Cross-Language, Type-Safe, and Transparent Object Sharing For Co-Located Managed Runtimes UCSB Technical Report June, 2010

Michal Wegiel
Computer Science Department
University of California, Santa Barbara
mwegiel@cs.ucsb.edu

Chandra Krintz
Computer Science Department
University of California, Santa Barbara
ckrintz@cs.ucsb.edu

Abstract

As software becomes increasingly complex and difficult to analyze, it is more and more common for developers to use high-level, type-safe, object-oriented (OO) programming languages and to architect systems that comprise multiple components. Different components are often implemented in different programming languages. In state-of-the-art multi-component, multi-language systems, cross-component communication relies on remote procedure calls (RPC) and message passing. As components are increasingly co-located on the same physical machine to ensure high utilization of multi-core systems, there is a growing potential for using shared memory for cross-language cross-runtime communication.

We present the design and implementation of Co-Located Runtime Sharing (CoLoRS), a system that enables cross-language, cross-runtime type-safe, transparent shared memory. CoLoRS provides object sharing for co-located OO runtimes for both static and dynamic languages. CoLoRS defines a novel language-neutral object/class model, which is a static-dynamic hybrid and enables class evolution while maintaining the space/time efficiency of a static model. CoLoRS uses type mapping and class versioning to transparently map shared types to private types. CoLoRS also contributes a synchronization mechanism and a parallel, concurrent, on-the-fly GC algorithm, both designed to facilitate cross-language cross-runtime object sharing.

We implement CoLoRS in open-source, production-quality runtimes for Python and Java. Our empirical evaluation shows that CoLoRS extensions impose low overhead. We also investigate RPC over CoLoRS and find that using shared memory to implement co-located RPC significantly improves both communication throughput and latency by avoiding data structure serialization.

1. Introduction

Large, scalable software systems are increasingly being built using collections of components to better manage software complexity through reusability, modularity, and fault isolation. Since each programming language has its own unique combination of performance, speed of development, and library support, different software components are often implemented in different languages. As evidence of this, Thrift [34] and Protocol Buffers [31] have been developed by engineers at Facebook and Google, respectively, to enable more efficient interoperation across multi-language components employed within their applications and back-end services. For web applications, different languages are better suited for the implementation of different tiers: Ruby, Python, Java, and JavaScript facilitate fast development of the presentation layer, Java, PHP, Perl, Python, and Ruby components commonly implement server-side logic, and Java, query languages, and C++ are used for a wide range of backend database technologies. The components of these multi-language, multi-component applications and mashups typically execute within independent runtime systems (language virtual machines (VMs), interpreters, etc.) and communicate and interoperate via remote procedure calls (RPC) and message passing.

Increasingly, administrators co-locate runtimes to better utilize multi-core resources. This makes it possible to use shared memory for such cross-component communication as well as for a cross-runtime language-neutral transparent object storage. However, despite its growing practical value, shared memory has not yet been investigated in either of these contexts. To evaluate the potential of using shared memory for cross-language, safe, transparent communication and object storage, we design and implement *Co-Located Runtime Systems (CoLoRS)*. CoLoRS provides direct object sharing across static and dynamic, object-oriented (OO) languages.

CoLoRS virtualizes VM components that assume a language-specific object/class/memory model. In CoLoRS, shared

objects retain their language-specific behavior including the semantics of virtual method calls, locking, and field access. In addition, builtin/library data structures, such as collections, transparently map to their shared counterparts in the CoLoRS object model.

Our key hypothesis is that sharing objects across static/dynamic OO languages using shared memory can be safe, transparent, and efficient. Our main contributions include:

- An object and memory model that enable language-neutral object and class sharing across dynamic and static languages. The CoLoRS object model is a static-dynamic hybrid, which provides the efficiency of a static model with the flexibility of dynamic class modifications. To enable this, CoLoRS uses an extensible static model with versioning and type mapping.
- A GC algorithm that is parallel, concurrent, and on-the-fly – one that is better suited for multi-VM memory management than extant GCs. CoLoRS GC is simpler than state-of-the-art on-the-fly GCs, does not require tight integration into a runtime, imposes no system-wide pauses, and guarantees termination of the on-the-fly collection.
- A synchronization mechanism that avoids the complexities of conventional approaches to monitor synchronization, while providing the same semantics and good performance.
- CoLoRS implementation for Java and Python. To investigate object sharing between dynamic and static OO languages, we integrate CoLoRS support within open-source, production-quality runtimes: HotSpot JVM and cPython.
- CoLoRS experimental evaluation. We have evaluated CoLoRS efficacy using standard Java and Python benchmarks and found that CoLoRS extensions impose low execution time overhead. We also provide experimental results for the CoLoRS GC algorithm and CoLoRS synchronization.
- RPC as a CoLoRS use-case. We have found empirically that CoLoRS can significantly (up to 2 orders of magnitude) improve the performance of cross-language RPC systems, such as CORBA [13], REST [24], Thrift [34], and Protocol Buffers [31]. This is because using shared memory in the co-located case avoids expensive object serialization. The improvements in communication throughput and latency due to CoLoRS significantly increase end-to-end transaction performance in Cassandra [10] (a key-value database), and the Hadoop Distributed File System (HDFS) server [26].

In the sections that follow, we present the design and architecture of CoLoRS, describe the key contributions of our system, including a language-neutral object/memory model, memory management, garbage collection, and synchronization support, as well as transparent object sharing via runtime/library virtualization. We then discuss CoLoRS implementation and its empirical evaluation, compare/contrast CoLoRS with related work, and conclude.

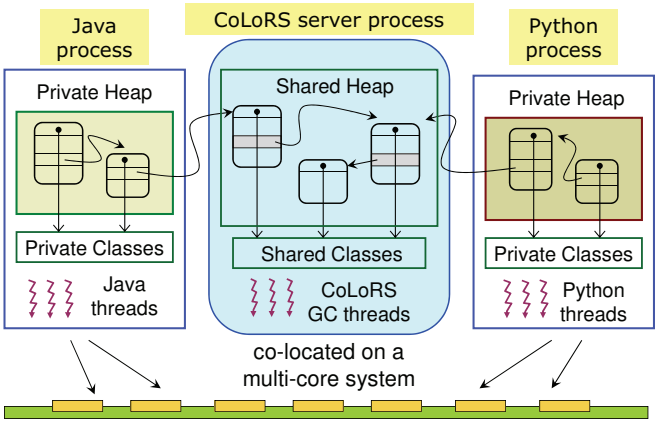

Figure 1. CoLoRS architecture. There is exactly one CoLoRS server process, which manages the shared memory segment and runs concurrent GC. Runtimes for different languages (Java and Python in this case) attach to the shared memory segment and allocate/use objects in the shared heap.

2. CoLoRS Overview

The primary design goal of CoLoRS is to provide type-safe, transparent, direct object sharing between co-located managed runtimes for different OO languages. This includes both statically-typed (e.g. Java) and dynamically-typed (e.g. Python) languages. The key challenge with providing such support are the major differences between language implementations, including object/class models, memory models, type systems, builtin types, standard libraries, and memory management (GC). For instance, dynamic languages support attribute (member) addition at runtime, while static languages permit class changes at compile-time only.

Figure 1 shows a high-level view of a CoLoRS system. In this example, two VM processes (one for Java and one for Python) are co-located on a multi-core system. There is exactly one CoLoRS server process which manages the shared heap (this includes the setup of the shared memory segment, data structure initialization, as well as support for garbage collection). Each VM process has its own private heap and a private object/class model and runs its applications threads. In the shared heap, there is a CoLoRS object/class model which is transparently translated to a private object/class model in each VM. We do not allow pointers from the shared heap to any private heap because of memory/type safety. All VMs map the shared memory segment at the same address in the virtual address space and use shared objects directly via pointers.

Static (class) fields are not subject to sharing because they often represent local resources and sharing them would break resource isolation. For instance (object) fields, how-

ever, CoLoRS supports fully transparent sharing with regard to allocation, GC, field access, (virtual) method invocation, monitor synchronization, standard libraries, and class loading. We do not support code sharing because that would require defining a VM-neutral language and checking whether two methods are equivalent, which is undecidable. In fact, sharing only instance fields makes CoLoRS more practical as the code and static data do not have to match across languages (note that builtin types, e.g. string, differ significantly across VMs in terms of both interface and implementation). The programming burden associated with the necessity of keeping instance fields in shared classes consistent across languages can be further reduced via tools that translate class implementations between languages.

A general approach we take in CoLoRS is to define a language-neutral, shared object model (with respect to non-static data) and then dynamically map it to each runtime-specific object model. To implement this, we virtualize all runtime components that rely on a specific object model. Modifications to runtimes are necessary to make object sharing transparent.

2.1 CoLoRS Usage

CoLoRS provides a simple application programming interface (API) for developers. The CoLoRS API for Java comprises the following methods in the $SharedMemory$ class (Python has equivalent API):

```
Object copyToSharedMemory(Object root);
Object allocate(Class objectClass);
Object allocate(Class containerClass, int length);
boolean isObjectShared(Object object);
ObjectRepository findOrCreateRepository(String key);
ObjectChannel findOrCreateChannel(String key);
Type getSharedType(Object object);
```

CoLoRS supports two ways of creating shared objects: via direct object allocation (the $allocate$ method) and via deep copying of a private object graph to shared memory (the $copyToSharedMemory$ method). The $allocate$ method has two variants: one for allocation of fixed-size objects and one for allocation of container objects (which takes the initial size of a container as a parameter).

Note that we do not support a state model where a thread can switch to the shared mode and issue regular object allocations to allocate in shared memory (as is done in related work on cross-JVM sharing [36]). The reason is that the state model requires complex rules specifying which allocations should target shared memory. For instance, in a JVM, we must exclude class loading, static initializers, and exception handling from leaking objects into shared memory.

CoLoRS provides two mechanisms to initiate communication between two runtimes: channels and repositories, both of which are named entities enabling exchange of a reference to a shared object. The $ObjectRepository$ class provides nonblocking get/set functionality while the $ObjectChannel$ class supports blocking send/receive cross-VM semantics. The following code fragments show an idiomatic repository usage for two Java processes. The client process:

```
ObjectRepository r = SharedMemory.findOrCreateRepository("db");
synchronized(r) { while(r.get() == null) r.wait(); }
```

The server process:

```
ObjectRepository r = SharedMemory.findOrCreateRepository("db");
synchronized(r) { r.set(root); r.notifyAll(); }
```

For object channels, we have a similar pattern but synchronization/waiting is not necessary because of the blocking behavior of send and receive.

Each repository holds a reference to its root object. Each channel has a fixed capacity for messages and blocks the sender when full. As long as a shared object is reachable from any repository, channel, or any VM, it stays alive. Unreachable shared objects are garbage collected. Channels and repositories are identified by a key (string).

The CoLoRS API enables reflective inspection of the shared type of a shared object via the $getSharedType$ method. We need this API method because in CoLoRS, expressions that evaluate to an object class, e.g. object.getClass() in Java, retrieve a private class to which a specific shared class currently maps. To see the shared class before mapping to a private class occurs, $getSharedType$ is used. Shared classes are regular objects – CoLoRS uses a three-level circular meta-data hierarchy that is fully traversable by programs wishing to inspect it.

A programmer can check whether an object is in shared memory via the $isObjectShared$ method. The system throws a $SharedMemoryException$ to prevent shared-to-private pointers as well as to signal type mapping failures, out-of-memory errors, and locking issues.

3. CoLoRS Design and Architecture

CoLoRS uses a dedicated process (CoLoRS server) to manage shared memory. There is one CoLoRS server per OS instance. This server creates, initializes, and destroys the shared memory segment, as well as runs concurrent, parallel GC. That is, GC continues to function even when no runtimes are currently attached.

To use shared memory, runtimes first connect to the CoLoRS server using TCP/IP sockets and then attach to the shared memory segment (by mapping it to their virtual address space at the pre-defined, fixed address). The shared memory segment contains three spaces: metadata space (for state variables and synchronization), classes space (for shared types, repositories, and channels), and objects space (for garbage-collected shared objects). Each VM runs a separate CoLoRS thread which is responsible for collaboration with the CoLoRS server during GC.

3.1 The CoLoRS Object Model (OM)

CoLoRS employs a new OM that aims at transparent and efficient cross-language object sharing, while supporting both static and dynamic languages. Our primary goal is maintaining the language-specific OM and object/class semantics while a VM interacts with shared objects. The rationale behind this is to avoid introducing a new unfamiliar programming model. In addition, CoLoRS combines certain characteristics of static and dynamic OMs in order to support the flexibility of a dynamic model while providing the efficiency and simplicity of a static model.

3.1.1 CoLoRS Type System

CoLoRS preserves language-specific type-safety without defining new typing rules by mapping shared types to private types. When mapping a shared type S to a private type P_1 in one VM and to private type P_2 in another VM, we guarantee that any field access permitted by P_1 does not violate the field typing constraints imposed by P_2 (and vice versa).

In the CoLoRS type system, every value is an object (there are no primitive types like in Java or C#). This is motivated by dynamic languages like Python and Ruby which treat everything as an object and therefore require that each value have a unique identity.

Unlike extant systems for cross-language data sharing, CoLoRS does not specify its own data definition language (DDL). Conventional approaches have resulted in a number of domain-specific DDLs, e.g., SQL in relational databases, WSDL in web services, and IDL in CORBA. The primary limitation of DDLs is their static nature and the necessity for a programmer to master another language. Instead, CoLoRS generates the shared data model automatically from the native language data model defined by the programmer. Moreover, this happens dynamically at runtime and only for types that are used in shared memory.

The CoLoRS OM strives to strike a balance between supporting diverse languages (both static and dynamic) and staying sufficiently close to each individual language so that costly runtime data conversions are avoided if possible. Another key design tradeoff is to support the flexibility of dynamic languages while leveraging the benefits provided by static typing. In fully static OMs (e.g. Java), object layout is completely described by classes, fields are efficiently accessed via offsets, each object consumes only as much memory as necessary for its attribute values, and the data model is fully documented by classes. On the other hand, in fully dynamic OMs (e.g. Python) classes do not describe object attributes, each object maintains a dictionary mapping attribute names to values, field access is expensive as it takes place via names, and space usage is suboptimal due to the redundancy across attribute dictionaries. However, unlike static OMs, dynamic OMs support dynamic attribute addition/removal as well as per-object attributes.

Several hybrid models have been introduced to mitigate the static-dynamic tradeoffs. A partially static/dynamic OM is used by Google App Engine, where each object has a static part (fields described by a class) and a dynamic part (per-object dictionary). On attribute access, the system first tries to use a static field then falls back to an object dictionary on failure. Dynamically created attributes do not become part of the static model. A similar concept has been introduced to Python (via the __slots__ declaration). The JavaScript V8 runtime implements hidden classes to enable fast, offset-based attribute lookup while supporting dynamic attribute addition and deletion.

3.1.2 Hybrid OM and Versioning

CoLoRS OM is a new static-dynamic hybrid, which can be described as an extensible static model with versioning and type mapping. Our goal is to keep CoLoRS OM as static as possible but still allow the flexibility of modifications (add/remove/change name/type of a field).

Shared classes are always created based on private classes when a private object gets allocated in (or copied to) shared memory. On each allocation in shared memory, we inspect the fields of the allocated object and look for a shared class being an exact match for a given type name and field set. If we do not find an exact match, we create a new class (or if a class with this name already exists, we create a new shared class version, having the same class name but a different field set). For example, suppose that we have the following class in Java:

```
class Employee { String name; double salary; }
```

and we perform shared allocation using:

```
Employee e = (Employee)SharedMemory.allocate(Employee.class);
```

If no $Employee$ class is present in shared memory yet, we create one, with two fields that correspond to the private $Employee$ class. Now assume that we add a new field to the $Employee$ class, say $Employee\ manager$; and we repeat the shared allocation as shown above. This time, CoLoRS will create a new version of the shared $Employee$ class, with three fields. Note that at any point in time there is exactly one private $Employee$ class (which may evolve in time) and there may be multiple versions of shared $Employee$ class (reflecting the schema evolution). Field removal is handled in a similar way.

Shared objects use shared classes to describe their layout. Different versions of a single shared class may have different layouts in memory and field sets. Shared classes are read-only, they do not change. However, shared objects may change their class pointers (from one version of a particular class to its another version). This can happen both in static and dynamic languages. For example, the following code in Python, which uses our two-field $Employee$ class:

```
e = shared_memory.copy_to(Employee('Smith', 100))
e.state = 'NY'
```

adds a new field (called $state$) dynamically. To support this in shared memory, CoLoRS creates a new version of the

$Employee$ class and changes the current class of the e object to the new class version. Dynamic field removal (via del in Python) is handled similarly.

The advantage of versioning over a pure OO model is lower space consumption. In conventional OO systems, class evolution takes place via subclassing: to add or hide a field a new class is created that inherits from the old class. As a result, it is not possible to remove any attribute and space is consumed forever by unused fields. In contrast, with versioning, even if classes evolve, the newly-created objects always consume the optimal amount of space.

3.1.3 Type Mapping

To correctly handle multiple class versions in shared memory, CoLoRS uses type mapping. Each private class P in a VM always has exactly one version which, at any given moment, may be mapped to several different versions of class P in shared memory (a one-to-many relationship). Except for builtins (e.g. Integer, String), mapping only occurs between classes with the same name – programs in different languages must agree on package/module and class names. We map a shared field to a private field if and only if both have the same name and the same (or convertible) type. In dynamic languages, we map solely on the field name basis as there are no static types available.

Since type mapping is a relatively expensive process, we perform it lazily, once per shared-class-version, and maintain the mapping in a private hash table in each VM. We also use a reverse mapping table, to avoid shared-type lookup/matching on every allocation in shared memory. Note that on allocation, we need to obtain the shared type based on a private type. In contrast, when accessing a field in a shared object, we perform the mapping from a shared type to the private type.

When CoLoRS allocates a new object in shared memory, it tries to find a shared class version that exactly matches the private field set of the newly-allocated object. If no exact match is found, it creates a new shared class version. Consequently, newly-created objects do not contain fields that were removed from a private class due to its evolution. The rationale behind this is that we want to keep the object size in shared memory optimal. However, when mapping a shared class to a private class in a context other than allocation, we allow both private and shared fields to remain unmapped (if they do not have a match). When a VM uses an unmapped field in a shared object, we dynamically add a field to a class. To do so, we create a new shared class version that contains the previously unmapped field, and change the shared object's class pointer to point to the new class version. Note that the shared object's type does not change, as seen from the VM's perspective – all versions of a shared class always map to the same private class (with the same name).

Although CoLoRS supports dynamic changes, once the data model is stable, both space usage and field access work exactly like a fully static model. Also, in the CoLoRS OM, all object attributes are always present in its class and can be introspected via reflection.

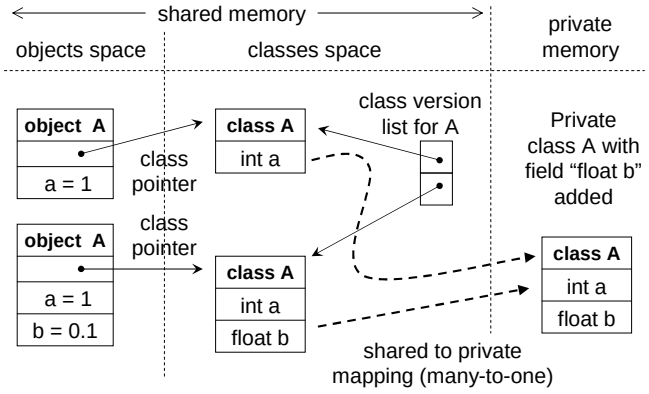

Figure 2. An example illustrating CoLoRS versioning and type mapping as private class A evolves by having a field added.

Figure 2 shows an example where private class A evolves from a single-field class containing "int a" into a class with two fields, "int a" and "float b". Private class A has exactly one version (the newest one with both fields). Shared class A has two versions. Both shared versions are mapped to the private class A so that they can be uniformly used, despite being distinct types in shared memory. The shared objects space contains two objects of class A – one allocated for the old version of A and one allocated for the new version of A. Note that each shared object uses only as much space as necessary for its attribute set. Both objects have the same type in a VM, and the VM may access both fields (a and b) in both objects. On access to a non-existent field (b in this case) in older shared objects, CoLoRS will expand the object to make room for the new field (initializing the new field to 0).

Reconsidering the example in Figure 2 in the case when class A evolves by having the b field removed, we have a similar situation. Private class A again has exactly one version (the newest one, with one field a). Shared class A has two versions, both mapped to the same private type A. Field b remains unmapped as it can never be used by the VM and this field is simply ignored in those shared objects that have it. Note that newly-allocated shared objects do not reserve a slot for field b, thus using optimal amount of space. In contrast, OO inheritance does not allow removal of a field from an object (unused inherited fields continue to consume slots in objects). Field renaming is equivalent to field removal followed by a field addition.

Note that using CoLoRS cannot lead to broken program invariants because matching fields can never remain unmapped. Thus, if class implementations across languages match and preserve some invariant in each language, CoLoRS will preserve this invariant too.

3.1.4 Built-In Types and Libraries

CoLoRS provides full transparency for builtin types (e.g. strings, integers, lists, and sets). Builtin types differ significantly across languages and at the same time are frequently used by programs and libraries. CoLoRS preserves language-specific interfaces for builtin types by virtualizing the builtin implementation and/or standard libraries in each runtime. Library virtualization amounts to modifying the code of library methods so that these methods check whether any of the method arguments (including the receiver, if any) is a shared object and, if so, to execute a different implementation of the method.

CoLoRS defines a set of builtin types which we identify in Table 1 with their mappings in Java and Python.

We support 64-bit integers, which can be mapped to Python int and to any integer type in Java, both primitive, e.g. int, $short$, and reference, e.g. $Long$, $Integer$. Having only one integer type allows us to avoid complex rules for field mapping during schema evolution. For example, if we supported int and $short$ as distinct integer types in shared memory, then we would have to define complex semantics for changing the field type from int to $short$ and vice versa, i.e. when we create a new field dynamically and when we reuse existing integer field.

We use a similar approach in case of floating-point types, supporting only 64-bit IEEE floats. The CoLoRS 64-bit float can be used in Java as any floating point type, e.g. $double$ or $Float$. We do overflow/underflow checks when reading/writing integer/float fields requires conversion.

For scalar types, we also provide a boolean and a string. As in Thrift [34], CoLoRS defines three container types: a list, set, and map. Containers are untyped (i.e. may contain objects of different types at the same time). This is because we cannot automatically infer the container element type (at least in Java and Python), even if the container is not empty. To support a compact byte array representation we provide the $binary$ type, suitable for blobs. Note that in Java, a shared $list$ can be used as an array (of any type) and as a $List$. The rationale behind this is transparency – we want to support Java arrays even though CoLoRS and Python do not have arrays so that we do not change the Java programming model. Scalar types (integer, float, boolean, and string) are immutable. Builtin objects always have exactly one version, one mapping to a private type, and do not have any programmer-visible fields.

In order to use shared objects along with private objects in a single hash-based container, hash codes and equal-to methods must agree across runtimes. We unify them for Java and Python builtin types. For shared objects, CoLoRS provides default hash code generation, equal-to methods, and less-then methods (all based on object addresses). They can be overridden by programmers.

For programmer convenience, CoLoRS automatically copies scalar types (e.g. integer, string) to shared memory.

shared	java	python
integer	byte, short, int, long, char, Byte, Short, Integer, Long, Character	int
float	float, double, Float, Double	float
boolean	boolean, Boolean	bool
string	String	str
binary	byte[]	bytearray
list	List, ArrayList, Object[], int[], float[], T[], ...	list, tuple
set	Set, HashSet	set, frozenset
map	Map, HashMap	dict

Table 1. Builtin types supported by CoLoRS and their mappings to Java and Python builtin types. For transparent and convenient use by programmers, multiple mappings are possible per shared type.

On field assignment/array store, the system checks whether the assignment uses a private r-value and a shared l-value. If so, and the r-value is of a scalar type, CoLoRS silently calls the $copyToSharedMemory$ method on the r-value, instead of throwing an exception. This mechanism is particularly useful for constructors.

3.1.5 Static Languages

In static languages, object fields are typed and typically accessed using field offsets. Since CoLoRS uses a mostly-static OM, it also identifies fields in shared objects by their offsets. Private and shared field offsets may differ so it is necessary to map between them. Unidirectional mapping from the private offset to the shared offset is sufficient because VMs always access shared fields using the context of a private type. To make this mapping efficient, we associate a field-offset-table with each pair (S,P) where S is a shared type mapped to private type P. Whenever we access a shared field in a shared object, we index the appropriate field-offset-table with the private field offset and obtain the shared field offset.

When inspecting a class of a shared object (e.g. via object.getClass() in Java) we always get a unique private class as a result. For example, $integer$ maps to $SharedInteger$ while $list$ maps to $SharedList$. However, to ensure transparency, shared builtins can map to multiple different private types. In OO languages, this can be implemented via multiple inheritance. For instance, if we can make $SharedList$ inherit from $List$, $Object[]$, $ArrayList$, etc. then representing shared $list$ as private $SharedList$ is correct in all possible mappings. However, some languages (e.g. Java) do not support multiple inheritance or inheritance of array types. We instead simulate both by modifying the runtime so that $SharedList$ can be cast to any of the private types that shared $list$ maps to. We apply a similar approach for $integer$ and $float$.

Each private class maps to a unique shared class. A general rule that we use is that whenever we allocate private type

P as shared type S, we must later be able to use the shared type S as P.

Type mapping may cause class loading in a VM. This is because whenever we encounter an instance of a shared type T, which maps to a private type U, we must load class U. Thus, CoLoRS introduces a new class loading barrier.

Since in static languages, the static type of a field is available, we permit certain conversions while mapping shared fields to private fields. Let us denote any private class to which a shared class S maps as map(S). For a given field of shared type S and of private type P, CoLoRS allows both upcasts and downcasts during mapping.

Upcasts occur if class P is a superclass of class map(S) or class map(S) implements interface P. For instance, we have an upcast when we map a field of shared type $list$ to a field of private type $List$ (because map($list$) = $SharedList$ and $SharedList$ implements the $List$ interface). Or we have an upcast when we map a field of shared type $string$ to a field of private type $Object$, because $Object$ is a superclass of class map($string$) = $String$. Upcasts are most useful to support interface-type private fields, such as $List$ in Java.

Downcasts take place if class P subclasses map(S). For example, there is a downcast if a field of shared type $list$ is mapped to a field of private type $String[]$, because $String[]$ subclasses $Object[]$ = map($list$). Thanks to downcasts, private arrays (whose elements are typed) can conveniently access shared lists (whose elements are untyped).

To ensure type safety, downcasts require a read barrier which checks the actual object type on each read access. Upcasts represent a covariant type operator (analogous to the array upcasts in Java) and therefore require a write barrier that checks the type of the stored object against the expected static type.

3.1.6 Dynamic Languages

In dynamic languages, fields are accessed by name (not by offsets) and static field types are not available. Therefore, when creating a new shared class or comparing to an existing one, CoLoRS relies on actual types of all non-null attributes in a particular object (i.e. the one being copied to shared memory). This results in type concretization – shared classes created by dynamic runtimes always have the most derived field types. We ignore NULL fields as for them no static type can be inferred. When looking for an exact type match (during shared allocation), we allow type conversions (upcasts and downcasts). No read barrier is necessary as dynamic languages do not guarantee any particular type for any field. However, each field store must verify the type of the stored object against an appropriate static shared type (via a write barrier).

When mapping a shared type S to a private type P, we do not map fields, as we do not have field types and offsets in P. Instead, we just create a hash table mapping field names to shared offsets. This speeds up attribute access (which is done via names). Since multiple private types can be mapped to a single shared type (e.g. $list$ and $tuple$ in Python both map to shared $list$), we employ multiple inheritance if possible (e.g. in Python) or we extend the runtime to simulate it for the types in question.

CoLoRS uses reverse mapping to avoid shared class lookup on each allocation. Reverse mapping can improve performance only if private instances of a single private class have similar attribute sets (a natural property but one that is not always enforced by dynamic languages). Otherwise, the system might end up relying on dynamic field addition frequently as some objects' types may be mapped to static types that have too few static attributes.

3.2 The CoLoRS Memory Model (MM)

CoLoRS defines a new memory model that builds on and simplifies memory models supported by mainstream languages. CoLoRS MM is equivalent to the Java MM for programs that do not contain data races. Java programs that rely on $volatile$ and $final$ fields or other race-related aspects of the Java MM may work incorrectly with CoLoRS because shared object fields drop their Java-specific modifiers. Python does not define any MM so using CoLoRS cannot break extant Python programs.

Following the Java Memory Model (JMM) approach and recent standardization effort for the C++ MM [9], CoLoRS guarantees sequentially consistent semantics only to programs that are properly synchronized (i.e. those that do not contain data races). A data race occurs when multiple threads can access the same object field at the same time and at least one of them performs a write.

Similarly to Java and C#, CoLoRS provides monitor synchronization. Monitors provide mutual exclusion for threads and restrict re-ordering of memory accesses. Monitor entry has load acquire semantics (downward fence) while monitor exit has store release semantics (upward fence). Full memory fence is not supported in CoLoRS (following Java and C# design). In CoLoRS, monitors are fault-tolerant: if a VM dies while holding a monitor, subsequent acquisitions of this monitor do not result in a deadlock or access to corrupted data, but throw a runtime exception before entering a critical section.

Like the JMM (and unlike the C++ MM), CoLoRS must guarantee basic type- and memory-safety even in the presence of data races. Therefore, in CoLoRS, all pointer stores and loads are always safe (even with data races). This property is relatively easy to implement (an aligned machine-word-wide load/store is atomic on most architectures). This property is not strictly necessary for type-safety in case of primitive values, like integer or float, and therefore CoLoRS does not guarantee it for non-pointer fields. Operations like shared class creation or dynamic field addition are always thread-safe because they are rare and can be internally protected by a lock.

Note that CoLoRS MM avoids many of the complexities of the JMM by supporting only instance field sharing (no

statics, no methods, no constructors) and ignoring field modifiers like $final$ and $volatile$. Unlike C++ MM, CoLoRS MM does not support atomic operations and the $trylock$ functionality, which simplifies the model significantly.

3.3 Monitor Synchronization

CoLoRS employs a new synchronization mechanism that is an adaptation and simplification of extant, commonly-used schemes. We have found these schemes inadequate for CoLoRS because of their complexity, tight integration with VM services, and reliance on the ability to stop all the threads.

State-of-the-art high-performance VMs, like HotSpot JVM, use biased locking [32] to avoid atomic CAS operations in the common case. However, biased locking requires safepoint support – it occasionally needs to stop all the threads to recover from its speculative behavior. One of the design goals in CoLoRS is to avoid stopping all VMs at once – such system-wide safepoints are inherently unscalable and introduce lengthy pauses. Therefore, biased locking is not suitable for CoLoRS.

Another commonly-used locking scheme is lightweight locking [32], which strives to avoid using OS primitives in the common case by relying on atomic CAS operations. We have investigated the efficacy of this approach and found that in modern OSes that provide futexes (fast user-mode locking primitives), lightweight locking performs worse that an OS mutex. In older OSes, OS-backed synchronization was slow because it required kernel entry/exit. Linux implements futexes that in the uncontended case perform one atomic CAS. In contrast, lightweight locking needs two atomic CASes, one for locking and one for unlocking [32] path. We have compared the cost of 2 atomic CASes with POSIX mutex lock and unlock. Our results show that two atomic CASes are slower: on a dual-core Intel Core2 by 31%, and on a quad-core Intel Xeon by 45%. Therefore, we have designed CoLoRS to use OS primitives (POSIX mutexes based on futexes) directly.

Most extant monitor implementations (e.g. HotSpot JVM) reserve a word in the object header to assign a lock pointer to an object once a lock is needed. The presence of such a pointer leads to significant design complexity in extant systems because once the pointer is set, one can only clear it when all threads are stopped or the object has become unreachable. CoLoRS does not ever stop-the-world (halt/safepoint all threads in the system), hence we take a different approach.

Instead of using a pointer to a monitor, we hash the object address (shared objects do not move in CoLoRS) into a fixed-size table of monitors kept in shared memory. Since few objects are used as monitors at a time, it is unlikely that multiple simultaneously locked objects will ever hash to the same monitor-table entry. Thus, we can multiplex OS locks without significant loss of concurrency level in the common case. Note that such multiplexing of OS locks due to hashing conflicts is correct and does not lead to a deadlock because CoLoRS translates notify operations to notify-all operations. Even if two or more threads block on the same monitor while synchronizing on different objects, each monitor notification awakens all blocked threads and a progress can be made. If a thread acquires locks on multiple objects then if two or more objects end up using the same monitor, locking remains correct because locks are recursive.

The above synchronization scheme can be transparently integrated into Java based on Java monitors. Python does not support the monitor abstraction (locks are not associated with objects) and therefore needs to be extended with dedicated API for monitors.

3.4 Garbage Collection

Since CoLoRS targets multi- and many-core systems and avoids system-wide safepoints, the most appropriate GC algorithm for shared objects is parallel (i.e. using multiple GC threads), concurrent (i.e. performing most work without stopping the application), and on-the-fly (i.e. stopping at most one thread at a time) GC. In addition, CoLoRS needs a non-moving, mark-sweep-style GC because some runtimes (e.g. Python) assume that objects do not move and other ones (e.g. Mono for C#) use conservative stack scanning.

We have found extant on-the-fly mark-sweep GCs to be unsuitable given the CoLoRS architecture and requirements. Therefore, we have designed a new variation of snapshot-at-the-beginning (SATB) GC, which is parallel, concurrent, and on-the-fly.

The state-of-the-art in on-the-fly GC systems include those that employ the Doligez-Leroy-Gonthier [17, 18] algorithm and its extensions by Domani et al. [19, 20] for generational heap layout and multiprocessors without sequential consistency.

State-of-the-art, snapshot-based, on-the-fly GC algorithms require multiple (three to start the collection cycle) system-wide handshakes with all the threads. The mutators must check whether they need to respond to handshakes regularly during their normal operation. For scalability, we designed CoLoRS to work at the granularity of VMs, not individual threads. The handshakes would require keeping track of all threads in all VMs. In addition, we do not want to require VMs to implement the per-thread handshake-polling mechanism, as it is not generally supported in VMs.

A design goal of CoLoRS GC is to abstract away private VM memory management to one operation: shared root report, without imposing any specific implementation details. As a result, we have designed an on-the-fly GC that does not use handshakes and works at the VM level (not thread level). In addition, the CoLoRS GC is simpler (as it does not have any phase transitions) and guarantees termination (some previous algorithms unreliably depend on the relative speed of the collector and mutation rate for termination).

CoLoRS uses thread-local allocation buffers (TLABs) to reduce allocation cost. Each thread performs bump-

pointer unsynchronized allocation in its own TLAB. Once the TLAB is exhausted, it is retired, and the thread requests a new one. VMs request TLAB or large-object allocation directly from the object space. The freelist contains all unallocated blocks whose size is at least the TLAB size. The freelist is protected by a lock.

3.4.1 GC Algorithm

Our GC comprises four concurrent phases: flag clearing, root report, marking, and sweeping. The CoLoRS server initiates a new GC cycle as soon as the heap usage crosses a specified threshold. The main GC thread is awoken by an allocating thread once this happens. CoLoRS GC imposes no pauses. If a VM is capable of reporting shared roots without causing internal pauses (e.g. as Python can), then the system never needs to pause any threads.

Flag clearing. The main GC thread first clears all GC-related flags in the heap. This operation is fully concurrent. Each object has three GC flags: pending (i.e. it needs to be recursively marked), marked (i.e. it has been recursively marked), and recent (it has been recently allocated).

Unlike in extant SATB GCs, in CoLoRS, the snapshot mode is active all the time. This simplifies the algorithm as it avoids complex state transitions and handshakes. The snapshot mode means that all objects are allocated live (i.e. with the recent flag set) and mutators use a write barrier: on pointer stores they mark the overwritten pointer as live (i.e. they set the pending flag). When GC scans a live object it sets its marked flag. During the flag-clearing heap scan, the main GC thread also computes a fully-balanced heap partitioning that is used later on for parallel scanning. The key system invariant is that it is always possible to sequentially scan all blocks in the heap, without any synchronization. We carefully design allocation procedures so that we do not break this invariant.

GC flag clearing has a similar effect to activating the snapshot mode from scratch in other algorithms, but does not require handshakes. Once GC flags are cleared, the main GC thread requests root dumps from all attached VMs.

Root report. Each VM must be able to identify pointers into shared memory in its private heap/stacks in an efficient way. In VMs using tracing GC this is straightforward – we either scan the whole heap (non-generational GC) or use a card table (generational GC). In the latter case, we extend the card table so that we can quickly find not only pointers from the old generation(s) to the young generation but also pointers from the old generation(s) to shared memory. To report shared roots, we simply trigger a fast minor collection and efficiently find all pointers to shared memory.

In VMs which use reference counting GC (e.g. cPython), CoLoRS can track shared roots as they are created and destroyed, thus being able to report them any time without any processing. For each shared reference, we create a small proxy object in private memory with reference count set to one. Once the proxy object becomes unreachable (which we know immediately thanks to reference counting) we reclaim it and forget the shared root. Note that only private references can exist to the proxy object since there are no shared-to-private pointers.

CoLoRS requests roots from each VM and waits until all reports arrive. To report a shared root, a VM sets the object's pending flag. To ensure store visibility, a memory fence takes place on both sides once the reporting completes. CoLoRS does not use timeouts because it detects VM termination in a reactive way via TCP/IP sockets. Termination is noticed right away and the exited VM is removed from the waiting-for-roots list.

Marking. As soon as all roots are reported, the main GC thread initiates parallel, concurrent marking done by several worker GC threads. Each worker thread scans its own heap partition looking for pending objects, and recursively marks them using depth-first search. To ensure dynamic load balancing during marking, worker GC threads employ randomized work stealing. GC threads use barrier synchronization to meet at subsequent GC phases.

Once first marking completes, the main GC thread enters a loop. During each iteration, CoLoRS performs parallel, concurrent marking from pending objects. However, this time it stops marking the object graph once its sees an object with the recent flag set. The loop terminates when no new objects have been marked. Stopping marking on recently-allocated objects guarantees GC termination – there is a finite number of "old" objects in the heap when the GC starts, and all the newly-allocated objects are being flagged as recent. Therefore, GC must finish in a finite number of steps.

For correctness, we must prove that a recently-allocated object cannot have a pointer to an object that is live but otherwise unreachable and invisible to GC (and thus it cannot be incorrectly left unmarked). Note that such a situation may occur during the first marking pass, which marks from the VM roots. Our snapshot write barrier (SATB WB) does not capture root pointer updates – it only captures heap stores. Suppose that root r points to object O, and a new object N is allocated having its only pointer set to O. If root r is later updated to point to N, we end up with a newly-allocated object N that has a pointer to a live object O that is reachable only through N. The reason for this is that we do not notice root updates. Such a situation is impossible from the second marking on, as during 2nd and subsequent markings we ignore roots and mark from the pending flags only (i.e. from heap objects that are protected by SATB WB). Reconsidering our example in the heap context: object O is marked as pending on r update, and will be marked/scanned even if we stop marking on object N (which has its recent flag set).

Sweeping. As soon as the marking loop terminates, CoLoRS moves on to concurrent, parallel sweep. Each worker GC thread scans its heap chunk trying to find the first potentially-free (candidate) block. This scan is done without synchroniz-

ing with mutators that are actively allocating objects. Once a GC thread finds a candidate block, it acquires the freelist lock and continues the scan as long as it encounters reclaimable blocks. Finally, it removes all found dead blocks from the freelist and inserts one coalesced block into the freelist. The GC thread releases the freelist lock and looks for the next candidate block. Our GC-mutator contract guarantees that all block headers are always parsable.

4. CoLoRS Implementation

CoLoRS can work under any OS that supports adequate IPC functionality. We have implemented CoLoRS in HotSpot JVM 1.6 and cPython 3.1.

We find that extending a VM with CoLoRS support is relatively straightforward. The first step is to determine the VM object/class model, its relationship to the CoLoRS OM, memory management (GC) algorithm(s), and operations that use objects, typically field access, method calls, synchronization, etc. Next, we define type mapping for builtins and user-defined types, and add any runtime extensions (such as multiple inheritance) to support it. The next step is heap access virtualization which amounts to extending an interpreter, a JIT compiler, or both, to provide a separate control path for handling shared objects. Depending on a VM, other components may need similar extensions, e.g. the GC subsystem. Typically, we must intercept all program instructions that read/write heap objects. Next, we insert calls to the CoLoRS API along the newly added control paths. This step translates VM-specific operations into VM-neutral operations (e.g. getting an attribute by name into getting a field by offset). Lastly, we add GC runtime support – we implement a dedicated CoLoRS thread and the shared-root-dump operation in the private GC system.

4.1 Shared Memory Segment

The CoLoRS shared memory segment contains three spaces: metadata, classes, and objects. The objects space is a garbage-collected mark-sweep heap with TLAB/free-list allocation. The classes space is a bump-pointer space for immortal objects that contains shared classes, class version lists, and registered object repositories/channels. The metadata space contains several pointers to objects allocated in the classes space: pointers to all builtin types, pointers to the repositories/channels hash tables (mapping names to repositories/channels), a pointer to class versions hash table (mapping names to class version lists), as well as user-level monitors, internal system locks, the freelist head and space usage statistics, and the bump-pointer top (for the classes space).

Each CoLoRS monitor has its POSIX mutex and condition variable. We use the PTHREAD_PROCESS_SHARED flag to make the POSIX mutexes and conditions work across OS processes. In addition, monitors use the recursion count (to avoid re-locking by the same thread) as well as owner ID (VM ID plus thread ID).

The CoLoRS server maintains additional state (metadata) in private memory to manage GC threads, and to track attached VMs. For each attached VM there is a dedicated monitoring thread, which detects VM termination using an open TCP/IP connection. On VM termination, the monitoring thread receives an error when reading from a closed socket. Note that OS-level IPC (e.g. sockets) is the only reliable way of detecting process termination without resorting to timeout/keep-alive solutions. This is because in Unix systems certain signals (e.g. the KILL signal) cannot be intercepted.

We group class versions into lists based on their name. Object repositories/channels and classes are permanent entities – we do not collect them as they are reusable. Object repositories/channels are treated as GC roots during GC.

GC flags are implemented as one-byte-wide fields because of concurrent access. We assume that writes issued by a particular thread are visible to other threads in the order they are issued (sequential consistency guarantees this).

The objects space is a contiguous sequence of blocks. Each block can be an object, a free chunk (part of the freelist), or a TLAB. The block header contains two fields: block length and block type. This enables quick traversal of the heap without parsing actual objects – a key property for our concurrent GC. TLAB blocks contain an owner ID, which identifies the VM that is currently using the TLAB. This enables us to reclaim TLABs orphaned by asynchronously terminated VMs.

To provide transparent object sharing, CoLoRS intercepts all VM operations that access heap memory. To efficiently check whether an object is shared, CoLoRS uses a constant border between private and shared area in the virtual memory. Each memory-related operation, such as field access, compares the pointer value against this constant border.

4.2 HotSpot JVM

In static runtimes with high-performance, adaptively optimizing compilers, border-checks may be expensive as they make the intermediate code larger and more difficult to optimize. Therefore, in our CoLoRS implementation in the HotSpot JVM server compiler, we compile methods in two modes: CoLoRS-aware and CoLoRS-safe. The CoLoRS-aware mode is used for methods in which shared memory has been determined (via profiling during interpretation) to be commonly-used. For such methods, border-checking overhead and the additional code that handles the shared pointers are acceptable.

The remaining methods (a vast majority in practice) are compiled in the CoLoRS-safe mode, where private pointers are the common case. The CoLoRS-safe methods contain only the minimum number of border-checks needed to take a trap on shared pointers. Such traps deoptimize the method and recompile it as CoLoRS-aware, running the method in the interpreted mode in the meantime. The CoLoRS-aware methods use fast upcalls to C to handle shared pointers

(CoLoRS is implemented in C). If fast upcalls fail (e.g. because class loading is needed), we bail out to the interpreter.

In CoLoRS-safe methods, we combine null checks with shared-border checks. Assuming that shared memory area is at lower virtual addresses than the private area, checking if a pointer is below the border detects both NULL pointers and shared pointers. If the check passes, we trap to the interpreter, which finds the actual cause of a trap itself (the trap cost is not a problem as it is the uncommon case path). In CoLoRS-aware methods we guard virtual method calls to prevent calling into a CoLoRS-safe method with a shared receiver (such calls need a trap). CoLoRS-safe methods must translate user-provided null checks into null-and-border checks to avoid eliding border checks along with null checks.

We also perform approximate data flow analysis which conservatively computes all methods which can operate on a pointer to a shared object. The analysis exploits the fact that shared pointers can only be produced by the methods from the CoLoRS API. We dynamically and incrementally build the call graph as classes are loaded. In the graph, nodes represent methods and there is an edge from node m to n, if method m can pass/return a reference to method n. In case of interface methods, we have additional edges leading to all implementors of a particular method. We divide all loaded methods into two classes: private and potentially-shared. Private methods can never reach shared objects. If any potentially-shared method contains the $putstatic$ bytecode, then we assume all methods containing the $getstatic$ bytecode to be potentially-shared. Otherwise, if a method is reachable from a potentially-shared method in the call graph, that method is also considered potentially-shared. Potentially-shared methods are compiled as either CoLoRS-aware or CoLoRS-safe, depending on the profiling data. Private methods do not contain any instrumentation. If class loading makes a previously-private method potentially-shared, we make the method non-entrant and recompile it.

CoLoRS intercepts all bytecode instructions that access objects in the heap (both fields and object header): putfield, getfield, arrayload, arraystore, invoke, monitor-related ones, arraylength, and objectclass. We extend the HotSpot template interpreter and the server compiler (both targeting amd64). In addition we virtualize the HotSpot runtime written in C (biased locking, GC, class loading, JNI, JVM, JMM, JVMTI). Several internal classes are not allowed to be in instantiated in shared memory (e.g. Thread, ClassLoader) – they are VM-specific and do not make sense in the context of other VMs.

4.3 cPython Runtime

We virtualize shared objects via private proxy objects, each containing a forwarding pointer to a shared object and a normal Python header (comprising private type and a reference count). This design choice is dictated by the fact that Python uses reference counting GC and CoLoRS uses tracing GC (so there is no reference counts in shared object headers). The cost of one level of indirection is fully compensated by the fact that we do not need to perform type mapping on each shared object access – proxy objects have their private type computed once. All proxy objects have the same size and are bucket-allocated in a dedicated memory region (for fast border checks). Deallocation takes place once a reference count drops to zero. Thus, the number of proxies never exceeds the number of private-to-shared pointers. Finding shared roots in such a setting is fast and amounts to a linear scan of the proxy object region.

Proxy objects also simplify Python runtime virtualization, as the Python interpreter dispatches basic operations such as field access, method call, and operator evaluation, based on object type (note that proxies already have the proper private type set). We provide a new private type for each builtin shared type, and the interpreter automatically invokes the right implementation (shared/private). Python VM allocates only one global TLAB because the interpreter is single-threaded and simulates multi-threading by context-switching between program threads. The Python runtime component most complex to virtualize are standard libraries and builtin types, which provide rich, complex interfaces (e.g. for sorting, set algebra, etc).

5. Related Work

CoLoRS is unique in that it is the first system to support type-safe, transparent, and direct object sharing via shared memory between managed runtimes of different object-oriented languages. To enable this, we design a new language-neutral object and class model, memory model supporting monitor synchronization, and a new parallel and concurrent, pause-free GC system.

CoLoRS takes a top-down approach to object sharing between runtimes for high-level languages. That is, we assume full isolation between the runtimes via operating system process semantics and provide a new mechanism for object sharing within this context. Several previous systems [6, 16, 23] took a bottom-up approach by executing multiple applications in a single OS process and providing software-based isolation between them.

Note that although one can use CoLoRS to implement an efficient cross-language RPC for the co-located case (similar in spirit to LRPC [7]), CoLoRS is more general than RPC systems and it focuses mainly on transparent object sharing (as opposed to message passing). CoLoRS is the first system to enable type-safe, transparent, shared memory across OO language runtimes. Thus, CoLoRS differs significantly from RPC systems (such as LRPC) in terms of both architecture and programming model.

State-of-the-art systems that support type-safe, cross-language communication for OO languages, such as OMG CORBA [13], Apache Thrift [34], Google Protocol Buffers [31], SOAP, and REST, target distributed systems and rely on

message-passing and data serialization. CoLoRS differs from these systems in that it targets co-location and shared memory (as opposed to message passing). CoLoRS is complementary to RPC frameworks as it can be used to optimize RPC in the co-located case. Note, however, that CoLoRS is not an RPC system and its primary goal is type-safe, transparent object sharing.

The XMem system by Wegiel and Krintz [36] is most related to ours. XMem provides direct object sharing between JVMs. XMem also takes a top-down and transparent approach, but does not support sharing between heterogeneous languages and requires global synchronization across runtimes (which CoLoRS avoids) for such operations as garbage collection, class loading, shared memory attach/detach, and communication channel establishment.

Systems supporting communication between isolated tasks within a single-language, single-process runtime include Erlang [3], KaffeOS [6], MVM [16], Alta [5], GVM [5], and J-Kernel [35]. These systems take a bottom-up approach which provides weaker isolation (i.e. weaker protection guarantees than the CoLoRS approach) and is more complex to implement. Unlike CoLoRS, they replicate operating system (OS) mechanisms within a single OS process instead of leveraging existing OS inter-process isolation.

Language-based operating systems also provide mechanisms for communication and interoperation between processes [8, 21, 23, 25, 27–29, 33, 37]. Such systems typically implement support for light-weight processes that share a single address space and provide compiler support to guarantee type and control safety within and between processes. To facilitate the latter, these systems require that the components (processes/tasks) be written in the same safe/checkable language. In addition, since CoLoRS is not an operating system, it is significantly simpler.

Some concurrent languages provide direct support for inter-process communication between light-weight processes [4, 14, 21] written in the same language. The key difference between these systems and CoLoRS is that they employ share-nothing semantics for message-based communication whereas CoLoRS provides support for direct object sharing when runtimes are co-located on the same physical machine.

CoLoRS is also distinct from distributed shared memory and single system image runtimes for clusters such as MultiJav [11], cJVM [2], JESSICA [30], Split-C [15], and UPC [22]. In contrast to them, CoLoRS provides a uniform cost for accessing all objects (private and shared) and does not target distributed computing. These systems provide sharing between code written in the same language, and focus on guaranteeing memory consistency and cache coherence for concurrent access to objects across multiple machines.

6. Experimental Evaluation

An important practical use case for CoLoRS is improving communication performance of RPC in the co-located case. We evaluate CoLoRS in this context because there are cross-language RPC frameworks, such as CORBA, Thrift, Protocol Buffers, and REST, to which we can compare. CoLoRS, however, provides significantly more functionality over extant cross-language RPC systems by enabling direct, type-safe, and transparent object sharing.

We compare CoLoRS-based RPC against extant RPC frameworks in terms of communication performance (i.e. latency and throughput). We also evaluate end-to-end server-client performance (response time and transaction rate) for two applications: Cassandra and HDFS. Finally, we measure the overhead of CoLoRS in programs that do not employ shared memory, using standard community benchmarks for Java and Python.

6.1 Methodology

Our experimental platform is a dedicated machine with a quad-core Intel Xeon and 8GB main memory. Each core is clocked at 2.66GHz and has 6MB cache. We run 64-bit Ubuntu Linux 8.04 (Hardy) with the 2.6.24 SMP kernel.

We use HotSpot JVM from OpenJDK 6 build 16 (April 2009) compiled with GCC 4.2.4 in the 64-bit mode. Our configuration employs the server (C2) compiler, biased locking, and parallel GC (copying in young generation and compacting in old). For the Python runtime we use the open-source cPython 3.1.1 (released August 2009) compiled with GCC 4.2.4 in the 64-bit mode.

To measure CoLoRS overhead in Java, we use DaCapo'08 and SPECjbb ('00 and '05). We set the heap size to 3.5x the live data size so that GC activity does not dominate performance and so that we capture all sources of overhead. We use the default input for DaCapo and 5 warehouses, with 90s runs, for SPECjbb.

In Python, we evaluate CoLoRS overhead using PyBench (a collection of tests that provides a standardized way to measure the performance of Python implementations), a set of Shootout cPython benchmarks (from [1]), and PyStone (a standard synthetic Python benchmark).

In all experiments, we repeat each measurement a minimum of seven times. For experiments that employ shared memory, we perform sufficient iterations to guarantee that GC is performed by CoLoRS. We report average values. The standard deviation is below 5% in all cases.

CoLoRS reserves 256MB in shared memory for objects and 64MB for classes. We use 32KB TLABs, and 2 parallel GC threads. In each experiment, we employ two co-located runtimes: Python and Java. Whenever running an unmodified (CoLoRS-unaware) JVM, we set its heap size to 300MB so that its private memory is comparable in size to the shared memory.

Note that our results underestimate CoLoRS potential since we implement CoLoRS in Python 3.1 and compare its communication performance with RPCs running on Python 2.6. This is because the RPC frameworks that we use have not yet been ported to Python 3.1. To quantify this difference we evaluate the performance of Python 3.1 relative to Python 2.6. The last Column in Table 5 shows the overhead of Python 3.1 relative to Python 2.6 across our set of benchmarks. On average, Python 3.1 is slower by 20%.

6.2 CoLoRS Impact on Communication Performance

We first evaluate the performance potential of CoLoRS-based RPC using communication microbenchmarks with a range of message types and sizes. We implement equivalent microbenchmarks using RPC frameworks for CORBA, Thrift, Protocol Buffers, and REST. We compare RPC latency and throughput (call rate).

For the implementation of the microbenchmarks, we use a Python client and a Java server. Whenever possible we employ RPC methods with fully symmetric input and output (i.e. returning a data structure similar to the data structure passed in as an argument). This ensures that the server and the client exercise data structure (de-)serialization in a symmetric way.

To evaluate RPC throughput, we vary method input/output size between 1 to 1024 units and measure mean time per method call. Next, we use least-squares linear regression to compute throughput from the coefficients in the equation $time = latency + size/throughput$. We calculate latency as the mean time needed per call for unit input/output. We employ this methodology because we have observed that for small input sizes the function $time(size)$ is sometimes non-linear and approximating it by a line leads to an inaccurate latency estimation.

Each RPC method call takes a list as input and returns a list as output. List sizes vary between 1 and 1024. For each list size we do 10 experiments and use their average in the calculation above. We use several different objects as list elements, including built-in primitive types (string, integer, float, and boolean) and user-defined types. For the latter we employ binary trees, the depth for which ranges between 1 and 4 levels, and each node contains 4 primitive fields. This enables us to investigate both shallow- and deeply-linked data structures. The above choice is also dictated by the limitations of extant RPC frameworks which support a small set of builtins and do not support recursive data structures. (Note that CoLoRS provides a richer and more flexible object model than these RPC systems.)

We implement an RPC endpoint in CoLoRS as a message queue on which a server waits for messages (call requests). Each message is an object encapsulating input and output. A client issues a call by allocating a message object (and the associated input) in shared memory, enqueuing it, and notifying the server. The server removes the request from the queue and generates the output in shared memory. Finally, the server notifies the client that the result is ready (as the output field in the message object).

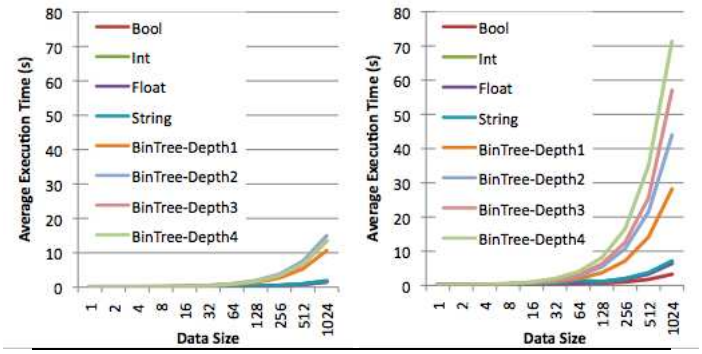

Figure 3. Average execution time (in seconds) for CoLoRS (left) and CORBA (right) experiments.

For all experiments, we report throughput as the number of calls per millisecond, and latency in milliseconds. We compute these values from the timings that we collect using the methodology described in the previous subsection. Due to space constraints, we only present timings graphs that compare CORBA to CoLoRS. This data is shown in Figure 3. The x-axis is message size and the y-axis is time in seconds. This data is representative of all of the RPC experiments. We summarize the latency and throughput of each below.

Table 2 shows throughput across all microbenchmarks and RPC systems. We report both absolute values and relative improvement due to CoLoRS. Table 3 uses a similar format but presents results for our latency measurements.

CORBA. The Common Object Request Broker Architecture (CORBA) [13] standardizes object-oriented RPC across different platforms, languages, and network protocols. A client and a server use automatically-generated stubs and skeletons to (de)marshall arguments and return values for methods specified in the Interface Definition Language (IDL). To implement our CORBA benchmarks, we use the $org.omg.CORBA$ package and the $idlj$ compiler in Java and the $Fnorb$ module and the $fnidl$ compiler in Python. Our measurements indicate that, compared to CORBA, CoLoRS achieves 11–27 times better throughput and 14–19 times lower latency.

Thrift. Thrift is a framework originally developed at Facebook for scalable cross-language RPC. Like CORBA, Thrift requires a language-neutral interface specification from which it generates client/server template code. However, Thrift is simpler and much more lightweight than CORBA. We use Apache Thrift version 2008/04/11. Our experiments show that CoLoRS improves throughput by 8–17 times and latency by 2–13 times, over Thrift. We also find that Thrift achieves much better performance for builtin types than for user-defined types.

	Throughput in calls/ms; CoLoRS/RPC in parenthesis							
	bool	int	float	string	nodes:1	nodes:2	nodes:3	nodes:4
CORBA	173.22 (11)	82.67 (26)	83.20 (27)	75.96 (15)	14.67 (13)	4.68 (15)	1.83 (17)	0.86 (17)
ProtoBuf	31.73 (59)	30.98 (70)	34.32 (65)	26.43 (43)	2.85 (68)	0.88 (78)	0.36 (85)	0.17 (91)
REST	23.17 (81)	22.45 (97)	21.89 (102)	22.94 (50)	8.73 (22)	2.66 (26)	0.91 (34)	0.31 (49)
Thrift	237.04 (8)	283.23 (8)	274.37 (8)	149.08 (8)	15.38 (13)	4.27 (16)	1.80 (17)	0.87 (17)
CoLoRS	1876.08 (1)	2175.32 (1)	2231.45 (1)	1144.87 (1)	193.66 (1)	68.61 (1)	30.61 (1)	15.08 (1)

Table 2. Throughput for Microbenchmark Programs. For each data type, we show the throughput in calls per millisecond; in parentheses, we show the CoLoRS/RPC throughput ratio. $nodes : n$ means the type is a binary tree of depth n.

	Latency in msecs; RPC/CoLoRS in parenthesis							
	bool	int	float	string	nodes:1	nodes:2	nodes:3	nodes:4
CORBA	0.62 (14)	0.65 (19)	0.62 (14)	0.63 (14)	0.68 (17)	0.82 (15)	1.13 (17)	1.92 (19)
ProtoBuf	0.22 (5)	0.31 (9)	0.21 (5)	0.23 (5)	0.55 (14)	1.32 (23)	2.90 (44)	6.02 (58)
REST	3.89 (90)	3.89 (113)	4.00 (89)	3.92 (90)	4.07 (101)	4.80 (85)	7.35 (111)	9.94 (96)
Thrift	0.09 (2)	0.10 (3)	0.11 (3)	0.12 (3)	0.19 (5)	0.35 (6)	0.74 (11)	1.38 (13)
CoLoRS	0.04 (1)	0.03 (1)	0.04 (1)	0.04 (1)	0.04 (1)	0.06 (1)	0.07 (1)	0.10 (1)

Table 3. Latency for Microbenchmark Programs. For each data type, we show the latency in milliseconds; in parentheses, we show the RPC/CoLoRS latency ratio. $nodes : n$ means the type is a binary tree of depth n.

Protocol Buffers. Protocol Buffers (PB) are a language-neutral, platform-neutral, extensible mechanism for serializing structured data developed by Google engineers as a more efficient alternative to XML [31]. To use PB, developers specify message types in a $.proto$ file, and a PB compiler generates data access classes that allow to parse/encode objects into a bytes buffer/stream. We use PB version 2.2.0, which includes message parsers and builders but does not support RPC. Therefore, we implement RPC on top of PB by using PB serialization and communication over TCP/IP sockets. Each message that we send from a client to a server, contains a method tag, message length, and PB-serialized data structure (method input). CoLoRS improves the throughput of PB-RPC by 43–91 times and latency by 5–58 times.

REST. REpresentational State Transfer (REST) [24] is a client-server architecture based on HTTP/1.0 where requests and responses are built around the transfer of representations of resources. REST provides stateful RPC by exchanging documents that capture the current or intended state of a resource. Individual resources are identified in requests by URIs. In our benchmarks, we define a single resource stored on a server and identified by `http://localhost:8080/db/items`. A representation of this resource is an XML document containing all stored items. Clients send GET requests to the resource URI, and parse the resulting XML document. This document contains a varying number of items (1–1024), where each item is either a primitive or a user-defined object. We employ the Python $restful\_lib$ to implement the client and the Java $restlet$ (version 1.1.6) for the server. Relative to REST, CoLoRS throughput is 22–102 times higher and latency is 85–113 times lower. REST has the highest latency among all of the RPC technologies that we investigate.

6.3 CoLoRS GC

We gathered basic GC statistics for our Java-Python microbenchmarks. The results are similar across all the payloads that we use (described in the previous Section). Below we discuss the experimental data obtained for 4-level binary trees.

We set the GC triggering threshold to 70%. Average time between subsequent GC cycles is 1458ms while average GC cycle time is 325ms (GC is active 18% of the time). Note that GC runs concurrently in a separate process. The clearing phase takes 94ms on average (29% GC cycle). The root dump phase was 1.2ms on average (below 0.4% GC cycle). In the HotSpot JVM, each root dump request causes a STW pause which averages at 0.8ms (with the maximum pause of 2.9ms). In cPython there is no pauses. The marking phase takes 116ms on average (36% GC cycle). Two object graph scanning iterations suffice on average (the maximum is 3). The sweep phase averages at 113ms (35% GC cycle). The dominating GC phases are marking, sweeping, and clearing, each taking around 1/3 of each GC cycle.

6.4 CoLoRS Impact on End-to-End Performance

To lend insight into the CoLoRS potential when used by actual applications, we investigate two popular server-side software systems: Cassandra [10] version 0.4.1 and HDFS [26] version 0.20.1. Cassandra is a highly scalable, eventually consistent, distributed, structured, peer-to-peer, key-value store developed by Facebook engineers. HDFS is the Hadoop Distributed File System – a file system server that provides replicated, reliable storage of files across cluster

resources. Both of these systems are employed for a wide range of web applications, e.g. MapReduce, HBase (open-source BigTable implementation), email search, etc.

Cassandra and HDFS both expose Thrift-based interfaces. These interfaces provide a set of query/update methods which use relatively complex data structures (e.g. maps). Query methods are natural candidates for in-memory result caching, recently a common approach to scaling up servers (e.g. MemchacheD, MySQL cache). If caching is used, then in the common case (i.e. on cache hit), server processing is minimal and therefore communication constitutes a large portion of the end-to-end performance.

In systems with in-memory caching, CoLoRS can improve performance in two ways. First, it can reduce RPC cost by avoiding serialization. Second, part of the in-memory cache can be kept in shared memory – immutable objects such as strings can be shared by multiple clients without the risk of interference. As a result, CoLoRS can provide copy semantics without actually copying data. To investigate both these scenarios, we extend Cassandra and HDFS with in-memory caches for particular queries and evaluate the efficacy of using CoLoRS for these queries, on end-to-end performance.

For Cassandra, we implement caching for the get_key_range query (parameterized by table name, column family, start value, end value, maximum keys count, and consistency level). The query returns a list of keys matching the given criteria. Updaters, such as insert and remove, detect conflicting modifications and invalidate the cache accordingly. The cache is kept on the server and maps inputs (serialized to a string) to responses. Cached responses are partially in shared memory (strings are immutable). Thus, CoLoRS has the potential for improving performance by avoiding serialization and reducing copying overhead.

For HDFS, we implement an in-memory cache for the $listStatus$ call, which, given a directory name, generates a list of $FileStatus$ objects, each describing file attributes, name, owner, permissions, length, and modification time. The cache is a map from path name to responses, which we partially store in shared memory. Cache invalidation happens on conflicting file system operations: create, append, write, rm, rename, mkdirs, chmod, and chown.

Figure 4 presents the timing data for Cassandra and CoLoRS (left graph) and HDFS and CoLoRS (right graph). The x-axis is message size and the y-axis is time in seconds. We use this data to compute latency and throughput, which we summarize in Table 4. Columns 2–3 show transaction rate (per millisecond) while Columns 4–5 present response time (in ms). We use one cache warmup iteration followed by 10 iterations during each of which we vary the query result size between 1 and 1024 entries. In each Column group, we report measurements for the server without CoLoRS and the relative improvement due to CoLoRS. For cache-enabled Cassandra, CoLoRS improves transaction rate by 19 times and reduces response time by 3 times. For cache-enabled HDFS, CoLoRS improves transaction rate by 20 times and decreases response time by 3 times.

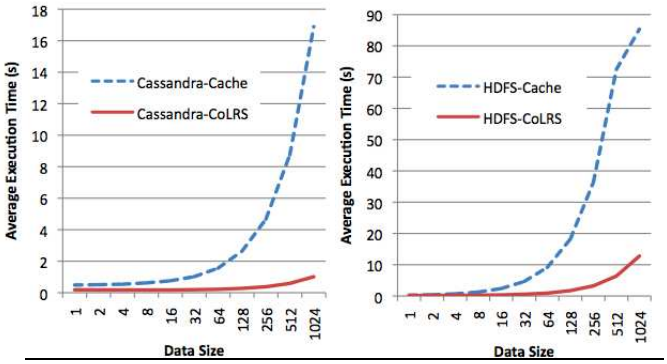

Figure 4. Average execution time (in seconds) for Cassandra (left) and HDFS (right) vs. CoLoRS .

Server Application	Throughput queries per ms	Throughput CoLoRS /App	Latency in ms	Latency App/ CoLoRS
Cassandra	249.50	19	0.12	3
HDFS	12.03	20	0.19	3

Table 4. End-to-End Performance for Cassandra and HDFS with Caching. The third and fifth Column show number of times improvement due to CoLoRS for throughput and latency, respectively.

6.5 CoLoRS Overhead

To implement CoLoRS, we virtualize components of Java and Python runtimes. This includes standard libraries, object field access, synchronization, method dispatch, interpreter, dynamic compiler, allocation, and GC. Doing so provides transparency, but introduces execution time overhead. To evaluate this overhead, we compare unmodified release versions of Python 3.1 and Java 1.6 with their CoLoRS counterparts.

Table 5 shows Python results. In Column 2, we report per-benchmark execution times for unmodified Python 3.1. Next, in Column 3, we present the CoLoRS overhead – percentage increase in execution times relative to Column 2. Across our benchmarks, the average CoLoRS overhead is 4%. Note that scripting languages are not concerned with enabling high-performance (they are interpreted and much slower than statically compiled code).

Table 6 shows the Java results. For each benchmark, we report its heap size and execution time (for DaCapo – the top 11 benchmarks) or throughput (for SPECjbb), and percentage CoLoRS overhead (Column 4). Across the benchmarks, the average CoLoRS overhead is 5%.

Bench-mark	Python 3.1 time (s)	CoLoRS 3.1 % OHead	Python 2.6 % Impr
binary-trees	6.79	3.39	-0.44
fannkuch	1.97	4.57	24.68
mandelbrot	15.32	7.18	66.52
meteor-contest	2.25	1.78	32.35
n-body	8.67	2.08	7.04
spectral-norm	14.31	5.73	18.85
pybench	3.92	5.20	1.18
pystone	4.09	5.87	12.98
Average	7.17	4.48	20.40

Table 5. The overhead of CoLoRS support for Python (and for the use of Python v3.1 over v2.6). Column 2 is execution time in seconds. Column 3 shows the percent degradation due to CoLoRS. Column 4 shows the percent improvement in performance when we use Python 2.6 (over 3.1).

Bench-mark	Heap Size	ET or TP	CoLoRS Support % OHead
antlr	7	2.40	8.4
bloat	28	6.34	6.3
chart	42	6.19	6.1
eclipse	115	24.54	4.7
fop	28	2.11	7.7
hsqldb	280	3.35	3.6
jython	3	8.35	4.5
luindex	7	7.50	9.0
lusearch	45	4.25	1.4
pmd	56	6.92	8.6
xalan	105	5.97	-0.6
jbb'00	900	112726	5.3
jbb'05	900	54066	1.3

Table 6. The overhead of CoLoRS runtime support for Java. Column 3 is execution time (ET) in seconds for all but jbb'00 and jbb'05 for which we report throughput (TP). Column 4 shows the percent degradation due to CoLoRS.

6.6 Sockets vs. Shared Memory

We also investigate the relative performance of shared-memory-based transport (SMTx) and local-socket-based transport (LSTx). This enables us to determine how much performance improvement is due to the use of shared memory versus of sockets and due to avoiding object serialization.

In this experiment, we extend the Thrift RPC framework for Java with SMTx and compared it with the LSTx built into Thrift using our microbenchmarks (described in Section 6.2). We have implemented the Thrift transport layer on top of a bidirectional FIFO channel based on a shared memory segment and POSIX mutexes/conditions. We focus on Java and Thrift here because of their high-performance characteristics.

We observe that Thrift over LSTx attains better throughput – the improvement ranges from 1.7x (for the integer payload) to 3.2x (for 4-level binary trees) and averages at 2.7x. At the same time, Thrift over SMTx has lower latency for small messages (up to 29% for the integer payload) and higher latency for larger payloads (up to 0.8x for 4-level binary trees), while averaging at 9% lower latency than Thrift over LSTx.

Thrift/LSTx achieves better communication performance than Thrift/SMTx because of the zero-copy network-stack optimizations in the Linux kernel as well as faster synchronization (kernel vs. user-land POSIX). Based on this experiment, we can conclude that CoLoRS improves throughput and latency because it avoids serialization and not because it uses shared memory instead of sockets.

6.7 Summary

CoLoRS can improve communication performance significantly when runtimes executing interoperating components (written in different languages) are co-located on the same physical system, compared to extant type-safe cross-language RPCs (latency 2–113 times and throughput 8–102 times). In systems with short request processing times (e.g. servers with caches) this improvement can translate to large end-to-end performance gains (19–20x for transaction rates and 3x for response times). As more and more components are co-located on multi-cores and caches become prevalent in servers, object sharing systems like CoLoRS have a growing potential for increasing performance of multi-component, multi-language systems.

7. Conclusions

We have presented the design and implementation of CoLoRS, the first system supporting cross-language, type-safe shared memory for co-located VMs. CoLoRS contributes a new language-neutral object/class/memory model for static and dynamic OO languages, as well as a novel pause-free concurrent GC and monitor synchronization mechanism.

We implement and evaluate CoLoRS within runtimes for Python and Java. CoLoRS imposes low overhead when there is no use of shared memory (4% for Python and 5% Java) due to virtualization of runtime services and libraries.

An important use case for CoLoRS is improving the performance of RPC protocols in the co-located case. We have found that for microbenchmarks CoLoRS increases throughput by 8–102 times and reduces latency by 2–113 times. CoLoRS improves the performance for the cache-enabled Cassandra database and HDFS by 19–20 times for throughput and 3 times for latency.

In summary, CoLoRS enables type-safe, object sharing across OO languages in a transparent and efficient way. As part of future work, we are extending name support to other OO languages (C++ and Ruby) and are investigating its use within multi-language distributed cloud systems such as

AppScale [12] for which components migrate dynamically (co-location is intermittent).

References

[1] Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *ICPP*, 1999.

[3] J. Armstrong. Erlang – a survey of the language and its industrial applications. In *9th ESIAP*, 1996.

[4] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[5] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. Technical report, Univ. of Utah, 1998.

[6] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *OSDI*, 2000.

[7] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1), 1990.

[8] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, 1995.

[9] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *PLDI '08*, 2008.

[10] Cassandra. Cassandra, 2009. `http://incubator.apache.org/cassandra/`.

[11] X. Chen and V. H. Allan. MultiJav: A distributed shared memory system based on multiple Java virtual machines. In *PDPTA*, 1998.

[12] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *International Conference on Cloud Computing*, Oct. 2009.

[13] CORBA. CORBA Specification. `http://www.omg.org`.

[14] I. Corporation. *Occam Programming Manual*. 1984.

[15] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel programming in Split-C. In *SC*, 1993.

[16] G. Czajkowski and L. Daynes. Multitasking without compromise: A virtual machine evolution. In *OOPSLA*, 2001.

[17] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL '94*, 1994.

[18] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *POPL '93*, 1993.

[19] T. Domani, E. K. Kolodner, and E. Petrank. A generational on-the-fly garbage collector for java. *SIGPLAN Not.*, 35(5), 2000.

[20] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanorer. Implementing an on-the-fly garbage collector for java. *SIGPLAN Not.*, 36(1), 2001.

[21] S. Dorward, R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. Inferno. In *COMPCON*, 1997.

[22] T. El-Ghazawi, W. Carlson, and J. Draper. UPC Language Specifications V, Feb. 2001. `http://upc.gwu.edu`.

[23] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*, 2006.

[24] R. T. Fielding. Architectural styles and the design of network-based software architectures. Technical report, Univ. of California, Irvine; PhD Dissertation, 2000. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[25] M. Golm, M. Felser, C. Wawersich, and J. Kleinoder. The JX operating system. In *USENIX Annual Technical Conference*, 2002.

[26] hdfs. Hadoop File System (HDFS). `http://hadoop.apache.org/common/docs/current/hdfs_user_guide.html`.

[27] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.

[28] javaos. *JavaOS : A Standalone Java Environment*. Sun Microsystems, 1996.

[29] JNode. JNode. `http://www.jnode.org`.

[30] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau. JESSICA: Java-enabled single-system-image computing architecture. *J. Parallel Distrib. Comput.*, 60(10), 2000.

[31] Protocol Buffers. Protocol Buffers. Google's Data Interchange Format. `http://code.google.com/p/protobuf`.

[32] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *SIGPLAN Not.*, 41(10), 2006.

[33] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. *Lecture Notes in CS*, 2001.

[34] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Apr. 2007. `http://incubator.apache.org/thrift/`.

[35] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, D. Hu, and D. Spoonhower. J-Kernel: A capability-based operating system for Java. In *Secure Internet Programming*, 1999.

[36] M. Wegiel and C. Krintz. XMem: Type-Safe, Transparent, Shared Memory for Cross-Runtime Communication and Coordination. In *PLDI*, 2008.

[37] N. Wirth and J. Gutknecht. *Project Oberon: the design of an operating system and compiler*. ACM Press/Addison-Wesley, 1992.