read accesses of immutable objects can be modeled as functional operators — the results of multiple getfield accesses of the form p.x (or of multiple array-load access of the form a[j]) with the same operands are guaranteed to be identical. Further, an immutable object is guaranteed to not be aliased with any object that is modified.

A brief summary of selected optimizations that can benefit from immutability assertions is included below:

1. **Global Value Numbering**: several approaches have been proposed in the literature to determine if two symbolic expressions are guaranteed to contain the same value [2, 21, 27]. Past approaches to global value numbering usually limited their attention to operations on local/scalar variables. However, with the availability of immutability assertions, those approaches can easily be extended to include read accesses of immutable objects. For example, the two accesses to array element S[i] in Figure 4 can be given the same value number if S is known to be immutable. The same approach can be used for array accesses with equivalent index expressions, and more generally for get accesses on collections such as Vector, Map, HashMap, etc., with equivalent index/key expressions.

2. **Load elimination**: the effectiveness of past algorithms for load elimination of heap accesses (e.g., [10]) is limited in the presence of call instructions, synchronization instructions, or potentially aliased store instructions. These limitations can be removed for accesses to immutable objects. For example, the second load of p.x in Figure 2 can be eliminated when object p is known to be immutable (but not otherwise).

3. **Elimination of Null Pointer Checks and Array Bounds Checks**: most JIT compilers for Java perform some level of optimization of null checks and bounds checks. However, as with load elimination, these optimizations are limited in the presence of call, synchronization, and store instructions. The enhanced global value numbering that results from the use of immutability assertions can be used to improve the effectiveness of the null check and bounds check optimization algorithms such as [5]. For example, if object reference p is known to be (deeply) immutable, then the null check performed on
4. **Loop-invariant Code Motion**: this optimization is very important for loop-intensive codes that are typically found in scientific applications. When an optimizing compiler finds an expression whose value is invariant in an enclosing loop, it can optimize the program by moving the loop-invariant computation outside the loop. Immutability assertions can enable loop-invariant code motion to be performed on object references. For example, when *this.M* is known to be immutable, the access to *this.M[i]* can be moved out of the innermost *j* loop in Figure 3.

5. **Dependence Analysis and Parallelization**: dependence analysis can be used to enable a wide range of optimizing and parallelizing transformations on loop nests. Dependences are usually computed on pairs of memory accesses. The three kinds of dependences that can prevent the application of transformations are *flow*, *anti*, and *output* dependences [29]. All three kinds of dependences require the presence of at least one write operation in the pair. Immutability assertions can greatly improve the effectiveness of dependence analysis, because any pair containing an access to an immutable object is guaranteed to not be a flow, anti, or output dependence. This is true even for strong memory models in which read operations are usually treated as "killing" [26]. Since no modifications (serial or concurrent) are permitted on immutable objects, there is no need to treat read accesses to immutable objects as killing, even in strong memory models.

6. **Data Transformations**: data transformations such as object inlining [8], object splitting [7], scalar replacement [10], and object caching, are becoming increasingly important due to the large memory hierarchy overheads present in current and future high-performance processors. Immutability assertions can greatly increase the applicability of data transformations. For example, immutable objects can be easily inlined, even if doing so leads to duplicate copies of immutable locations. This is different from the safety conditions for inlining of mutable objects, which usually require that each subobject have exactly one parent object.

Finally, we observe that even for cases in which the immutability properties can be inferred automatically by an advanced compiler, there is a benefit in using the annotations so as to avoid the run-time overhead of performing the immutability analysis in a JIT compiler.

5. **Experimental Results**

In this section, we present preliminary experimental results to support our approach of using immutability assertions. Section 5.1 presents the results of a *limit study* of dynamic read and write accesses in a range of benchmark programs. These results show that a significant number of read accesses could be classified as accesses to immutable objects (using the more
Table I. Benchmark programs and their dynamic characteristics

general dimensions of immutability introduced in our framework). Section 5.2 presents dynamic read/write counts and execution times for a small set of annotated test programs.

5.1. Limit Study on Immutability

Table I lists the benchmarks used for the limit study and their dynamic characteristics. The dynamic characteristics were obtained by instrumenting the Jikes Research Virtual Machine [16] to generate traces of all field and array element read and write operations in the benchmark programs. The column titled Locs lists the number of distinct dynamic field and array element locations encountered in the program. The columns titled Reads and Writes give the total number of dynamic read and write operations observed for fields and array elements. The column titled Reads/Locs provides the ratio of dynamic reads to dynamic locations for each benchmark program. When this ratio is large, it suggests an increased opportunity for optimization of read operations. In fact, this ratio varies from 23.9 to 586.1, with an average value of 59.1, which is reasonably large. The column titled Writes/Reads provides the ratio of dynamic writes to dynamic reads for each benchmark program. When this ratio is small, it suggests an increased opportunity for identifying immutable locations. This ratio varies from 2.3% to 57.7%, with an average value of 16.6%, i.e., on average one write operation is performed for every 6 read operations, which is reasonably small. Therefore, both ratios are promising in suggesting potential opportunities for exploiting object immutability.

Since read operations are the focus of immutability-based optimizations, we define an Immutability Ratio (IR) as follows:

\[ IR = \frac{\# \text{ of immutable read operations}}{\text{total \# of read operations}} \]

Thus, the IR will range from 0% to 100%, with a larger value indicating a larger degree of immutability.
One question that arises is: which read operations should be considered immutable for the purpose of the limit study? In the most liberal interpretation, we can consider every read operation to be immutable for the interval between its previous and next write operations. However, this would degenerate to an IR value of 100% for all programs, which is not a useful interpretation. Instead, for this limit study, we only consider as immutable those read operations that follow the last write operation on a given location.

Figure 7 shows two IR values for each benchmark program:

- **IR actual** — this value is obtained by examining each dynamic location allocated during program execution, and counting all reads following the last write to the dynamic location as immutable. This is a fine-grained limit on IR because each location is examined separately, even for locations that are instances of the same field. The average value of IR actual across all benchmarks in Figure 7 is 60.8%, which suggests that approximately 3 out of 5 read operations can potentially be classified as immutable.

  One approximation made in computing the IR actual limit value is that read and write operations on the same array object were not distinguished by array index values. A
more fine-grained measure would treat each array element as a separate location, which could lead to larger IR values in the limit, especially for array-intensive programs.

- **IR uniform** — this value is a hypothetical "expected" value of IR computed by assuming that the write operations are uniformly distributed among the read operations, and by estimating how many read operations appear between two write operations on average. The IR uniform value, $E(\text{IR})$, can be computed directly from the number of locations, reads and writes listed in Table I. Specifically,

$$
E(\text{IR}) = \frac{\text{Locs} \times \text{Reads}/\text{Locs}}{\text{Reads}/\text{Writes}} = \frac{\text{Locs}}{\text{Writes}}
$$

i.e., $E(\text{IR})$ is simply the average ratio of dynamic locations to dynamic write operations. As can be seen in Figure 7, the $E(\text{IR})$ values are quite small, with an average value of 12.1% across all benchmarks. It is encouraging to note that the IR actual values are larger than the IR uniform values, because it suggests that the actual read-write distributions observed in benchmark programs are more favorable for immutability than uniform distributions.

Figures 8, 9, 10, 11 and 12 provide more details on the IR actual results for the four biggest benchmark programs in Table I: 202.jess, 209.jdb, 213.javac, 228.jack, and DOMcount. These figures show how the cumulative number of dynamic immutable read operations (on the vertical axis with a logarithmic scale) vary as function of the number of abstract locations (on the horizontal axis). Recall that an abstract location is a static representative for a set of dynamic location instances. In this study, each declared array type and field was treated as a distinct abstract location.

In each figure, the fine-grained immutability curve shows how the cumulative number of immutable reads rises with the inclusion of additional abstract locations. The abstract locations are sorted in descending order of their contribution to the number of dynamic immutable read operations, so we would expect each fine-grained immutability curve to rise quickly and then flatten out as the cumulative value approaches the total number of immutable reads used to compute IR actual in Figure 7. The data in these curves establish a new 90-10 rule viz., less than 10% of abstract locations account for more than 90% of the dynamic immutable reads. This is encouraging, because the programmer effort required to specify immutability annotations is proportional to the number of abstract locations that need to be annotated.

While the fine-grained immutability curves provide upper bounds on the number of read operations that can be classified as immutable, the coarse-grained immutability curves serve as lower bounds. The number of reads considered immutable according to the coarse-grained definition is obtained by merging all dynamic instances of the same abstract location, and only considering those read operations to be immutable that follow the last write on the merged location. The rationale for this measurement is to evaluate how much of the potential immutability in a program can be exploited by an optimizer without using additional context to refine the dynamic instances of an abstract location. As can be seen in the figures, the number of reads classified as coarse-grained immutable is about 3x to 30x smaller than the reads classified as fine-grained immutable. These results confirm that additional context information would be needed to fully exploit immutability annotations for optimization.
Figure 8. Cumulative distribution of dynamic immutable reads for _202_jess benchmark

Figure 9. Cumulative distribution of dynamic immutable reads for _209_db benchmark
Figure 10. Cumulative distribution of dynamic immutable reads for _213_javac benchmark

Figure 11. Cumulative distribution of dynamic immutable reads for _228_jack benchmark
Figure 12. Cumulative distribution of dynamic immutable reads for DOMcount benchmark

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Abstract Location</th>
<th>Imm. Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>spec.benchmarks.202.jess.jess.Value[]</code></td>
<td>16730296</td>
</tr>
<tr>
<td></td>
<td><code>spec.benchmarks.202.jess.jess.Value._type</code></td>
<td>6312771</td>
</tr>
<tr>
<td></td>
<td><code>java.lang.String.count</code></td>
<td>1113720</td>
</tr>
</tbody>
</table>

Table II. Most immutable locations for largest benchmarks: _202.jess_

Tables II and III show fields and array types associated with the ten most immutable abstract locations (those that contribute the most dynamic immutable reads) for each of the five largest benchmarks.

5.2. Preliminary Optimization Results

In this section, we present preliminary results on dynamic read counts and execution times obtained by inserting immutability annotations into selected benchmark programs. The inherent limitation in adding more benchmarks is that all results in this section require hand-coded insertion of annotations, which in turn requires an in-depth understanding of the
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Abstract Location</th>
<th>Imm. Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>_209.db</td>
<td>char[]</td>
<td>8512939</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.count</td>
<td>7367265</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.value</td>
<td>7117295</td>
</tr>
<tr>
<td></td>
<td>byte[]</td>
<td>5062810</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.offset</td>
<td>4752364</td>
</tr>
<tr>
<td></td>
<td>java.util.Vector.elementData</td>
<td>3619159</td>
</tr>
<tr>
<td></td>
<td>spec.benchmarks._209.db.Entry.items</td>
<td>3380587</td>
</tr>
<tr>
<td></td>
<td>java.util.Vector.elementCount</td>
<td>3302652</td>
</tr>
<tr>
<td></td>
<td>java.lang.Object[]</td>
<td>3184556</td>
</tr>
<tr>
<td></td>
<td>spec.benchmarks._209.db.Database.index</td>
<td>2946860</td>
</tr>
<tr>
<td>_213.javac</td>
<td>char[]</td>
<td>7778109</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.count</td>
<td>3008441</td>
</tr>
<tr>
<td></td>
<td>java.io.BufferedInputStream.buf</td>
<td>2360959</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.value</td>
<td>2021506</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.offset</td>
<td>1664637</td>
</tr>
<tr>
<td></td>
<td>byte[]</td>
<td>1214173</td>
</tr>
<tr>
<td></td>
<td>java.io.FilterInputStream.in</td>
<td>1187866</td>
</tr>
<tr>
<td></td>
<td>spec.benchmarks._213.javac.Scanner.in</td>
<td>1125723</td>
</tr>
<tr>
<td></td>
<td>java.util.HashMap.Entry[]</td>
<td>807524</td>
</tr>
<tr>
<td></td>
<td>java.util.HashMap.elementData</td>
<td>678225</td>
</tr>
<tr>
<td>_228.jack</td>
<td>java.util.HashMap.Entry[]</td>
<td>8793701</td>
</tr>
<tr>
<td></td>
<td>java.util.HashMap.Entry[].array</td>
<td>87837296</td>
</tr>
<tr>
<td></td>
<td>java.util.HashMap.Entry[].end</td>
<td>8705454</td>
</tr>
<tr>
<td></td>
<td>java.util.Vector.elementData</td>
<td>5447891</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.count</td>
<td>3653959</td>
</tr>
<tr>
<td></td>
<td>char[]</td>
<td>3553651</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.value</td>
<td>3405006</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.offset</td>
<td>2937270</td>
</tr>
<tr>
<td></td>
<td>java.util.HashMap.elementData</td>
<td>2783631</td>
</tr>
<tr>
<td></td>
<td>java.util.Vector.elementCount</td>
<td>1821537</td>
</tr>
<tr>
<td>DOMcount</td>
<td>byte[]</td>
<td>1273087</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.value</td>
<td>959667</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.count</td>
<td>953368</td>
</tr>
<tr>
<td></td>
<td>char[]</td>
<td>882621</td>
</tr>
<tr>
<td></td>
<td>W internal data structure</td>
<td>731215</td>
</tr>
<tr>
<td></td>
<td>org.apache.xerces.utils.UTF8DataChunk.fgTempBuffer</td>
<td>383345</td>
</tr>
<tr>
<td></td>
<td>org.apache.xerces.utils.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMLCharacterProperties.fgAsciiCharData</td>
<td>380542</td>
</tr>
<tr>
<td></td>
<td>int[]</td>
<td>374780</td>
</tr>
<tr>
<td></td>
<td>java.lang.String.offset</td>
<td>343734</td>
</tr>
<tr>
<td></td>
<td>org.apache.xerces.framework</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMLDocument.Scanner.ContentDispatcher.this$0</td>
<td>295080</td>
</tr>
</tbody>
</table>

Table III. Most immutable locations for largest benchmarks: _209.db, _213.javac, _228.jack, DOMcount.
Table IV. Dynamic read access counts (in thousands)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Unoptimized</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GetStatic</td>
<td>GetField</td>
</tr>
<tr>
<td>utf8</td>
<td>0.7</td>
<td>1968.7</td>
</tr>
<tr>
<td>opttests:sieve</td>
<td>40950.2</td>
<td>2.5</td>
</tr>
<tr>
<td>heapsort</td>
<td>0.2</td>
<td>1303221.7</td>
</tr>
<tr>
<td>lufact:dmxpy</td>
<td>0.2</td>
<td>3.2</td>
</tr>
</tbody>
</table>

Figure 13. Dynamic read access counts for benchmark programs

benchmark implementations. The results reported in this section were obtained by modifying version 1.11 of the Jikes source-to-bytecode compiler to propagate immutability annotations through classfiles, and version 2.0.3 of the Jikes Research Virtual Machine to use immutability annotations for optimization. Only three phases of the Jikes RVM optimizing compiler [1] were augmented to use immutability annotations — Global Value Numbering, Load Elimination, and Loop-invariant Code Motion (items 1, 2, 4 in Section 4).

The dynamic read counts are presented in Table IV, with a graphical display shown in Figure 13. The applications were run with and without using the annotations, and read accesses to fields through getfield and getstatic bytecodes were counted for both cases. Two of the applications are part of the Jikes RVM regression tests (utf8 and opttests:sieve), and two are part of the Java Grande Forum benchmark set (heapsort and lufact:dmxpy). The utf8 test reads characters from a generated Unicode string and verifies that they are read correctly by comparing them to an expression based on the index. The opttests:sieve test
is a prime number computation kernel based on the Sieve of Eratosthenes method, and is  
part of the opttests benchmark for which results were presented in Section 5.1. Heapsort is  
a sorting benchmark, and lufact:dynmxpy is a vector-matrix multiplication method extracted  
from the lufact benchmark. The results in Table IV show that immutability annotations can  
potentially lead to large reductions in the number of dynamic read accesses (in the range of  
33% to 99% of the accesses targeted).

The execution times presented in Table V were obtained using a RS/6000 model S80  
system with 450MHz PowerPC RS64II processors, and do not include compile time. Though  
the standard SizeB data size was used for heapsort, the input sizes for the remaining  
benchmarks in Table V were increased compared to the versions used in Table I so as to  
obtain execution times that are large enough to be reproducible with little variation. The  
results show that immutability annotations delivered speedups in the range of 1.05× to 1.10×  
for these benchmarks. Additional performance improvements can be expected in the future  
when additional optimizations in Jikes RVM are extended to use immutability annotations.

The following annotations were used for the applications:

1. utf8 — an @immutableField tag was specified for field test, as well as three fields of  
the String class: count, value, and offset. These immutability annotations enabled loop-  
varying code motion and common subexpression elimination of getstatic operations on  
test and getfield operations on String.count. In the absence of immutability annotations,  
the optimizer is forced to assume that these fields could be updated in another method  
or in a concurrent thread, thus inhibiting loop-invariant code motion and common  
subexpression elimination.

2. opttests:sieve — an @immutableField tag was specified for the field sieve.flags for  
the lifetime of the method go. This annotation enabled loop-invariant code motion for  
getstatic operations on the flags field.

3. heapsort — an @immutableField tag was specified for field NumericSortTest.TestArray  
for the lifetime of method NumSift. The knowledge that this field is immutable can enable  
common subexpression elimination and loop-invariant code motion of most getfield  
operations on TestArray in the while loop. The result of the transformation is a single  
getfield operation that is performed only for the first iteration of the loop.

4. lufact:dynmxpy — an @immutableParam tag was specified for parameters x and m of  
method dynmxpy, for the lifetime of the method. The knowledge that the one-dimensional  
array x and the two-dimensional array a are immutable enabled common subexpression  
 elimination of the elements of the array x, x[j], and the one-dimensional  
sub-arrays of m, m[i]. The Jikes RVM optimizing compiler does not perform this  
transformation by default, because it has to be prepared for the possibility that elements  
of x or subarrays of m may be updated by a concurrent thread.

6. Related Work

As mentioned earlier, many programming languages offer rudimentary mechanisms for  
specifying immutability, e.g., the final declaration in Java [14], and the const declaration in
C++ [9]. However, neither of these mechanisms is flexible enough to specify the immutability properties discussed in this paper - Java's final can only specify single reference reachability for a limited set of lifetimes, while C++'s const can also provide object reachability for the current (this) object and method parameters. A more closely related approach is immutable class specification [13, 28]. The idea is similar to ours, but the specification is statically checked and lacks the fine granularity of our approach.

The C pragma feature [19] is a notable mechanism for providing source-level annotations. Source code annotations were also used heavily in previous work on dynamic compilation, e.g., [15]. One recent proposal of note is the Java Metadata specification JSR [4]. The mechanism proposed is similar to Java attributes, and would allow very expressive annotations in Java programs. The propagation mechanism proposed in the JSR is similar to the one outlined in this paper.

Past work on static analysis and detection of immutability properties (e.g., [25]) is relevant to our framework in two important ways. First, as discussed in Section 3.3, static analysis can be used as the first stage of verification of immutability properties. Second, immutability assertions can be used to improve the results of static analysis. Immutability analysis stems from pointer and side-effect analysis (e.g., [11, 12]). There is a very large body of work on side-effect analysis spanning the last three decades. The immutability specification approach is different because side-effect information is specified by the programmer and verified by the compiler or the virtual machine. Finally, there has been additional work on the use of Java bytecode attributes to communicate the results of off-line analysis or profiling information to just-in-time (JIT) compilers [24, 3, 17].

7. Conclusions

We have presented a framework for specifying immutability assertions at the level of Java source code. The use of assertion information in optimization produced significant reductions (33% to 99%) in the number of dynamic read accesses, and also measurable speedups in the range of 5% to 10% for certain benchmark programs. A limit study suggests that many more opportunities for immutability assertions are present in other benchmarks, including the SPECjvm98 suite.
Acknowledgments

We are grateful to Robert O'Callahan and to the anonymous reviewers for their comments on an earlier version of this paper. The @ArrayImmutableField and @ArrayImmutableParam tags were the result of a specific suggestion from one of the reviewers. We would also like to thank all members of the Jikes RVM team at IBM for their contributions to the Jikes Research Virtual Machine, and Philippe Charles and David Shields for their assistance with extending the Jikes source-to-bytecode compiler.

REFERENCES


