XMem: Type-Safe, Transparent, Shared Memory for Cross-Runtime Communication and Coordination

Michal Wegiel Chandra Krintz

Computer Science Department
Univ. of California, Santa Barbara
{mwegiel,ckrintz}@cs.ucsb.edu

Abstract

Developers commonly build contemporary enterprise applications using type-safe, component-based platforms, such as J2EE, and architect them to comprise multiple tiers, such as a web container, application server, and database engine. Administrators increasingly execute each tier in its own managed runtime environment (MRE) to improve reliability and to manage system complexity through the fault containment and modularity offered by isolated MRE instances. Such isolation, however, necessitates expensive cross-tier communication based on protocols such as object serialization and remote procedure calls. Administrators commonly co-locate communicating MREs on a single host to reduce communication overhead and to better exploit increasing numbers of available processing cores. However, state-of-the-art MREs offer no support for more efficient communication between co-located MREs, while fast inter-process communication mechanisms, such as shared memory, are widely available as a standard operating system service on most modern platforms.

To address this growing need, we present the design and implementation of XMem – type-safe, transparent, shared memory support for co-located MREs. XMem guarantees type-safety through coordinated, parallel, multi-process class loading and garbage collection. To avoid introducing any level of indirection, XMem manipulates virtual memory mapping. In addition, object sharing in XMem is fully transparent: shared objects are identical to local objects in terms of field access, synchronization, garbage collection, and method invocation, with the only difference being that shared-to-private pointers are disallowed. XMem facilitates easy integration and use by existing communication technologies and software systems, such as RMI, JNDI, JDBC, serialization/XML, and network sockets.

We have implemented XMem in the open-source, production-quality HotSpot Java Virtual Machine. Our experimental evaluation, based on core communication technologies underlying J2EE, as well as using open-source server applications, indicates that XMem significantly improves throughput and response time by avoiding the overheads imposed by object serialization and network communication.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features – Dynamic Storage Management, Classes and Objects; D.3.4 [*Programming Languages*]: Processors – Run-Time Environments, Memory Management (Garbage Collection), Compilers, Optimization

General Terms Design, Experimentation, Languages, Management, Measurement, Performance

Keywords Interprocess Communication, Managed Runtimes, Shared Memory, Transparent, Type-Safe, Garbage Collection, Synchronization, Class Loading, Parallel

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.

1. Introduction

Developers today predominately construct modern, enterprise, component-based, middleware for portable, distributed applications using type-safe, object-oriented languages, such as Java, which users execute within managed runtime environments (MREs). These MREs typically support garbage collection (GC), dynamic class loading, incremental compilation, as well as high-level threading and synchronization primitives, among other runtime services. One popular example from this application domain is JBoss, an application server that provides a complete implementation of the J2EE [32] specification, and that is architected on top of the Java platform [34].

A common architectural design pattern employed by administrators of enterprise applications is multi-tier deployment that partitions the system into independent domains, typically run using separate MRE instances (OS processes). Such isolation enables fault containment and modularity as well as more aggressive specializations in the MREs (e.g., using the best-performing compilation strategy or garbage collection system for a particular application, set of activities, or domain). J2EE-based applications typically comprise at least three tiers: a web container (front-end presentation layer), an application server (business logic), and a database engine (back-end data source) [34, 8, 60].

Multi-tier decomposition, however, necessitates expensive interprocess communication (IPC) between MREs (isolated components). Since most general-purpose servers (e.g., web, application, database) are designed for online transaction processing, in which many clients perform many short transactions simultaneously, communication overhead can constitute a significant portion of the observed, end-to-end response time (especially when multiple isolation units are involved).

To reduce the overhead of MRE IPC, administrators commonly co-locate multiple tiers on a single machine. Co-location simplifies administration and configuration, enables efficient use of local network communication for IPC, and makes better use of multiprocessor architectures through increased thread-level parallelism. Emerging multi- and many-core systems are likely to make MRE co-location increasingly commonplace.

Cross-MRE IPC mechanisms cannot depend on co-location, however, since MREs may alternatively be distributed across different cluster nodes or be migrated to achieve load balancing and more

effective utilization of server resources, an increasingly important operation in virtualizing systems today [44, 47, 57]. Thus, MRE IPC employs high-overhead implementations of standard communication protocols, such as remote procedure calls and object serialization, regardless of the proximity of the communicating MREs.

We introduce support for transparent and type-safe, cross-MRE communication and coordination, called XMem. XMem is an IPC mechanism that enables object sharing between MREs co-located on the same machine and communication via extant distributed protocols when physically separated. XMem is transparent in that shared objects are the same as unshared objects (in terms of field access, synchronization, GC, and method invocation, among others), except that XMem disallows pointers from shared objects into MRE-private storage. To enable efficient object sharing, XMem manipulates virtual memory mapping to avoid indirection, i.e., all object references in the system are direct. Moreover, existing communication technologies, e.g., those employed by J2EE or network sockets, can use XMem without application modification.

XMem guarantees type-safety by ensuring that the MREs employ the same types for shared objects when the communication medium is shared memory. XMem is also compatible with core MRE services such as GC, dynamic class loading, and thread synchronization. XMem coordinates MREs through infrequent, synchronized global operations that include GC and class loading.
In summary, we make the following contributions with this work:

- Improved integration of MREs with the underlying system. XMem provides a new, efficient communication mechanism while maintaining standard, portable interaction with the lower-level layers of the software/hardware stack.
- Direct object sharing via isolated channels between co-located MREs isolated as distinct OS processes that avoids the trade-offs inherent to previous approaches [17, 4] by enabling communication without serialization and data copying.
- Extensions to the MRE services and abstractions, including parallel, cross-process class loading and garbage collection. XMem implements changes to MRE subsystems and system libraries to enable transparent and efficient use of shared memory support when available.
- Empirical evaluation of XMem that quantifies the reduction in response time and the increase in throughput in the context of the most commonly-used J2EE communication technologies, such as JDBC, JNDI, and RMI, as well as a database server and a web server.

In the sections that follow, we describe the necessary support of object sharing (Section 2), of multi-threading (Section 3) and management of the shared memory segment (Section 4). In Section 5, we present the implementation details of our prototype as well as our experimental methodology and empirical evaluation of XMem. Finally, we contrast related work (Section 6) and present our conclusions and our future directions in Section 7.

2. XMem: Transparent Object Sharing

The goal of XMem is to improve communication performance for enterprise-class, object-oriented, software systems, a popular application domain for web services. XMem enables transparent IPC via shared-memory between isolated MREs that are *co-located* on the same machine; such co-location of related processes is an increasingly common technique for the exploitation and better utilization of multicore systems. Using XMem, MREs share objects directly to avoid the overhead that is imposed by distributed communication protocols due to object marshalling and serialization.

To enable direct object sharing, XMem maps the shared memory segment at the same location in the virtual address space (VAS) of all attached MREs. Figure 1 depicts an example instance of an XMem system. Two MREs attach to the same shared memory segment (gray area of the VAS) to share objects. We refer to the VAS of each MRE that is not mapped to the shared memory segment (white area of the VAS) as MRE-private. XMem systems share per-instance, non-static data only – static (per-class) data is MRE-private since static fields typically record program-specific or MRE-specific state. Sharing of such fields can violate both type safety and inter-process resource isolation.

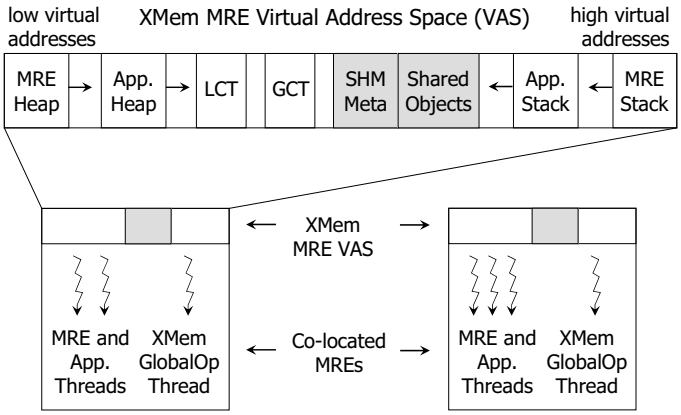


Figure 1. Co-located XMem MREs, and their virtual address spaces (VAS), that are attached to a shared memory segment (gray area). The shared region contains meta-data (SHM-Meta) and shared objects and is mapped at the same virtual address in each MRE. The GlobalOp thread in each MRE performs infrequent global operations that XMem synchronizes across attached MREs.

Since we map shared memory to the same virtual address in all MREs, objects within the shared memory have the same addresses in all MREs. To guarantee memory and type safety, we disallow pointers from shared objects to private objects via a write barrier (described further below), since the address space of the non-shared areas in MREs is independent and unrelated across MREs. Regardless of this constraint however, XMem MREs provide services, such as class loading, GC, allocation, synchronization, compilation, uniformly for shared and MRE-private objects, i.e., XMem provides object-level transparency.

Key to enabling such transparency efficiently is that (i) the internal representations of object types (classes) are the same across all attached MREs, and that (ii) the underlying operating system provides support for virtual memory paging and its manipulation by user-level processes (the MRE in our case).

2.1 VAS Manipulation

XMem manipulates the virtual address space to enable direct access to objects as well as to their class representations. Objects in most object-oriented language systems typically contain a reference to an internal representation of the class (type) from which they are instantiated. This reference enables direct retrieval of object metadata for fast implementation of common operations such as dynamic dispatch, field access, and dynamic compilation. These internal representations of classes, however, are MRE-specific and cannot be shared, as they commonly record application- or MRE-specific state and provide access to static (private) data. Class pointers, therefore, must resolve to the MRE-private internal representation of the class.

To avoid moving (reordering) existing class objects (internal representations) within each attached MRE (which can be complex and expensive), yet to ensure that the same virtual address refers to the same MRE-private internal representation of the class in all

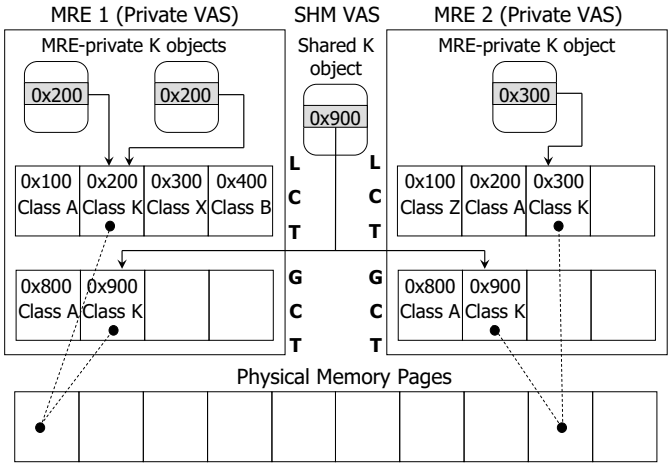


Figure 2. Manipulation of VAS mapping so that class pointers resolve to equivalent MRE-private class representations across attached MREs without copying or moving, and while enabling direct retrieval of object metadata (for dynamic dispatch, field access, etc.). Each box is a virtual page (4KB in size), potentially mapped to physical memory. Blank boxes are unmapped. We omit mapping lines (dotted with round ends) for classes other than K, for clarity.

MREs, XMem aligns class objects to virtual memory page boundaries (we assume traditional 4KB pages) and manipulates virtual address mapping as depicted in Figure 2 via double mapping. In the virtual address space (VAS) of each attached MRE in XMem, there is a global class table (GCT) and a local class table (LCT), both of which are MRE-private. The LCT holds the representations of both MRE-private and global classes. LCTs across MREs are independent and unrelated. In contrast, the GCT in each MRE is identical in structure and layout (class order, count) and has the same virtual address in MRE-private space.

XMem maps the physical page of a particular (global) class to a virtual page in both LCT and GCT, to achieve resolution of class pointers within shared objects to private class representations without copying or moving and without introducing pointer indirection. In the example, the class pointer of unshared objects (instances of class K) refers to the internal class representation in the LCT in their MRE (address 0x200 in MRE 1 and 0x300 in MRE 2). When the two MREs share an object of type K, XMem adds an entry for class K to the GCT at the same location in each MRE. Since the GCTs are identical in each MRE and start at the same virtual address, the class pointer in the shared object is the same for both MREs (0x900). We overview the class loading process that makes use of this implementation in Section 4.2.

There are two side-effects of this double-mapping. First, in the worst case, XMem consumes twice the VAS needed for classes (worst case is when each MRE-private class is a globally shared class). This case is uncommon in our experience as the number of MRE-private classes typically far exceeds that of globally shared classes. Moreover, such VAS use is negligible for machines with large address spaces (64-bit platforms). Second, class alignment to virtual page boundaries limits the class size to that of a virtual page and can cause fragmentation in the LCT (when classes are smaller than the page size). In practice, we have never found a class object to be larger than our virtual page size. However, if this proves to be a limitation, we can reserve a multiple of the page size for each class. In our implementation, the LCT is the permanent generation of the MRE which stores other long-lived data (e.g., MRE data structures, static strings) in addition to class objects. This data consumes part of each page which helps to reduce fragmentation. We measure and report the space overhead of fragmentation in Java benchmarks in Section 5.2. We plan to investigate the impact of large page sizes on XMem systems as part of future work.

2.2 Shared-to-Private Pointers

To guarantee that shared objects never refer to private heaps (since such references are particular to a specific MRE process), XMem piggy-backs on the extant write barrier implementation of generational garbage collection (GC). Generational GCs are in widespread use in modern MREs as they provide superior performance which they achieve by exploiting similarity in object lifetimes and by partitioning the heap into distinct, contiguous spaces called generations [58, 62, 37]. These systems allocate most objects from the young generation, and collect this region frequently since a majority of objects die young [6, 36]. To enable efficient, independent collection of generations, generational GCs use a write barrier at every reference store in a program to track references from older to younger generations. Modern MREs typically also employ a permanent generation that is rarely collected and that holds long-lived objects such as internal class representations, constant strings, and MRE data structures.

XMem extends write barriers with two checks needed to compare the source and destination of a particular pointer against the constant boundary between MRE-private and shared part of the heap. We need the source check for each pointer store, and the destination check only for stores to the shared memory. If a program makes an assignment that violates the XMem constraint, the runtime throws an exception and the instruction fails. Since we map the shared memory segment to the same location in each MRE and the segment has a fixed size, this check is very efficient: it consists of a register and constant comparison. Such checks impose negligible overhead on modern architectures because there is no memory access and the branch direction is typically highly biased and thus, easily predictable.

2.3 Using XMem

Developers make use of XMem via a simple application programming interface (API). The XMem API for Java comprises the following public static methods declared in the `ipc.SharedMemory` class:

```
void sharedModeOn();      boolean isSharedModeOn();
void sharedModeOff();     boolean isShared(Object o);
Object accept(int p);     void connect(int p, Object o);
void bind(int p);         Object copyToShared(Object o);
```

To support transparency and backward-compatibility, programs within XMem allocate objects using the conventional `new` operator, regardless of whether they are allocating shared or private memory. XMem determines from which region (shared or private) to allocate using a per-thread allocation mode. Initially, the allocation mode is private. Programs or libraries change the allocation mode explicitly via the `sharedModeOn` and `sharedModeOff` methods. The system throws an `ipc.SharedMemoryException` to prevent shared-to-private pointers as well as signal binding/connection failures and out-of-memory errors.

XMem makes use of the concept of *ports* to enable co-existence of multiple, isolated communication channels in a single shared memory segment. To initiate communication, two distinct MREs (to which we refer as a client and a server) must obtain a reference to a shared object (to which we refer as a root). A client allocates a root in shared memory and passes a reference to it to the `connect` method along with a port to which a specific server has been bound via the `bind` method. The server retrieves the root from the `accept` method. Once the root is exchanged, further communication proceeds according to an application-specific protocol which commonly includes monitor synchronization (wait/notify) on the root. Objects shared through a particular channel are reachable only

to threads/MREs that have established the connection. However, an arbitrary number of threads/MREs can share a specific object if a server makes a reference to a shared object available to multiple clients (which use distinct channels for communication with the server).

To enable interoperability with libraries that do not guarantee immutability of the objects they take as arguments, XMem provides a mechanism for recursive (deep) copying of objects into the shared memory via the `copy` method. Object cloning, commonly available from the underlying language (e.g., Java or C#) platform, by default creates shallow object copies and must be overridden on a per-application basis to support deep coping. XMem provides this general service uniformly across applications. XMem uses stack-based, depth-first copying and handles cycles in the object graph by maintaining a hash table that maps the already-visited objects to their copies. We describe how such copying to shared memory interacts with shared-memory garbage collection in Section 4.3.

Although, in this work, we focus primarily on shared memory, other IPC mechanisms such as signals and message queues can be built on top of XMem in a straightforward way. We have integrated XMem, through the use of its API, into existing communication mechanisms, such as RMI, applying only minimal library modifications. Such XMem-aware implementations provide two paths of execution that the library routine selects based on the proximity of the communication target: one that employs shared memory and one that uses traditional distributed communication.

3. XMem Runtime Support

To enable cross-MRE object sharing, XMem extends the MRE multi-threading implementation by adding dual mode (shared or private) object allocation and support for cross-process synchronization based on shared object monitors. XMem automatically preserves the guarantees provided by the memory consistency model of a specific MRE (e.g., the Java Memory Model [43]) since the system consists of homogeneous MREs.

3.1 Dual Mode Object Allocation

XMem extends the common allocation technique of thread-local allocation buffers (TLABs). TLABs are used by modern MREs to reduce contention between threads that concurrently perform linear (bump-pointer) allocation from a common area. This approach requires no synchronization when allocating within a TLAB. The system allocates TLABs to threads linearly, using more expensive atomic operations. XMem associates two TLABs with each application thread, one in private and one in shared memory. We do not initialize the latter until the thread performs its first allocation into shared memory, e.g., when it first executes a `new` bytecode within the XMem shared mode (`sharedModeOn()`). XMem excludes objects that the system creates by side-effect of other operations, such as class loading or lazy data structure initialization, from allocation in shared memory to prevent unintended object leaks. XMem also uses private mode for allocation of all internal data structures (data commonly stored in a permanent area of the heap).

3.2 Thread Synchronization

Two locking schemes are commonly used to implement language-level (e.g., Java) monitors in extant MREs: lightweight locking [52] and biased locking [52]. Biased locking optimistically assumes that a single thread uses a monitor (i.e., there is no contention); when this proves not to be the case, biased locking falls back to lightweight locking. Both lightweight and biased locking require a re-design to work with shared memory. XMem adapts and employs lightweight locking since it is the basis for both schemes. We first overview lightweight locking and then describe its implementation in XMem.

Lightweight Locking. To avoid using OS primitives (a pair consisting of a mutex and a condition variable) in the common case of uncontended locking, lightweight locking employs atomic compare-and-swap (CAS) operations. Only when two threads attempt to lock the same object does the MRE inflate the lightweight lock into a heavyweight OS-backed monitor. Lightweight locking improves performance as user-mode locking is significantly more efficient than system calls.

The MRE stores basic locking information in the object header which occupies one machine word. The lowest two bits encode one of the three possible states: unlocked (UL), lightweight-locked (LL), and heavyweight-locked (HL). When an object is LL (by a `monitorenter` bytecode), the system inserts a lock record into the stack of the thread performing the lock acquisition operation. During stack unwinding (which takes place when an exception is thrown), the system uses lock records to unlock the objects that are locked in the discarded stack frames. Normally, objects are unlocked by a `monitorexit` bytecode generated as part of the epilogue of block-structured critical sections. Each lock record holds a pointer to the locked object and the original value of the overwritten object header.

During locking, a thread attempts an atomic CAS on the object header to replace it with a pointer to the stack-allocated lock record. Lock records are word-aligned, therefore the two lowest bits are always cleared and do not conflict with the locking state bits kept in the header. If the CAS succeeds, the thread owns the monitor. Otherwise, a slow path is taken and the lock is inflated – the object header is CAS-updated to point to a data structure containing a mutex and a condition variable. This data structure is stored in private, MRE-managed, memory.

During the unlock operation of an LL object, a thread tries to CAS-restore the header that it has stored on the stack. On success, no fall-back is needed and the fast path is complete. The CAS failure indicates that the lock was contended for (and inflated) while it was held. Under such circumstances, it is necessary to notify the competing (and now waiting) threads that the object is unlocked. These threads are blocked on the condition variable. When awakened, they unlock the mutex and resume execution by trying to re-acquire the mutex. The mutex and the condition variable are multiplexed here: first they are used to wait until the LL object becomes unlocked and then they are used in a standard way to provide mutual exclusion along with the wait/notify functionality.

Recursive locking in the lightweight case is based on implicit lock ownership – if the object header points into the stack of the current thread then the current thread already owns the lock and in a new lock record on the stack the header field is set to `NULL`. When unlocking, a lock record with the `NULL` header field is ignored. Recursive locking in the heavyweight case uses a counter located in the aforementioned MRE data structure.

Lightweight Locking in XMem. The challenges to lightweight locking in XMem shared memory are three-fold. First, the header of an LL object points into a private thread stack. Such references cannot be interpreted properly across MREs directly. Second, heavyweight data structures allocated in MRE-private memory must now be accessible to other MREs. Finally, POSIX synchronization primitives by default work within a single process.

To address these issues, XMem allocates a lock data structure (LDS) in shared memory, both in case of lightweight and heavyweight locking and uses POSIX object attributes to enable cross-process synchronization. LDS reserves space for a mutex and a conditional variable (which are initialized only in case of inflation). LDS contains a process identifier (PID) and a thread identifier (TID), that together unambiguously identify the owner, as well as a recursion count, the locked object reference, the binary flag used for mutex/condition variable multiplexing, and the original object

header. Lock records that are stored on the stack contain only the address of the locked object. An object header, instead of pointing into a stack, always refers to the corresponding LDS, when locked.

XMem maintains an LDS pool in SHM-Meta (metadata area in shared memory). Application threads atomically bulk-allocate multiple LDSes at once from the global pool to reduce synchronization overhead. Each thread holds several LDSes in a local queue with a FIFO discipline. An LDS of an LL object is returned to the thread-local queue when unlocking succeeds (i.e., no contention is detected). An inflated LDS can be freed only during shared memory GC when the HL shared object becomes unreachable.

Invoking wait/notify on an LL object results in the lock inflation. This is necessary as these operations require support from the OS. An important aspect of LL is the hash value computation. MREs typically store the hash value in the object header and lazily initialize it. The hash code, once computed, should never change. Since LL displaces an object header, a race condition arises when an LL object is simultaneously unlocked and its hash code is being initialized. Such circumstances force lock inflation and safe initialization of the hash code (inflated locks are more stable as their unlocking does not change the object header).

XMem uses this modified LL scheme only in the shared memory – each MRE uses the original scheme internally as it is more space efficient. Each lock/unlock operation checks whether the corresponding object is shared or not and dynamically applies the appropriate locking scheme.

4. Shared Memory Segment Management

Each XMem MRE executes a Global Operations (GlobalOp) thread that performs five coordinated *global operations*: attachment, detachment, connection, class loading, and garbage collection (GC). XMem serializes these, relatively rare, operations using a global lock (a mutex and a condition variable located in the shared memory). The system performs every global operation in parallel by all currently attached MREs using the GlobalOp thread in each MRE. Since MRE attachment and detachment are global operations, there is a well-defined set of attached MREs with respect to the current global operation. This is an important property, as global operations terminate only when all attached MREs report operation completion. With the exception of GC, global operations execute concurrently with application threads (i.e., without stopping them).

4.1 Attachment, Detachment, and Connection

Two JVM properties, `ipc.shm.file` and `ipc.shm.destroy`, control MRE-OS interaction. The first one identifies a shared memory segment to create or attach to (we employ Linux System V IPC [49]). The second one specifies if an MRE should mark the segment for destruction upon termination. The OS releases only marked segments whose attachment count reaches zero. Upon startup, each MRE attempts to create a new shared memory segment. The creation process fails if the segment already exists, which causes a fall-back to attachment. The MRE that succeeds in segment creation, initializes the shared data structures (located in SHM-Meta).

MREs that attach/detach to/from an existing segment perform a global attach/detach operation. Attachment takes place after completing the MRE bootstrap procedure and before invoking the program's `main` method. Detachment is performed upon program termination. These two global operations are automatic and not accessible via the XMem API. An MRE can attach only to one segment at a time. However, XMem supports multiple communication channels over a single shared memory segment. A configuration with a single segment per host is most memory-efficient but multiple segments can be used if needed. Attach and detach operations update a global counter that tracks the number of attached MREs.

The connection operation establishes a communication channel. Connection allows two MREs to obtain a reference to a shared object while guaranteeing privacy (other MREs cannot reach that shared object). It implements semantics similar to that of a network socket. The arguments passed to the `connect` (i.e., a port number and a shared object), are propagated to other MREs as parameters of the global operation. Each MRE maintains a list of ports to which it is bound. When a connection request to a locally bound port is detected, an MRE adds the corresponding shared object to a local queue and awakens the threads that are blocked on the `accept` call on the port. The shared object is then dequeued and returned by the `accept` method. XMem ensures that only one MRE is bound to any port (an atomically-updated boolean table is kept for this purpose in the shared memory). Since connection is a global operation, it is serialized with respect to GC, and as a result, the shared object (root) has a stable location while the operation is in progress.

4.2 Global Class Loading

Through global class loading, XMem ensures that a specific class is privately loaded by, and is the same in, all attached MREs, to guarantee type safety. XMem implements the latter by comparing the 160-bit SHA-1 hash value computed for the class bytecode, across MREs. If an MRE encounters a bytecode mismatch, global class loading fails and an exception is thrown.

Since XMem places no restrictions on MRE-private class loading, the class of a shared object may or may not be loaded in all attached MREs when it is instantiated in shared memory. Therefore, following each object allocation, XMem executes a class loading barrier which checks if the new object resides in the shared memory. If the object is shared, the MRE checks whether its class has been loaded globally. To make this check fast (note that it is done for each allocation), XMem adds a field (a forwarding pointer) to each private class object. The forwarding pointer is initially set to `NULL` to indicate that the class is loaded only privately. After global class loading, the forwarding pointer is set to the GCT address of the class. Following each allocation in shared memory, the MRE updates the class pointer of the new shared object to the forwarding pointer. If the check fails, i.e., the class of the new shared object has not been loaded globally, the MRE initiates global class loading.

Global class loading uses the default system class loader (which corresponds to the `CLASSPATH` variable). XMem permits classes defined by user-defined class loaders to be instantiated in the shared memory as long as the corresponding user-defined class loaders are themselves allocated in the shared memory. However, even though class loaders can be shared, the internal class representations are always MRE-private. XMem relies on MRE-private class loader constraints to guarantee type safety in the presence of lazy class loading, user-defined class loaders, and delegation [39]. No extension is needed because we first locally load all globally loaded classes and thus, local constraints are always a superset of global constraints.

4.3 Global Garbage Collection

Global GC in XMem identifies and reclaims dead, shared objects (i.e., those that are not reachable from any attached MRE). The GC is initiated by one of the attached MREs when allocation of a new TLAB in shared memory fails. In order to interoperate with different GC algorithms and heap layouts [63, 37], XMem provides a generic mechanism for identifying root objects in the shared memory. Root objects in this context are objects directly reachable from one or more MREs by following pointers that are located on thread stacks, in registers, or in the live part of a private heap. Once a snapshot of the root objects is obtained, shared memory can be collected in a conventional way using any tracing collector.

The key challenge is in identifying the root objects without resorting to scanning all the live objects in each MRE. Note that pointers into shared memory can be scattered across all genera-

tions. At the same time, we can expect the number of such pointers to be relatively small.

XMem identifies roots by piggy-backing on a fast minor collection (the one confined to the young generation). To enable this, XMem extends a card table mechanism [64] that supports generational GC so that it tracks pointers from older generations that point into the young generation or into shared memory. As a result, a young generation collection is able to detect all root objects that originate from a given MRE without an exhaustive scan of older generations. For each global GC, XMem triggers a minor collection in the attached MREs. To perform a minor GC, state-of-the-art MREs typically employ a parallel copying collector [23] that is executed in a stop-the-world (STW) fashion as it imposes very short pause times (i.e., concurrent collection [19, 48] is not necessary).

An XMem system implements global GC of the shared memory segment using STW parallel copying collection. All attached MREs perform GC in parallel, each contributing multiple GC threads. MREs synchronize only before and after collection. A full barrier is needed after all MREs reach a safepoint (i.e., state where application threads are suspended) because one cannot start moving the shared objects while other MREs are actively using them. For similar reasons, all GC threads from all MREs synchronize when leaving a safepoint. Any additional coordination depends on the GC algorithm used. Although, global GC stops all MREs, it is not unscalable or deadlock-prone since bringing an MRE to a safepoint is a low-delay operation robust with regard to I/O.

Since global GC can interrupt an XMem deep copy from private to shared memory, we must be careful to avoid introducing temporary shared-to-private pointers during the copy process. To this end, when we copy an object to its new location, we clear its reference fields (as they may still point to private objects). We update these fields with the correct values (new locations) when we copy the corresponding objects to shared memory. Global GC needs to update the entries in the stacks and hash tables used by XMem copy operation because it is moving objects. We provide a new object header to each shared memory replica to preclude them from inheriting the synchronization state of original objects.

Non-global GCs (both minor and major) do not follow pointers that point into the shared memory. Because of the XMem invariant that no shared-to-private pointers are allowed, it is correct to stop tracing when a shared object is encountered. GC completeness is preserved because scanning of objects in the shared memory cannot lead to the discovery of any additional live objects in the private heap. Local GC performance, is thus the same regardless of the number of objects in shared memory.

The most suitable GC algorithm for shared memory collection depends on the demographics and total size of the live shared objects [36]. If XMem is used to share a large amount of long-lived data, then compacting collectors are most appropriate. On the other hand, if the primary purpose of XMem is communication between strongly isolated MREs, then copying collection is a better choice [62]. This is because the communicating MREs exchange a small number of objects which exhibit relatively short lifetimes. Generational collection can be used to support a wide range of object lifetimes. To accommodate short-lived communication behavior typical of J2EE applications, we implement a non-generational, parallel copying in our XMem prototype.

Parallel copying collectors employ several GC threads to evacuate live objects from the currently-used source space(s) to the currently-unused target space [28]. Since most objects are expected to be unreachable, the target space is typically smaller than the source space(s) and the worst-case scenario is handled by falling back to the promotion of overflow objects into older generation(s). In the absence of a generational heap layout, half of the space needs to be set aside as a copy reserve.

XMem employs two equal-sized semi-spaces in the shared memory and the collection of the source semi-space is performed in parallel by all attached MREs. This process is interleaved with local minor GCs so that the object graph is traversed only once. Each MRE uses multiple GC threads, which correspond to schedulable kernel threads and whose total number equals the number of processors/cores available or dedicated to each MRE.

XMem employs a two-level load balancing scheme in the form of work stealing [23]. GC threads that become idle attempt to steal object references from non-empty marking stacks of other GC threads. Each GC thread is associated with two marking stacks, which we refer to as the local and shared stack. Intra-MRE load balancing is limited to local stacks while inter-MRE work stealing uses shared stacks only. MREs push references to objects residing in the shared memory onto the shared stacks to make them available to other MREs. Local load balancing is preferred and global stealing is done only when all local stacks become empty. The stealing target (i.e., the marking stack/stack entry) is selected randomly.

Global GC is an STW operation that comprises three barriers: prologue, epilogue, and GC termination. The GC prologue flips the semi-spaces. The GC epilogue forwards the pointers in SHM-Meta and deflates heavyweight monitors associated with dead objects.

To ensure that each live object is processed exactly once, GC threads claim objects atomically. Atomic CAS instructions are supported by most processors and can be used across processes (as they are based on physical rather than virtual addresses). To reduce contention, each GC thread owns a parallel local allocation buffer (PLAB) where it copies the objects it has claimed. We allocate PLABs linearly, atomically, and on-demand, from the target semi-space. The GC first copies an object to its destination, and then pushes the addresses of its reference-type fields onto the marking stack (local and/or shared). Then, a GC thread tries to CAS-forward the original object header to the new location. If a thread loses a race to another thread, the GC removes the object from the PLAB and pops the new pointers off the stack. This order of operations is motivated by fault-tolerance (Section 4.5).

4.4 Global Meta-Data Management

The SHM-Meta data structures support the runtime and global operations of XMem. They include a descriptor for the current global operation, which contains the operation code, its input arguments, barrier counters, state flags, and a mutex and condition variable with which the system serializes the execution of global operations. In addition, SHM-Meta holds the marking stacks for global GC and a table that records the meta-information for all globally loaded classes including the class name, defining class loader (set to `NULL` if the default system class loader is used), and a bytecode hash for type-safety verification. SHM-Meta also holds a list of the bound ports that are currently in use for communication sessions between co-located MREs. Finally, SHM-Meta contains single-word entries for (i) the number of attached MREs, (ii) the number of globally loaded classes, (iii) the boundaries of and current position in the shared heap (for allocation of new TLABs), and (iv) the start and end of a pool of global locks that enable cross-MRE monitor synchronization.

4.5 Fault Tolerance

XMem tolerates unexpected MRE termination, between and during global operations, by implementing a timeout mechanism (based on the `pthread` timed wait on a condition). If an MRE fails, the next global operation times out. Upon timeout, XMem subtracts the number of not-responding MREs from the counter of the attached MREs and releases any shared locks that were held by the terminated MRE. Connection, and class loading are global operations that do not require any additional handling upon timeout.

Bench-mark	Generation Size [MB] Young+Old	Permanent	Execution Time [s]	Number of GCs Minor	Major	Number of Classes	XMem Overhead Time [%]	Space [MB]
bloat	40	5	55.3 ± 0.5	528 ± 2	1 ± 0	827	3.5	2.24
pmd	34	6	19.5 ± 0.1	495 ± 1	8 ± 1	1186	3.5	2.94
xalan	42	6	49.2 ± 0.3	1480 ± 8	107 ± 4	1179	3.4	3.11
antlr	8	4	4.5 ± 0.1	380 ± 1	5 ± 0	679	2.8	1.79
chart	30	9	15.7 ± 0.1	355 ± 7	9 ± 0	1440	1.9	3.74
eclipse	68	16	66.5 ± 0.2	551 ± 3	11 ± 0	2627	2.3	7.04
hsqldb	336	5	13.6 ± 0.1	9 ± 0	5 ± 0	736	0.3	2.00
fop	20	6	2.0 ± 0.0	19 ± 0	0 ± 0	1423	2.8	3.45
luindex	8	4	7.2 ± 0.1	260 ± 8	3 ± 0	689	1.6	1.83
lusearch	18	4	8.6 ± 0.1	706 ± 1	1 ± 0	683	2.6	1.79
jython	8	8	43.1 ± 0.3	3539 ± 2	1 ± 0	1325	3.0	3.14
jbb/6wh	476	8	90 ± 0.0	149 ± 0	4 ± 0	1296	0.64	3.59
jbb/8wh	636	8	90 ± 0.0	116 ± 1	3 ± 0	1296	1.78	3.59
jbb/10wh	780	8	90 ± 0.0	98 ± 0	3 ± 0	1296	0.82	3.59

Table 1. The overhead introduced by XMem in terms of application throughput (Jbb) or execution time (Dacapo) and occupancy of the permanent generation. For each benchmark we report generation sizes, execution time (note that Jbb runs for a fixed period of time), the number of minor/major collections, and the number of loaded classes.

In case of timed-out detachment and attachment operations, the system needs to determine whether it was the detaching/attaching MRE that failed (to correctly keep track of the number of live MREs). This is done based on the PID of the process which initiated attachment/detachment (XMem sends a signal using the `kill` system call and gets an error if the process is dead).

GC requires more complex handling, as the shared stacks of a terminated MRE can contain pointers stolen from other MREs. These stacks are located in shared memory so they are not lost and can still be processed. During GC, objects are forwarded to their new locations only when they have been copied and when their content has been scanned (and pushed onto a stack). Therefore, copying collection can be interrupted at any time without losing correctness, provided that whatever is on the stack(s) is eventually processed. If a global GC times out, it is sufficient to empty all the marking stacks located in the shared memory.

5. Implementation and Evaluation

We have implemented XMem in HotSpot [46], an open-source, high-performance JVM written in C/C++. The heap in HotSpot [28] comprises three generations: young (where new object allocations take place), old (where long-lived objects are promoted), and permanent (where classes are stored). HotSpot reserves two words per object. The first word (the header) contains the locking state, age bits, and the hash code. The second word is a pointer to a class object located in the permanent generation. Class objects encapsulate static fields, a virtual method table, a class loader reference, and pointers to other meta-objects that describe methods and fields (among others).

The `PTHREAD_PROCESS_SHARED` attribute is set on the POSIX mutexes and condition variables to enable cross-process synchronization. To create or look up a shared memory segment, XMem employs `shmget`. This system call is used with the `IPC_PRIVATE` key to implement double mapping in LCT and GCT. Global shared memory segments are identified by a key generated by `ftok` based on a file name. XMem supports multiple global segments on a single host, differentiated by a file name (the JVM `ipc.shm.file` property). We implement attachment with `shmat`, which allows to specify a virtual address that a segment is mapped to. MRE-private memory is allocated using `mmap`, which is called with the `MAP_FIXED` flag when pinning GCT at a specific location. For atomic operations we use the x86 `cmpxchg` instruction. LCT corresponds to the permanent generation.

5.1 Methodology

Our experimental platform is a dedicated machine with a dual-core Intel Core 2 Duo (Conroe B2) processor clocked at 2.66GHz, equipped with 4M 16-way L3 cache, 32K 8-way L1 cache, 2GB main memory, and running Debian GNU/Linux 3.0 with the 2.6.17 kernel. We use the HotSpot OpenJDK [46] v7-ea-b18 (Aug. 2007) compiled with GCC 3.2.3 in the optimized client-compiler (C1) mode. This version of HotSpot implements a highly-optimized, state-of-the-art serialization mechanism and uses standard (not process-shared) mutexes/condition variables.

For our experiments, we employ standard community benchmarks from the Dacapo [18] and SPECjbb2005 [55] suites to evaluate the impact of XMem on programs that do not communicate across MREs. We use the large input for Dacapo and 6, 8, and 10 warehouses, with 90s runs, for Jbb. To evaluate the impact of using shared memory, we develop a number of benchmarks ourselves (an approach taken in [41] in a similar context), which exercise shared memory and implement the J2EE communication protocols. We describe these benchmarks with each experiment. We evaluate XMem-aware implementations of RMI and CORBA, serialization and XML, JNDI, and TCP/IP sockets. Finally, we evaluate XMem for two server-side applications: Hsqldb [29] and Tomcat [1]. In all experiments, there are 2 MREs and the shared memory size is 30MB. Whenever running the original HotSpot JVM, we set the young generation to 30MB.

5.2 XMem Overhead

To investigate the overhead introduced by adding support for XMem, we compare the performance of shared-memory-oblivious applications run on top of unmodified HotSpot JVM against our implementation of XMem. Table 1 summarizes the results. For each benchmark, we employ a heap size (i.e., total size of the young and old generation) of twice the minimum (the same methodology that is used in [56]). We employ this methodology to ensure some GC activity without having GC dominate performance – so that we are able to measure other sources of overhead potentially introduced by XMem. We set the permanent generation size to the minimum required for XMem to load all the necessary classes. The young generation constitutes one fourth of the heap. We report generation sizes, the number of minor and major GCs, and the number of loaded classes. For timings, we execute 5 warm-up runs then compute the average and standard deviation of the next 5 runs.

XMem imposes negligible time overhead which we express as the percentage of total execution time (for DaCapo) or throughput

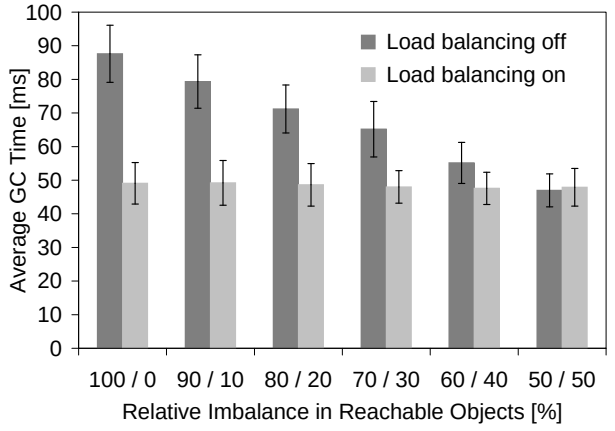


Figure 3. Global GC pause times with and without inter-MRE load balancing for different distributions (percentage) of shared objects reachable from individual MREs.

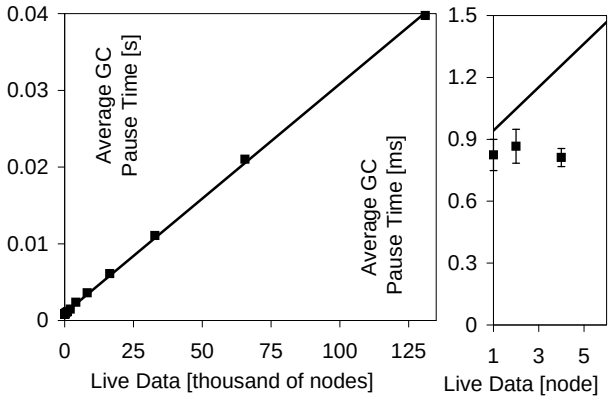


Figure 4. Impact of the size of live shared objects on global GC pause times. We present two views of the same graph to show both copying throughput and global safepoint latency.

(for SPECjbb2005). The sources of overhead are two additional checks per write barrier and internal checks for whether or not an object is shared. We report absolute values for the space overhead introduced in the permanent generation (by the page alignment implementation) as this overhead does not depend on generation sizes (only on the number of loaded classes). The space overhead ranges from 1MB to 7MB and on average is 3.1MB across the 14 programs. This overhead is bounded by the meta-data size (as opposed to the application working set size).

5.3 Global GC Performance

Figure 3 shows the impact of inter-MRE GC load balancing (work stealing) on average pause times of global GC. In this experiment, each MRE executes a single GC thread and can reach only a specific fraction of shared objects. We express the distribution of reachable objects (imbalance) as a pair of percentage values. For perfect balance (50/50), load balancing adds a small overhead. For the most imbalanced configuration (100/0), inter-MRE work stealing reduces GC pause time by 44%. We report average GC pause times (and standard errors) from 15 GCs. This result indicates that cross-MRE load balancing is important for efficient GC in an XMem system.

XMem implements STW parallel copying collection and therefore its GC pause times increase linearly with live data size. Figure 4 presents measurements obtained using two MREs, each with a single GC thread, where live data consists of a binary tree comprising a specific number of nodes. We report average GC pause times for different live data sizes. Global GC latency (computed by extrapolating GC pause time for live data size equal to zero) is 0.9ms. Safepoint latency in a single MRE is 0.7ms on average. Safepoints are reached concurrently by two MREs (they do not add up). Thus, there is 0.2ms overhead imposed by XMem to coordinate global GC across MREs. Copying throughput is 3.3 million nodes/second (where each node corresponds to 5 small objects). This throughput is identical in case of a single MRE (XMem does not degrade it).

5.4 Communication Efficiency for Microbenchmarks

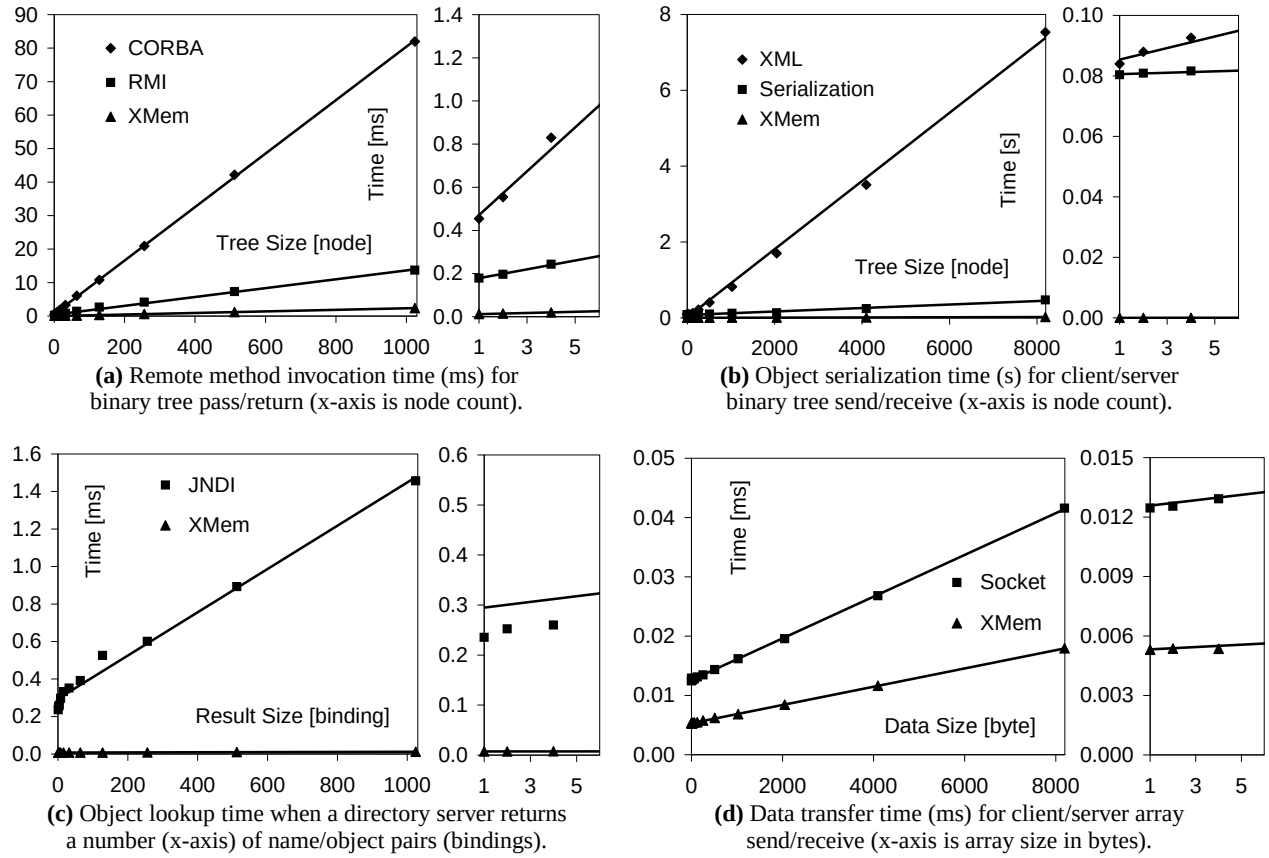
We next evaluate the impact of XMem on the performance of Java communication technologies using our microbenchmarks.

RMI and CORBA. RMI [51] enables inter-MRE type-safe remote method calls. A server registers a remote object using a directory service which is later consulted by the client to look up the remote object by name. Once a remote reference (proxy) is constructed, the client can call remote methods. A client and a server use automatically-generated stubs and skeletons to (de)marshall arguments and return values. CORBA [14] employs a more portable transport protocol (IIOP) to interoperate with other runtimes. Our microbenchmark times a remote method call that takes a binary tree of objects as an input argument and returns another binary tree as an output value. We employ binary trees as the microbenchmark since they represent a middle-ground in common data structures: they are neither sparsely-connected (like linked lists) nor densely-connected (like complex graphs).

Figure 5(a) shows the average invocation time (y-axis) for an increasing number of nodes in the binary tree (x-axis). We implement the remote call using XMem by allocating binary trees directly in the shared memory. Client-server interaction is coordinated by monitor synchronization. Having allocated a tree, the client notifies the server that the input is ready. Once the server allocates the output tree, the client is notified that the call is complete. XMem reduces latency 15x and 37x while increasing throughput (calls/second) 6x and 35x, compared to RMI and CORBA, respectively, since XMem avoids argument marshalling and network communication.

Serialization and XML. Object serialization [54] provides a type-safe mechanism for transforming an arbitrary graph of objects implementing the `java.io.Serializable` interface into a binary byte stream which then can be used to reconstruct the original data structure. A runtime-portable alternative to binary representation is XML. We compare default and XML-based serialization against their XMem implementation. Our microbenchmark times the exchange of an object graph between a server and a client. A client allocates a binary tree of objects, serializes it and sends the result to the server over a socket. The server deserializes the tree, allocates a response (being a binary tree of the same size) and sends it back to the client in a serialized form. In XMem, we allocate the tree in the shared memory and notify the other side that the data is ready (we consider the overhead of copying below). Figure 5(b) presents the average serialization time (in msecs on y-axis) for a tree of 1–1024 nodes (x-axis). XMem eliminates the need for serialization and data transfer and thus improves throughput (calls/second) 20x and 391x while reducing latency by around 7000x compared to default and XML serialization.

JNDI. JNDI provides access to directory services, such as LDAP or RMI registry, where clients can look up objects by name as well as evaluate search queries. Our microbenchmark first binds a specific number of objects in an RMI registry and then performs a query that lists all available bindings (name/object pairs). We time the latter operation only as it is more important (directories are rarely modified). XMem keeps the bindings in the shared memory and returns an enumeration of their subset in response to each query.



(a) Remote method invocation time (ms) for binary tree pass/return (x-axis is node count).

(b) Object serialization time (s) for client/server binary tree send/receive (x-axis is node count).

(c) Object lookup time when a directory server returns a number (x-axis) of name/object pairs (bindings).

(d) Data transfer time (ms) for client/server array send/receive (x-axis is array size in bytes).

Figure 5. Microbenchmark communication performance. We blow up the axes using a second graph snapshot to make latency visible. We include regression lines on each graph.

This enumeration is allocated in the shared memory and returned to the client by means of a notification. Figure 5(c) shows the average results gathered for a varying number of bindings (1–1024). XMem reduces latency 32x and increases throughput (lookups/second) 240x which can be attributed to copy and transfer avoidance.

TCP/IP Sockets. Network sockets operate at the byte level (as opposed to the object level) and therefore have no notion of type-safety. However, we compare their efficacy with XMem for completeness. Our microbenchmark measures the time needed to transfer a byte array of a certain length from a client to a sever and vice versa using TCP/IP sockets. We implement XMem-based communication by allocating a shared byte buffer. Each party writes into the shared buffer and then notifies its peer that the new data is available. Figure 5(d) compares the transfer time (in ms) using conventional sockets for data sizes 1 to 8192 bytes (x-axis). XMem increases throughput and decreases latency both by 2x by avoiding network stack interposition and redundant data copying.

Copying Overhead. Occasionally, an object graph needs to be copied to the shared memory to ensure full transparency of communication. In case of remote method invocation and object serialization this translates to allocating locally and then copying an object tree to the shared memory just before notification. In case of sockets, two copies are necessary in the worst case: first from a local buffer (client side) to a shared buffer and then from the shared buffer to a local buffer (server side). Since bindings in directory services are immutable, it is sufficient to copy only the enumeration object encapsulating the query result while leaving the bindings intact. Table 2 shows the impact of copying on latency and throughput. We report relative performance of XMem with copying to existing technologies run on top of HotSpot (HS) and the non-copying version of XMem. Columns 2 and 4 show latency and throughput for XMem with copying vs. HotSpot and columns 3 and 5 show these metrics for XMem without copying vs. XMem with copying. When copying is used, XMem still significantly outperforms the extant mechanisms (Columns 2 and 4).

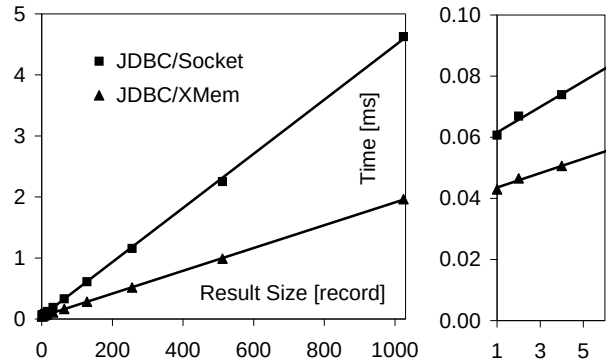
Bench-mark	Latency		Throughput	
	vs. HS	vs. XMem	vs. HS	vs. XMem
RMI	2.5x	5.8x	2.4x	2.4x
CORBA	6.4x	5.8x	14.3x	2.4x
Serial.	1152x	6.1x	8.3x	2.4x
XML	1204x	6.1x	164x	2.4x
JNDI	6.9x	4.6x	71x	3.4x
Socket	2.2x	1.1x	1.5x	1.6x

Table 2. Impact of copying shared data (so that both client and server operate on their own instance) on latency and throughput. Columns 2 and 4 show these metrics for XMem with copying vs. HotSpot and columns 3 and 5 show these metrics for XMem without copying vs. XMem with copying.

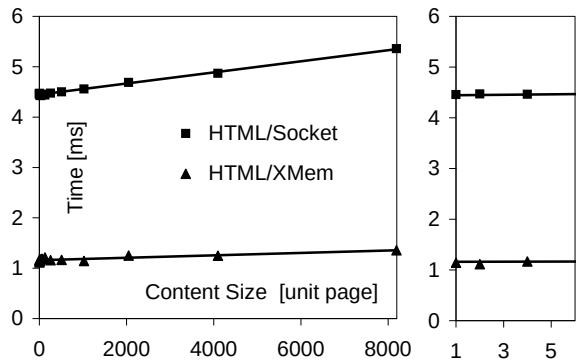
5.5 Application Performance

We next evaluate the impact of XMem on the performance of two enterprise applications. We quantify the improvement in user-perceived throughput and response time by comparing an unmodified database server (Hsqldb) and a web server (Tomcat) with their XMem-based variants.

Hsqldb [29] is a relational SQL database management system that supports in-memory and disk-based data storage. JBoss uses an



(a) Database query processing time (ms) when a server returns a set of records (x-axis is number of records)



(b) Request processing time (ms) when a web server retrieves a web page (x-axis is page size)

Figure 6. Application performance: (a) shows Hsqldb data; (b) shows Tomcat data. We blow up the axes using a second graph snapshot to make latency visible. We include regression lines on each graph.

Bench-mark	Latency		Throughput	
	HS	XMem	HS	XMem
RMI	0.18 ms	15x	75.8 call/s	6x
CORBA	0.45 ms	37x	12.5 call/s	35x
Serial.	80.4 ms	6977x	21.8 object/s	20x
XML	84.0 ms	7292x	1.11 object/s	391x
JNDI	0.24 ms	32x	833 lookup/s	240x
Socket	0.01 ms	2.3x	279 kB/s	2.3x
Hsqldb	0.06 ms	1.4x	227 query/s	2.3x
Tomcat	4.46 ms	3.9x	10^4 request/s	4.2x

Table 3. Summary of XMem impact on latency and throughput for microbenchmarks and applications. We report average baseline performance (Columns 2 and 4) and XMem improvement as a multiple of the baseline (Columns 3 and 5).

embedded Hsqldb database engine by default for persistence and caching. We have modified Hsqldb 1.8.0 to employ shared memory. A client allocates an SQL query as a shared string. The server is then notified, parses the query, and computes the result in the shared memory. Hsqldb maintains an object cache in the shared memory. Internal representation of leaf data in the object cache is based on immutable objects (strings, integers, dates that model SQL objects). Clients can be given a reference to such objects without a risk of modification and therefore most data (and metadata) does not need copying. We have encapsulated interaction over XMem into a JDBC driver for Hsqldb to achieve full transparency. The server listens for connections both on a network socket and in the shared memory. For Hsqldb we measure the impact of XMem on end-to-end throughput (queries/second). Our microbenchmark times the `SELECT * FROM T` statement executed against a table `T` which contains between 1 and 1024 3-field records. Figure 6(a) shows the results. XMem increases throughput 2.3x and decreases latency 1.4x. The Hsqldb JDBC driver performs proprietary data serialization, which is unnecessary in XMem.

Apache Tomcat [1] is an industrial-strength web and servlet container. We have modified Tomcat 6.0 to optimize local request handling using XMem. A client and a server share a byte array and notify each other when sending data. We measure end-to-end performance (requests/second) when retrieving (HTTP `GET` method) static web pages of different sizes (multiples of a unit page size). We use the Apache `httpclient` package to generate conventional HTTP requests. Figure 6 (b) shows the time needed to retrieve a page of a given size. XMem achieves 4x better throughput and 4x shorter latency.

5.6 Results Summary

Table 3 summarizes our results in terms of average latency and throughput. We use least-squares linear regression to obtain latency and throughput as the coefficients in the equation $time = latency + size/throughput$, following [10]. We report throughput in the units appropriate for each protocol. While microbenchmarks focus on communication efficiency (the only additional processing is initialization/allocation of the exchanged data), Hsqldb and Tomcat provide insight into the end-to-end application performance. We observe very significant reduction in latency (over three orders of magnitude) in case of serialization (default and XML-based) – RMI, CORBA, and JNDI use their own, more efficient, serialization and thus benefit less due to XMem. XML-based serialization yields the most significant throughput increase (over two orders of magnitude) since it uses a verbose representation of the object graph and thus transfers more data.

6. Related Work

The key difference between XMem and previously reported systems that coordinate co-located and isolated applications written in type-safe languages is that XMem takes a top-down approach by assuming full isolation between MREs and providing an efficient and straightforward mechanism for direct object sharing while preserving strong OS-assisted resource protection as much as possible. Prior work has focused on bottom-up approaches by introducing weak isolation implemented through replication of basic OS facilities within a single OS process. Such systems are much more complex than XMem, have weaker protection guarantees, and duplicate existing OS mechanisms.

KaffeOS [4] and the Multi-tasking Virtual Machine (MVM) [17] employ a single-application MRE and add support for isolation and multi-tasking. MVM provides isolation via the Isolate API [31]. Multiple programs (tasks) execute in a single MRE instance (OS process) and the MRE manages resources and sharing across them. MVM introduces a level of indirection when accessing static fields and does not support direct object sharing. The system introduces links (communication channels between tasks) but cannot eliminate the object serialization and data copying. KaffeOS supports direct object sharing by means of shared heaps. However, shared heaps are not garbage collected and are coarse-grained entities reclaimed in full when they become unreferenced. KaffeOS lacks support for many state-of-the-art MRE mechanisms like parallel GC and modern synchronization.

Other systems that implement the process/task model within a JVM, include Alta [5], GVM [5], and J-Kernel [59], as well as a multi-tasking JVM described in [7]. These systems strive to provide

resource management and isolation within a single process without relying on hardware/OS protection. Class-loader-based isolation [16] is a standard technique commonly employed by applications servers in order to avoid name space pollution/conflicts between multiple web applications hosted within a single JVM. Such isolation, however, does not prevent interference through static fields of classes loaded by the system (bootstrap) class loader. This last problem was addressed in [11] by introducing a control access model called object spaces where cross-space object accesses are mediated by a security policy. This approach, however, provides weak isolation and imposes overhead on inter-space method calls.

XMem does not have the aforementioned limitations and is significantly simpler than multi-tasking approaches as it leverages the existing infrastructure both at the MRE and OS level. At the same time, XMem offers better fault containment – critical errors do not automatically propagate to other MREs unless a fault affects the shared memory. This decreases the probability of a failure escalating to multiple components. In XMem, MREs are not completely isolated as they share part of their virtual address spaces. However, XMem is significantly more robust than multi-tasking approaches, given that resources other than memory are fully isolated and memory itself is only partially shared. XMem achieves stronger isolation, while providing direct object sharing without introducing any level of indirection (unlike the MVM).

The notion of transparent global and local objects in the context of distributed shared memory (DSM) multi-processors has been used in Split-C [15] and UPC [21]. Unlike XMem, these systems are not type-safe and provide access to global objects at a different cost than to local objects. JavaSpaces [24] provide DSM for applications that implement object flows. Object repositories in JavaSpaces are type-safe but the system uses serialization and provides no shared memory support for co-located application components. Other DSM systems for type-safe languages include single-system-image approaches to implementing a global object space such as cJVM [2], JAVA/DSM [67], JESSICA [40], Hyperion [42], JavaParty [50], and MultiJav [12]. While XMem targets sharing between co-located MREs, software DSM focuses mostly on distributed protocols necessary to guarantee memory consistency and cache coherence models defining certain semantics for concurrency in a distributed system.

Runtime systems for concurrent languages that offer built-in constructs for inter-process communication include Erlang [3], Occam [45], and Limbo [20]. These systems build on the algebra of communicating sequential processes [27] and provide a point-to-point message passing mechanism for lightweight processes with share-nothing semantics. In contrast, XMem adheres to the shared memory programming model. Unlike XMem, Erlang is a functional language and requires the shared objects to be immutable. XMem targets general-purpose imperative procedural languages.

In language-based operating systems [53], such as Singularity [22, 30], JX [25], JNode [35], Inferno [20], SPIN [9], Oberon [65], and JavaOS [33], processes share a single address space and use type and control safety provided by a trusted compiler (via static analysis) to guarantee memory protection and resource isolation without implementing a hardware-assisted reference monitor. Singularity is a micro-kernel OS, implemented mostly in C#, supporting efficient communication between multiple isolated processes. Its design differs from XMem in several ways. First, XMem leverages hardware memory protection, while Singularity provides lightweight software-based isolation via type-safety (multiple applications execute in a single address space). Second, Singularity provides message-passing via typed channels and explicit communication primitives. In contrast, XMem provides a shared-memory-based implicit communication where only the initial handshake employs the channel abstraction. Next, in Singularity communication is limited to two endpoints and involves the transfer of ownership of a memory block (there is no data sharing between the sender and the receiver). XMem enables direct and transparent object sharing between any number of threads, potentially from distinct MREs. Finally, Singularity employs block-based reference counting garbage collection while XMem uses more fine-grained tracing GC.

To date, virtual memory manipulation (which is used by XMem to implement double mapping of the GCT and LCT) has been used in MREs mostly in the context of GC [61, 38, 66, 26, 13]. For example, the Compressor [38] employs double mapping to enable concurrent compaction, and the Mapping Collector [61] compacts free space by remapping to avoid object copying.

7. Conclusions

We present XMem, type-safe and transparent shared memory for isolated, co-located MREs. The motivation behind XMem is more efficient, cross-component interaction and communication in enterprise multi-tier applications deployed on a single host. XMem provides stronger fault and resource isolation than previously reported systems, while enabling efficient direct object sharing over private channels. To guarantee type-safety, XMem extends state-of-the-art MRE services such as synchronization, class loading, object allocation, and garbage collection, as well as introduces global operations to coordinate MREs using a single shared segment. XMem manipulates virtual memory mapping (using a standard OS interface) to avoid indirect memory access. XMem is transparently integrated within the MRE infrastructure and can be used to optimize existing communication protocols, such as RMI. We implement XMem in the HotSpot JVM and evaluate it empirically. XMem introduces tolerable space/time overhead while improving efficiency (latency and throughput) of extant J2SE/J2EE communication mechanisms by up to several orders of magnitude.

As part of future work, we plan to extend XMem to heterogeneous MREs. Currently XMem requires that the attached MREs implement the same object model and have the same internal representation of classes (types). We are investigating ways to support shared memory communication and coordination between different Java Virtual Machines as well as between different language virtual execution environments to enable better cross-language interaction that is simple and easy to use. In addition, we are investigating similar approaches for IPC between other virtualization systems, e.g., virtual machine monitors and full system virtual machines.

Acknowledgments

We thank the anonymous reviewers for providing insightful comments on this paper. This work was funded in part by NSF grants CCF-0444412, CNS-CAREER-0546737, and CNS-0627183.

References

[1] Apache Tomcat. `http://tomcat.apache.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, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

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

[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] H. G. Baker. Infant mortality and generational garbage collection. *SIGPLAN Not.*, 28(4), 1993.

[7] D. Balfanz and L. Gong. Experience with secure multi-processing in Java. In *ICDCS*, 1998.

[8] BEA WebLogic Application Server. `http://www.bea.com`.

[9] 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.

[10] F. Breg and C. D. Polychronopoulos. Java virtual machine support for object serialization. In *Java Grande*, 2001.

[11] C. Bryce and C. Razafimahefa. An approach to safe object sharing. *SIGPLAN Not.*, 35(10), 2000.

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

[13] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *VEE*, 2005.

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

[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. Application isolation in the Java virtual machine. In *OOPSLA*, 2000.

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

[18] The DaCapo benchmarks. `http://dacapobench.org`.

[19] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM*, 2004.

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

[21] T. El-Ghazawi, W. Carlson, and J. Draper. UPC Language Specifications V, 2001.

[22] 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.

[23] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *JVM*, 2001.

[24] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice (Jini Series)*. Pearson Education, 1999.

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

[26] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *PLDI*, 2005.

[27] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1), 1983.

[28] HotSpot Java Virtual Machine GC. `http://java.sun.com/javase/technologies/hotspot`.

[29] Hsqldb. `http://www.hsqldb.org`.

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

[31] Isolate API. JSR-121. `http://jcp.org`.

[32] Java 2 Enterprise Edition. `http://java.sun.com/javaee/`.

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

[34] JBoss Enterprise Middleware. `http://www.jboss.com`.

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

[36] R. Jones and C. Ryder. Garbage collection should be lifetime aware. In *ICOOOLPS*, 2006.

[37] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[38] H. Kermany and E. Petrank. The Compressor: Concurrent, incremental and parallel compaction. In *PLDI*, 2006.

[39] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA*, 1998.

[40] 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.

[41] J. Maassen, R. V. Nieuwpoort, R. Veldema, H. E. Bal, T. Kielmann, C. J. H. Jacobs, and R. F. H. Hofman. Efficient Java RMI for parallel programming. *Programming Languages and Systems*, 23(6), 2001.

[42] M. Macbeth, K. McGuigan, and P. Hatcher. Executing Java threads in parallel in a distributed-memory environment. In *CASCON*, 1998.

[43] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. *SIGPLAN Not.*, 40(1), 2005.

[44] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX Technical Conference*, 2005.

[45] *Occam Programming Manual*. Inmos Corporation, 1984.

[46] Open Source J2SE. `http://openjdk.java.net`.

[47] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *OSDI*, 2002.

[48] Y. Ossia, O. Ben-Yitzhak, and M. Segal. Mostly concurrent compaction for mark-sweep GC. In *ISMM*, 2004.

[49] M. Perry. Shared Memory Under Linux, 1999. `http://fscked.org/writings/SHM/shm.html`.

[50] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11), 1997.

[51] Java RMI Specification. `http://java.sun.com`.

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

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

[54] Java Object Serialization Specification. `http://java.sun.com`.

[55] SPEC. `http://www.spec.org`.

[56] D. Stefanovic, M. Hertz, S. M. Blackburn, K. S. McKinley, and J. E. B. Moss. Older-first garbage collection in practice: Evaluation in a Java virtual machine. In *MSP*, 2002.

[57] T. Suezawa. Persistent execution state of a Java virtual machine. In *Java Grande*, 2000.

[58] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Not.*, 19(5), 1984.

[59] 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.

[60] IBM WebSphere Application Server. `http://www.ibm.com`.

[61] M. Wegiel and C. Krintz. The Mapping Collector: Virtual memory support for generational, parallel, and concurrent compaction. In *ASPLOS*, 2008.

[62] P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, Univ. of Texas, 1994.

[63] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *IWMM*, 1995.

[64] P. R. Wilson and T. G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Not.*, 24(5), 1989.

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

[66] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *OSDI*, 2006.

[67] W. Yu and A. L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency - Practice and Experience*, 9(11), 1997.