Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations

Ilya Lipkind, Igor Pechtchanski, and Vijay Karamcheti

Courant Institute of Mathematical Sciences
New York University
715 Broadway, 7th Floor
New York, NY 10003

{lipkind, pechtcha, vijayk} @ cs.nyu.edu

ABSTRACT

Object-based parallel and distributed applications are becoming increasingly popular, driven by the programmability advantages of component technology and a flat shared-object space. However, the flat shared-object space introduces a performance challenge: applications that rely on the transparent coherent caching of objects achieve high performance only on tightly coupled parallel machines. In distributed environments, the overheads of object caching force application designers to choose other solutions. Consequently, most applications sacrifice programmability, relying instead on either the explicit coherence management of cached objects, or on vastly different middleware abstractions such as multicast and events.

In this paper, we describe object views — language support for efficient object caching in parallel and distributed computations. Object views specify restrictions on *how* computation threads can use an object, providing the underlying implementation with information about the potential side effects of object access, and thereby enabling construction of scalable, low-overhead caching protocols customized to application requirements. We present extensions to the Java programming language for expressing object views, and describe the design and implementation of a translator and run-time system for executing view-augmented Java programs on a distributed cluster of workstations. Experimental results based on a shared whiteboard application demonstrate that view-based object caching can achieve performance superior to multicast- and event-based implementations, while retaining essentially a shared object interface.

Keywords

Distributed Objects, Object Caching, Java, Shared Objects, Views, Collaborative Applications, Object Representation.

1. INTRODUCTION

Sequential object-oriented languages and component technologies have become widespread because they allow abstraction, modularity, and reuse. These same advantages also make object technology very useful in parallel and distributed processing, particularly in light of the growing popularity of parallel servers and distributed network-based services. Parallel applications can leverage encapsulation to separate core logic from the details of resource management, data distribution, and concurrency decisions. Similarly, the construction of large-scale distributed applications is simplified by relying upon component-based middleware to orchestrate remote interactions. However, realizing these potential benefits requires overcoming some fundamental performance challenges.

One of the primary challenges in both parallel and distributed object environments is that of ensuring data locality. Objects must be placed (statically, or dynamically via caching) to minimize the amount of communication incurred for their access. Bad placement or large caching costs can often limit application scalability. Traditional approaches for dealing with this problem fall into one of two extremes based on the amount of application knowledge they rely on. The first approach, typically adopted in parallel systems, is more or less transparent to the application developer, relying on application-independent coherence schemes such as entry consistency [5,10,17,19] to guarantee that each object access sees consistent state. Coherence actions are triggered on demand, and, in the absence of knowledge about how the application accesses the object, tend to take conservative actions such as keeping the entire object state consistent after an update and/or serializing potentially conflicting accesses. The consequence is that applications do not scale very well and achieve good performance only on tightly coupled parallel machines.

The second approach, typically adopted in distributed environments, relies on explicit specification of placement and caching policies. The designer considers the implementation behavior of the objects (breaking encapsulation) and a model of their usage to make these decisions. Although such systems perform well, these performance advantages come at the cost of programming complexity. In addition, *ad hoc* placement and caching decisions typically do not translate to good performance in heterogeneous environments where resource characteristics change dynamically. The same tradeoffs between performance, programming complex-

ity, and platform dependence of placement decisions also hold true for applications relying on multicast or event-based middleware abstractions built on top of remote method invocations. The fundamental problem with all of these custom solutions is that in order to achieve efficiency on diverse architectures such as small-scale tightly coupled parallel machines and large-scale loosely coupled distributed environments, they require radically different application structures. Such divergence in the application code base complicates program development as well as subsequent maintenance.

In this paper, we present a new programming and execution abstraction called *object views*, that allows a single program specification, written assuming a shared object space, to achieve good performance across a wide range of target platforms. This work is motivated by the recognition that the key limitation preventing the transparent use of efficient object caching schemes is a lack of information about *how* the object is accessed by computation threads. Object views are an attempt to introduce minimal language support to enable the application developer to specify such usage information. Object views can be thought of as generalized interfaces, which, in addition to advertising object functionality, also specify restrictions on its use by computation threads (e.g., a thread can only invoke a subset of the methods supported by the object). All accesses to the object are through its views; from the perspective of the computation threads, the latter are virtually indistinguishable from regular objects. The usage information contained in object views provides the underlying implementation with information about potential side effects of object access, facilitating construction of scalable, low-overhead caching protocols customized to application requirements. For example, information about the subset of methods invoked by a thread allows the implementation to cache a version of the object that only maintains consistency on the object fields accessed by these methods. In general, object views improve caching performance by reducing the amount of object state that needs to be transferred, by incurring fewer invalidations/updates, and by increasing the number of concurrent operations. Moreover, object views also provide an elegant abstraction for expressing application-specific caching optimizations, while retaining the intuitively simple programming interface of a cache-coherent global shared object space.

Object views can be supported in an existing object-oriented language with minimal extensions. We describe these extensions in the context of the Java programming language. The extended language, VJava, introduces two new keywords: **`view`** and **`represents`**. The first keyword declares views as class-like structures, and the second indicates correspondence between view functionality and fields and methods of objects; this correspondence expresses restrictions on object usage. We also describe the design and implementation of a translator, which converts VJava programs into base Java that uses JNI calls to access a software shared memory layer that provides composable primitives for building custom coherence protocols [11]. The run-time environment involves independent JVMs running on each of the nodes of a distributed cluster of workstations, sharing objects using the software shared memory layer.

To assess the convenience of using these extensions and quantify their performance impact, we have also built a collaborative whiteboard application. The collaborative environment is a shared space of graphical objects that is simultaneously accessed by many different users who modify, add, and delete objects according to various usage patterns. Object views help reduce coherence traffic by allowing consistency to be maintained at the granularity of the subsets of object state that are actually accessed by computation threads. For example, a thread inspecting a whiteboard object at a coarse-level need not interfere with another thread that might be modifying the internal state of the object. Our experiments, using several parameterized common collaborative usage patterns, find that object views reduce the amount of coherence message traffic by as much as a factor of 5 over traditional object caching models. These message traffic levels are comparable to those obtained by whiteboard implementations using event-based middleware[1] abstractions, where a thread explicitly identifies the subset of objects it is interested in receiving updates for. Note, however, that unlike a multicast- or event-based approach, object views reduce message traffic without compromising on object consistency in the face of concurrent updates.

The organization of the rest of this paper is as follows. Section 2 provides relevant background on object caching and discusses related approaches for specifying constraints on object behavior. Section 3 describes, in a language-neutral fashion, the essential ideas behind object views and discusses how the information contained in object views can be utilized to improve caching performance. Section 4 presents VJava, Java augmented with language extensions for expressing views. The VJava-to-Java translator is discussed in Section 5. Section 6 describes the design and performance of the whiteboard application. Section 7 places our work in context and finally we conclude in Section 8.

2. BACKGROUND AND RELATED WORK

2.1 Object Caching

Object-based parallel and distributed applications require good data locality for performance. Although it is possible for application developers to specify the placement and caching of objects, such decisions increase programming complexity, and often are not portable across changes in usage scenarios or resource characteristics of the underlying environment. Consequently, a great deal of attention has focused on the development of efficient object-caching strategies that transparently maintain the consistency of cached copies.

In contrast to hardware cache-coherence solutions that require expensive custom hardware and are not applicable in distributed environments, software coherence schemes are much more flexible. Starting from the original work on Ivy [13], several software distributed shared memory systems have been developed that support a flat shared address space maintaining coherence either at fixed granularity or at variable granularity (objects). Example systems in the former category include TreadMarks [1], Shasta [18], and AURC [9], while those in the latter category include systems such as SharedRegions [17], Midway [5], SAM [19], and CRL [10]. Recent research in both types of systems relies on the use of weak consistency protocols such as lazy release consistency [1], message-driven release consistency [12], and entry consis-

[1] Event services are sometimes also referred to as publisher-subscriber systems [4].

tency [5] that postpone the propagation of object updates until program synchronization points. Entry consistency in particular is well suited to object-oriented programming models, guaranteeing consistency of object state only upon method entry. Although such relaxed consistency models achieve good performance on small-scale applications, the primary factor limiting their scalability is that, in the absence of any information about application behavior, they are forced to rely on conservative decisions. For example, most schemes maintain coherence at the granularity of the entire object, propagating updates in response to all object accesses or disallowing potentially conflicting operations. A few approaches, such as Munin [6] and Orca [3] have demonstrated the advantages of incorporating object usage information (e.g., an object is read-only) to reduce coherence traffic. However, this reliance on global object properties implies that such optimizations can only be used if all accesses to the object satisfy the desired property. Our work shares the same goals as Munin and Orca but relaxes this latter restriction considerably.

2.2 Constraints on Object Behavior

In the context of object-oriented programming models, two types of mechanisms have been most commonly used to specify constraints on object behavior—reflection and synchronization specifications.

Reflection mechanisms allow an object to inspect and manipulate itself at run time. The Java reflection API [2] permits inquiry about object fields and methods, allows assignments to fields and invocation of specific methods, and provides an object-factory service. In some actor-based languages such as ABCL/r [14] and HAL [8], reflection also allows specification of resource management policies orthogonal to the core functionality of the object. Examples of such specification include the use of reflection to control location of new actors, and influencing the processing order of messages delivered to the actor's mailbox. Unlike these uses of reflection, which *extend* object functionality by monitoring object accesses at run-time, object views focus on *restricting* object usage to facilitate more efficient data transfer.

Object views are more similar to specifications such as enabled method sets [20] and transition specification [15] used in concurrent object-oriented languages to express synchronization constraints among methods of the object. Synchronization specifications constrain the order in which object methods can be invoked, but do not provide any guidance on how to improve data transfer costs. Our work focuses on the latter, motivated by the observation that, given the growing costs of communication, data transfer costs often end up dominating the overall costs of concurrent execution in distributed environments.

Our use of the view terminology is only peripherally related to its namesake in the Emerald programming language [21. Emerald views represent an object casting mechanism, called narrowing and widening, between abstract types of the same inheritance chain. The Emerald keyword *restrict* is more relevant to our work. In Emerald, once restricted an object cannot be widened (up-casted) beyond the minimum of all the restrictions that were applied to it. In essence, it becomes a restricted version of an original object, thereby permitting optimizations such as transferring reduced state whenever the object is migrated to a different node. As we shall see in Section 3, our notion of views subsumes this restriction mechanism, while providing a more general and flexible description of object usage.

3. OBJECT VIEWS

The main idea behind this work is to provide users with a new programming abstraction, an *object view*, which restricts the functionality of an object that is visible to a particular computation thread. All accesses to the object are performed through its views; from the perspective of the computation threads, the latter are virtually indistinguishable from regular objects. Views implicitly provide the underlying implementation with information about potential side effects of concurrent object accesses. A compiler can then use knowledge of these side effects to generate scalable, low-overhead caching protocols customized to application requirements.

Object views can be thought of as generalizations of interfaces, providing restricted handles through which objects are accessed.[2] Adopting this perspective, we first describe three different kinds of object views in object-oriented programming languages (see Figure 1). Our discussion is language-neutral; we defer a description of language-support for views in the Java programming language to Section 4.

Level 0 views:

Level 0 views, representing the current notion of classes, provide a baseline, where no additional information about object usage is expressed. In this case, the view consists of all publicly accessible methods and fields of the object. The consequence of not having any more information is that the underlying implementation must maintain object coherence making only conservative assumptions about thread behavior (i.e., that any thread can invoke any method on the object).

Level 1 views:

Level 1 views, corresponding to the current notion of interfaces (or signatures in ML[22]), restrict the behavior of a computation thread by exposing only a subset of the publicly available functionality of the object. Figure 1 shows an instance of such a view. Only the methods present in the view can be invoked upon objects of the class that the view represents. This is akin to invoking methods on the object using interfaces. As mentioned in Section 2, this is somewhat related to the restriction mechanism in Emerald [21]. The difference is that, in Emerald, restriction serves as a permanent limitation on the way the object is used, while in our framework the programmer is also allowed to subsequently cast views into other less restrictive views. What is also different from both interfaces and Emerald's mechanism, is how this restriction is used by the underlying system. The information that a thread only accesses certain methods of the object enables the compiler to determine the minimal amount of object state required to

[2] We draw this analogy recognizing that what we refer to as Level 1 views in the following discussion can also be implemented in Java-like languages using interfaces. However, given that the latter use represents a departure from the intended use of interfaces, and to express Level 2 views in a uniform fashion, we have chosen to introduce new language primitives.

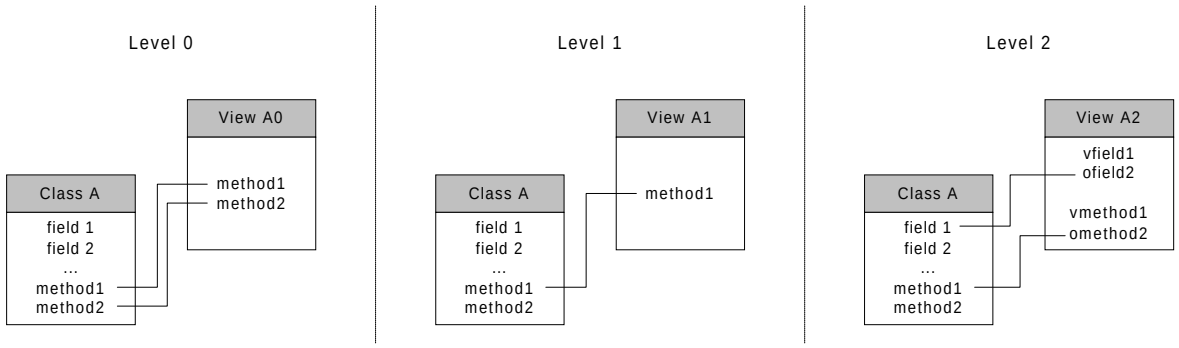

Figure 1: Object views provide a restricted handle through which to access object functionality. Level 0 views expose all publicly available object methods, Level 1 views expose only a subset of the object methods, and Level 2 views permit augmentation of Level 1 views with view-local fields and methods.

maintain consistency between object copies cached by each of the threads. The intuition is that a cached object copy supporting a particular view need only be invalidated (or updated) if there is concurrent access to the object by another thread using a *conflicting* view. Two views *conflict* if they access an overlapping subset of object state in a conflicting fashion (i.e., at least one of the accesses is a write). In addition to determining whether two views are conflicting, a compiler can also optimize data transfer by transforming the view methods to use a compact state layout in the cached copies.

Level 1 views provide an intuitive abstraction for expressing the behavior of threads that access a shared object: each thread accesses the object using a cleanly describable interface. In addition, the underlying implementation can aggressively cache the object while maintaining coherence at sub-object granularity. In the absence of Level 1 views, the implementation would have to disallow concurrent execution of threads that *might* perform conflicting operations on the object.

Level 2 views:

Level 2 views augment Level 1 functionality by allowing each view to contain its own set of fields and methods. *View-local* methods can also access object fields that are remapped into the view space. Level 2 views permit the expression of a variety of sophisticated application-specific caching protocols by enabling the thread to compute using view-local state. These views support a caching model where threads obtain a snapshot of the object, compute with respect to this snapshot, and merge their changes into the object as required by the computation. Note that unlike approaches where the programmer explicitly manages cached copies, Level 2 views still present the programmer with the abstraction of accessing a single shared object. Moreover, reconciliation of view state is triggered automatically whenever another thread accesses the object using a conflicting view. Acquiring the object snapshot and reconciling local changes into the global object can be specified in a flexible fashion by allowing custom callbacks to perform these operations. The underlying implementation can invoke these callbacks as required.

Level 2 views admit two popular applications. First, views that define only view-local methods, which access remapped object state, provide a convenient abstraction for incorporating thread-specific processing into each object access, for example to short-circuit a remote object invocation with some local processing. The second application transfers all functionality to the views, with objects themselves simply acting as containers for shared data. The former presents a convenient model for transforming existing objects and interfaces, as well as dividing object functionality between server and client nodes in a distributed application. The latter represents a scenario where objects and their views are developed simultaneously. In this model, in the limit, objects are abstracted away entirely with views becoming the primary units of processing.

Figure 2 highlights the coherence traffic savings achieved by Level 1 and Level 2 views as compared to using only Level 0 views (the baseline). In the figure, threads running on three nodes, P1, P2, and P3 access a single object using views that cache copies of the object. The figure shows the cached copies on the different nodes corresponding to accesses by processors P2, P3, P1, and P2 respectively. The first three accesses involve different subsets of the object, while the last access by P2 conflicts with all previous accesses. Using only Level 0 views, the underlying implementation may have to invalidate previous copies on each access. With Level 1 views, only those copies that represent conflicting views need to be invalidated, saving on some data transfer and invalidation operations as compared to Level 0 views. With Level 2 views, the number of invalidations is further reduced when the application uses custom protocols that take advantage of reconciliation callbacks.

3.1 Example Applications of Object Views

Example 1: Consider the natural parallelization of a five-point stencil computation on a 2D grid, an important primitive operation in most numerical solvers. If each partition of the grid represents a different object, then the thread operating on a partition needs to access the row and column values contained in the objects to the north, south, east, and west of it. In the absence of views, caching the entire object is likely to be an expensive operation, particularly since the cached values are updated on each iteration. Accessing the object using views that permit access to the appropriate rows and columns separately drastically reduces the amount of data that must be transferred. In addition, since a row or a column view is now shared by only two threads, it lends

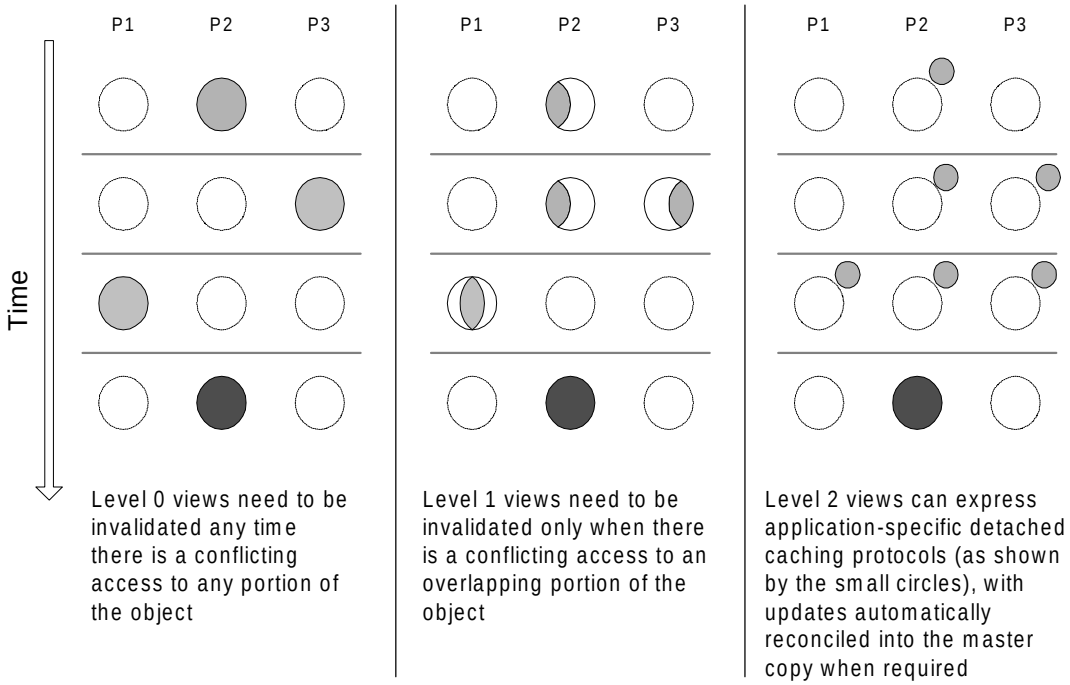

Figure 2: Coherence traffic requirements for different kinds of views. The gray circles represent accesses to the object that cause invalidations with Level 0 views, but allow cached copies to coexist using Level 1 views as long as they represent non-conflicting views. Level 2 views allow cached copies (shown using the small circles) representing even conflicting views to coexist as specified by application-specific coherence protocols. The black circles represent accesses to the object that always invalidate all other views.

itself to optimizations such as replacing an invalidation-based coherence protocol with an update-based protocol.

Example 2: Consider a collaborative application such as a shared whiteboard, which provides users with the ability to concurrently inspect and modify the same document, and to see changes made by others reflected in their own view. Such an application lends itself naturally to a variety of views corresponding to different operations. For example, different views can be used depending on whether the object of interest is being inspected superficially or at a high level of detail. Similarly, views can be used to distinguish between modification operations that change the position of the object and those that modify some of its internal attributes. These multiple views enable a considerable reduction in overall message traffic, while still providing the computation threads with consistent accesses to shared objects. We provide additional details about the use of views and their performance impact using a simple whiteboard application in Section 6.

Example 3: Consider a Web-server that serves dynamically generated pages constructed from multiple segments. Each segment can be used to generate multiple pages. Because of the many-to-many relationship between segments and pages, the performance of such a web server can be greatly improved if, instead of the segments, the pages themselves are cached. However, this introduces a consistency problem whenever the underlying segments get modified. A solution to this problem is to consider each page as a *view* that spans multiple segment objects. Invalidation of a segment now only affects the views that map that segment, reducing total network traffic and the load on the server.

4. VJAVA: LANGUAGE SUPPORT FOR VIEWS

In this section, we describe extensions to the Java programming language to support object views. Besides providing an intuitive abstraction for the programmer, specifying views at the language level enables a compiler to collect the maximum amount of information about the views, their interdependence and their usage. The compiler could catch the majority of errors related to view specifications early on, as well as provide the necessary information for run-time correctness checks. The view-augmented version of Java, referred to as VJava from this point on, aims to closely follow the base language philosophy.

VJava defines two new keywords: **view** and **represents**. A view definition can also contain two optional callback methods **mergeView()** and **extractView()**, the role of which will be explained later. View definitions in VJava match class definitions in Java, except for an additional argument reflecting the view's underlying class. To allow specification of both Level 1 and Level 2 views, the view definition consists of two types of fields and two types of methods: fields and methods that are remapped from the object, and view-local fields and methods. Remapping allows the introduction of new names for the restricted object functionality exposed through the view. Below we introduce necessary language

extensions in stages, according to the types of views that we defined earlier.

Level 0 views:

A Level 0 view is the class itself with all of its fields and methods. Expressing Level 0 views requires no additional language support.

Level 1 views:

A Level 1 view allows mapping of the class fields and methods into view fields and methods. The language support required is the addition of **view** and **represents** keywords. Level 1 views define restricted interfaces through which the object can be accessed and do not contain any state. The syntax required for Level 1 views is shown below:

```
view ViewName represents ClassName
                    extends SuperViewName {
  // remapped object fields, with optional side-effect information
  [final] remappedField1 represents type objectField;

  // remapped object methods
  remappedMethod1 represents type objectMethod( signature );
}
```

Notice that the **final** keyword is overloaded to indicate read-only access to the field. The **represents** keyword is used both to associate a view with the class and to map class fields and methods into renamed view fields and methods.

Level 2 views:

A Level 2 view additionally allows the specification of fields and methods that are completely local to the view. To provide the support for Level 2 views, the syntax is extended to support view-local fields and methods and optional **mergeView()** and **extractView()** methods to provide the view callback functionality, as described in Section 3. User-defined views can override the default *extractView()* and *mergeView()* methods of *SharedView* if special treatment of invalidation is required. The most general view syntax required to support all three types of views is shown below:

```
view ViewName represents ClassName
                    extends SuperViewName {
  // remapped object fields, with optional side-effect information
  [final] remappedField1 represents type objectField;

  // view-local fields
  type viewField2;

  // remapped object methods
  remappedMethod1 represents type objectMethod( signature );

  // view-local methods
  type viewMethod2( signature );

  // optional callback methods for Level 2 views
  [ void extractView(); ]
  [ void mergeView(); ]
}
```

4.1 Example: Expressing Views in VJava

Figure 3 contains an example showing the use of these extensions. The figure contains two sample classes on the left (*Component* and *Circle*) and six view definitions on the right (*InspectComponent*, *MoveComponent*, *EditComponent*, *InspectCircle*, *EditCircle*, and *CachedEditCircle*). The views have been defined to allow the maximum degree of concurrency among them.

InspectComponent, *MoveComponent* and *EditComponent* are Level 1 views, whereas *CachedEditCircle* is an example of a Level 2 view. Figure 4 pictorially depicts the inheritance relationships between these views and shows which of the views conflict because of conflicting uses of object state. A compiler can derive these relationships automatically from the view declarations.

A computation thread uses these views as shown in the code fragment below. The compiler defines coercion operations from an object reference to each of the views of the corresponding class. In the following code fragment, the thread first obtains an *EditCircle* view on the object referred to by the reference *obj*, and subsequently invokes a method supported by the view.

```
....
EditCircle circleV = (EditCircle) obj;
circleV.updateRadius( 45 );
....
```

To understand how the language extensions are used, let us examine Figure 3 in more detail. *Circle* is an object that can be shared across multiple nodes by different threads that each may use it in a specific way. The views provide a way of specifying (constraining) how each thread will use this particular object, including which fields it will read or write. The first line of the view specification defines the *view-of* relation to the object using the **represents** keyword. View inheritance is specified using the **extends** keyword, similar to object inheritance. The view member declarations are more complicated. One can define *view-local* fields (such as *local_rad* in *CachedEditCircleView*) or map a field of a represented object or its superclass (such as the *rad* field in *EditCircleView*) into a view field and specify how this field will be used in the view (read/write). Similarly, methods can also be view-local or mapped from the class methods. View-local methods can only operate on view fields. The compiler can make sure that this is indeed the case, so that no run-time overhead is incurred. If view-local fields are defined by the programmer, he is responsible for initializing these fields and reconciling any changes with the global object by overriding the predefined *extractView()* and *mergeView()* methods. The default behavior of these methods is to do nothing.

Classes	Views

abstract class Component *extends Object* {
  Point pos; // position
  int c; // color
 	
  Point getPos() { **return** pos; }
  void setPos(Point npos) { pos = npos; }
  int getColor() { **return** c; }
  void setColor(**int** nc) { c = nc; }
}
 	
class Circle **extends** Component {
  int r; // radius
 	
  void setRadius(**int** nr) { r = nr; }
  int getRadius() { **return** r; }
}
	view InspectComponentView **represents** Component {
	  getPos represents **int** getPos();
	}
	
	view MoveComponentView **represents** Component {
	  setPos **represents void** setPos(Point npos);
	}
	
	view EditComponentView **represents** Component {
	  setColor **represents void** setColor(**int** nc);
	}
	
	view InspectCircleView **represents** Circle **extends** InspectComponentView {
	  getRadius **represents int** getRadius();
	}
	
	view EditCircleView **represents** Circle **extends** EditComponentView {
	  rad **represents int** r; // remapping of object's r
	
	  void updateRadius(**int** nr) { rad = max(rad, nr); }
	}
	
	view CachedEditCircleView **extends** EditCircleView {
	  int local_rad; // cached copy of rad
	
	  void updateRadius(**int** nr) { local_rad = max(local_rad, nr); }
	
	  void mergeView() { rad = max(rad, local_rad); }
	  void extractView() { local_rad = rad; }
	}

Figure 3: Example of classes and views in VJava.

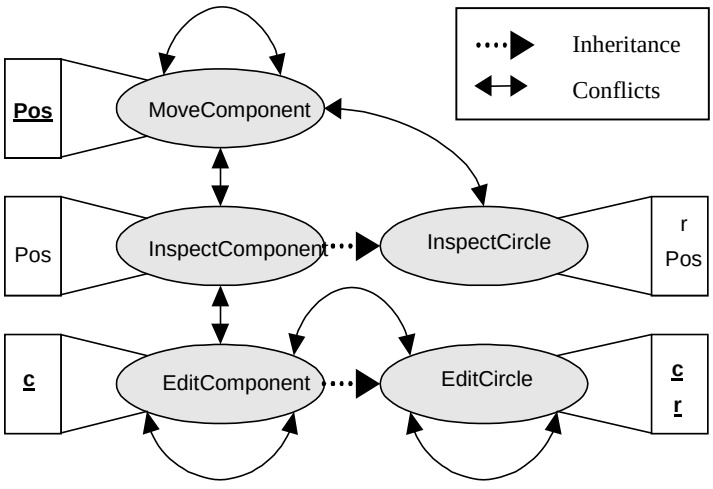

Figure 4: View hierarchy showing the inheritance links (dashed lines) and view conflicts (solid lines). Cached copies that support non-conflicting views can coexist. The boxes show the object state read and written by the corresponding view. Fields that are written to are shown in underlined boldface.

5. IMPLEMENTING OBJECT VIEWS

This section describes the source-to-source VJava to Java translator and our implementation of the framework that supports shared objects and views. The last sub-section covers in detail the architecture and algorithm for shared region allocation.

5.1 Overall Design

The run-time environment of VJava programs consists of independent JVMs running on the nodes of a distributed cluster, sharing objects using a flat region-based software distributed shared memory (DSM) layer at the lowest level. Shared VJava objects are serialized into a set of flat regions according to the algorithm described later in this section. We have provided a thin JNI (Java Native Interface) wrapper on top of the C-based software DSM layer to enable VJava objects to interface with the corresponding shared regions. The DSM library that we used, called VCache, provides an efficient implementation of entry consistency protocols for shared regions [11,16]. To supply a bootstrap mechanism for shared applications, we implemented a simple name server that registers names of global objects at a known address in the Java RMI Registry; joining nodes can look up the remote reference using this name server. Figure 5 shows the overall architecture.

When an application is started, it accesses the top-level objects of the shared hierarchy through the name-server. These name-server queries are the only RMI calls - after bootstrapping, all communication happens through the VCache library using region ids (RIDs) to uniquely identify system objects. In fact, RIDs are also used in place of Java references when accessing shared objects. The object view framework is implemented by five special Java classes—*Region*, *SharedObject*, *SharedClass*, *SharedView*, and *SharedArray*. The *Region* class encapsulates the JNI interface to the VCache library, allowing creation, destruction, mapping, unmapping, and locking of regions, and includes methods to retrieve typed data from regions. *SharedObject* is the base class for any objects that are intended to be shared across multiple nodes using our framework. *SharedArray* supports shared arrays of objects, and *SharedView* is the base class for all view declarations.

5.2 VJava-to-Java Translator

Instead of implementing a compiler for VJava, we chose to implement a source-to-source translator. The primary reason for this decision is that the implementation of a compiler would require modifications to the JVM itself (to support view run-time data) which would make such an implementation non-portable. To avoid additional recompilation, the translator can generate bytecodes directly. However, because there is a one-to-one correspondence between the bytecodes and the Java source, we will use Java in the following text to better illustrate how views are supported in the JVM.

The *SharedObject* and *SharedClass* classes provide the support necessary to implement shared object operations. To associate a view with a particular class (**represents** keyword in VJava) the translator will emit a call to the **addView()** method of the *SharedClass* object stored in the static field *sclass* of each *SharedObject* (similar to the *class* member in Java *Object*) and define a cast method that can be used to obtain the corresponding view object. For each of the fields intended to be shared, a call to the **addField()** method will be emitted. Figure 6 sketches the translation of a sample VJava class and a corresponding view into Java. The interested reader is referred to Appendix A for the complete translation of the *Component* class and some of its associated views, previously described in Section 4.

View translation proceeds similarly. In the translated code, a Java object represents a VJava view if it has *SharedView* as its root superclass. Remapped object fields are added to a view field map, stored in a statically allocated *FieldMap* object. Remapped object methods are transformed to use the object fields remapped into the view object, relying on typed field accessors provided by the *SharedView* class. View-local fields and methods are passed through unchanged. Fields of superviews are also available in the views, as are fields from superclasses of the actual object (as long as they are appropriately mapped).

The translation of the view calling sequence is shown in the code fragment below. For the VJava code sequence:

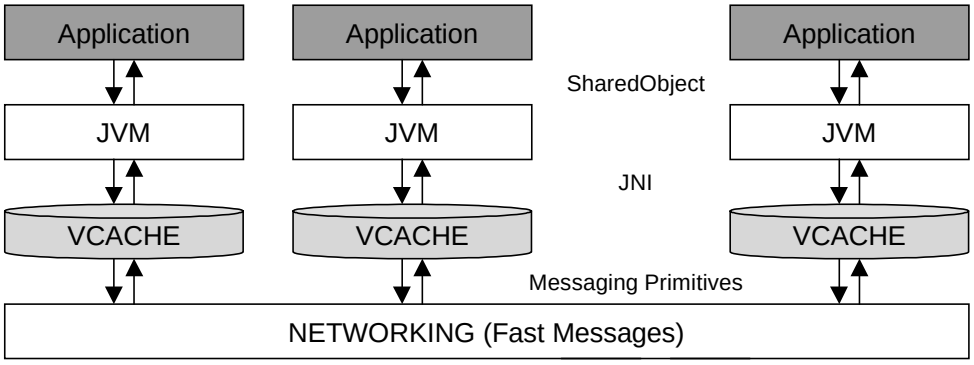

Figure 5: Overall design of the VJava run-time system. The translated Java code runs on top of independent JVMs, which access globally shared objects using a flat region-based software shared-memory layer (VCACHE).

VJava	Java

abstract class C *extends Object* {
  int f; // position
 	
  int getF() { **return** f; }
  void setF(int new_f) { f = new_f; }
}
	abstract class C extends SharedObject {
	  // SharedClass is where static field and view information is stored
	  static SharedClass sclass = new SharedClass(SharedObject.sclass);
	  static {
	  sclass.addIntField("f");
	  sclass.addView(V);
	  }
	// obtaining views from the object: these methods are virtual
	  V toV() {
	  return new V(this);
	  }
	}

view V **represents** C *extends View* {
  setF **represents void** setF(**int** new_f);
}
	class V **extends** SharedView {
	  // FieldMap is where static view field information is stored
	  static FieldMap fm = **new** FieldMap(C.sclass);
	  static {
	  fm.mapReadWriteField("f", "f");
	  }
	  void setF(**int** new_f) { setIntField("f", new_f); }
	}

Figure 6: Sample class and view skeletons in Java (left) and translated Java (right)

....
EditCircleView circleV = (EditCircleView) obj ;
circleV.updateRadius(45);
....

the translator would emit the following code:
....
EditCircleView circleV = obj.toEditCircleView() ;
circleV.acquire();
circleV.updateRadius(45);
circleV.release();
....

The *toEditCircleView*() method is the result of translating the cast of the object to the appropriate view. The *acquire*() method of a view is used to atomically lock all the regions associated with the view with appropriate permissions, and execute any update callbacks. No locking is done at the time of view instantiation to avoid creating a new view object for each use. The translator locks the view regions in a sorted order to avoid deadlocks across nodes. A smart compiler can move the location of *acquire()* and *release()* calls, merging any that happen to be adjacent.

5.3 Mapping Views to Flat Regions

The sharing of object fields is accomplished by mapping them into a set of flat regions at the VCache level. The intuition is that the VCache layer provides consistency at the granularity of regions, which is in turn used to implement consistency at the level of views. A translated VJava object does not store any of the field data of its own. All it contains is a list of VCache region ids (RIDs) and offsets into the corresponding regions.

5.3.1 Region Allocation

Based on the views defined for a particular object, the translator *statically* computes the optimal grouping of fields in the region. Clearly, two non-overlapping views should not interfere with each other's operation, therefore their fields should be stored in different regions. Each field of the object is labeled with the set of views that use it and the type of use (i.e. read or write). The fields are then grouped into equivalence classes based on the labels assigned to them. Figure 7 illustrates an object that has fields A, B, C, D, E, F and three views.

Each equivalence class will have a separate VCache region allocated for it. In our example, four separate shared regions will be created. At the time of static class initialization, the field grouping information is used to create a map from fields to regions (and offsets) for that object, which becomes the representation of that object in the VCache. At run time every shared object is initialized with the map from fields to the regions in VCache.

5.3.2 System Internals

All accesses to shared regions are through the *Region* class, which provides type-sensitive access to the shared region contents. All primitive types are written trivially, and objects are stored into

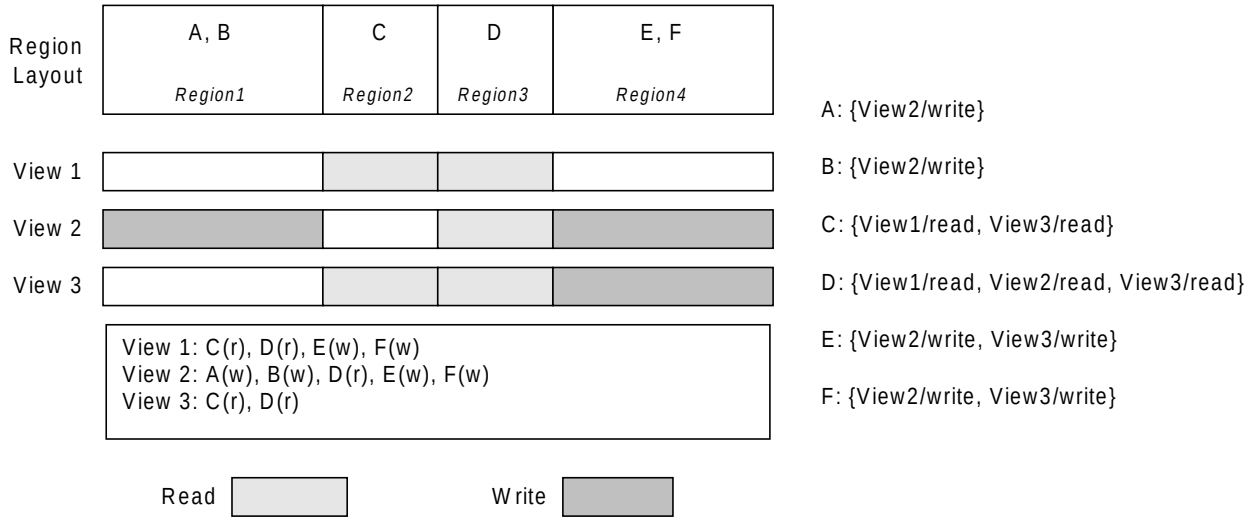

Figure 7: Mapping of views to regions automatically generated by the translator based on a grouping of fields that minimally expresses conflicting access constraints. Field labels used for grouping into equivalence classes are on the right.

regions using the Java serialization mechanism to provide type consistency (i.e. the type read from the region is the type written to the region). We actually overwrite serialization of shared classes because only the mapping from field ids to regions (ids and offsets) needs to be stored. All other object information (field types, views, etc) is either statically stored in the *SharedClass* or can be recomputed on the fly.

The Java reflection mechanism is used in the translated code to register a view with an object, to return an appropriate view of the object (custom constructor call), and to check for correct usage of views at runtime.

6. SHARED WHITEBOARD: EXAMPLE OF USING OBJECT VIEWS

To verify the programmability of object views, and assess their performance impact, we have implemented a collaborative whiteboard application. The whiteboard objects are similar in spirit to the sample classes discussed in Sections 4 and 5, modulo additional state and a richer inheritance hierarchy. The whiteboard implementation leverages object views to support additional concurrency. For instance, a user can move an object while another is changing its properties, because the *MoveComponent* and *EditComponent* views do not conflict with each other. All graphical objects are derived from the *Component* class, and there exists a parallel hierarchy of corresponding views. The user of the whiteboard sees, by default, a high-level *InspectComponent* view of each whiteboard component. The fact that an object is being edited by someone (i.e. *EditComponent* view is enabled on that object) is communicated to other users via color change. To implement the whiteboard application, we started off with a standard sequential version, and then converted it to a shared application by defining appropriate object views. We found the process quite straightforward and intuitive, and most of the application logic carried over unchanged during the transformation.

6.1 Performance Results

To demonstrate that the object views abstraction not only provides an easier programming model, but also reduces unnecessary communication, we conducted a series of experiments that compare message traffic under different usage scenarios. All experiments were run on a cluster of 16 200 MHz Pentium Pro workstations running Windows NT 4.0, connected with the Myrinet interconnect. The whiteboard was initialized with four hundred shared objects, and each thread (one per physical node) generated 1000 accesses to some subset of these objects, selecting both the object and the view according to the usage patterns described below.

We compared message traffic incurred with our whiteboard implementation (that caches object views) against three other approaches. The first two represent commonly adopted approaches for scalable implementations of whiteboards. The *multicast* approach propagates each change to every node using a multicast protocol. In this case, N messages have to be sent for every write operation on a node (one to the central manager and N-1 updates to other nodes by the update manager), but none of the reads require any network traffic. The *event-based* approach represents a model where a node subscribes to receive updates only about the events of interest. In this case, a central node sends updates only to the nodes that have subscribed to receive them. An example of a whiteboard framework that uses the event model is NCSA's Habanero [7]. Note that neither the multicast nor the event-based approaches provide any consistency guarantees. The third point of comparison was a traditional weak consistency-based DSM, which maintains coherence at the granularity of the entire object. In the rest of this section, we refer to this third model as "object caching". We expected that our approach would greatly reduce

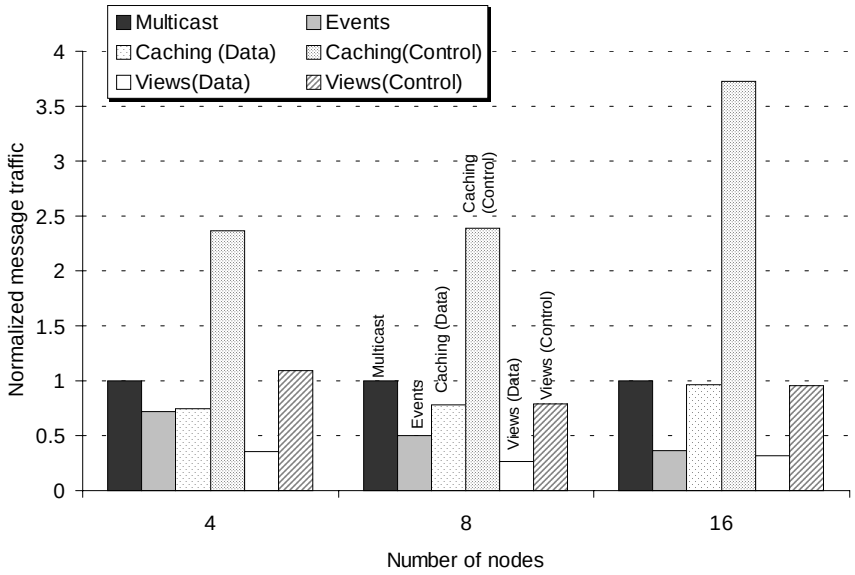

Figure 8: Normalized message traffic for multicast, event-based, and caching-based whiteboard implementations as the number of competing threads is varied (one thread per node). The other parameters are kept fixed at *write-fraction* = 10%, *access-pattern* = random, and *subset-size* = 25%.

the number of messages for certain patterns of access as compared to the conventionally used approaches of multicast, object caching or event subscription for collaborative environments.

For each of the four implementations of the whiteboard, we compared message traffic that would be generated for different numbers of threads with variation in the following three parameters that affect the overall usage scenario: *write-fraction*, *access-pattern* (*hotspot size*), and *subset-size*. The *write-fraction* parameter controls the ratio of read view accesses to write-view accesses. The *access-pattern* (*hotspot size*) parameter simulates two major patterns of access:

- random access on objects simply chooses a random object, and then reads or writes that object as dictated by the other parameters (*hotspot size* = 0)
- hotspot access is a pattern that more closely resembles a typical collaborative environment where a small fraction of objects are likely to receive a disproportionately higher fraction of all accesses (*hotspot size* > 0)

The *subset-size* parameter determines the cardinality of the region of interest for each thread. Given a particular *subset-size*, each of the threads randomly chooses a set containing that many objects to access. A larger *subset-size* value indicates an increased likelihood of overlap between the regions of interest of two threads, and, consequently, higher coherence traffic.

Figures 8-11 show the performance of each of these four approaches for interesting sub-ranges of the parameter space described above. The graphs show normalized message traffic; the number of messages incurred by the multicast model is used as the normalizing factor. For the two caching implementations—caching and views—the overall message traffic is split into two categories: control messages and data messages. The former are required by the underlying coherence protocol and the latter carry updates. In real applications, data messages are likely to be much larger than control messages. With the availability of high-performance messaging layers such as Fast Messages [16], which can support small messages with very small overheads, the size of the message (rather than the number of messages) becomes the dominant contributor to messaging overheads.

The graphs verify our intuition: exploiting object views reduces both data and control message traffic as compared to traditional object caching (without views) by as much as a factor of 5. The benefits are higher when the system is stressed, i.e., in the presence of increased hotspots, higher write fractions, and larger subset sizes. What is quite surprising, however, is how well the view-based implementation performs in comparison to the multicast- and event-based approaches. The view implementation is always competitive and occasionally superior in terms of data traffic as compared to even the event-based approach that propagates updates only to those nodes that have expressed an interest in the object. The view-based implementation does incur some overhead due to the larger number of control messages; however, as mentioned above, the increased number of control messages is likely to account for only a small fraction of the total overhead. A distinct advantage of the view-based approach is that it guarantees consistency of the whiteboard objects, while achieving the message traffic levels of the two communication-efficient approaches. Our results show that object views can enable simpler construction of efficient scalable distributed and parallel object applications. The same program can now run well on a range of architectures ranging from tightly coupled SMPs to loosely coupled distributed environments; traditionally, such applications have required rewriting, achieving performance at the cost of programming convenience and maintainability.

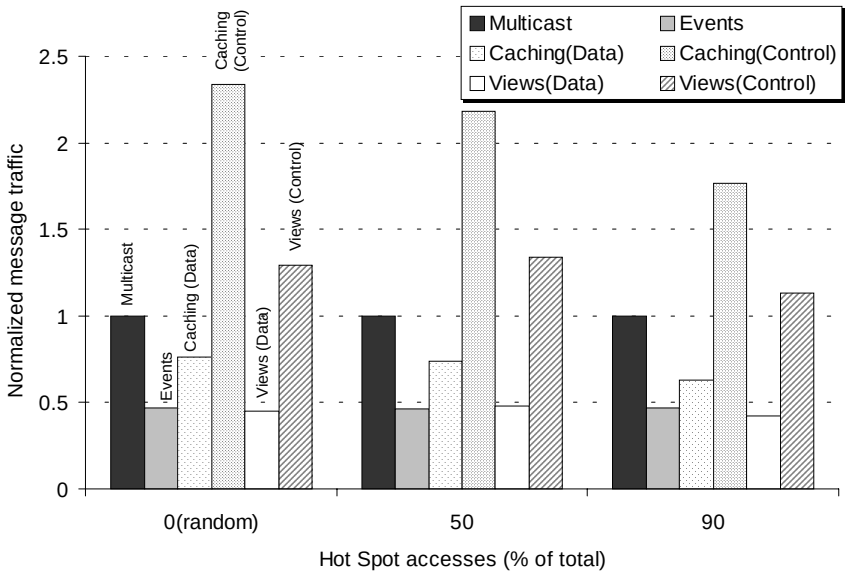

Figure 9: Normalized message traffic for multicast, event-based, and caching-based whiteboard implementations as the percentage of hotspot accesses is varied. The other parameters are kept fixed at *write-fraction* = 10%, *num-nodes* = 8, and *subset-size* = 25%.

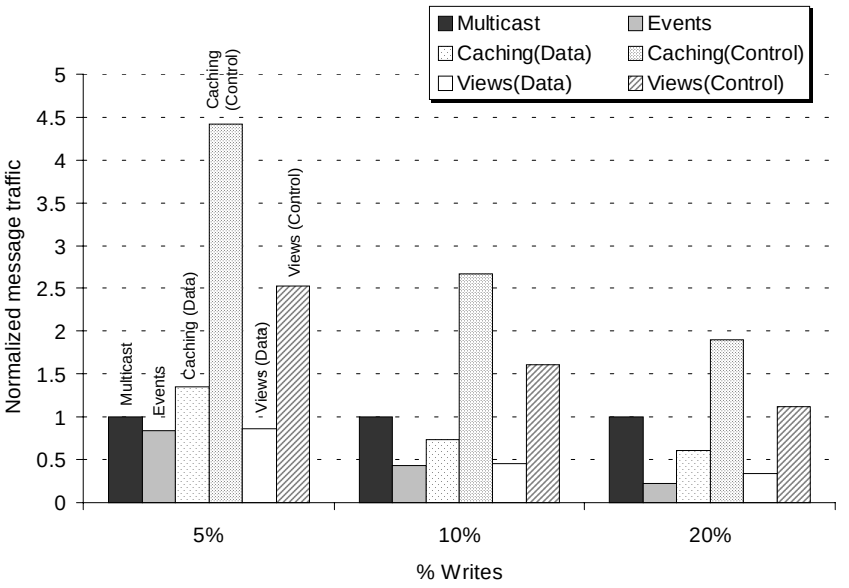

Figure 10: Normalized message traffic for multicast, event-based, and caching-based whiteboard implementations as the percentage of write accesses is varied. The other parameters are kept fixed at *access-pattern* = random, *num-nodes* = 8, and *subset-size* = 25%.

7. DISCUSSION

We chose to implement object views using an approach that combines language extensions with intelligent compiler analyses coupled with some run-time system support. Although a subset of the functionality provided by object views can also be achieved using alternate approaches such as a library-based interface or design patterns, we believe that our approach affords significant advantages over these alternatives.

A library-based interface to object views would require additional effort from the programmer. Our translator essentially produces

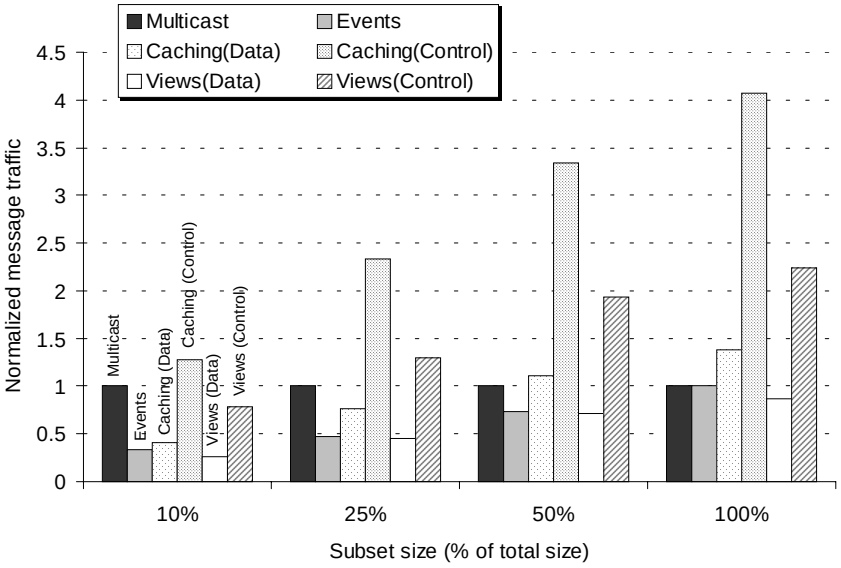

Figure 11: Normalized message traffic for multicast, event-based, and caching-based whiteboard implementations as the subset size is varied. The other parameters are kept fixed at *write-fraction* = 10%, *num-nodes* = 8, and *access-pattern* = random.

the calls to such a library interface. However, as the translated code in Section 5 demonstrates, a compiler performing relatively simple static analyses can significantly simplify the programming task of expressing views. Moreover, the compiler can perform several optimizations in its choice of view coherence protocols by relying upon global program analyses. For instance, a compiler can determine that a view accesses object state that is written once and is not subsequently modified, permitting the use of efficient coherence protocols for obtaining such a view. A programmer would be hard-pressed to provide similar information. It is this potential for large performance gains that motivated us to favor an approach integrating views at the language level and provides the programmer with a natural interface for accessing shared objects.

The information contained within object views can also be communicated using stylized coding practices, similar to design patterns. However, the latter do not allow the compiler to associate additional semantics with program fragments; doing so is equivalent to extending the language. In contrast, integrating views into the language provides the compiler much more flexibility in the kinds of analyses and optimizations that it can perform.

8. CONCLUSION

In this paper, we have introduced the concept of *object views*—language support for specifying constraints on object usage, enabling the underlying implementation to generate efficient object caching protocols customized to application requirements. We presented VJava — the extension to the Java programming language for expressing object views, and described a translator that converts VJava programs to base Java augmented with calls to a flat software shared-memory system running on a cluster of workstations. A shared whiteboard application was used to verify the programmability of object views, and to quantify their performance impact. Our results show that object views reduce message traffic required for coherence to levels comparable to that achieved by approaches such as publisher-subscriber based multicast without compromising the consistency of shared objects.

This research was motivated by the observation that although a flat shared object model provides good programming abstraction, it is difficult to implement it in a efficient scalable fashion to permit its use in large-scale distributed applications. A consequence has been that programmers have grown to accept approaches that permit efficient execution even if it makes the programming task difficult and unintuitive. We have shown that a simple and intuitive notion of a view can simplify programming by providing the necessary abstraction, yet still permit efficient communication. Current work focuses on integrating view specifications with compiler analyses to generate per-view custom protocols, and generalizing the view interface to express constraints involving consistent behavior of multiple objects.

ACKNOWLEDGMENTS

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-96-1-0320 and F30602-99-1-0517; by the National Science Foundation under grant number CCR-9411590 and CAREER award number CCR-9876128; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

REFERENCES

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamoney, Y. Wu, and W. Zwaenopoel. Tread-marks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2), February 1996.

[2] Ken Arnold and James Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, 1998.

[3] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[4] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multi cast protocol for content-based publish-subscribe systems. In *Proceedings of ICDCS'99, International Conference on Distributed Computing Systems*, 1999.

[5] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proceedings of COMPCON 1993*, pages 528–537, March 1993.

[6] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 152–164, 1991.

[7] A. Chabert, E. Grossman, L. Jackson, and S. Pietrovicz. NCSA Habanero: Synchronous collaborative framework and environment. Software Development Division at NCSA, Champaign, IL, 1997.

[8] C. Houck and G. Agha. HAL: A high-level actor language and its distributed implementation. In *Proceedings of the 21st International Conference on Parallel Processing*, pages 158–165, St. Charles, IL, August 1992.

[9] Liviu Iftode, Cezary Dubnicki, Edward W. Felten, and Kai Li. Improving release-consistent shared virtual memory using automatic update. In *Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, pages 14–25, February 1996.

[10] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the Symposium on Operating Systems Principles*, pages 213–228, 1995.

[11] Vijay Karamcheti and Andrew A. Chien. View caching: Efficient software shared memory for dynamic computations. In *Proceedings of the International Parallel Processing Symposium*, pages 483–489, 1997.

[12] Povl T. Koch, Robert J. Fowler, and Eric Jul. Message-driven relaxed consistency in a software dis-tributed shared memory. In *First Symposium on Operating Systems Design and Implementation*, pages 75–85, November 1994.

[13] Kai Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[14] H. Masuhara, S. Matsuoka, T. Watanaba, and A. Yonezawa. Objected-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of OOPSLA '92*, pages 127–144, 1992.

[15]] S. Matsuoka, K. Taura, and A. Yonezawa. Highly efficient and encapsulated re-use of synchroniza-tion code in concurrent object-oriented languages. In *Proceedings of OOPSLA'93, Object-Oriented Programming Systems, Languages and Architectures*, 1993.

[16] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and mpps. *IEEE Concurrency*, 5(2):60–73, April-June 1997.

[17] H. S. Sandhu, B. Gamsa, and S. Zhou. The shared region approach to software cache coherence on multiprocessors. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'93)*, pages 229–238, July 1993.

[18] Daniel Scales, Kourosh Gharachorloo, and Chandramohan Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of ASPLOS-VII*, pages 174–185, October 1996.

[19] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for dis-tributed memory machines. In *First Symposium on Operating Systems Design and Implementation*, pages 101–114, 1994.

[20] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In Proceedings of OOPSLA'89, Object-Oriented Programming Systems, Languages and Architectures, 1989.

[21] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A General-Purpose Programming Language. *Software - Practice and Experience 21(1)*,January 1991.

[22] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

APPENDIX A

Figure 12 shows a complete translation from VJava to Java of the *Component* class and some of its associated views, previously described in Section 4. Section 5 provides details about the translation process.

VJava	Java

abstract class Component *extends Object* {
  Point pos; // position
  int c; // color
 	
  Point getPos() { **return** pos; }
  void setPos(Point npos) { pos = npos; }
  int getColor() { **return** c; }
  void setColor(**int** nc) { c = nc; }
}
	abstract class Component **extends** SharedObject {
	  // SharedClass is where static field and view information is stored
	  static SharedClass sclass = **new** SharedClass(SharedObject.sclass, 2);
	  public SharedClass getSharedClass() { **return s**class; }
	  static {
	  sclass.addObjectField("pos");
	  sclass.addIntField("c");
	  // this associates views with the class
	  sclass.addView(EditComponentView);
	  }
	  // obtaining views from the object: these methods are virtual
	  EditComponentView toEditComponentView() {
	  return new EditComponentView(**this**);
	  }
	}

view EditComponent **represents** Component
  extends View {
  setColor **represents void** setColor(**int** nc);
}
	class EditComponentView **extends** SharedView {
	  // FieldMap is where static view field information is stored
	  static FieldMap fm = **new** FieldMap(Component.sclass);
	  public FieldMap getFieldMap() { **return** fm; }
	  static {
	  fm.mapReadWriteField("c", “c”);
	  }
	  void setColor(**int** nc) { setIntField(“c”, nc); }
	  public EditComponentView(SharedObject o) { **super**(o); }
	}

view EditCircle **represents** Circle
  extends EditComponent {
  rad **represents int** r;
 	
  void updateRadius(**int** nr) {
  rad = max(rad, nr);
  }
}
	class EditCircleView **extends** EditComponent {
	  // FieldMap is where static view field information is stored
	  static FieldMap fm = **new** FieldMap(Circle.sclass);
	  public FieldMap getFieldMap() { **return** fm; }
	  static {
	  fm.mapReadWriteField("rad", “r”);
	  }
	
	  void updateRadius(**int** nr) { rad = max(rad, nr); }
	  public EditCircleView(SharedObject o) { **super**(o); }
	}

view CachedEditCircle **extends** EditCircle {
  int local_rad;
 	
  void updateRadius(**int** nr) {
  local_rad = max(local_rad, nr);
  }
 	
  void mergeView() { rad = max(rad, local_rad); }
  void extractView() { local_rad = rad; }
}
	class CachedEditCircle **extends** EditCircle {
	  // FieldMap is where static view field information is stored
	  static FieldMap fm = **new** FieldMap(Circle.sclass);
	  public FieldMap getFieldMap() { **return** fm; }
	  static {
	  }
	
	  int local_rad;
	  void updateRadius(**int** nr) { local_rad = max(local_rad, nr); }
	  void mergeView() { rad = max(rad, local_rad); }
	  void extractView() { local_rad = rad; }
	
	  public CachedEditCircleView(SharedObject o) { super(o); }
	}

Figure 12: VJava (left) and translated Java (right) for some of the classes and views from Figure 3