Exploiting Reflection to Add Persistence and Query Optimization to a Statically Typed Object-Oriented Language

Gökhan Kutlu J. Eliot B. Moss

Department of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
{kutlu,moss}@cs.umass.edu

Abstract

It is popular and appealing to design and construct a persistent programming language by extending the semantics of a non-persistent language appropriately and then modifying its compiler and run-time system to implement the extended semantics. We describe here how to achieve this, and furthermore, how to support query optimization, which is typically neglected in persistent programming language implementations, through judicious exploitation of reflection. Significantly, we avoid modifying the structure of the source language or its compiler in any way, and minimize and localize the modifications to the run-time system. We work in the context of the Java programming language, and conclude that the key features required in our approach are: a typed intermediate representation (as provided by Java class files), reflection supporting code inspection (an extension to the standard Java virtual machine), and dynamic loading of code generated at run-time. We also require virtual machine extensions to support read and write barriers and to trigger our reflective optimization and code generation. Further, we argue that optimization at the reflective level can remove much of the overhead of the read and write barriers.

1 Introduction

We are interested in building a *database programming language* (DBPL) by minimal extension of an existing (statically typed, object–oriented) language. A DBPL is a *persistent programming language* (PPL) that also provides one or more *collection* types and supports the processing (optimization) of queries on collections. A PPL is a programming language in which created objects are persistent—they continue to exist and retain their values between runs of a program. The data are kept in a *persistent store*, which the programmer views as an extension of volatile memory. The language implementation and run-time system work together to make persistent data memory-resident on demand, and to propagate the program's modifications of persistent data into the persistent store. As shown in Figure 1, one appealing way to achieve the transition from programming language to database programming language is to extend the language with persistence and query processing.

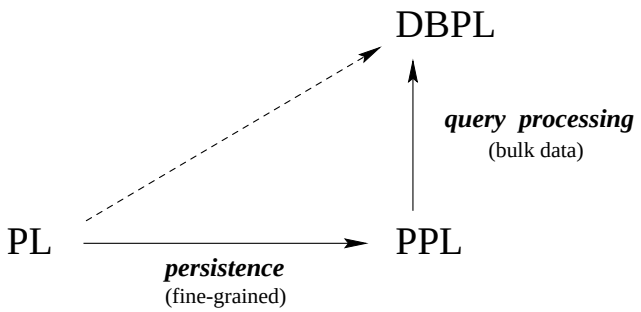

Figure 1: Transition from programming language to database programming language.

Supporting a PPL (persistence) demands implementation of at least the following:

Movement/translation of objects to/from the persistent store. Objects need to be converted into the persistent store format before they are written. Similarly, they need to be translated into the language's heap object format as they are read in from the store. We call these transformations *packing* and *unpacking*[1] (see Figure 2). Since our persistent store (Mneme [Moss, 1990]) uses OIDs to reference objects, these OIDs may need to be *swizzled*—converted into direct memory pointers—as part of unpacking. Pointer references are *unswizzled* during packing.

Read barrier. Not all objects referenced within a program will be resident at all times. A *read barrier* detects uses of references to non-resident objects (object faults) and triggers actions to make these objects resident.

Write barrier. A *write barrier* records which objects have been modified since the last update. At a checkpoint, only modified objects need to be packed and written back to the store.

Transaction/checkpoint model. A transaction/checkpoint model dictates the way in which and when modifications to persistent objects are made permanent (committed) in the persistent store. We do not consider such models in detail.

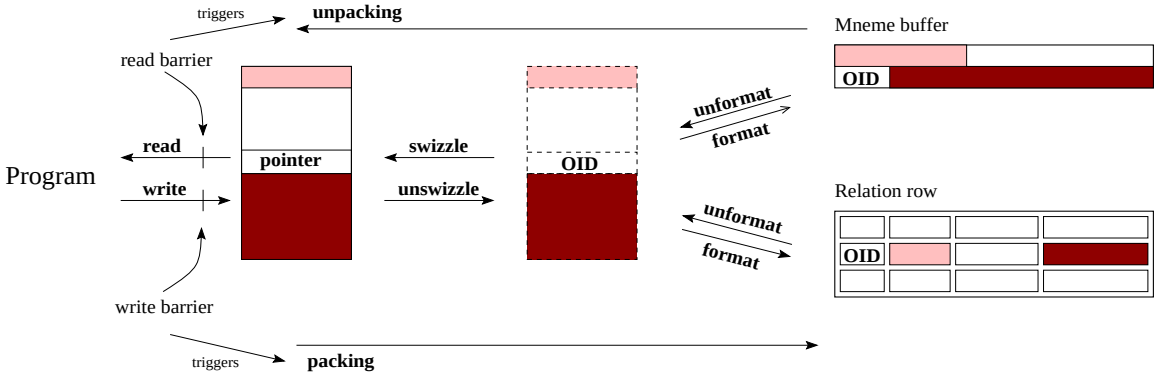

Figure 2: Read and write barriers trigger packing and unpacking.

Supporting query optimization, on the other hand, requires an understanding of the following:

The operation to be performed. Because the query is defined imperatively, its intention (meaning) has to be inferred from an analysis of the program, which requires access to (seeing) the query code.

The data operated upon. We need to know the (static/dynamic) type of the data in order to identify possible optimizations, and the objects being queried (e.g., collections) so as to know the specific associated operations and tools available for use in optimization, such as indices.

The language, its compiler, or both can be extended to provide the kinds of support listed above. Unfortunately, modifying the language and its compiler this way is a tedious and painful process. Besides, the code written in the extended language is not readily portable.

1.1 Reflection

The term *reflection* refers to the ability of a program to observe or change aspects of its own behavior and execution environment at run time. A programming language is said to be *reflective* when it provides its programs with reflection. The ability of a program to generate new program fragments and integrate these into its own execution is known as *linguistic reflection* [Stemple *et al.*, 1992]. In *compile-time* linguistic

[1]The metaphor is that of an object being packed (arranged, enclosed) into a case or box upon write, and unpacked on read; it does not imply compression.

reflection, reflective constructs are compiled and executed during compilation. When generation of new code takes place at run-time, the process is termed *run-time* linguistic reflection. A good example of run-time linguistic reflection is the use of a run-time callable compiler together with the ability to bind and execute newly compiled program fragments within the running program [Stemple *et al.*, 1993]. With *behavioral reflection*, a program alters its own meaning by changing the behavior of its interpreter.

Reification is the process by which aspects of a program, which were implicit in the translated program and the run-time system, are brought to the fore using a representation (data structures, procedures, etc.) expressed in the language itself and made available to the program, which can inspect and modify them as ordinary data [Malenfant *et al.*, 1996]. Linguistic reflection is concerned with the ability of the language to reify both the program (code) currently executing as well as its data types. Behavioral reflection, on the other hand, is concerned with the ability of the program to reify its own execution state and that of the (abstract) interpreter on which the program is running.

1.2 Java

Java[2] is a class-based, statically typed, object-oriented programming language [Lindholm & Yellin, 1997]. The Java compiler generates *bytecode*s for the Java *virtual machine* (JVM) [Gosling *et al.*, 1996]. Bytecodes are typed abstract machine code executed by the Java interpreter and run-time system that implement the JVM. The Java compiler compiles each Java class definition written in the Java programming language into a *class file*, which contains a symbol table of type definitions and method bytecodes. The class file contains adequate information for strong type checking upon loading. The JVM can preload the class file, or load it dynamically on demand. The Java heap is managed automatically, by a garbage collector.

Each class has a set of associated literals and constants, such as the names of classes instantiated in the methods of the class and constant values used. These constants are kept in a *constant pool*. The class constants have to be resolved to the objects that they represent before they can be used. A class name, for example has to be converted to a reference to the class object. This process of binding constants to their corresponding objects is called *resolution*.

Java is statically type-checked (i.e., at compile- and load-time, in advance of execution), but allows dynamic loading of classes. This nice tension between static and dynamic behavior allows interesting features to be realized in Java. In particular, it is possible to support reflection in the language using dynamic class (method) generation and loading. Reflection, in turn, lends itself to interesting implementation of functionalities such as persistence and query optimization.

1.3 Overview

In this paper, we describe our planned implementation of persistence and query optimization using reflection without modifying Java, its compiler, or its class file format. Java currently does not support reflection (in the sense in which we use the word). Extending Java with reflection will imply changing the virtual machine to some extent, but we will keep these changes minimal. Also, we will add new bytecodes to the original bytecode set, and, with the help of reflection, insert them appropriately into Java code to help implement read and write barriers.

Reflection supports more transparent and general implementation of persistence, because it allows automatic generation of code that carries out the transformation between memory and disk data formats, and incorporation of that code into the run-time system. Neither a compiler nor a pre-processor needs to be involved.

Reflection can provide support for query optimization and also eliminate the need to extend the base language with incompatible (query language) features. Reflection does not require new language constructs. We can build reflection in the virtual machine over Java's existing reflective package, and use it to provide

[2] Java is a trademark of Sun Microsystems, Inc.

query optimization. Furthermore, with run-time reflection, the optimizer can take advantage of the run-time specifics of the data store, such as the existence of indices, the clustering of data on disk, the distribution of data among the sets in the data store, or values of parameters to the query.

This paper is structured as follows. Section 2 reviews related work. In Section 3, we describe our extensions to the Java VM. Sections 4 and 5 discuss object faulting and object updates. Section 6 describes our query optimization model. Extending Java with powerful reflective capabilities raises several issues. We discuss these issues in Section 7 and conclude in Section 8.

2 Related work

The work presented in this paper relates to work on implementing persistent programming languages, for example, E [Richardson *et al.*, 1993], Persistent Smalltalk [Hosking, 1995], PJama [Atkinson *et al.*, 1996], etc. Our work differs from all the above in that it applies a new technique to implement persistence, namely reflection. None of the persistent programming languages above support query optimization.

Stemple, et al. [Stemple *et al.*, 1992] are the first to mention the use of reflection as a viable technique for query optimization. Recently, Kirby, et al. [Kirby *et al.*, 1998a] describe how linguistic reflection may be provided in Java and used to implement natural join [Kirby *et al.*, 1998b]. Similarly, reflection has been used to implement new functionalities in programming languages, such as atomic types [Stroud & Wu, 1995], and fault tolerance [Fabre *et al.*, 1995].

We take advantage of compiler optimization techniques to build a query optimizer. In particular, we use loop optimizations to transform join loops in a query. Lieuwen [Lieuwen, 1992] has used similar techniques to transform loops in an object-oriented database programming language.

The use of Java's dynamic, late binding nature as an opportunity for improving program performance is similar to the approach taken in Self [Ungar & Hözle, 1987]. In Self, methods that are used frequently are dynamically re-compiled to optimize their execution.

3 Reflection Extensions

Java supports limited reflection,[3] allowing access to the class meta-information via the `Class` class. `Class` objects provide access to (public) field and method definitions via the `Field` and `Method` classes. The `Field` class provides access to the name, type, and modifiers of a field, and allows reading or modifying the corresponding field in an instance of the class. Similarly, Java provides means to access the definition of a class method through the `Method` class. A `Method` object provides access to the name, parameter types, and return types of a method. It also allows invoking the method, but it does not allow access to its bytecodes.

We are extending Java with true linguistic reflection, that is, the ability to change the definition of a class and its methods at run-time (or load-time). Furthermore, we provide behavioral reflection by allowing changes to the virtual machine execution state.

As shown in Figure 3, the reflective loop consists of class reification, editing, and reflection. The reified representation for a class, field, and method is an instance of the extended reflection support classes `XClass`,[4] `XField`, and `XMethod`, respectively. Unlike the ones provided by Java, these classes, along with their corresponding editors (`XClassEditor`, `XFieldEditor`, and `XMethodEditor`), allow the program to add, remove, or modify fields and methods via class editing. The program commits these changes to the class run-time definition via reflection.

Below, we describe in more detail the various phases a method goes through during the reflective process.

[3] This is in fact a misnomer for Java's package, because Java provides mostly reification and only limited reflection.

[4] The prefix `X` stands for e`X`tended.

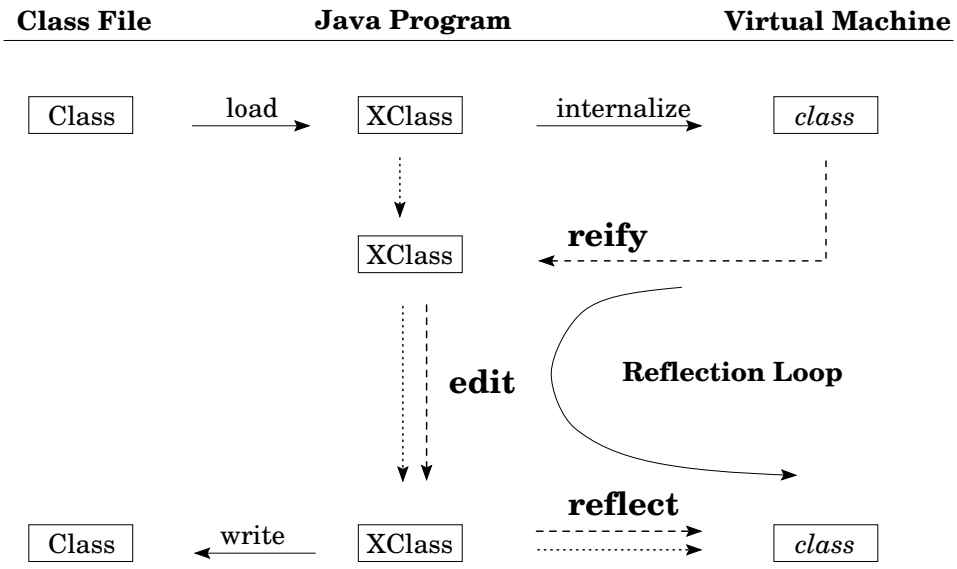

Figure 3: Run-time and load-time reflection.

3.1 Reification: Code Inspection

In order to work on a Java method at the program level, we need access to a representation of the method. As mentioned above, we allow an extended reification of method bytecodes via the `XMethod` class. `XMethod` provides essentially all of the information found in the class file. In particular, it includes method bytecodes and a handle to the class constant pool. The `XMethodEditor` object puts the method bytecodes in a form that supports the inspection, analysis and optimization, and/or transformation of a method, and production of a new class (file) with the new method.

3.2 Editing: Code Generation

The `XMethodEditor` class supports creation and editing of `XMethod` objects. Among the operations supported are local variable creation, and bytecode insertion, deletion, and replacement. The editor commits the generated or modified method back into the `XClassEditor`, which, in turn, commits the updated class via reflection.

3.3 Linguistic Reflection: Code Insertion

Once a new method is generated dynamically, it needs to be reflected back into the run-time environment. One way to implement reflection in Java is to clone (at run time) the original class, extend it with the method, and load it via the usual loading mechanism to overwrite the previous definition. The Java VM supports dynamic loading of classes, provided as a sequence of bytes, typically (but not necessarily) originating from a class file, so the new class definition may be written to a file or buffer before being loaded.

It is also possible, though more difficult, to extend the original class incrementally with the new method. Java does not allow incremental extension of class definitions. However, it may be more efficient to add a new method into a class rather than generate a new class and load it in its entirety. This requires adding to Java support for incremental class extension via code insertion.

We support code insertion via the `XClassEditor` class. An instance of this class acts as an editor for a particular class at run-time. As described above, the program uses the `XMethodEditor` to create a new `XMethod` instance from its bytecodes and constant pool handle. It then commits the edits by passing the `XMethod` to the class editor, which internalizes the method.

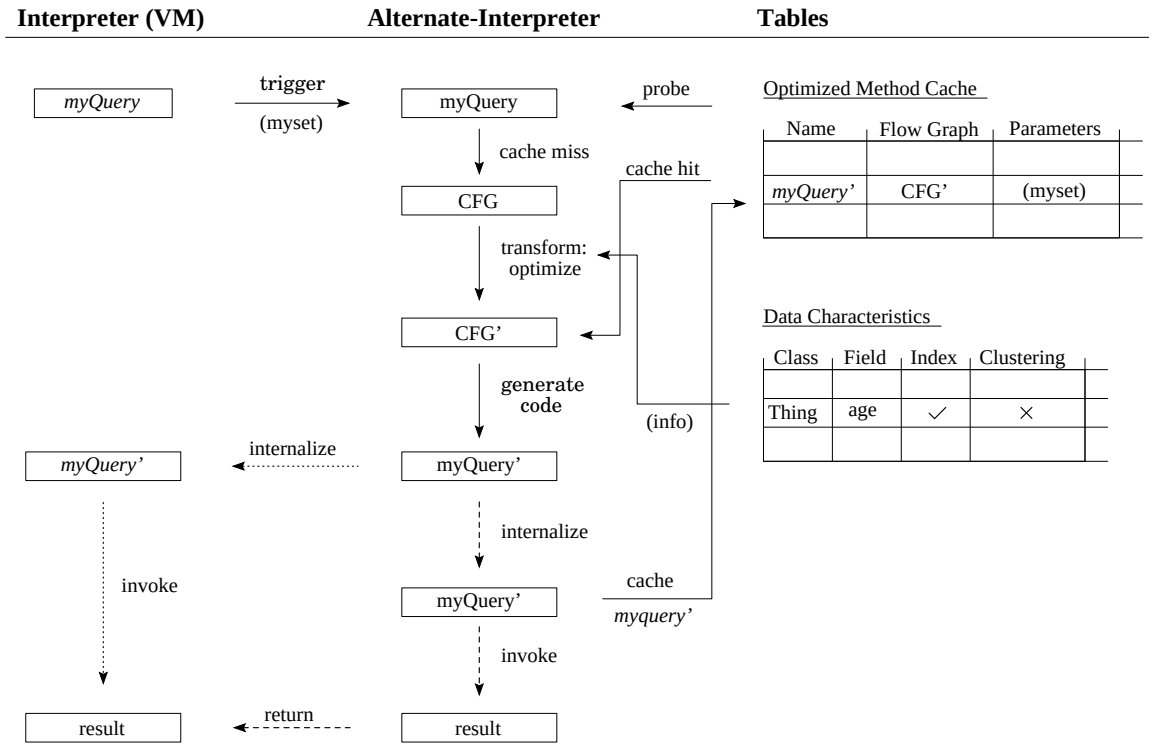

Figure 4: Alternate interpreter and optimization example.

The incorporation of a new method into the definition of a class is similar to loading a method from the class file. In particular, the VM verifies the method and (possibly later) resolves its constants. During verification, the method is type checked to make sure that its argument and return types are valid, any returned object matches the return type, and so on. Once the new method passes all checks, it is ready for execution using the existing invocation mechanisms.

3.4 Behavioral Reflection: Alternate Interpreter

We extend Java with behavioral reflection via the *alternate interpreter* mechanism. In our run-time model, each method has an associated interpreter. For most methods this is the base (built-in) interpreter and nothing unusual happens when the method is invoked. However, one can associate a (suitably typed) Java method as an *alternate* interpreter. When the original method is invoked, the alternate interpreter code is run, passing the original method's code (as an `XMethod`) and the original arguments. The alternate interpreter can proceed in an arbitrary fashion, and whatever it returns is returned as the result of the original method invocation.

As shown in Figure 4, the optimizing alternate interpreter receives an `XMethod` object (the method code) and arguments, transforms the code, and incorporates the new method into the run-time environment of the already running program through the reflective mechanisms described above. The alternate interpreter then executes the method and returns the result as the result of the original. It also caches information about the optimization, to reduce effort on optimizing the same or similar calls in the future.

The only requirement is that the return type and argument types are the same as those of the original method. Ideally, the alternate interpreter executes code that is semantically equivalent to the original, although, one can think of degrees of permission that can be granted by the programmer, to give the interpreter more flexibility.

Annotations provided with the class file can be used to assign interpreters to methods. Alternatively, interpreters can be assigned through an analysis step. In the query optimization example, any of the triggers

mentioned in Section 6 will result in the optimizing interpreter being assigned the method.

3.5 Code Transformation Using Alternate Interpreter

A special form of editing is code transformation. We assign a method we want transformed to a transforming interpreter for execution. In this paper, we focus primarily on an interpreter that optimizes queries, but this idea is easily extended to, for example, code optimizations that exploit the target machine architecture (e.g., *just in time* (JIT) compilation). In the extended model, one can think of the program running on a number of such specialized optimizing interpreters and being passed from one interpreter to another according to the type of optimization or transformation that gets triggered.

3.6 Reflection Issues

Internalization of transformed methods at run-time raises a number of issues. First, active invocations of the original method being transformed need to be completed even if the new one is in effect. This requires keeping the environments in use by these invocations active until they all finish executing. This means that multiple versions of a method may be available and may need to be managed. Once completed, all versions of the method except the most recent one would be reclaimed.

Second, we have to deal with type (class) evolution within an object-oriented programming language. This is an interesting context in which one can explore versioning. We need a mechanism to trap uses of old objects and an object substitution operator to convert them to new formats. Similarly, one must be careful with the subclasses of a modified class, since they are bound to changes in the superclass.

Third, the optimized method cache manager needs to be able to evict an entry associated with a method that is no longer valid.

4 Object Faulting

The Java run-time system loads persistent data on demand, using the program's attempts to access non-resident data as triggers for retrieval. In order to access a persistent data item, the program needs to dereference a *pointer*. Traversing a pointer to a non-resident object causes (via the read barrier) an *object fault*, which triggers the retrieval of the object. As described below, the methods that carry out object packing and unpacking during retrieval are generated automatically via the reflection mechanism.

Although it incurs a time penalty, the translation can carry out conversions between foreign and native formats. This is especially valuable when the object may be from a remote source, or a different data model (e.g., relational). A further generalization of this idea is to allow the user to choose the source and target databases for their program data. The programmer could for example, replace the Mneme client with a client of a different database, or access multiple databases.

4.1 Residency check bytecode

Supporting object faulting requires implementing the read barrier, i.e., detecting references followed to non-resident objects. We currently implement a read barrier via a *residency check* on each pointer followed to see if it points to a valid object. We introduce a `ResidencyCheck` bytecode, which ensures an object is resident, faulting if necessary. This simple barrier is coded inline at load time into the method code using method editing.[5]

The `ResidencyCheck` is inserted after every bytecode that pushes an object reference onto the stack. If the reference is an OID, the bytecode triggers faulting via a callback to the (Java) `getObject` method

[5]We currently use editor classes provided by the BLOAT [Nystrom & Hosking, 1998] package. We will use the `XMethodEditor` class instead once it is implemented.

of the `PersistenceManager` class. `getObject` interacts with Mneme to retrieve the object with the specified OID, and uses its type information to create an empty Java object. The (Java) object constructs itself by unpacking the Mneme object. The memory pointer to the Java object is swizzled (pushed back into the stack).

4.2 Object Reading

In order to unpack and swizzle an object, the run-time system must know the object's type structure.[6] One way to provide this information to the run-time system is to have the object reveal its structure via a method that is generated by the compiler or a preprocessor, or hand-coded by the programmer. This technique does not require any changes to the run-time system, but has to be carried out for every persistent class. Moreover, the methods have to be re-generated every time the type definition changes.

Alternatively, the information Java provides via reflection can be used to iterate over the type information of an object, translating or swizzling its fields one by one. This interpretive technique, based on run-time type reflection, is inefficient, however, because it requires examining the type definition of an instance every time, and it makes many Java method invocations.

Instead, we generate (transparently) a method for each class that is called to carry out unpacking. We support two ways of doing this. In the approach we call *load-time reflection*, we extend each class with unpacking methods generated automatically at load-time and inserted into the class definition via the reflective mechanism described in Section 3.3.

In our *just-in-time reflection* approach, we further eliminate the cost of generating methods for classes having no persistent instances. We do this by delaying the generation of unpacking methods until such a method is invoked. In this scheme, a class does not have an unpacking method when first loaded, but inherits one from `Object`. The job of the `Object` unpacking method is to generate and insert the unpacking methods into the class definition of the object originating the call. The first call to an unpacking method invokes the one in the `Object` class, thereby triggering the generation of the methods in the object's actual class. Subsequent calls to the unpacking methods will invoke the generated methods.

4.2.1 Overriding the standard translation

The programmer can override the standard mappings that govern object packing/unpacking. In particular, one can imagine this as an opportunity for support of non-standard mappings, such as the compression/decompression (one-to-one) of an object, the compaction/decompaction of a set object (many-to-one), or the linearization of the subcomponents of a larger object (one-to-many).

A subclass method can use the generated packing and unpacking methods for its superclasses and provide code only for its own fields:

```
class Subclass {
    ...
    public void unpack () {
        super.unpack();
        ... unpack self ...
    }
    ...
}
```

[6] Type information is needed for locating pointers, possible length and format conversions, type-checking, etc.

4.2.2 Multiple object stores

We intend to support bindings to more than one object store simultaneously. For this, we need to be able to determine which store an object originated from just by looking at its OID. Currently, the two most significant bits in an OID are available and can be used to distinguish up to four data sources/targets. We could use more general OID mappings to make this more flexible in the future.

5 Object Updating and Flushing

At certain points in its execution, a program may invoke a checkpoint operation to make permanent all modifications it has made to persistent objects. Updating modified objects at a checkpoint involves unswizzling and packing (copying) modified (subranges of) objects and newly-created objects back to the store (see Figure 2). The transformation on write back is carried out by packing methods automatically generated for the object's class. As in the read case (see Section 4.2), it is possible to delay generation of these methods until their first use.

In order to minimize the number of objects examined (considered) for packing, we use a simple write barrier. We extend the Java bytecode set with a `NoteUpdate` bytecode. Upon a write into one of the fields of a persistent object, the `NoteUpdate` bytecode enters the modified object[7] into a *remembered set* [Ungar, 1984].[8] We insert a `NoteUpdate` bytecode before each object field write or array store bytecode. For object-based remembering schemes, a block of writes to the same object results in the insertion of unnecessary bytecodes. It is possible to collapse several such `NoteUpdate` bytecodes into one. At checkpoint time, we need to scan only remembered set entries to identify all the modified objects.

After checkpointing makes the memory and store copies of data consistent, *flushing* may be used to reclaim persistent objects or release object locks. Flushing involves tracing the stack and transient (heap) objects and unswizzling references to persistent objects. Flushing can be full or partial. We can unswizzle all persistent space, or, e.g., limit our attention to the object graph rooted at a given object. Once we have unswizzled all references to a set of resident persistent objects, ordinary heap garbage collection will reclaim the space they consumed.

Although we will not go into detail here, we should mention that we may also need features to support multi-user access, such as a `Lock` bytecode to acquire read or write privileges to persistent objects.

6 Query Optimization

Note that our goal is to demonstrate that query optimization can be supported in a PPL with the help of reflection. Therefore, our optimizations provide primarily proof of concept.

Optimization starts with an alternate interpreter invocation. A number of ways are possible for setting up the triggers. For example, class file annotations can be used to set triggers on (certain) methods. The method code and arguments are passed to the optimizing interpreter (see Figure 4). The interpreter analyzes the query code, and its operands and their properties, to identify possible improvements. It extracts the query into an abstract query plan representation and applies transformations to the plan to yield a semantically equivalent but improved plan in terms of performance. It then converts the optimized query plan into an executable sequence of bytecodes.

The interpreter creates a new method with the transformed code as its body and inserts (reflects) it into the class definition of the object. The method is also cached, indexed by the values of various database parameters and operands used in the optimization (similar to caching optimized queries in database man-

[7] Although we remember objects, one can also remember slots, i.e., record the modified slot of an object in the remembered set.

[8] Card marking [Sobalvarro, 1988] or other representations of the set of modified (portions of) objects are also valid implementations of the `NoteUpdate` bytecode.

agement systems). The cached method can be used next time if these values have not changed to render the optimization invalid.

6.1 Kinds of analyses

We focus on queries expressed as a (possibly nested) loop iteration over one or more collections. We analyze bytecodes of a method to identify the query loop structure and build an abstract query representation amenable to transformations. We identify loops in this representation and analyze it for loop-invariant code elimination, index identification, loop tiling, etc. In particular, we reorder nested loops in order to take advantage of relational join optimization algorithms available. This is possible, it turns out, when the loop statement(s) satisfy self-commutativity constraints [Lieuwen, 1992]. Consider the following simple group-by loop (expressed in pseudo-code):

for (x_1 **of** Set_1) **suchthat** ($P_1(x_1)$)
...
for (x_m **of** Set_m) **suchthat** ($P_m(x_1, \ldots, x_m)$)
Stmt;

The statement Stmt is defined to be *self-commutative* relative to $x_1, x_2, \ldots, x_m$ if the code segment above and the code segment below reach an identical, deterministic state from any starting state and under any order of binding of variables:

for (x_1 **of** Set_1; ... ; x_m **of** Set_m) **suchthat** ($P_1(x_1)$ && ... && $P_m(x_1, \ldots, x_m)$)
Stmt;

If Stmt is self-commutative, Lieuwen writes the above statement as a join followed by a sort. Self-commutativity can be identified easily. A statement is self-commutative if any two adjacent instantiations of it can be permuted without changing the final computation of the program.

6.1.1 Bytecode versus Source Code

While finding loop patterns in general may be quite hard, finding them from code written in a somewhat stylized or idiomatic way should be much easier. On the other hand, there is little or no difference between finding these patterns in source code versus bytecodes. Bytecodes are a nearly complete representation of the source code form, differing only in minor aspects such as the availability of variable names, line numbers, etc. All the typing and control flow structure is there.

6.2 Example

Figure 5 shows a sample query written in Java, and its optimized version, which uses an index. The `Enumeration` *e* is initialized to the first set item at the beginning of the loop and the `nextElement` operator allows iteration through the set elements. In this case, the optimizer identifies an index on the query attribute `Thing.age` and uses it to speed up the query. Using the index, the original full scan of a possibly large set of objects is reduced to an iteration over a selected subset of the original set.

7 Additional Directions

7.1 Optimizing read/write barriers

We plan to analyze residency checks to eliminate redundant ones. For example, if an object remains resident once it is faulted, then many residency checks will be redundant. Suitable data flow analysis should help remove many such checks. The exact rules depend on the contract between the optimizer and the VM

Before:

```
Enumeration e = myset.elements();
while (e.hasMoreElements()) {
   Thing elt = (Thing) e.nextElement();
   if (elt.age >= 13 && elt.age <= 19)
      do stuff with elt
}
```

After:

```
Enumeration e = myset.indexEnumerateIntegerRange ("age", 13, 19);
while (e.hasMoreElements()) {
   Thing elt = (Thing) e.nextElement();
   do stuff with elt
}
```

Figure 5: A set iteration loop in Java before it is transformed and after.

regulating when the VM can flush resident objects. Hosking, et al. [Hosking *et al.*, 1998] have explored possible contracts and measured their performance impact.

7.2 What about JIT compilers?

Query optimization as described above is a form of just in time optimization. It is possible to conceive of JIT compilation performed by yet another interpreter. The concepts are essentially the same, although the JIT compiler outputs native code, while the interpreter stays with Java bytecodes. It is always possible to combine the optimization phase with native code generation to improve performance further.

Code generated at load-time is seen as ordinary code to a JIT compiler. JIT compilers do not cause a problem for persistence if the residency checks are implemented via special bytecodes. These bytecodes would be translated as part of compilation. The alternate interpreter concept limits the optimizations a JIT compiler can do with respect to method dispatch, since the compiler cannot inline a call if the binding between methods and their interpreters can be changed dynamically. The JIT compiler can be informed if the binding is static and supported more directly by compiler.

7.3 Interactions With the Garbage Collector

The garbage collector must understand OIDs and not follow them as references. We currently use the least significant bit as a flag to indicate that a reference is an OID. It is very easy to incorporate a filter into the collector that ignores OIDs encountered while tracing roots and heap objects.

The garbage collector can help provide support for the write barrier, object write back, and swizzling and unswizzling. At the end of a checkpoint, all inter-object (heap) references and stack references need to be unswizzled. This requires a stack and heap scan, a process also used in garbage collection. The checkpoint operation can be optimized if unswizzling can be combined with (performed during) the heap and stack scan phases of garbage collection. Similarly, during a heap and stack scan, the garbage collector can help swizzle references to objects that are made resident. Finally, the collector can guarantee that inaccessible objects are written back to disk by calling their pack methods before deallocating them. One might also extend or exploit Java finalization mechanisms to help with writing back inaccessible resident persistent objects.

The reflective mechanisms described above can also be used to generate helper methods for the garbage collection at load time. For example, a method that iterates through the references of its class instances would be helpful in heap scans.

We need to research further the interactions of our mechanisms with Java finalization in main memory and the store. Zigman and Blackburn [Zigman & Blackburn, 1998] have considered this issue in more detail. We hope that a store collector can be written in Java exploiting our reflective mechanisms.

8 Summary and Conclusions

We described reflection as an approach for implementing persistence and query optimization in a programming language without modifying the compiler and with minor modifications to the run-time system. We suggested approaches to extending Java with more powerful reflective capabilities such as class, field, and method creation, modification, and removal. We argued that judicious use of linguistic reflection can also help support implementation of read and write barriers, and use of behavioral reflection can allow support for transparent query optimization.

References

[Atkinson *et al.*, 1996] M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM Sigmod Record 25*, 4 (December 1996), 68–75.

[Fabre *et al.*, 1995] J. Fabre, V. Nicomette, T. Perennou, R. J. Stroud, and Z. Wu. Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming. In *Proceedings of the 25^{t}h IEEE Symposium on Fault Tolerant Computing* (1995).

[Gosling *et al.*, 1996] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Hosking *et al.*, 1998] A. Hosking, N. Nystrom, Q. Cutts, and K. Brahnmath. Optimizing the Read and Write Barrier for Orthogonal Persistence. To appear in *8th International Workshop on Persistent Object Systems (POS8)*, Tiburon, CA, August 1998.

[Hosking, 1995] A. L. Hosking. *Lightweight Support for Fine-Grained Persistence on Stock Hardware*. PhD thesis, Computer Science Department, University of Massachusetts at Amherst, 1995.

[Kirby *et al.*, 1998a] G. N. C. Kirby, R. Morrison, and D. W. Stemple. Linguistic Reflection in Java. *Software – Practice & Experience 28*, 10 (1998), 1045–1077.

[Kirby *et al.*, 1998b] G. N. C. Kirby, R. Morrison, and D. W. Stemple. Linguistic Reflection in Java: A Quantitative Assessment. In *Proceedings of the 5th International IDEA Workshop, Fremantle, Western Australia* (1998).

[Lieuwen, 1992] D. F. Lieuwen. *Optimizing and Parallelizing Loops in Object-Oriented Database Programming Languages*. PhD thesis, University of Wisconsin — Madison, 1992.

[Lindholm & Yellin, 1997] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[Malenfant *et al.*, 1996] J. Malenfant, M. Jacques, and F.-N. Demers. A Tutorial on Behavioral Reflection and its Implementation. In *Proceedings of the First International Conference Reflection '96* (San Fransisco, CA, April 1996), pp. 1–20.

[Moss, 1990] J. Eliot B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems 8*, 2 (April 1990), 103–139.

[Nystrom & Hosking, 1998] N. J. Nystrom and T. Hosking. BLOAT: Bytecode Level Optimization and Analysis Tool. Department of Computer Sciences, Purdue University, 1998.

[Richardson *et al.*, 1993] Joel E. Richardson, Michael J. Carey, and David T. Schuh. The Design of the E programming Language. *ACM Transactions on Programming Languages and Systems 15*, 3 (July 1993), 494–534.

[Sobalvarro, 1988] P. Sobalvarro. A Lifetime-Based Garbage Collector for Lisp Systems on General-Purpose Computers. Technical Report AITR-1417, MIT, AI Lab, February 1988.

[Stemple *et al.*, 1993] D. Stemple, R. Morrison, G. N. C. Kirby, and R. C. H. Connor. Integrating Reflection, Strong Typing and Static Checking. In *Proceedings of the 16th Australian Computer Science Conference* (Brisbane, Australia, 1993), pp. 83–92.

[Stemple *et al.*, 1992] D. Stemple, R. B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G. N. C. Kirby, L. Fegaras, R. L. Cooper, R. C. H. Connor, M. P. Atkinson, and S. Alagic. Type-Safe Linguistic Reflection: A Generator Technology. Technical Report FIDE/92/49, ESPRIT BRA Project 3070 FIDE, 1992.

[Stroud & Wu, 1995] R. J. Stroud and Z. Wu. Using Metaobject Protocols to Implement Atomic Data Types. In *Proceedings of the European Conference on Object-Oriented Programming* (1995), pp. 168–189.

[Ungar, 1984] D. M. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices 19*, 5 (April 1984), 157–167.

[Ungar & Hözle, 1987] D. M. Ungar and U. Hözle. Self: The Power of Simplicity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (1987), pp. 227–241.

[Zigman & Blackburn, 1998] J. N. Zigman and S. M. Blackburn. Java Finalize Method, Orthogonal Persistence and Transactions. To appear in *3rd International Workshop on Persistence and Java (PJW3)*, Tiburon, CA, September 1998.