Object-Specific Redundancy Elimination Techniques*

Rhodes H. F. Brown and R. Nigel Horspool
{rhodesb,nigelh}@cs.uvic.ca

Department of Computer Science
University of Victoria, P.O. Box 3055
Victoria, BC, Canada V8W 3P6

Abstract. Traditionally, code optimization techniques have focused on the analysis and manipulation of low-level, simply-typed values. In some cases though, and especially for object-oriented languages, the semantics of the source language and/or execution model may provide additional constraints on low-level values which can be leveraged to produce further optimizations. We believe that numerous opportunities exist to induce program optimizations based on the static and dynamic properties of objects. This paper introduces several new approaches to redundancy elimination based on object-specific properties.

1 Introduction

Broadly speaking, redundancy elimination optimizations are those that remove or reduce the occurrence of unnecessary computations in a program. These optimizations focus on identifying situations where the result of a computation is pre-determined, or where the result of an equivalent computation is readily available in the same context. If a computation is deemed to be redundant, and it has no other side-effects, then it can be replaced with a direct reference to the previous or known result. Of course, such replacements are only valid if a program analysis can prove that the previously computed result is equal to the one that would be computed in the target context. In other words, an analysis must identify an *invariant* that persists from the previous computation through to the target context and guarantees the equivalence of the two results.

Traditional redundancy elimination relies on the idea that computations involving simple values are invariant as long as the participating variables are not assigned new values. Other, more sophisticated optimizations such as static method binding and method inlining [2, 3] (i.e., elimination of polymorphism & dispatch mechanisms) rely on the notion that an object's run-time type is also an invariant property. If it is possible to associate the use of an object reference with a particular instantiation, then the precise type of the object can be determined and dynamic tests of the object's type become redundant. We build

* This research was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

on this concept by observing that "type" (i.e., a class label) is certainly not the only invariant that an object reference inherits at the point of instantiation.

One obvious, yet overlooked property of statically-typed, object-oriented languages is that the method implementations for a specific class type are fixed over the lifetime of any particular instance of the class. To see how this property can be leveraged to eliminate redundant computations, consider the common virtual dispatch table approach to implementing dynamic polymorphism. In a given context (see Figure 1) an object's specific type may not be known. Therefore, upon reaching an invoke expression, the actual target address of the method is loaded from the virtual dispatch table associated with the object—essentially, a computation involving one or more indexed memory loads. If another invoke of the same method applied to the same receiver (e.g., the second invoke of `o.m1()`) occurs in this context then the second query of the dispatch table is clearly redundant since the target method address remains invariant. Additionally, some partial redundancy may occur if the receiver of a dispatch was previously involved in a different dispatch computation or a run-time type test. In both cases, it is likely that some relevant data was already loaded as part of the earlier computation.

```
void foo( T o ) {
  o.m1(); // unknown target
  ...
  o.m1(); // known target
  ...
  o.m2(); // known vtable
}
```

Fig. 1. Dynamic Dispatch Redundancy

An important observation regarding the elimination of redundant virtual table queries is that the invariant for a specific object reference (i.e., the contents of its dispatch table) cannot be statically determined for languages with open-ended hierarchies and dynamic class loading, such as Java and C#. In this case, even though the invariant persists throughout the lifetime of a given object, it must be dynamically detected. This is in stark contrast to classic redundancy elimination which usually considers statically detectable, yet transient invariant properties.

There are numerous other kinds of object/reference properties that can also be inferred from the dynamics of a given execution context. For example, local elimination of nullness checks and array-bounds checks [4] relies on an inspection of earlier referencing and indexing operations.

In the following discussion, we explore a variety of static and dynamic schemes for identifying and exploiting persistent, modal, and transient object-specific properties to perform new kinds of redundancy elimination transforms. In some cases we exploit existing features of the semantics of Java. We also propose several semantic extensions that allow programmers to further constrain the use of object references in certain contexts. Our examples focus on Java, however our proposals are intended to be applicable to a wide range of object-oriented languages. Our goal is to identify the kinds of properties that can be associated with object values, detected through some combination of static and dynamic measurements, and leveraged to eliminate redundant computations.

2 Static Invariants

Consider a typical example of iteration over an array in Java, as illustrated in Figure 2. Clearly, the value of `array.length` is unchanged over the lifetime of the array object. Thus it is only necessary to load the length value once in the current context rather than every time the loop condition is evaluated. In fact, similar reasoning can be applied to any Java field declared as `final`.[1]

```
for (i=0; i<array.length; i++)
 // use array[i]
```

Fig. 2. Redundant Constant Loading

There is, however, one caveat to consider when attempting to eliminate redundant loads of final fields. In Java, it is possible for a final field to be loaded before the field has actually been initialized with its "final" value. A constructor (or class initializer) may invoke other methods, which access fields, before completing the field initialization. Figure 3 illustrates this dilemma. Moreover, with the possibility for open-ended extensions to the class hierarchy, it may be impossible to statically determine if a given field load is being applied to a fully initialized object. To be both valid and effective, an optimization that removes redundant final field loads needs to establish another form of invariant, namely that the target object is, in fact, fully initialized (i.e., constructed) at the initial load point and that this condition persists through to any subsequent loads in the same context. Obviously the "constructed" property is persistent—once constructed, objects do not later revert to being unconstructed—but, unlike type and method bindings, the property does not arise immediately—it follows initialization, not allocation. Thus, we require a mechanism to deduce which object references have definitely completed this modal transition and are fully initialized.

```
class X {
  final int x;
  X() {
    this.m();
    x = 1;
  }
  void m() {
    // x: 0 or 1?
  }
}
```

Fig. 3. Incomplete Initialization

One approach to detecting the constructed status of an object might be to include a new bit in the object's status header and set the bit on completion of construction. However, a purely static approach is also possible if the source language can offer guarantees regarding the constructed status of object references. Java attributes (introduced in version 5 of the language [5]) provide a basis for establishing new language semantics to identify and control the proliferation of references to objects that may not be fully initialized. Just as the existing access modifiers (e.g., `private`, `protected`, etc.) restrict the use of certain methods, a `@Constructed` attribute could be used to indicate that a method can only be invoked on a fully-constructed receiver. Applying such an attribute

[1] Note that any private field that is only assigned a value inside a constructor (or class initializer for statics) is effectively final.

to method `m()` in Figure 3 would forbid its use in the constructor, and hence disambiguate the result of loading `x`. Alternatively, since methods are usually applied to constructed objects, it might be more practical to consider the converse: only methods with an `@Incomplete` attribute can be invoked inside constructors. If the compiler (or verifier) can be relied on to enforce such attributes transitively across invocations and down the inheritance hierarchy, it would enable the safe and effective removal of most redundant final field loads.

Another example of a compiler-enforced attribute that would have significant potential for redundancy elimination is a `@Constant` attribute, with similar semantics to C++'s `const` modifier [6] (as applied to methods). To see the utility of such an attribute, consider that a common practice in object-oriented design is to restrict direct access to field values and instead provide indirect access via "getter" methods. In many cases, the object interface may also provide access to values that are not actually stored in fields, but merely derived from them (for example, the polar coordinates of a Cartesian point). Figure 4 provides a simple illustration of methods that always yield the same constant value.

```
class Point {
  private double x, y;

  @Constant
  double getX() {
    return x;
  }

  // non-constant:
  void setX(double x) {
    this.x = x;
  }

  // y access...
}

class Polar extends Point {
  @Constant
  double getRadius() {
    return sqrt(
      getX()*getX()+
      getY()*getY());
  }
  ...
}
```

Fig. 4. Constant-Valued Methods

Just as the "final" property implies an invariant over the use of field members, enabling the removal of redundant loads, a "constant" property would imply invariance over the use of method members, enabling the removal of entire redundant invocations—for example, the second uses of both `getX()` and `getY()` in computing the `getRadius()` result in Figure 4. The one important difference is that while finality is persistent (post-construction, of course), the precise return value of constant methods can only be guaranteed as long the receiver has not been mutated (directly, or indirectly). In particular, to leverage the full potential of a `@Constant` method attribute, it must be combined with knowledge of whether the receiving object can be concurrently modified. While a whole-program thread-escape analysis [7] may be useful in this regard, we can also leverage the potential of attributes to bound the possibilities for concurrent mutation.

As with construction, objects that escape the thread context in which they were created undergo a modal transition: once a reference is passed/assigned to an external context, any subsequent uses of the object must assume that concurrent modifications are possible. To make a `@Constant` method attribute

effective, it could be combined with a "does-not-escape", or more practically, an `@Escapes` method attribute that identifies when references that may influence an object's state escape. Again, if this property is enforced transitively and down the hierarchy, it could be used to assert that a locally allocated object definitely does not undergo any concurrent modifications. For example, given a locally allocated `Point` (as in Figure 4), a call to `setX()` may alter the state of the object but, it does not do anything to expose the state beyond the current thread context. Thus, repeated use of the same `@Constant` method (e.g., `getX()`), following a `setX()` call, may still result in some guaranteed redundancy. For non-local objects, the situation is more complicated. A thread-escape analysis may be useful in providing an approximation of which references can be concurrently accessed, but a more effective approach may be to detect the synchronization status dynamically.

3 Dynamic Invariants

Consider an extension of our previous redundant load example, this time using a two-dimensional array. Clearly the repetitive loading of `array[i]` in the inner loop of Figure 5 is redundant. However, Java semantics dictate that, in general, the redundant access cannot be eliminated due to the potential for concurrent modification. Again, similar reasoning applies to all forms of non-final field loads and stores.

```
for (i=0; i<array.length; i++)
  for (j=0; j<array[i].length; j++)
    // use array[i][j]
```

Fig. 5. Redundant Field Access

A static thread-escape analysis may provide an approximation of the contexts in which an object may be modified concurrently but, an alternative is to simply inspect the synchronization status in the object's header before executing the first load (or store) operation. Then, assuming the object is not unlocked in the current context, subsequent re-loads (or re-stores of the same value) are unnecessary and can be safely eliminated. Figure 6 illustrates a replacement for the inner loop above that safely eliminates the redundant field loads. Note that this approach bears a strong resemblance to the guarded inlining scheme of Detlefs and Agesen [8]. Both approaches focus on dynamic detection of incidental object-properties.

```
if (array is locked) {
  tmp = array[i];
  for (j=0; j<tmp.length; j++)
    // use tmp[j]
}
else {
  // as before
}
```

Fig. 6. Redundant Load Elimination

Of course, to be effective, a scheme based on dynamic detection of an object's locked status requires that the target object actually gets synchronized at some point. Arbitrarily inserting synchronization statements would violate the

semantics of the input program. However, if the target object is locally allocated and does not escape, it may be possible (and preferable) to introduce artificial locks to enable the elimination of redundant loads, stores, or invocations (as in Figure 4) performed in nested routines. Furthermore, it is reasonable to assume that the overhead of such additional synchronization would be negligible when compared to the cost of several redundant loads, stores or invocations, given a fast mechanism for securing uncontended locks [9, 10].

Yet another potential optimization based on dynamic detection of object invariants is to consider specialized object implementations based on the values assigned to final fields. From a static point of view (as seen in Figure 7), it is often difficult to eliminate polymorphic dispatches, in favor of direct method bindings and/or inlining, due to the potential for dynamic class loading. However, from the point of view of a particular object instance, dispatches on final fields will bind to specific and invariant method targets. Thus, for long-lived objects, it may be worthwhile to detect the precise types assigned to final fields and generate specialized method implementations with direct dispatches or inlined code. Such an approach would necessitate object-specific rather than type-specific dispatch tables but, the approach is quite feasible given the right object layout.

```
class LongLived {
  final T f;
  LongLived( T f ) {
    this.f = f;
      // type of f is fixed,
      // but unknown
  }
  void m1() {
    f.m2();
      // target of invoke is
      // fixed for each LongLived
  }
}
```

Fig. 7. Instance Specialization

Su and Lipasti [11] describe a similar notion of specialization based on object state. In their approach, specializations are viewed as dynamic extensions of the class hierarchy. Unfortunately, this perspective can be misleading since these specializations do not introduce any new type semantics. Our view differs in that we recognize that certain, possibly dynamic, object properties and state values permit automated generation of more efficient object implementations. Dean, *et al.* [12] propose a similar, type-based (rather than object-based) approach. In another related idea, Maurer [13] describes an explicitly programmed form of specialization, called *metamorphic programming*, where the implementation associated with a given object appears to evolve over the lifetime of the object.

4 Conclusion

In the previous sections, we have identified a number of opportunities for exploiting both static and dynamic object-specific properties to enable redundancy elimination transforms. For the most part, these proposals build on existing opti-

mization schemes. Our contribution is to present (and advocate) a holistic[2] view of object-oriented optimization. Taken individually, properties such as the synchronized status of objects or the invariance of method results may yield some potential optimization. However, to achieve truly effective redundancy elimination, it is necessary to combine this knowledge, along with other static and dynamic properties, to expose all of the opportunities for optimization.

It is quite reasonable to presume that the static language extensions we propose would be adopted by the programming community since they offer benefits beyond just improved redundancy elimination. Clearly, each of the "constructed," "constant," and "escapes" attributes allow a refinement of the programmer's intent when architecting a class, enhancing maintainability and other software engineering concerns. The information provided by these attributes could also be exploited by a variety of other analyses and transforms. For example, the constraints provided by an `@Escapes` attribute (or a lack thereof, indicating no escaping references) could be used to achieve more precise model checking of a program's concurrent behavior [15]. In another example, a `@Constant` attribute could be used to enhance the effectiveness of speculative multithreading [16] by identifying methods that have (guaranteed) predictable results, even when executed concurrently. In short, our proposals demonstrate the need to consider new forms of "object modifiers" that extend beyond the basic, type-associated, static access modifiers (e.g., `private`, `final`, etc.).

Our initial investigations suggest that a focus on object-specific properties has the potential to reveal a number of new and significant optimization opportunities. In the near term, we aim to quantify this potential by estimating the utility of our combined approach on actual benchmark programs. Ultimately, our goal is to develop a collection of effective, object-specific optimization techniques using a combination of ahead-of-time and just-in-time compilation strategies.

References

1. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco, CA (1997)
2. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), New York, NY, ACM Press (2000) 264–280
3. Qian, F., Hendren, L.: A study of type analysis for speculative method inlining in a JIT environment. In: Proceedings of the International Conference on Compiler Construction (CC). Lecture Notes in Computer Science, Springer (2005) 255–270
4. Qian, F., Hendren, L., Verbrugge, C.: A comprehensive approach to array bounds check elimination for Java. In: Proceedings of the International Conference on

[2] From Reference.com (`http://www.reference.com/browse/wiki/Holism`): "Holism ... is the idea that the properties of a system cannot be determined or explained by the sum of its components."

Compiler Construction (CC). Lecture Notes in Computer Science, Springer (2002) 325–342
5. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java^{TM} Language Specification. Third edn. Addison-Wesley (2005)
6. Stroustrup, B.: The C++ Programming Language. Third edn. Addison-Wesley, Reading, MA (1997)
7. Blanchet, B.: Escape analysis for Java^{TM}: Theory and practice. ACM Transactions on Programming Languages and Systems **25**(6) (2003) 713–775
8. Detlefs, D., Agesen, O.: Inlining of virtual methods. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, Springer (1999) 258–278
9. Bacon, D.F., Konuru, R., Murthy, C., Serrano, M.: Thin locks: Featherweight synchronization for Java. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), New York, NY, ACM Press (1998) 258–268
10. Gagnon, E., Hendren, L.: SableVM: A research framework for the efficient execution of Java bytecode. In: Java^{TM} Virtual Machine Research and Technology Symposium, USENIX Association (2001)
11. Su, L., Lipasti, M.H.: Dynamic class hierarchy mutation. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), Washington, DC, IEEE Computer Society Press (2006) 98–110
12. Dean, J., Chambers, C., Grove, D.: Selective specialization for object-oriented languages. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM Press (1995) 93–102
13. Maurer, P.M.: Metamorphic programming: Unconventional high performance. IEEE Computer **37**(3) (2004) 30–38
14. Ruf, E.: Effective synchronization removal for Java. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), New York, NY, ACM Press (2000) 208–218
15. Dwyer, M.B., Hatcliff, J., Robby, Ranganath, V.P.: Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. Formal Methods in System Design **25**(2-3) (2004) 199–240
16. Pickett, C.J.F., Verbrugge, C.: Return value prediction in a Java virtual machine. In: Proceedings of the Value-Prediction and Value-Based Optimization Workshop. (2004) 40–47
17. Pominville, P., Qian, F., Vallée-Rai, R., Hendren, L., Verbrugge, C.: A framework for optimizing java using attributes. In: Proceedings of the International Conference on Compiler Construction (CC). Lecture Notes in Computer Science, Springer (2001) 334–554