Design and Performance Analysis of a Distributed Java Virtual Machine

Mihai Surdeanu and Dan Moldovan
Language Computer Corporation and
Department of Computer Science
University of Texas at Dallas
mihai@languagecomputer.com
moldovan@utdallas.edu

Abstract

This paper introduces DISK, a distributed Java Virtual Machine for networks of heterogenous workstations. Several research issues are addressed. A novelty of the system is its object-based, multiple-writer memory consistency protocol (OMW). The correctness of the protocol and its Java compliance is demonstrated by comparing the non-operational definitions of Release Consistency, the consistency model implemented by OMW, with the Java Virtual Machine memory consistency model (JVMC) as defined in the Java Virtual Machine Specification. An analytical performance model was developed to study and compare the design tradeoffs between OMW and the lazy invalidate Release Consistency (LI) protocols as a function of the number of processors, network characteristics and application types. The DISK system has been implemented and running on a network of 16 Pentium III computers interconnected by a 100Mbps Ethernet network. Experiments performed with two applications: parallel matrix multiplication and traveling salesman problem confirm the analytical model.

Index terms: object-oriented distributed shared memory, Java Virtual Machine, performance analysis, memory consistency protocols, consistency models

1 Introduction

The desire for software simplicity in complex projects is in part responsible for the large popularity of the Java language. Complex projects, on the other hand, often require parallel or distributed processing. Unfortunately, so far, parallelism in Java has been limited to either multi-threading on symmetric multiprocessors (SMPs) or distributed computing using Remote Method Invocation (RMI). None of these options is extremely attractive: the first comes with an additional hardware cost, and the second implies increased software complexity. Our goal in this project is to take advantage of the inherent parallelism available in a network of workstations (NOW) without requiring any modifications to Java multi-threaded applications.

This paper focuses on the design of a *correct* (meaning compliance with the Java Virtual Machine Specification (JVMS) [17]) and *performant* distributed shared memory (DSM) Java Virtual Machine (JVM). Several research issues are addressed.

A new object-based multiple-writer *memory consistency protocol* (OMW) for a distributed Java Virtual Machine is presented. OMW is adapted from the update-based, multiple-writer Lazy Release Consistency protocol. Taking advantage of the object framework offered by the JVM, OMW adds the following improvements to the original protocol: (i) The protocol *minimizes its overhead* by automatically classifying objects as shared or unshared. Because unshared objects have no consistency overhead, the protocol's strategy to maintain objects unshared as much as possible leads to significant performance improvement. (ii) *Access faults are completely avoided.* Based on the thread structure, OMW detects if an object may be accessed by a remote processor and provides object copies as necessary. (iii) *The OMW protocol is completely decentralized.* Due to the update nature of the protocol, all nodes have up-to-date information about shared objects protected by synchronization variables, hence object directories

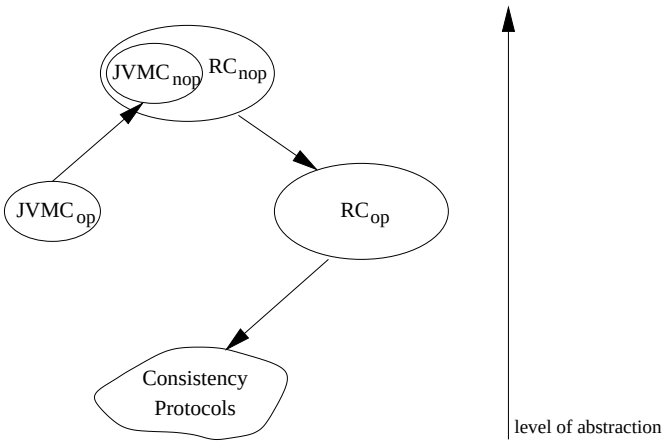

Figure 1: Transition from the JVMC operational definition to consistency protocols

are not required.

Correctness becomes a very important issue when one realizes that the JVMS consistency model is not compatible with a distributed shared memory architecture. The JVMS provides an operational definition of the Java Virtual Machine memory consistency model (JVMC) by defining the interaction rules between two memory layers: global shared memory and thread private memory (see Appendix A). This definition assumes that shared memory is available, hence it is not applicable to architectures without shared memory (i.e. NOWs). To mitigate this architectural incompatibility we introduce an architectural-independent, non-operational definition for JVMC (JVMC_{nop}). A comparison of JVMC_{nop} with a non-operational definition of release consistency (RC_{nop}) indicates that JVMC is stronger than RC, but, for data-race-free programs [2, 13], these two consistency models are equivalent. Even if it is not explicitly stated in the JVMS, the data-race-free program assumption is generally true for programs written for weak consistency models, hence the applicability of this result is not reduced. As shown in Figure 1, this equivalence is the bridge that links JVMC with consistency protocols originally designed for RC. For asynchronous algorithms, the JVMS is not feasible for a distributed shared memory implementation. In order to be JVM compliant for asynchronous programs, release consistency protocols required additional constraints which affect their performance. To our knowledge, this is the first work on distributed JVMs that reports this issue.

An *analytical performance model* was developed. It defines the parallel execution time in terms of the computation time and the overhead consisting of synchronization, thread creation time, access fault overhead, and consistency barrier overhead. The model allows us to compare the OMW protocol with a lazy invalidate (LI) protocol and realize possible tradeoffs as a function of the number of processors, network characteristics and application types. The *experimental results* measured on two different benchmarks confirm the analytical model.

The OMW protocol has been *implemented in DISK* (DIStributed Kaffe[1]), a distributed Java Virtual Machine, running on a network of sixteen Pentium III processors. For flexibility and portability, DISK has been implemented entirely in user space.

2 Related Work

A distributed Java system is appealing because it combines high performance with language elegance and simplicity. Figure 2 shows how both DSM systems and Java environments have evolved towards this common ground: DSM systems seek the elegant object-oriented framework, while Java environments indicate a tendency towards distributed computing.

[1] We use Kaffe[19] version 1.0.b5.

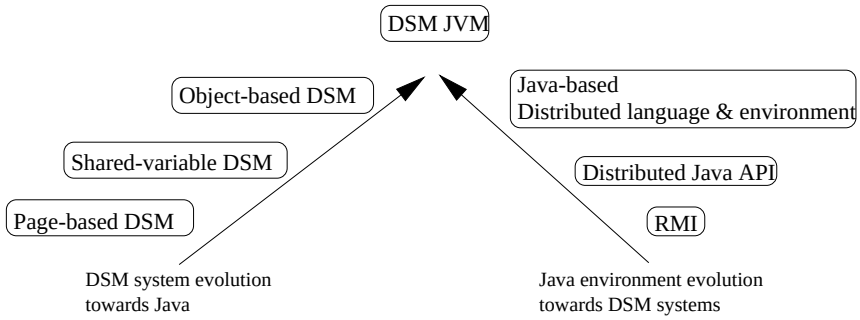

Figure 2: Evolutionary tendencies in DSM and Java systems

Page-based DSM systems (IVY, TreadMarks, Coherent Virtual Machine, Millipede) were chronologically the first to appear, and they still enjoy a large popularity. IVY was the first system to introduce the notion of distributed shared memory to networks of workstations [21]. IVY emulates the cache of a multiprocessor architecture using the workstation's memory management unit and the operating system. The virtual memory fault handler is redirected to IVY's routine, which is aware that a missing page may not be available on the local disk (like a traditional virtual memory system), but on a remote workstation. While page-based DSM systems have evolved significantly since IVY, the basic idea of using the virtual memory management system to implement consistency remained the same. IVY implements *sequential consistency* using a *single writer multiple reader* protocol. Modern page-based DSM systems attempt to correct IVY's two main disadvantages: first, sequential consistency protocols have poor performance due to the large number of consistency messages, and second, single writer protocols have the *false sharing* problem. The false sharing problem is caused by two (or more) variables being located on the same page. If each variable is used by a different processor, the page bounces between nodes even though conceptually there is no shared data. The TreadMarks page-based DSM system avoids both these disadvantages [5]. TreadMarks achieves performance close to equivalent message-passing architectures by replacing sequential consistency with *lazy release consistency* [20]. Lazy release consistency delays the propagation of consistency messages until a synchronization variable is acquired by a different processor. The size of consistency messages is reduced because only pages known to be out of date on the remote node are updated/invalidated. The false sharing problem is mitigated by the usage of *multiple writer protocols.* Multiple writer protocols allow a page to be concurrently written, as long as different processors access different parts of the page. The latest research from the TreadMarks group focuses on run-time *adaptive* memory consistency protocols. Based on the observed page accesses, the TreadMarks memory consistency protocol is capable of selecting between single or multiple writers, and between invalidate or update strategies [6]. Other page-based DSM systems, such as Coherent Virtual Machine (CVM) [11, 26] and Millipede [22], reduce the number of consistency messages through *thread migration.* Based on their page access profile, threads are migrated on the nodes that maximize the number of local pages, hence reducing the number of consistency messages.

Shared-variable DSM systems (Munin [10], Midway [9]) brought language support for DSM. Even though it is variable-based, Munin uses the virtual memory management unit to implement consistency. To avoid the false sharing problem, the Munin compiler allocates every shared variable to a separate page. Several protocols are implemented in Munin for shared variables: (i) *read-only* variables are freely replicated without any consistency management; (ii) *migratory* variables are not replicated: they migrate from node to node as critical regions are entered; (iii) *write-shared* variables are maintained consistent through a multiple writer protocol implementing eager release consistency [13]; and (iv) *conventional* variables, which are sequentially consistent. The protocol selection is performed by the programmer which annotates shared variables and the desired consistency protocol with special keywords. Midway is another well known variable-based DSM system, in many aspects similar to Munin. Midway's novelty is the implementation of a new consistency model: *entry consistency.* Entry consistency associates a synchronization variable with every shared variable. Similar to lazy release consistency, consistency messages are propagated when synchronization variables are acquired, but, unlike release consistency, entry consistency updates/invalidates only the shared variable associated with the acquired synchronization variable. Similarly to Munin, Midway relies on the programmer to annotate shared variables.

Object-based DSM systems (Linda, Orca) provide a more elegant programming framework for DSM environments. Linda provides a conceptual view of the distributed shared memory as a *tuple space* [3]. Tuples are sets of fields of certain types, roughly similar to C structures. Tuples can be added to the shared space with the *out* operation, and retrieved with the *in* operation, which implements an associative search over the tuple space. The tuple space is maintained consistent by broadcasting changes on all nodes part of the Linda system. Unlike Linda, whose programming interface is an extension of the C language, Orca proposes a completely new language [8]. Orca is a parallel environment designed from scratch as an object-oriented DSM system. Orca's programming language is based on Modula-2, with additional distributed programming features. The Orca runtime system is implemented on top of the Amoeba operating system. A very important feature of this operating system is that it has support for reliable broadcast. This feature is used to broadcast the operations performed on shared objects, such that they are simultaneously performed on all object copies. Because the reliable broadcast mechanism offers a unique order for all operations performed on shared objects, Orca implements sequential consistency. Orca objects can be in one of two states: *single-copy* or *replicated*. Replicated objects are maintained consistent with the broadcast mechanism described above. Single-copy objects are accessed through remote procedure calls. Based on object access profiles, the Orca runtime system decides if an object should be replicated or maintained in a single copy.

From the Java perspective, Sun offers support for distributed computing in Java with *Remote Method Invocation (RMI)* since release 1.1, and release 1.2 is Corba-compliant. Even though both RMI and Corba hide message exchanges behind method calls to remote objects, they are still explicit client-server architectures lacking the transparency required from a distributed system. RMI and Corba also lack true support for optimizations like data replication and computational load balancing commonly seen in distributed systems.

The next evolutionary step is enhanced *distributed Java Application Programming Interfaces (API)*. An interesting representative in this category is the Aleph toolkit [15]. Aleph provides distributed services in the form of an API including both data and control shipping. Jini(tm) is Sun's version of a distributed API [18]. Instead of providing a shared space for objects, Jini is a platform for sharing *services*. A service is an entity that can be used by a user or a program. Any kind of network device can make a service available through Jini, regardless of the connection type or the software network interface (RMI, Corba etc). Jini also helps clients find and access the posted services. The Jini service protocol includes secure access to services, and accesses in the form of leases or transactions. Distributed Java APIs, such as Jini and Aleph, do not offer complete transparency, but they are attractive due to their flexibility.

JavaParty [24] implements a *Java-based distributed environment*, built as a cluster of regular JVMs connected through RMI. JavaParty uses a new keyword to mark objects as shared. A preprocessor translates this code into RMI-based Java code. JavaParty provides the programmer with a simple interface for distributed programming. However, JavaParty does not offer complete transparency, the programmer having to separate local objects from shared objects. To our knowledge the only other project that qualifies as a distributed shared memory Java Virtual Machine is Java/DSM [27]. Java/DSM is a JDK 1.0.2-compliant distributed JVM implemented on top of TreadMarks. Java/DSM allocates objects and classes with TreadMarks shared-memory allocation routines. This modular design allows TreadMarks to provide the required memory consistency, while Java/DSM implements the JRE. However, we believe that DISK's approach, where the JVM is tightly connected to the consistency protocols allows for better tuning and higher performance. Java/DSM also fails to indicate whether or not the consistency protocols used are JVMS compliant.

The theoretical foundation of this paper builds upon the work published by Gontmacker and Schuster [14] who introduce a non-operational definition for JVMC when no locks are involved (first two subsections of Appendix A) and compare this definition with traditional consistency models. When locks are involved, they show that JVMC is stronger than RC, but no definition is provided. Unlike Gontmacker and Schuster, we provide a complete, self-contained, non-operational definition for JVMC, including both the non-lock and lock constraints. Furthermore, we show that this definition is indeed stronger than RC, but, for data-race-free programs these two models are equivalent.

3 DISK Design Overview

The DISK system consists of a set of nodes connected through software point-to-point TCP channels. Figure 3 shows the block diagram of a DISK node. The modules part of a DISK node are detailed next.

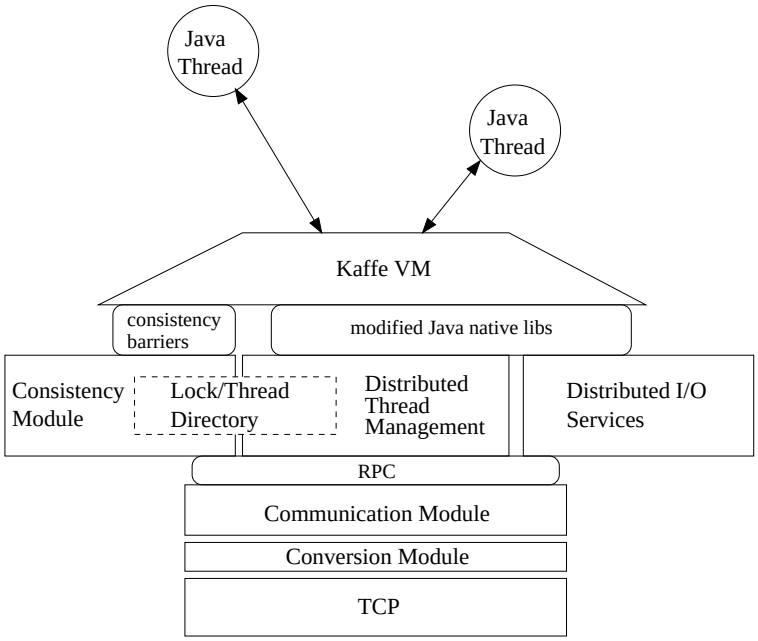

Figure 3: DISK node block diagram

DISK uses the *Kaffe Java Virtual Machine* [19] to execute the Java bytecode on each node. This module contains the Just-In-Time (JIT) compiler and the garbage collector.

The *consistency barriers* link the JVM with the consistency modules. To implement the software barriers we modified Kaffe's JIT compiler to insert consistency function calls before object and class write-access instructions. We show in section 6 that the average overhead of the software-implemented consistency barriers for the tested applications is only 3%. This is possible due to the fact that the consistency barriers affect only write instructions to Java objects and classes. Write accesses to Java primitive types, or JVM internal data are not affected.

The *modified Java native libraries* redirect the JVM thread, lock and I/O native functions to our own distributed implementations. Both this module and the software barrier module require minimal changes to the JVM source code.

The *consistency module* contains the core of the system: the memory consistency protocol. This protocol is described in detail in section 4.

The main function of the *distributed thread management module* is *processor allocation* for Java threads. DISK currently supports only a round-robin processor allocation algorithm. Both this module and the consistency module use the services of the *lock and thread directory.* This directory tracks the location of every lock owner, and the processor where every thread is allocated. We currently use a centralized directory for threads in order to provide up-to-date information for a possible load-balancing processor allocation algorithm. The lock directory is distributed on all N nodes in a round-robin strategy based on the lock id modulo N.

The *distributed I/O module* provides I/O transparency to Java application threads. The goal in designing this module was to have as little inter-node interaction as possible. I/O accesses generated during class loading are performed "locally", on an NFS-mounted file system. The protocol for application-based I/O accesses is based on the

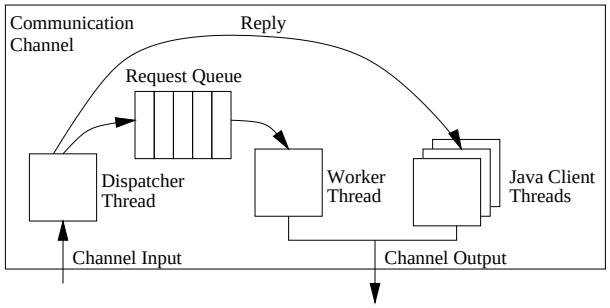

Figure 4: DISK communication channel end-point

following algorithm:

- When a file is opened, the upper $log(N)$ bits of the file descriptor are set to the local node id (N being the number of nodes in the system). This algorithm limits the number of possible file descriptors to $\frac{2^{word\ size}}{N}$ on each node. However, the limit imposed by the operating system is usually much smaller, hence we do not consider this a major drawback.
- Read or write accesses to a file descriptor are performed locally if the upper $log(N)$ bits of the file descriptor are equal to the node id. Only if the node id stored in the file descriptor is different than the local node id the I/O request is forwarded to a remote node. The standard I/O streams (input, output and error) are exempt from this algorithm: they are all redirected to the node where the main Java thread runs.

The DISK modules introduced above communicate with their peers on other nodes using our own implementation of a multi-threaded *Remote Procedure Call (RPC) protocol*. This protocol hides the details of the communication layer behind the stub/skeleton RPC abstraction.

As mentioned before, DISK uses a fully-connected logical topology, each node-to-node link being a TCP channel. The channel end-points are managed on each node by the *communication module*. The description of a communication channel end-point is presented in Figure 4. Each channel input is monitored by a dedicated *dispatcher thread*. Incoming requests are enqueued by the dispatcher thread in the *request queue*. One *worker thread* per channel is responsible for dequeuing and serving request messages. This dispatcher/worker strategy provides better CPU utilization than the more simple approach where the dispatcher thread serves the incoming requests. If the incoming message is a reply, the dispatcher thread wakes up the corresponding Java thread waiting for this response.

The *conversion module* is an optional module. If plugged in, this module is in charge of translating data to/from network standard. The conversion module allows heterogenous systems to be part of the same DISK system.

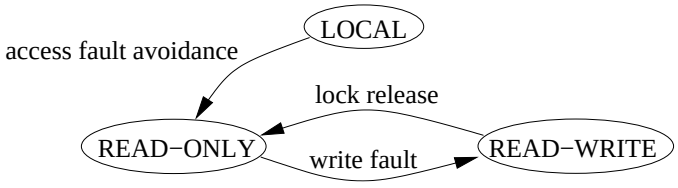

Figure 5: Object State Transitions

4 Object-Based Multiple-Writer Memory Consistency Protocol (OMW)

4.1 Protocol Description

The *object-based, multiple-writer* memory consistency protocol (OMW) implemented in DISK is adapted from the update-based, multiple-writer Lazy Release Consistency protocol introduced by Keleher in [20]. Taking advantage of the object framework offered by the JVM, OMW adds the following improvements to the original protocol: (i) *access faults are completely avoided.* Based on the thread structure, OMW detects if an object may be accessed by a remote processor and provides object copies as necessary. (ii) Until an object is accessible by a remote thread, the protocol marks it as unshared, thus *reducing the consistency overhead.* (iii) *The OMW protocol is completely decentralized.* Due to the update nature of the protocol, all nodes have up-to-date information about shared objects protected by synchronization variables, hence object directories are not required.

The object state transitions permitted by the protocol are shown in Figure 5. Initially, all objects are in the LOCAL state. An object maintains this state as long as it is accessible from a single thread. There is no consistency overhead for objects in the LOCAL state. The addition of this state to the OMW protocol was determined by the fact that more than 90% of the Java heap is allocated to objects that are not reachable outside the thread that created them [12], hence they can be ignored by the consistency protocol. To our knowledge OMW is the only consistency protocol that uses this observation to reduce the consistency overhead. A LOCAL object changes its state to READ-ONLY if the *access-fault-avoidance algorithm* detects that the object has become shared. An object becomes shared in one of the following two conditions:

Conditions for access fault avoidance:

1. The object is directly or indirectly reachable from a thread object, and the thread attached to this object is started. If the newly started thread is allocated to a remote processor, OMW provides remote copies for all objects not available on the destination processor but reachable from the thread object. This operation is needed to avoid possible access faults from the destination processor. The initial state for remote copies is READ-ONLY.

2. The object is directly or indirectly reachable from an updated object that is propagated to a remote processor. Similarly with the previous case, OMW provides remote copies for all objects not available on the remote processor but reachable from the update.

The first write access to a READ-ONLY object generates a write fault. Upon a write fault, OMW creates a copy of the object (a *twin*), and changes the object state to READ-WRITE. A READ-WRITE object moves back in the READ-ONLY state when a synchronization variable is acquired by a remote processor. When a synchronization variable is yielded to a remote node, a *write-notice* for each modified object is created (the write-notice being a bitmap of the modified object fields), the twin is discarded, and the object state is moved back to READ-ONLY. Based on the object write-notices, OMW propagates only the modified object fields (the *diff*) to the remote processor. According to the access fault avoidance algorithm, copies of objects reachable from the diffs are also propagated to the remote node. To avoid excessive communication, OMW maintains for each shared object a list of processors having a copy of the object (the *copyset*). Objects reachable from diffs are sent to the remote node only if the destination processor is not already in the copyset.

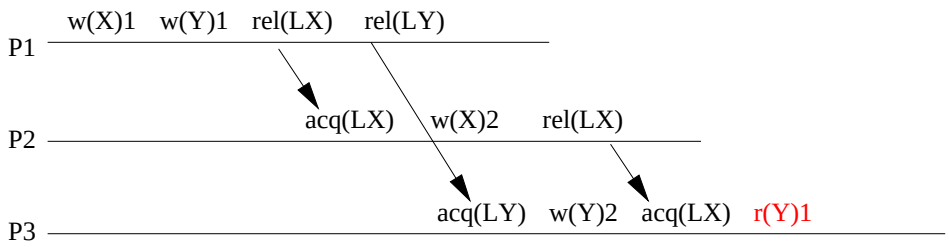

Figure 6: Example of incorrect diff combining

OMW uses *diff combining* to reduce consistency network traffic. An example of diff combining and the dangers generated by this method is presented in Figure 6. In Figure 6, three processors share two Java variables, X and Y, part of the same Java object O. Accesses to variable X are synchronized through lock LX, and accesses to variable Y are maintained consistent through lock LY. Java consistency requires that before any lock acquire, *all* modified variables be updated. Hence, when processor P2 acquires lock LX, both X and Y are propagated from P1 to P2. The inconsistency arises when processor P3 acquires lock LX. Since up to this point there has been no communication between node P2 and P3, processor P2 is not aware that P3 has seen the write to variable Y from P1. Hence the combined diff for object O sent from P2 to P3 includes both variable X with value 2 and variable Y with value 1. Applying the received diff on the local copy of object O on node P3 overwrites the w(Y)2 operation, an illegal action since, according to the data-race-free model, the operation w(Y)1 on node P1 "happened before" w(Y)2 on node P2 [2]. The problem illustrated in Figure 6 is avoided in DISK by *not accepting write notices that have already been seen on the receiving node.* Formally, each processor maintains an interval matrix im with one row per processor. For example, if $im[p][q] = 2$, the local node knows that processor p has seen all write notices generated by processor q up to interval 2. On each node, a new interval starts when a lock ownership is either accepted from, or yielded to a remote node. The algorithms for sending and receiving update messages are the following:

Algorithm 1 Algorithm for sending combined diffs to processor p

1. For all processors q, send to p all write notices belonging to interval i_q, where $i_q > im[p][q]$.
2. For each object o modified in the sent intervals, build the combined write notice $sentwn_o$ as the bit OR of all write notices for this object in the sent intervals:
   $sentwn_o = OR(wn_o,\ wn_o \subset i_q\ and\ i_q > im[p][q])$
3. For each object o modified in the sent intervals, build the combined diff to include all fields marked in $sentwn_o$.
4. Send the combined diffs.

Algorithm 2 Algorithm for receiving combined diffs

1. Receive all intervals i_q. If $i_q > im[this][q]$ accept this interval. Otherwise, discard interval i_q.
2. For each object o modified in the received intervals, build the combined write notice $recvwn_o$ as the bit OR of the write notices part of accepted intervals:
   $recvwn_o = OR(wn_o,\ wn_o \subset i_q\ and\ i_q > im[this][q])$
3. For each object o modified in the received intervals, receive the combined diff. Apply on the local copy only fields marked in $recvwn_o$.

The diff sending/receiving operations are performed when a synchronization variable is acquired by a remote processor. In Java, every object is associated with a synchronization variable. To reduce the overhead of the synchronization operations, OMW marks synchronization variables as shared *only* when the corresponding Java object becomes shared. For unshared synchronization variables, OMW uses the JVM built-in MONITORENTER and MONITOREXIT operations. Only when a shared synchronization variable is not available on the local processor a request is sent to the corresponding synchronization variable directory. The request includes the interval timestamps of the node requesting the synchronization variable. This information is updated on the node granting the lock before diffs are constructed and sent. The Linearizability required by the JVM specification for synchronization variables (see section 5) is provided by the lock directories. Figure 7 details the situation when two requests for the same synchronization variable are sent to the synchronization variable directory. The lock directory records the last request for each shared synchronization variable. When processor P2 requests the lock owned by P1, the lock directory records P2's address and forwards the request to P1. When processor P3 requests the same lock, the requests is forwarded to processor P2, even though P2 may not own the synchronization variable yet. This simple mechanism of forwarding a synchronization variable request to the node with the last known request implements a distributed ownership queue, as illustrated by the lock yield messages in Figure 7. The order given by the distributed queue guarantees Linearizability.

The advantages of the OMW protocol are the following:

- first, the protocol minimizes its overhead by automatically classifying objects as shared or unshared. Previous work reports that in Java 90% of the objects are unshared objects [12]. Because unshared objects have no consistency overhead, the protocol's strategy to maintain objects unshared as much as possible leads to significant performance improvement. To our knowledge, OMW is the only consistency protocol that automatically detects when an object becomes shared and uses this information to reduce its overhead. Other DSM systems either consider all objects shared [8], or they require the programmer to annotate the shared objects [10, 23, 24].

- second, the protocol does not require consistency barriers for read accesses, and the consistency barriers for write accesses become very simple: their only purpose is to initiate the transition from the READ-ONLY to the READ-WRITE state. This feature is very important for a DSM system with all-software implementation for consistency barriers. As shown in section 6, OMW's consistency barriers take an average of 3% of the sequential execution time, versus an average of 110% for the traditional invalidate-based protocols.

- third, the protocol is almost completely decentralized. Besides the lock directories required to provide Linearizability for synchronization variables, the protocol maintains no centralized information. Due to the update nature of the protocol, processors have up-to-date information about all shared objects. Hence there is no need to maintain centralized object directories or implement other strategies to locate valid object copies.

The main disadvantage of the protocol is the potential propagation of more objects than necessary. This may happen in one of the two following situations:

- objects that are not accessible anymore by a processor are still updated during lock acquire operations performed by the processor. This can be avoided by a garbage collector that interacts with the OMW protocol. When a shared object copy is no longer accessible by a processor, the garbage collector can remove the local processor from the object copyset. During future update operations, the copyset gets propagated to remote nodes which thus become aware that this node is no longer interested in the object. The DISK garbage collector does not currently update object copysets, but the framework for a possible interaction with the garbage collector is prepared.

- shared objects that are accessible by a processor, but are not actually accessed, are still updated. Because the objects are still accessible the garbage collector can not modify their copyset and the processor continues to receive useless updates. The only solution to avoid this problem is to use an invalidate approach. However, for a system where most objects are small [12], and the overhead of consistency barriers for read accesses is large, an update approach is an attractive solution. Other projects have taken similar approaches [8].

The overhead of the additional object propagation is analyzed in section 6 where we compare OMW with a traditional home-based, multiple-writer, lazy invalidate Release Consistency protocol (LI) [5, 20]. The conclusion of this comparison is that the invalidate protocol outperforms OMW only for configurations with large number of processors and applications with tightly connected threads, i.e. applications where the size of the shared objects is a significant percent of the overall heap space.

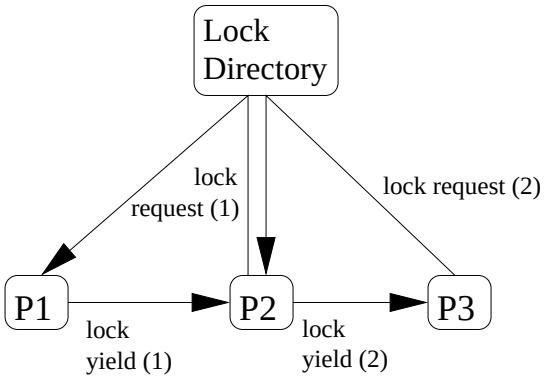

Figure 7: Processors P2 and P3 request the same synchronization variable from P1

4.2 Comparison with Other Memory Consistency Protocols

In order to emphasize the novelty of the OMW protocol, we compare it against the protocols used in three well-known DSM systems: Munin [10], Midway [9], and Orca [8].

Comparison with Munin

Munin relies on the programmer to identify shared and unshared variables. OMW is capable to automatically identify shared objects based on the Java thread structure. The consequences are two-fold: first, OMW is able to execute Java applications without any changes to the source code, and second, the consistency overhead is reduced because OMW automatically maintains objects in the unshared state as much as possible.

OMW requires less messages than Munin's consistency protocol. At lock release time, Munin sends object updates to all processors in the object copyset. OMW sends updates only to the processor acquiring the lock. Starting from the assumption that programs are data-race free, all processors eventually receive the updated information during synchronization operations [20].

When Munin accesses an invalid object copy, a "follow the probable owner" strategy is used to locate the actual object owner. The same strategy, which takes an indefinite number of messages, is used to locate synchronization variables. OMW does not need to locate valid object copies. Due to its update nature, nodes have valid copies for all accessible shared objects. Synchronization variables are located using a distributed lock directory, strategy that takes a constant number of 3 messages. For this reason, OMW is a better candidate for a real-time system.

Munin uses a timer-based algorithm to mark unused pages. OMW is capable to use much more accurate information from a garbage collector to mark unneeded objects.

Comparison with Midway

Midway uses the entry consistency model, while OMW uses the lazy release consistency model. The main difference between entry consistency and lazy release consistency is that entry consistency associates every shared object with a distinct synchronization variable. When a synchronization variable is acquired *only* the corresponding object is updated. Similarly, OMW associates every Java object with a synchronization variable, but when a synchronization variable is acquired acquired *all* modified objects are updated. This is a constraint imposed by the Java Virtual Machine specification.

Similarly with Munin, Midway introduces new keywords in the C language to mark shared variables. OMW does not require any changes to the Java language. Shared objects are identified at runtime with the same benefits mentioned in the Munin comparison: ability to run unmodified Java programs and consistency overhead minimization.

Midway uses the same mechanism as Munin to locate the owner of a synchronization variable. Again, OMW's strategy requires a constant number of messages to locate synchronization variables, thus it has better worse-case behavior and it is a better candidate for real-time protocol.

OMW and Midway share similar strategies to reduce the overhead of the update messages. Both rely on the "happens before" relation to reduce the number of objects included in update messages, but with different implementations. Midway uses a global logical clock, while OMW uses an interval matrix, where each matrix line corresponds to one processor's view of the system.

Comparison with Orca

Orca implements sequential consistency with a reliable broadcast mechanism that replicates all object operations. OMW implements lazy release consistency, which is known to achieve better performance. OMW sends update information only during synchronization variable acquire operations. The size of the consistency messages is reduced by considering only objects known to be present but out-of-date on the destination node.

Orca considers all objects shared, but is capable of changing object states between replicated and single-copy based on the object access profiles. This runtime adaptation is the closest behavior we found to OMW's runtime classification of objects in shared or unshared. The difference is that Orca uses a heuristic based on the ratio of read to write accesses, while OMW uses the thread structure to identify newly shared objects.

Both Orca and OMW use an update strategy instead of the invalidate strategy typically used in page-based systems. Both systems decided on the update strategy based on the small average object size, which translates into small overhead for the consistency messages and simpler consistency barriers.

5 Correctness of the OMW Protocol

This section demonstrates the compliance of the OMW protocol introduced in section 3 with the Java Virtual Machine Specification. In order to prove the equivalence of Release Consistency, the consistency model implemented in DISK, with JVMS, the consistency model defined in the JVMS, we provide non-operational definitions for both models, and show that these new definitions are equivalent for data-race-free programs.

The following conventions are used throughout this section:

The term *processor* identifies a Java thread. Processor operations are limited to the following four: READ, WRITE, ACQUIRE and RELEASE. The READ operation corresponds to the JVMS `load`[2] abstract operation, and the WRITE operation corresponds to the JVMS `store` operation. Note that no assumptions are made about the underlying implementation of the memory system (we do not use the `read` and `write` global memory access JVMS operations). Similarly, the ACQUIRE and RELEASE synchronization operations correspond to the `lock` and `unlock` JVMS abstract operations.

The memory access unit is a *variable*. A Java variable is defined in the JVMS as a storage location having either a primitive type or a reference (pointer) type. Java *objects* are collections of variables. From a memory perspective, an object is similar to a page, the only major difference being that Java objects have variable sizes while pages have fixed sizes.

A *local history* of processor p, denoted H_p, is a sequence of operations executed in the program order by processor p. If $o1$ and $o2$ are two operations in H_p and $o1$ appears before $o2$, then $o1$ *precedes* $o2$ *in program order*. The notation used is $o1 \xrightarrow{po} o2$. A *history* H is a collection of all local histories. $H|x$ is the subset of history H containing only operations on variable x.

If H is a history, then S is a *serialization* of H if S is a linear program sequence containing exactly the operations in H. The notation $o1 \xrightarrow{S} o2$ indicates that $o1$ appears before $o2$ in S. S is a *legal serialization*, if each READ operation returns the value written by the most recent WRITE to the same location.

The notation $o1 \xrightarrow{E} o2$ indicates that operation $o1$ appears before operation $o2$ in the *wall-clock execution*. This relation is used to describe the Linearizability required for synchronization operations in JVMC.

5.1 JVMC Non-Operational Definition

We define JVMC based on an abstract relation named $\xrightarrow{jvmc}$. Intuitively, the $\xrightarrow{jvmc}$ relation describes the JVMC bytecode instruction dependencies. This is formally specified in Definition 1.

Definition 1. For any two operations $o1$ and $o2$ in a history H, $o1 \xrightarrow{jvmc} o2$ if one of the following conditions holds:

(jvmc1) $o1$ and $o2$ are data-access operations (READ or WRITE) on the same variable, and $o1 \xrightarrow{po} o2$.

(jvmc2) $o1$ is a READ operation, $o2$ is a WRITE operation, and $o1 \xrightarrow{po} o2$.

(jvmc3) $o1$ and $o2$ are two synchronization operations (ACQUIRE or RELEASE) and $o1 \xrightarrow{po} o2$.

(jvmc4) $o1$ and $o2$ are two synchronization operations and $o1 \xrightarrow{E} o2$.

[2] We use typewriter font for notations from the JVMS.

(jvmc5) $o1$ is a data-access operation, $o2$ is a RELEASE operation, and $o1 \xrightarrow{po} o2$.

(jvmc6) $o1$ is an ACQUIRE operation, $o2$ is a data-access operation, and $o1 \xrightarrow{po} o2$.

(jvmc7) there is an operation o' in H, such that $o1 \xrightarrow{jvmc} o'$ and $o' \xrightarrow{jvmc} o2$.

Definition 2. A history H is JVMC if there is a legal serialization S of H, such that for any two operations in H $o1$ and $o2$, if $o1 \xrightarrow{jvmc} o2$ then $o1 \xrightarrow{S} o2$. A memory is JVMC if all acceptable histories under this memory are JVMC.

Informally, conditions (jvmc1) and (jvmc2) describe the data-access constraints from the JVMS (see also Appendix A). Conditions (jvmc3) through (jvmc6) describe the JVMS synchronization constraints, and condition (jvmc7) provides the transitive closure for the $\xrightarrow{jvmc}$ relation. Lemma 1 formalizes these observations.

Lemma 1. JVMC, as defined in Definition 2, is equivalent with the memory consistency model of the Java Virtual Machine defined in the JVMS.

The proof of Lemma 1 is presented in Appendix B.

For the sake of simplicity, Definition 1 ignores Java `volatile` variables. The JVMS requires Linearizability for volatile variable operations [14], so they can be added to Definition 1 with two conditions similar to (jvm3) and (jvm4).

5.2 RC Non-Operational Definition

To make the comparison between JVMC and RC possible we provide a non-operational RC definition using the same formalizations as section 5.1.

Gharachorloo makes two implicit assumptions before defining Release Consistency (condition 3.1 in [13]):

1. The memory is coherent. This is formally expressed in Definition 3.
2. Uniprocessor data dependencies are respected on each processor. This is formally stated in Definition 5. Uniprocessor data dependencies are formally expressed by relation $o1 \xrightarrow{D} o2$ introduced in Definition 4.

Definition 3. A memory is *coherent* if for every variable x, there is a legal serialization S_x of $H|x$ such that, if $o1$ and $o2$ are two operations in $H|x$ and if $o1 \xrightarrow{po} o2$, then $o1 \xrightarrow{S_x} o2$ [4].

Definition 4. For any $o1$ and $o2$ two operations in a history H, $o1 \xrightarrow{D} o2$ if one of the following holds:

DATA $o1$ is a write, $o2$ is a read on the same variable, and $o1 \xrightarrow{po} o2$.

ANTI $o1$ is a read, $o2$ is a write on the same variable, and $o1 \xrightarrow{po} o2$.

OUTPUT $o1$ and $o2$ are write operations on the same variable, and $o1 \xrightarrow{po} o2$.

INSTR $o1$ is a read, $o2$ is a write, $o1 \xrightarrow{po} o2$ and the value stored by $o2$ is dependent on the value loaded by $o1$.

The first three conditions in Definition 4 are an immediate translation of the classical data, anti, and output dependencies between atomic high level instructions. INSTR dependency is needed here because we use lower level memory access operations. The INSTR dependency specifies that $o1$ is on the right-hand side and $o2$ is on the left-hand side of the same high-level instruction. For example, the addition $a = b + c$ implies the following INSTR dependencies: READ c $\rightarrow$ WRITE a, and READ b $\rightarrow$ WRITE a. Note that at this level of formal specification, control dependencies are ignored. In Java, bytecode control instructions do not act on memory locations (variables), hence they are not visible from the memory perspective. Of course, control instructions have an important role in generating processor histories. We assume that all histories are computed with respect to control dependencies.

Definition 5. A history H_p respects the uniprocessor dependencies if there is a serialization S_p of H_p such that, if $o1$ and $o2$ are two operations in H_p and $o1 \xrightarrow{D} o2$, then $o1 \xrightarrow{S_p} o2$.

Because JVMC defines Linearizability for synchronization operations, we need to compare JVMC with Release Consistency with Linearizability for synchronization operations (RC_l). Making the observation that the operational statements *"performed before"* and *"previous"* used in [13] are translated as *"appearing before in a legal serialization"* and *"previous in the program order"* in a non-operational interpretation, we define the $\xrightarrow{rcl}$ relation and RC_l in Definitions 6 and 7.

Definition 6. For any two operations $o1$ and $o2$ in a history H, $o1 \xrightarrow{rcl} o2$ if one of the following conditions holds:

(rcl1) $o1$ and $o2$ are two synchronization operations and $o1 \xrightarrow{po} o2$.

(rcl2) $o1$ and $o2$ are two synchronization operations and $o1 \xrightarrow{E} o2$.

(rcl3) $o1$ is an ACQUIRE operation, $o2$ is a data-access operation, and $o1 \xrightarrow{po} o2$.

(rcl4) $o1$ is a data-access operation, $o2$ is a RELEASE operation, and $o1 \xrightarrow{po} o2$.

(rcl5) there is an operation o' in H, such that $o1 \xrightarrow{rcl} o'$ and $o' \xrightarrow{rcl} o2$.

Definition 7. A history H is RC_l if there is a legal serialization S of H, such that for any two operations $o1$ and $o2$ in H, if $o1 \xrightarrow{rcl} o2$ then $o1 \xrightarrow{S} o2$. Additionally, history H must be coherent, and uniprocessor dependencies must be respected in all processor histories H_p part of H. A memory is RC_l if all acceptable histories under this memory are RC_l.

5.3 Comparison between the JVMC and RC Consistency Models

Lemma 2. JVMC is strictly stronger than RC_l.

Proof. It is obvious that the conditions from Definition 6 are similar with conditions (jvmc3) through (jvmc7) from Definition 1. However, JVMC is stronger than RC_l for the following reasons:

First, JVMC is stronger than Coherence. JVMC requires a legal serialization of the *global* history H with respect to condition (jvmc1), while Coherence requires legal serializations of the same condition only for *subsets* $(H|x)$ of the global history H.

Second, JVMC is stronger than Definition 5. Definition 5 requires that legal serializations of the $\xrightarrow{D}$ relation exist for *local* histories, while JVMC requires a legal serialization of the *global* history with respect to conditions (jvmc1) and (jvmc2). Furthermore, condition (jvmc1) is more restrictive than the DATA, ANTI, and OUTPUT dependencies because it applies also to two READ operations on the same variable. Condition (jvmc2) is stronger than the INSTR dependency because it applies to all (READ, WRITE) pairs if the READ operation precedes the WRITE operation in program order, while the INSTR dependency imposes an ordering only if the READ and WRITE operations are part of the same high-level instruction.

□

Lemma 3. For data-race-free programs (or properly-labeled programs in [13]), JVMC is equivalent with RC_l.

Proof. The proof of this lemma follows from the observation that RC_{sc} is equivalent with Sequential Consistency (SC) for data-race-free programs [13]. RC_l is stronger than RC_{sc}, hence RC_l is stronger than SC. SC is obviously stronger than conditions (jvmc1) and (jvmc2) in Definition 1, the only conditions that make JVMC stronger than RC_l. Hence RC_l is stronger than JVMC. We have shown in Lemma 2 that JVMC is stronger than RC_l, hence, for data-race-free programs, RC_l is equivalent with JVMC.

□

Lemma 3 shows that under the assumption that programs are data-race free, any consistency protocol implementing RC_l is correct according to the JVMS. This statement proves the JVMS compliance of the OMW protocol, an update-based, lazy Release Consistency protocol. However, for asynchronous algorithms, OMW is weaker than JVMC. An informal proof of this statement is that multiple writer protocols do not offer Coherence for concurrent data accesses, and JVMC is stronger than Coherence (see Lemma 2). Formally, for a history H of an asynchronous algorithm, OMW (or any other Release Consistency protocol) does not guarantee a legal serialization with respect

to conditions (jvmc1) and (jvmc2) from Definition 1. Conditions (jvmc1) and (jvmc2) are too strict for a distributed implementation because they require inter-node communication outside synchronization intervals. In this paper we do not consider asynchronous algorithms, thus we do not focus on modifying the protocol to support asynchronous algorithms.

The importance of the theoretical analysis performed in this section is two-fold:

1. Lemma 3 proves that for "regular" (i.e. data-race free) programs OMW, or for that matter any release consistent protocol, is JVMC compliant. This result provides an important theoretical foundation for this paper, considering that Java is a relatively new language with a unique specification.
2. The uniqueness of the Java language is emphasized by Lemma 2, which shows that for asynchronous programs the constraints imposed by the JVMC are stronger than those required by release consistency. This observation proves that there are circumstances under which release consistency protocols are incorrect according to the JVMC. While we do not handle these special situations, a JVMC-compliant distributed JVM should implement special consistency protocols to address these issues. To our knowledge, no distributed JVM project implements, or even mentions, this problem.

6 Performance Analysis

6.1 Performance Model

In this section we analyze the performance of the OMW protocol introduced in section 3. The OMW executions are compared with DISK executions using a traditional home-based, multiple-writer, lazy invalidate Release Consistency protocol (LI).

The performance of a software-implemented DSM system is modeled using the following variables:

- N - the number of threads,
- P - the number of processors,
- t - the thread granularity,
- t_s - the synchronization overhead,
- t_c - the distributed thread creation overhead,
- t_o - the thread execution overhead,

and the following constants:

- η - the interconnection network overhead, defined as the number of clock cycles needed to transmit one byte of data, and
- β - the software consistency barrier overhead, defined as the number of clock cycles needed to verify the validity of an object.

The thread granularity is an application dependent parameter, a function of the problem size n and the number of threads created by the application N. The synchronization overhead t_s can be expanded as $t_s = (c_s + d_s)\eta$, where c_s is the overhead for control synchronization messages, and d_s is the overhead for data appended to the synchronization messages. Similarly, t_c can be expressed as $t_c = (c_c + d_c)\eta$, where c_c is the overhead for the control messages required for thread creation, and d_c is the overhead for the data messages involved in the thread creation process. The thread execution overhead t_o is expanded as $t_o = t_f + t_b$, where t_f is the access fault overhead and t_b is the software consistency barrier overhead. Both t_f and t_b are application depended. However, t_b can be further expanded as $t_b = (r + w)\beta$, where r is the number of consistency barriers performed before read accesses, and w is the number of consistency barriers performed before write accesses.

Using these notations the speedup of a DSM system is defined as:

$$S = \frac{T_1}{T_{N,P}}, \tag{1}$$

where T_1 is the sequential execution time defined as $T_1 = N\,t$, and $T_{N,P}$ is the distributed execution time on P processors with N threads, defined by the following formula:

$$T_{N,P} = t_s + t_c + (t + t_o)\lceil \tfrac{N}{P} \rceil, \text{ or} \tag{2}$$

$$T_{N,P} = (c_s + d_s)\eta + (c_c + d_c)\eta + [t + t_f + (r + w)\beta]\lceil \tfrac{N}{P} \rceil. \tag{3}$$

Hence the speedup of the DSM system is:

$$S = \frac{t\,N}{(c_s+d_s)\eta+(c_c+d_c)\eta+[t+t_f+(r+w)\beta]\lceil \frac{N}{P} \rceil} \tag{4}$$

The DISK platform is targeted for computational intensive natural language applications. Our previous work indicated that the exploitation of intra-task parallelism is beneficial for an interactive question/answering system [25]. Nevertheless, the distributed design introduced in [25] required source code changes of the sequential question/answering system. DISK can provide the same benefits without requiring any source code modifications. Because the question/answering system is not yet ported to Java, we evaluate the performance of the OMW protocol on two applications: a matrix multiplication program and the traveling salesman problem. These applications were selected because they are similar to the question/answering application in two regards: first, they are computational intensive, and second, intra-task parallelism is exploited using a master/worker strategy. An analysis of the system behavior is performed for the OMW protocol and the LI protocol on these two applications. In section 6.4, this theoretical model is compared with the results measured on DISK.

6.2 The Parallel Matrix Multiplication Benchmark (MM)

Both benchmarks use the a master/worker strategy. In the MM benchmark, each worker thread computes a distinct band from the output matrix. The main thread waits until all workers are finished to read the output matrix. The thread granularity t is $t = \frac{n^3}{N}$, where n is the matrix dimension.

6.2.1 Speedup for the OMW protocol

The application and protocol dependent variables in the speedup expression have the following values for the OMW protocol:

- $c_s = N$; since N synchronization control messages are required to synchronize the main thread with all workers.
- $d_s = n^2$; each worker thread appends its corresponding band from the output matrix to the synchronization message sent to the main thread.
- $c_c = N$; since N control messages are needed to create the N worker threads.
- $d_c = 2n^2P$; during thread creation time, the two input matrices are distributed to all P processors.
- $t_f = 0$; as discussed in section 4, the OMW protocol avoids access faults completely.
- $r = 0$; the OMW protocol does not require consistency barriers before read accesses.
- $w = \frac{n^2}{N}$; each worker thread performs a write access to a shared object only when writing to its corresponding band in the output matrix.

Using these expressions the speedup becomes:

$$S = \frac{\frac{n^3}{N}N}{(N+n^2)\eta+(N+2n^2P)\eta+[\frac{n^3}{N}+0+(0+\frac{n^2}{N})\beta]\lceil \frac{N}{P} \rceil}. \tag{5}$$

After simplification and ignoring the least significant terms, the speedup expression is:

$$S \cong \frac{n}{\eta+2P\eta+\frac{n+\beta}{P}} > 1. \quad (6)$$

The analysis of this inequality in P indicates that the following two conditions have to be true to have speedup larger than 1 for the OMW protocol:

$$P > \frac{n-\eta-\sqrt{n^2+\eta^2-10n\eta-8\eta\beta}}{4\eta}, \text{ and} \quad (7)$$

$$P < \frac{n-\eta+\sqrt{n^2+\eta^2-10n\eta-8\eta\beta}}{4\eta}. \quad (8)$$

6.2.2 Speedup for the LI protocol

For the LI protocol, the application/protocol dependent variables have the following values:

- $c_s = N$; since N synchronization control messages are required to synchronize the main thread with all workers.
- $d_s = n^2$; in the LI protocol no data is appended to synchronization messages, but the synchronization messages received by the main thread are immediately followed by access faults on the output matrix, access faults that will retrieve the computed matrix from the worker threads.
- $c_c = N$; since N control messages are needed to create the N worker threads.
- $d_c = 0$; no data is propagated during thread creation.
- $t_f = 2n^2\frac{P}{N}$; the access fault overhead to retrieve the two input matrices on each processor is $2n^2$. This overhead is equally divided among all $\frac{N}{P}$ threads running on each processor.
- $r = \frac{n^3}{N}$; the LI protocol requires consistency barriers before read accesses to shared objects. For the MM benchmark this translates to read accesses to the two input matrices.
- $w = \frac{n^2}{N}$.

The MM speedup for the LI protocol is:

$$S = \frac{\frac{n^3}{N}N}{(N+n^2)\eta+(N+0)\eta+[\frac{n^3}{N}+2n^2\frac{P}{N}+(\frac{n^3}{N}+\frac{n^2}{N})\beta]\lceil\frac{N}{P}\rceil}. \quad (9)$$

After simplification, the speedup is:

$$S \cong \frac{n}{3\eta+\frac{n(1+\beta)}{P}} > 1 \quad (10)$$

Solving this inequality indicates that the following condition has to be true to have a speedup larger than one for the LI protocol:

$$P > \frac{n(\beta+1)}{n-3\eta}. \quad (11)$$

6.3 The Traveling Salesman Problem Benchmark (TSP)

This program exhaustively searches for the shortest path that contains all nodes in a graph, starting at any node. The algorithm works as follows: the main thread builds a shared stack of partial paths and then creates a number of worker threads. Each worker repeatedly pops a path from the stack and solves it storing the local best path found. When the stack empties, the worker threads compare their local best paths and the shortest path is reported to the main thread. To simplify our analysis, we use a fully connected graph of n vertices, and the stack of partial paths is initialized with n^2 elements. The thread granularity is $t = \frac{n^n}{N}$.

6.3.1 Speedup for the OMW protocol

The following expressions describe the OMW protocol:

- $c_s = n^2 + N$; all n^2 accesses to the stack of partial paths are synchronized. The synchronization between the main thread and the worker threads requires N control messages.
- $d_s = Nn$; at the final synchronization between each worker thread and the main thread, the local best path found is appended to the control messages.
- $c_c = N$; N control messages are needed to create the N worker threads.
- $d_c = 2n^2P$; during thread creation time, the graph and the shared stack are distributed to all P processors.
- $t_f = 0$; as discussed in section 4, the OMW protocol avoids access faults completely.
- $r = 0$; the OMW protocol does not require consistency barriers before read accesses.
- $w = n$; each worker thread performs a write access to a shared object only when storing the local best path in the main thread.

The TSP speedup for the OMW protocol is:

$$S = \frac{n^n}{(n^2+N+Nn)\eta+(N+2n^2P)\eta+(\frac{n^n}{N}+0+n\beta)\lceil\frac{N}{P}\rceil}. \tag{12}$$

The exponential terms are obviously dominant, hence the speedup can be simplified as:

$$S \cong P > 1 \tag{13}$$

6.3.2 Speedup for the LI protocol

The following expressions describe the LI protocol:

- $c_s = n^2 + N$; all n^2 accesses to the stack of partial paths are synchronized. The synchronization between the main thread and the worker threads requires N control messages.
- $d_s = Nn$; the main thread access faults following the final synchronization between worker threads and the main thread retrieve all local best paths.
- $c_c = N$; N control messages are needed to create the N worker threads.
- $d_c = 0$; no data is propagated during thread creation.
- $t_f = 2n^2\frac{P}{N}$; the access fault overhead to retrieve the shared stack and the input graph is divided among all threads running on the same processor.
- $r = \frac{n^n}{N}$; the LI protocol inserts consistency barriers before all read accesses to graph elements.
- $w = n$.

The TSP speedup for the LI protocol is:

$$S = \frac{n^n}{(n^2+N+Nn)\eta+(N+0)\eta+(\frac{n^n}{N}+2n^2\frac{P}{N}+(\frac{n^n}{N}+n)\beta)\lceil\frac{N}{P}\rceil}. \tag{14}$$

Ignoring the non-exponential terms the speedup becomes:

$$S = \frac{P}{1+\beta} > 1, \tag{15}$$

which means that the condition

$$P > 1 + \beta \tag{16}$$

has to be true to have a speedup larger than one for the LI protocol.

6.4 Measured Results

The tests presented in this section were performed on sixteen 450 MHz Pentium III computers running Linux kernel 2.2.12, connected by a 100 Mbps Ethernet network. For this system $\eta = 36$. The software consistency barrier is implemented as a jump to the consistency routine, a load (to load the object status word), a comparison, and a return, an overhead of roughly 10 clock cycles, hence $\beta = 10$.

The MM benchmark was used with 640 x 640 matrices and 64 worker threads, hence $n = 640$ and $N = 64$. For this values conditions (7) and (8) from section 6.2.1 become:

$$P > 1.33, \text{ and} \tag{7'}$$
$$P < 7.11. \tag{8'}$$

This conditions indicate that MM will show speedup for the OMW protocol for up to 7 processors. For the LI protocol condition (16) becomes:

$$P > 13.23, \tag{16'}$$

which indicates that only if more than 13 processors are used, MM will show any speedup for the LI protocol.

The measured execution times for MM are presented in Figure 8. Figure 8 compares the execution times for the

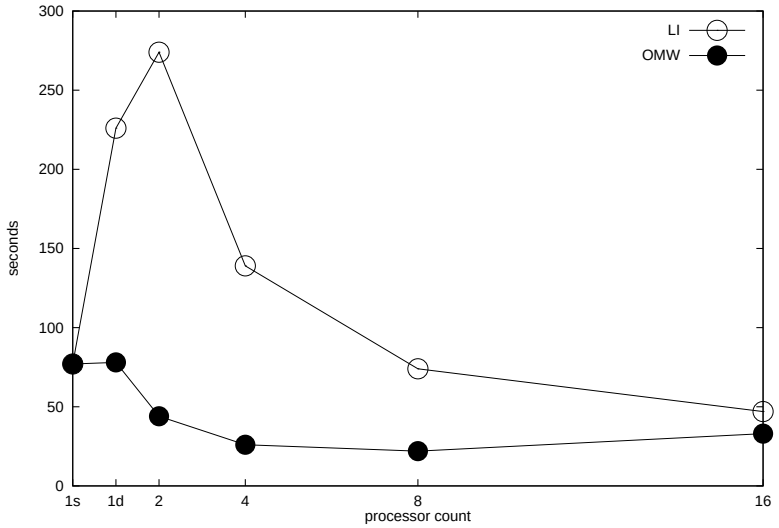

Figure 8: Measured execution times for MM

OMW and LI protocols with sequential executions denoted "1s". One-processor DISK executions are denoted with "1d". Due to the single-node topology, the difference between the "1s" and "1d" execution times is a good indication of DISK's software consistency barrier overhead. The measured execution times are very close to the results obtained from our performance model. Indeed, OMW starts losing performance for more than 8 processors and LI shows some speedup over the sequential execution only for 16 nodes. OMW performs slightly better than the model due to two improvements not covered by the performance model:

1. Even if the OMW protocol always sends more data than the LI protocol, this data is always packed into fewer messages than the LI protocol. For example, OMW transmits almost three times more data than LI for the MM application, but this data is packed into more than 50 times fewer messages (see Table 1) . This is caused by the fact that OMW appends all consistency information to synchronization messages, while LI retrieves this information only in response to an access fault. The strategy used by the OMW protocol is a better match for the TCP/IP protocol, a protocol optimized for batch data transfer.

	LI	OMW
Transmitted data (MB)	30	63
Transmitted message count	23827	439

Table 1: Measured message count and size for the MM benchmark

2. Due to the large message sizes, OMW messages are good candidates for compression. For example, the largest message in the MM application has 4.8 MB. Compressing this message with the Lempel-Ziv coding method yields an output of 45 KB, a buffer more than 100 times smaller than the original. We currently use only a very simple form of compression which detects similar array elements.

For the TSP benchmark we used a 10-node fully connected graph, $n = 10$, and 64 worker threads, $N = 64$. For $\beta = 10$, the LI speedup condition from section 6.3.2 becomes $P > 11$. The measured execution times are presented in Figure 9. Figure 9 indicates that OMW show speedup starting with 2 processors, while LI shows speedup over

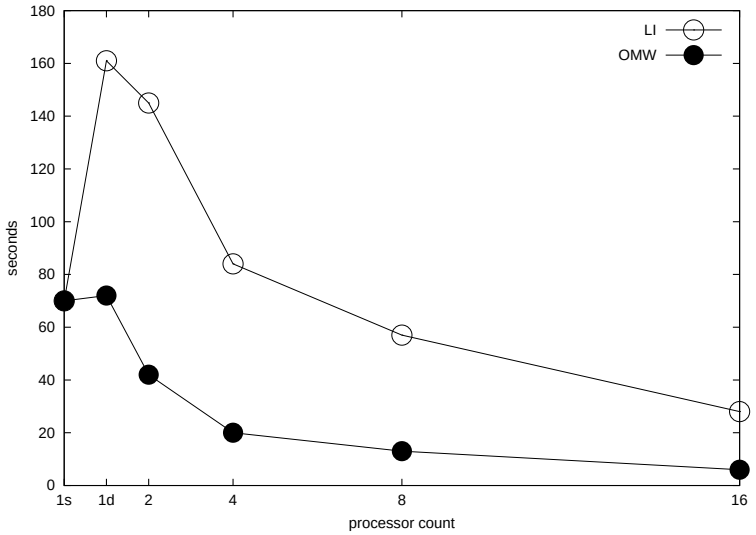

Figure 9: Measured execution times for TSP

the sequential execution only over 8 processors.

The conclusion of this analysis is that the OMW protocol eliminates access faults and reduces the software barrier overhead with the expense of a higher overhead for thread creation and synchronization messages. For applications with loosely-connected threads (i.e. applications where the size of the shared data is significantly smaller than the heap space) OMW always outperforms LI. TSP and our target application, the question/answering system, fall in this category. For problems where the size of the shared data is significant (i.e. MM), OMW outperforms LI only for a relatively small number of processors. The observed results indicate that for our target applications, OMW is a more atractive solution.

7 Conclusions

This paper describes DISK, a distributed Java Virtual Machine for networks of heterogeneous workstations. DISK is entirely implemented in user space with no references to the virtual memory management unit, which makes it portable to various classes of architectures. The novelty of the system is the object-based, multiple-writer memory consistency protocol (OMW). The OMW protocol minimizes its overhead by automatically classifying objects as shared or unshared. Because unshared objects have no consistency overhead, the protocol's strategy to maintain objects unshared as much as possible leads to significant performance improvement. To our knowledge, OMW is the only consistency protocol that automatically detects when an object becomes shared and uses this information to

reduce its overhead. The OMW protocol is almost completely decentralized. Besides the lock directories required to provide Linearizability for synchronization variables, the protocol maintains no centralized information. The protocol does not require consistency barriers for read accesses, and the consistency barriers for write accesses are simplified. This feature is very important for a DSM system with all-software implementation for consistency barriers.

The Java compliance of the protocol is demonstrated by comparing Release Consistency, the consistency model implemented by OMW, with the Java Virtual Machine memory consistency model (JVMC) as defined in the Java Virtual Machine Specification. The comparison of these two consistency models indicates that for data-race free programs, OMW is compliant with the JVM specification. The applicability of this observation is large because most programs written for weak consistency models are data-race-free. However, for asynchronous programs the constraints imposed by the JVMC are stronger than those required by release consistency. While we do not handle asynchronous programs yet, a JVMC-compliant distributed JVM should implement special consistency protocols to address such situations. To our knowledge, no distributed JVM project implements, or even mentions, this problem.

An analytical model for the distributed shared memory system was introduced. The model shows that for applications with loosely-connected threads (i.e. applications where the size of the shared data is significantly smaller than the heap space) OMW always outperforms invalidate protocols. The traveling salesman problem and our target application, the question/answering system, fall in this category. For problems where the size of the shared data is significant (i.e. matrix multiplication), OMW outperforms invalidate protocols only for a relatively small number of processors. The experimental results obtained on a network of 16 Pentium III computers validate the analytical model. Through these experiments it became clear that a single protocol is not always the best solution for an object oriented DSM. We envision a distributed Java Virtual Machine with adaptable protocols, depending on on the application type and size.

Appendix A JVMS Memory Coherence Constraints

Figure 10 shows a graphical representation of the abstract memory hierarchy defined in the JVMS. Throughout the paper we assume that the Java bytecode is correctly generated by a Java compiler, hence our focus is on the constraints imposed on the interaction between the lowest two layers of the memory hierarchy shown in Figure 10: the thread working memory and the global shared memory.

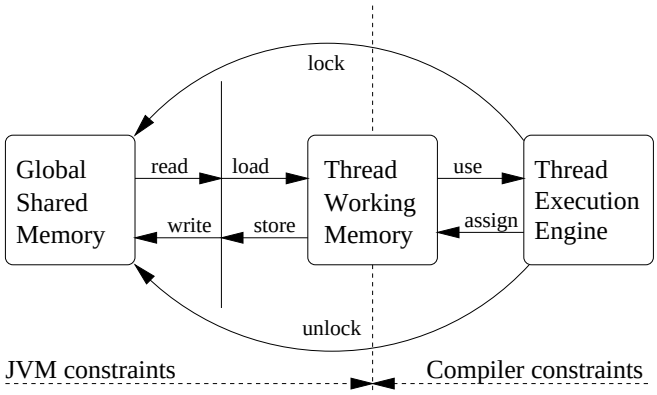

Figure 10: Java memory hierarchy

The JVMS defines the following constraints[3]:

General Constraints

(JVMS1) The actions performed by any one thread are totally ordered.

[3]We use T to denote a Java thread, V to denote a variable, and L to denote a lock.

(JVMS2) The actions performed by the main memory for any one variable are totally ordered.

(JVMS3) The actions performed by the main memory for any one lock are totally ordered.

(JVMS4) It is not permitted for an action to follow itself.

Data-Access Constraints

(JVMS5) For every `load` performed by T on its working copy of V, there should be a corresponding preceding `read` by the main memory on the master copy of V.

(JVMS6) For every `store` performed by T on its working copy of V, there should be a corresponding following `write` by the main memory.

(JVMS7) Let A be a `load` or `store` of V by T, and let P be the corresponding `read` or `write` by the main memory. Let B and Q be two other such operations by T and the main memory (on V), correspondingly. Now, if A precedes B, then P precedes Q.

Synchronization Constraints

(JVMS8) Each `lock` and `unlock` action is performed jointly by some thread and the main memory.

(JVMS9) `Locks` and `unlocks` of L are performed in some sequential order which is consistent with the program order of all threads.

(JVMS10) A `lock` of L by T may occur only if for every other thread the number of preceding `unlocks` equals the number of preceding `locks`.

(JVMS11) An `unlock` of L by T may occur only if the number of preceding `unlocks` of L by T is strictly less than the number of preceding `locks`.

(JVMS12) A `store` must intervene between an `assign` of V by T and a subsequent `unlock` of L by T, and the `write` which corresponds to the `store` must occur before the `unlock` by the main memory.

(JVMS13) Between a `lock` of L by T and a subsequent `use` or `store` of V by T, an `assign` or `load` of V must appear. If what appears is a `load` then its corresponding `read` should appear after the `lock` by the main memory.

Appendix B Proof of Lemma 1

The proof of this lemma is organized in two parts: the first part shows that the JVMS definition is no weaker than Definition 2. The second part shows that Definition 2 is no weaker than the JVMS definition, hence the two definitions are equivalent.

Part 1: *The JVMS definition is no weaker than Definition 2.*

Given a JVMS history H we show that there is a corresponding serialization conforming with Definition 2. We note with τ the timing of this history H. According to constraint (JVMS4), the application of the JVMS constraints on timing τ can not contain a cycle, hence the JVMS constraints build a directed acyclic graph (DAG) on H. Serialization S is obtained as follows:

1. Construct a serialization of all JVMS operations in history H (S_{all}) by applying a topological sort on the above DAG.
2. Consider S_{main} the subset of serialization S_{all} containing only the main-memory operations (`read`, `write`, `lock`, `unlock`).

3. Construct a serialization S_{thread} containing only thread operations (`load`, `store`, `lock`, `unlock`). Obtain serialization S_{thread} by replacing the main memory data-access operations from S_{main} with their corresponding thread operations (`read` with `load`, `write` with `store`). This correspondence is defined by constraints (JVMS5) and (JVMS6). The `lock` and `unlock` operations are defined as atomic (constraint (JVMS8)) hence no replacing is needed.

4. Obtain serialization S by replacing the operations in S_{thread} with the corresponding operations used in Definitions 1 and 2(`load` with READ, `store` with WRITE, `lock` with ACQUIRE, and `unlock` with RELEASE).

We claim that S is a legal serialization of relation $\xrightarrow{jvmc}$.

Serialization S is legal and conditions (jvmc1), (jvmc2) and (jvmc7) are satisfied in this serialization [14].

Synchronization operations are defined as atomic (constraint (JVMS8)). This observation coupled with constraints (JVMS3) and (JVMS9) implies Linearizability for synchronization operations [16]. Linearizability provides an ordering consistent with all local program orders and the wall-clock execution [16]. Hence conditions (jvmc3) and (jvmc4) are satisfied in serialization S.

Condition (jvmc5) is satisfied in serialization S: let $o1$ be a data-access operation (`load` or `store` in history H), and $o2$ a RELEASE operation (`unlock` in history H), such that $o1 \xrightarrow{po} o2$. If $o1$ is a `store` operation, then $o1 \xrightarrow{S} o2$ due to constraint (JVMS12). If $o1$ is a `load` operation, then $o1 \xrightarrow{S} o2$ due to constraints (JVMS5) and (JVMS1). Similarly, using constraints (JVMS1), (JVMS6), and (JVMS13), one can show that condition (jvmc6) is satisfied in serialization S, hence serialization S conforms with Definition 2. □

Part 2: *Definition 2 is no weaker than the JVMS definition.*

Given a history H of READ, WRITE, ACQUIRE, and RELEASE operations conforming with Definition 2, we show that there is a timing τ of JVMS operations complying with the JVMS constraints. If history H is JVMC, than there exists a legal serialization S such that if $o1 \xrightarrow{jvmc} o2$ then $o1 \xrightarrow{S} o2$. Based on serialization S timing τ is constructed as follows:

1. Construct serialization S_{main} by replacing all operations in S with the corresponding JVMS main-memory operations: READ with `read`, WRITE with `write`, ACQUIRE with `lock`, and RELEASE with `unlock`.

2. Let τ be initially S_{main}.

3. Iteratively extend τ as follows: for every local history in H, for each READ or WRITE operation add the corresponding `load` or `store` JVMS operation to timing τ. The operations of each local history are processed in program order. Use the following insertion rules:

   (a) A `load` is inserted immediately after the corresponding `read` already in τ, or after the last `load/store` operation inserted, whichever came last.

   (b) A `store` is inserted immediately after the last `load/store` operation inserted.

We claim that timing τ complies with the JVMS constraints.

Timing τ complies with constraints (JVMS1), (JVMS2), (JVMS4), (JVMS5), (JVMS6), and (JVMS7) [14].

Condition (jvmc3) provides an ordering for synchronization operations consistent with all local program orders, hence constraint (JVMS9) is satisfied. Condition (jvmc4) enforces this ordering to match the wall-clock execution, which in the JVMS is the order in which the synchronization operations are seen by the main memory. Hence, constraints (JVMS3) and (JVMS8) are satisfied.

Constraints (JVMS10) and (JVMS11) define *mutual exclusion* and *reentrancy* for Java locks. We do not consider these issues strictly related to the memory consistency model. We just assume that serialization S is constructed with respect for lock mutual exclusion and reentrancy, hence constraints (JVMS10) and (JVMS11) are satisfied in timing τ.

If $o1$ is an ACQUIRE operation, $o2$ is a READ operation in serialization S, and $o1 \xrightarrow{po} o2$, then $o1 \xrightarrow{S} o2$. In timing τ we replace $o1$ with a `lock` JVMS operation, $o2$ with a `read` operation, and we insert a `load` operation after the `read`. Hence, constraint (JVMS13) is satisfied.

If $o1$ is a WRITE operation, $o2$ is a RELEASE operation in serialization S, and $o1 \xrightarrow{po} o2$, then $o1 \xrightarrow{S} o2$. In timing τ we replace $o1$ with a `write` JVMS operation, $o2$ with an `unlock` operation, and we insert a `store` operation. The `store` operation is inserted before the `write` because constraint (JVMS6) is satisfied [14]. Hence, constraint (JVMS12) is also satisfied. □

References

[1] S. Adve and M. Hill. *Weak Ordering - A New Definition.* In *Proc. of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[2] S. Adve and M. Hill. *A Unified Formalization of Four Shared-Memory Models.* University of Wisconsin Computer Sciences Technical Report #1051, September 1991, revised September 1992.

[3] S. Ahuja, N. Carriero, and D. Gelernter. *Linda and Friends.* In IEEE Computer 19(8), August 1986.

[4] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. *The Power of Processor Consistency.* In *Proc. of the 5th ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[5] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. *TreadMarks: Shared Memory Computing on Networks of Workstations*, In IEEE Computer, Vol. 29, No. 2, February 1996.

[6] C. Amza, A.L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. *Adaptive Protocols for Software Distributed Shared Memory*, In Proceedings of IEEE, Special Issue on Distributed Shared Memory, March 1999.

[7] H. Attiya and J. Welch. *Sequential Consistency versus Linearizability.* In *ACM Trans. on Computer Systems*, May 1994.

[8] H. E. Bal, M. F. Kaashoek, A. Tanenbaum. *Orca: A Language for Parallel Programming of Distributed Systems.* In IEEE Transaction on Software Engineering, vol. 18, no. 3, March 1992.

[9] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. *The Midway Distributed Shared Memory System.* In Proceedings of the IEEE CompCon Conference, 1993

[10] J.B. Carter. *Design of the Munin Distributed Shared Memory System.* In *Journal of Parallel and Distributed Computing*, special issue on distributed shared memory, 1995.

[11] *Coherent Virtual Machine (CVM).* http://www.cs.umd.edu/projects/cvm/

[12] S. Dieckemann and U. Holzle. *A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks.* UCSB Technical Report TRC98-33, December 1998.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.* In *Proc. of the 17th Annual International Symposium on Computer Architecture*, 1990.

[14] A. Gontmacker and A. Schuster. *Java Consistency: Non-Operational Characterizations for Java Memory Behavior.* Technion Technical Report, CS-0922, August 1997.

[15] M. Herlihy. *The Aleph Toolkit: Support for Scalable Distributed Shared Objects.* In Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing 1999

[16] M. Herlihy and J. Wing. *Linearizability: A Correctness Condition for Concurrent Objects.* In *ACM Trans. on Programming Languages and Systems*, July 1990.

[17] *The Java Virtual Machine Specification.* http://java.sun.com

[18] *Jini(tm) Network Technology.* http://www.sun.com/jini

[19] *The Kaffe Project.* http://www.kaffe.org

[20] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency.* Ph.D. Thesis, Rice University, December 1994.

[21] K. Li and P. Hudak. *Memory Coherence in Shared Virtual Memory Systems.* In ACM Transactions on Computer Systems, 7(4), November 1989.

[22] *The Millipede Project.* http://www.cs.technion.ac.il/Labs/Millipede/

[23] N. Nagaratman and A. Srinivasan. *Remote Objects in Java.* In IASTED Networks '96, January 1996.

[24] M. Philippsen and M. Zenger. *JavaParty - Transparent Remote Objects in Java.* In Concurency: Practice and Experience, vol. 9, no. 11, November 1997.

[25] M. Surdeanu, D. Moldovan, and S. Harabagiu. *Performance Analysis of a Distributed Question/Answering System.* In Proceedings of the 15th International Parallel & Distributed Processing Symposium, April 2001.

[26] K. Thitikamol and P. Keleher. *Thread Migration and Communication Minimization in DSM Systems.* In Proceedings of the IEEE, March 1999.

[27] W. Yu and A. L. Cox. *Java/DSM: A Platform for Heterogenous Computing.* In ACM 1997 Workshop on Java for Science and Engineering Computation, June 1997.