Runtime Optimizations for a Java DSM Implementation

R. Veldema† R.F.H. Hofman† R.A.F. Bhoedjang* H.E. Bal†

†Department of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
{rveldema,rutger,bal}@cs.vu.nl

*Department of Computer Science
Cornell University
Ithaca, NY, USA
raoul@cs.cornell.edu

ABSTRACT

Jackal is a fine-grained distributed shared memory implementation of the Java programming language. Jackal implements Java's memory model and allows multithreaded Java programs to run unmodified on distributed-memory systems.

This paper focuses on Jackal's runtime system, which implements a multiple-writer, home-based consistency protocol. Protocol actions are triggered by software access checks that Jackal's compiler inserts before object and array references. We describe optimizations for Jackal's runtime system, which mainly consist of discovering opportunities to dispense with flushing of cached data. We give performance results for different runtime optimizations, and compare their impact with the impact of one compiler optimization. We find that our runtime optimizations are necessary for good Jackal performance, but only in conjunction with the Jackal compiler optimizations described in [24]. As a yardstick, we compare the performance of Java applications run on Jackal with the performance of equivalent applications that use a fast implementation of Java's Remote Method Invocation (RMI) instead of shared memory.

1. INTRODUCTION

Jackal is a compiler-supported, fine-grained distributed shared memory (DSM) system for Java. The system can run unmodified, multithreaded Java programs on a cluster of workstations. Together, Jackal's compiler and runtime system (RTS) hide the distributed nature of the cluster: Jackal programs use threads and shared variables instead of message-passing abstractions like Remote Method Invocation [19]. This paper focuses on the implementation of the RTS and its optimizations, which mainly consist of discovering opportunities to dispense with flushing of cached data to main memory.

Jackal resembles fine-grained DSM systems like Shasta [22] and Sirocco [13] in that it uses a small unit of coherence that is managed entirely by software. In Jackal, the unit of coherence is called a *region*. Each region contains either a complete Java object or a section of a Java array. In contrast with page-based DSMs, Jackal uses software access checks to determine if a region is present in local memory and up-to-date. If an access check detects that a region is absent or out-of-date, it invokes Jackal's runtime system which implements a multiple-writer cache coherence protocol that resolves read and write misses. A region is managed by its *home node*, which is the processor that created the associated object. Jackal does not use a single-writer protocol, because that would require the compiler to inform the runtime system when a read/write operation has finished; that would increase code size and protocol overhead, and pose complications for the compiler in (re)moving access checks.

Jackal conforms to the Java memory model, which allows caching of objects in (thread) local memory and logically requires complete flushing of local memory upon each entry and exit of a synchronized block. In our system, main memory equates to an object's home node, and local memory to the requesting machine's memory. Flushing regions and subsequently requesting them again may cause a large overhead under a naive implementation (especially using the class libraries which perform many unnecessary synchronizations [1]). To reduce this overhead, we investigate possibilities offered by the Java memory model to cache regions across a synchronization operation. This is possible for regions that are read-shared and regions that are accessed by a single machine.

Jackal uses an optimizing Java compiler to generate access checks. In the optimization passes of the compiler, access checks may be removed, lifted or combined. For example, array accesses may be combined and lifted from a loop that (partially) traverses the array, or accesses may be aggregated when the compiler determines that an object is used together with its referenced subobjects. The compiler opimizations are described in detail in [24].

The contributions of this paper are as follows:

- We describe various RTS optimizations to reduce the number of region flushes.
- We measure the impact of the RTS optimizations for several Java applications and compare them to the impact of compiler optimizations.

The paper is structured as follows. Section 2 treats Java's memory model. Section 3 describes Jackal and its implementation. Section 4 summarizes Jackal's compiler optimizations and describes our new RTS optimizations. Section 3 and an extended version of Subsection 4.1 appeared earlier in [24], but we repeat these introductory sections here to make this paper self-contained. Section 5 studies the impact of the RTS optimizations on Jackal's performance on a Myrinet-based cluster computer. Section 6 discusses related work. Finally, Section 7 concludes.

2. JAVA'S MEMORY MODEL

We briefly summarize Java's memory model; for a detailed description we refer to the language specification [10] and Pugh's critique of the memory model [21].

Java's memory model specifies that each thread has a *working memory*, which can be considered a thread-private cache. The entire program has a *main memory* which is used for communication between threads. The data modified by a thread is flushed to main memory upon encountering a synchronization point. (In this respect, the model resembles release consistency [8, 16].) Synchronization points in Java correspond to the entry and exit of synchronized blocks. These are implemented as calls that lock and unlock an object. A lock operation conceptually copies all of a thread's working memory to main memory and invalidates the working memory. For each storage location, the first access to that location after a lock operation will copy the storage location's value from main memory into working memory.

Both lock and unlock operations must flush a thread's working memory, but an implementation is allowed to flush earlier, even after every write operation. If a thread updates an object from outside a synchronized block, Java does not specify when other threads will see the update.

In contrast with entry consistency [3], Java's memory model does not couple locks to specific objects or fields. In particular, different fields of one object may be protected by different locks, so that those fields can be updated concurrently without introducing race conditions.

3. IMPLEMENTATION

Jackal consists of an optimizing Java compiler and a runtime system. The compiler translates Java sources directly into executable code rather than Java bytecode. (The Jackal runtime system, however, contains a dynamic bytecode compiler [19] to support dynamic class loading.) The compiler also generates software access checks and performs several optimizations to reduce the number and cost of these checks. The runtime system implements Jackal's multiple-writer cache-coherence protocol. The following sections describe the main components of the implementation. Optimizations are described separately in Section 4.

3.1 Regions

A region is Jackal's unit of coherence. A region is a contiguous chunk of virtual memory that contains one Java object or a contiguous section of a Java array. Jackal partitions arrays into fixed-size, 256-byte regions (to reduce false sharing inside large arrays).

Every region has a *region header* that contains a pointer to the start of the Java data stored in the region, a pointer to the region's twin (see Section 3.3), and DSM status information. Each object or array has a Java object header that contains a pointer to a virtual-function table and object status flags. To keep array data contiguous, regions and their headers are stored separately (see Fig. 1).

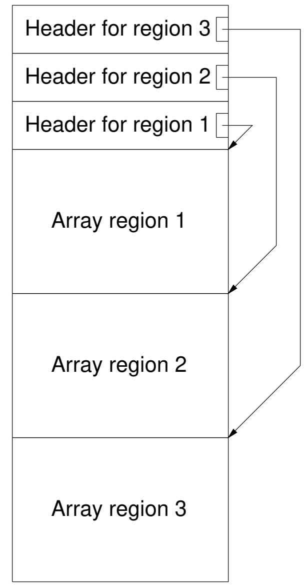

Figure 1: Array layout.

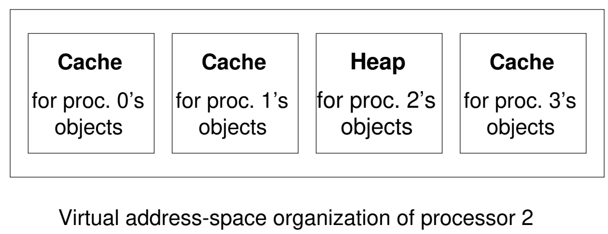

Figure 2: Address-space layout.

The processor that allocates a region is called the region's *home node*. The home node always provides storage for the region and plays an important role in Jackal's coherence protocol (see Section 3.3). Non-home nodes can cache the region and may discard their copy and its memory when they see fit (e.g., during garbage collection).

3.2 Address-Space Management

Jackal stores all regions in a single, shared virtual address space. Each region occupies the same virtual-address range on all processors that store a copy of the region. Regions are named and accessed through their virtual address; this scheme avoids translation of object pointers.

Fig. 2 shows a processor's address-space layout. The shared virtual address space is split into P equal parts, where P is the number of processors. Each processor owns one of these parts and creates objects and arrays in its own part. This way, each processor can allocate objects without synchronizing with other processors.

When a processor wishes to access a region created by another machine, it must (1) potentially allocate physical memory for the virtual memory pages in which the object is stored, and (2) retrieve an up-to-date copy of the region from its home node. Region retrieval is described in Section 3.3. Physical memory is allocated using the *mmap()* system call. Un-

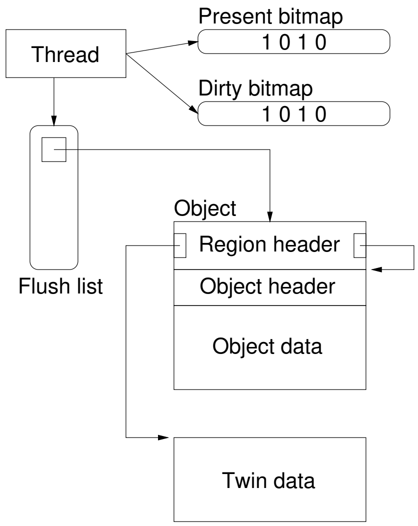

Figure 3: Runtime system data structures.

mapped pages are detected through MMU traps which result in an operating-system signal that is processed by Jackal's runtime system. If a processor runs out of free physical memory, it initiates a global garbage collection that frees both Java objects and physical memory pages.

3.3 Coherence Protocol and Access Checks

Jackal employs an invalidation-based, multiple-writer protocol that combines features of HLRC [26] and TreadMarks [15]. As in HLRC, modifications are flushed to a home node; as in TreadMarks, twinning and diffing is used to allow concurrent writes to shared data. Unlike TreadMarks, Jackal uses software access checks inserted before each object/array usage to detect non-local and stale data. The run-time data structures related to the coherence protocol are shown in Fig. 3.

The coherence protocol allows processors to cache a region created on another processor (i.e., the region's home node). All threads on one processor share one copy of a cached region. The home node and the caching processors all store this copy at the same virtual address.

Although all threads on a processor access the same copy of a given region, each thread maintains its own cache-state vector for that region. This is required because Jackal allows multiple threads per processor and the JMM is defined with respect to threads, not processors. For this purpose, each thread maintains a *present* and a *dirty* bitmap, each of which contains one bit per 64 bytes of heap. Objects are 64-byte aligned to map a single object to a single bit in the bitmap. To reduce memory usage, pages for these bitmaps are allocated lazily.

The present bit in thread T's bitmap indicates whether thread T retrieved an up-to-date copy of region R from R's home node. A dirty bit in thread T's bitmap indicates whether thread T wrote to region R since it fetched R from its home node. If the present bit is not set, the access-check code invokes the runtime system to retrieve an up-to-date copy from the region's home node. When the copy arrives, the runtime system stores the region at its virtual address and sets the accessing thread's present bit for this region. This cached region copy is called a processor's *working copy* of a region. The runtime system stores a pointer to the region in the accessing thread's *flush list*. In the case of a write miss, the runtime system also sets the region's dirty bit and creates a *twin*, a copy of the region just retrieved, unless such a twin already exists.

A cached region copy remains valid for a particular thread until that thread reaches a synchronization point. At a synchronization point, the thread empties its flush list. All regions on the thread's flush list are invalidated for that thread by clearing their present bits for that thread. Regions that have their dirty bits set are written back to their home nodes in the form of *diffs*, and the dirty bits are cleared. A diff contains the difference between a region's working copy and its twin. The home node uses the incoming diff to update its own copy. To speed up flushing, region flushes to the same home node are combined into a single message.

When two threads on a single processor miss on the same region, *both* threads must request a fresh copy from the home node, because region state is maintained per thread, not per processor. The data accessed by the second thread may have been modified on another processor *after* the first thread requested its copy. (As explained in Section 2, this is not a race condition if these parts are protected by different locks.) To see the modification, the second thread must fetch an up-to-date copy from the home node. The second copy is stored at the same virtual address; the newly arrived data is merged into the twin and into the working copy.

4. OPTIMIZATIONS

To improve performance, Jackal removes superfluous access checks, prefetches regions, flushes regions lazily, and employs computation migration to improve locality. The compiler optimizations are described in detail in [24] and are briefly summarized here. The RTS optimizations are described in detail below.

4.1 Compiler Optimizations

Jackal's front-end inserts access checks before all heap accesses. Since these access checks add considerable runtime overhead, the backend's optimization passes try to remove as many checks as possible.

The compiler performs interprocedural program analysis to discover opportunities to lift access checks. The front-end of Jackal's compiler can determine sets of virtual-function call targets and maintain label lists for switch statements. This information is passed on to the compiler back-end which uses it to remove access checks. An access check for address a at program point p can be removed if a has already been checked on *all* paths that reach p, but only if no path contains a synchronization statement.

Access checks to array elements that are accessed in a loop may be lifted into one aggregate array check before the loop.

The compiler also performs heap analysis [9] to discover when subobjects referenced by an object are *always* accessed through that outer object. If this is the case, an aggregate access check is generated to fault in the outer object and all its referenced subobjects. This may greatly increase granularity, and may save a number of network round-trips. The applicability of this optimization strongly depends on interprocedural analysis. Escape analysis [6] in combination with heap analysis is used to remove checks on objects that remain local to the creating thread.

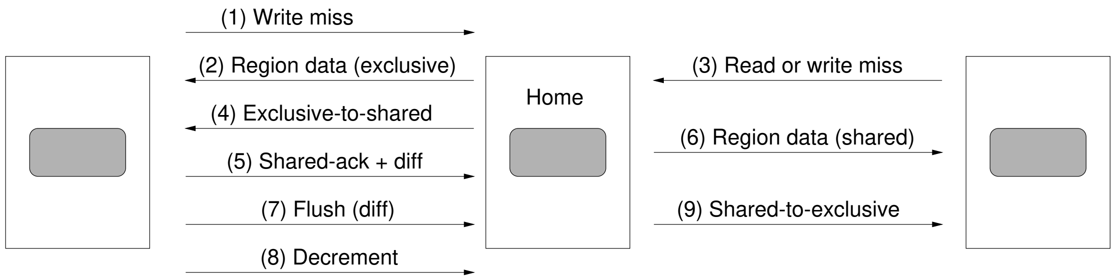

Figure 4: Lazy flushing.

The compiler may generate code for computation migration [12]: part or all of a method invocation is moved to the machine where the data resides. This may be especially effective for synchronized blocks and thread object constructors.

In Jackal, the home node of the lock object acts as the manager of the lock. Lock, unlock, wait and notify calls are implemented as control messages to the lock's home node. When the data protected by the lock resides on the same node as the lock, it is often more efficient to ship the whole synchronized computation to its home: only two messages are involved.

A comparable optimization is applied to calls to thread object constructors. These constructor calls are shipped to the machine where the new thread will run. The result is that the thread object and data created from the constructor have the machine where the thread will run as their home node.

4.2 Runtime Optimizations: Adaptive Lazy Flushing

The coherence protocol described in Section 3.3 invalidates and possibly flushes *all* data in a thread's working memory at each synchronization point. That is, the protocol exactly follows the specification of Java's memory model, which potentially leads to much interprocessor communication. The implementation, however, can relax this protocol without violating the memory model. In particular, it is not necessary to invalidate or flush a region that is accessed by a single processor, or that is only read by any accessing threads. This covers several important cases:

- *home-only* regions that are accessed only at their home node,
- *read-only* regions that are accessed in read mode only,
- *exclusive* regions that have been created and initialized by one node, but are currently accessed by one other node.

Each of these cases corresponds to a *region state*. In general, a region is in *shared* state; if a region is in any of the other states, the thread(s) holding the region apply *lazy flushing*: the region is not flushed on a synchronization operation. *Home-only* is a special case of *exclusive*. It is profitable to make this distinction, however, since the protocol to support *home-only* is much simpler than the protocol for *exclusive*.

A processor is said to *share* a region if the region occurs in the flush list of one or more of the threads on that processor. In its optimized version, the RTS tracks which machines share a region; moreover, it distinguishes between read and write sharers.

The optimized version brings a performance trade-off. In the unoptimized version, regions are always mapped at their home node; they are never faulted or flushed by the home node. To detect any of the other states, the RTS must be aware whether the home node also shares the region (for *read-only*, it must monitor whether the home node is a writer). Now, threads must also flush and fault regions at their home node: present or dirty bits must be set and cleared in home node thread bitmaps, and a pointer to the region must be added to the threads' flush list. However, lazy flushing may be capable of removing most of the flushes and faults at the home node.

A second penalty for the optimized version is that non-home machines must track the number of threads that share a region; if this number drops to zero, the home node must be notified, even when the region has been mapped for read access only. We alleviate this penalty by combining release notices during a flush into a single message per home node, like we did with diff messages.

A region state can be changed by its home node only when a new sharer requests the region, or when a machine gives notice that it no longer shares the region. The new state is computed based on the number of read or write sharers, with the home node as a special case. Some state changes have only local effect (to and from *home-only*), for some state changes the information can be piggy-backed on the data reply (to *read-only*).

Two state transitions bring extra communication with them. First, for a region that goes from *read-only* state to *shared* state, all sharers must be notified; the region is restored on the flush lists of all threads that access the region on all sharer machines. Second, transitions to and from *exclusive* state are rather complicated (see Fig. 4). If a region is shared by zero nodes and some node requests a copy for write access (1), then the home node makes the requesting node the region's owner and gives it an exclusive copy (2). The region remains in *exclusive* state until another node requests another copy from the home node (3). In that case, the home node first sends a message to the owner, informing it to move the region to *shared* state (4). The owner replies with an acknowledgement or a diff (5). The home node merges the diff into its own copy and sends the resulting copy to the requesting node (6). Since the region is now in *shared* state, modifications will be flushed to the home node at synchronization points (7). The region remains in *shared* state until there is only one sharing node, or there are only read sharers left. If any node no longer shares the region, the node informs the home node that there is one sharer less (8). If the last thread on this node had write access to the region, this information is piggybacked onto the diff that is sent home. When only one write sharer remains, the home node puts the region in *exclusive* state and informs the remaining sharer that it is now the region's owner (9). Since the new owner will not invalidate the region from now on, its copy must be brought up to date, so the home node includes the region data in message (9). When only read sharers remain after a release notice, the home node puts the region in *read-only* state; sharers are not explicitly notified, and they will find out the next time the region is accessed.

Frequent transitions to and from *exclusive* state may cause thrashing. We arbitrarily limit the number of times a region is allowed to go to *exclusive* state to 5. From then on, such a region is allowed to go to all region states except *exclusive* state.

5. PERFORMANCE

In this section we study the impact of RTS optimizations on Jackal's performance. All tests were performed on a cluster of 200 MHz PentiumPros, running Linux, and connected by a Myrinet [5] network. We use LFC [4], an efficient user-level communication system. On our hardware, LFC achieves a null roundtrip latency of 20.8 μs and a throughput of 27.6 Mbyte/s (for a 256 byte message, including a receiver-side copy).

Jackal was configured so that each processor has a maximum of 32 Mbyte of local heap and 32 Mbyte of cache available for mapping pages from other processors.

We quote results on Jackal's basic performance from [24]. The time to fault and retrieve a region that contains only one pointer as data is 35 μs. Throughput for a stream of array regions is 24 MByte/s (768 user bytes per 1K packet). Jackal's compiler generates good sequential code; sequential speed of code without access checks is at least as good as the performance of IBM's JIT version 1.3 for Linux, which is the fastest JIT compiler system currently available [7, 23]. Generation of access checks without optimization creates a large performance penalty: up to a factor of 5.5 for the applications described below. The compiler optimization passes reduce the overhead for access checks to 9 % on average for these applications.

5.1 Application Suite

Our application suite consists of four multithreaded Java programs: ASP, SOR, TSP, and Water. Besides the multithreaded, shared-memory versions of these programs, we also wrote equivalent RMI (message-passing) versions of these programs. The data set for each application is small. Fine-grained applications show protocol overhead much more clearly than coarse-grained applications, which communicate infrequently. The differences for the various optimizations come out markedly; also, the comparison with the RMI implementations becomes extremely competitive, since RMI has substantially smaller protocol overhead.

5.2 Parallel Performance

This section compares, for each application, the performance of various Jackal configurations, and presents the performance of an equivalent, hand-optimized RMI program as a yardstick. The RMI programs use a highly optimized RMI implementation [19] and run on the same hardware and communication platform (LFC) as Jackal. On this platform, an empty RMI costs 38 μs. Both the Jackal and the RMI programs were compiled using Jackal's Java compiler.

RMI has its own sources of overhead: parameters and return values must be marshaled and unmarshaled and at the server side a thread is created to execute the method invoked by the client. Nevertheless, RMI has several important advantages over Jackal: data and synchronization traffic can be combined; large arrays can always be transferred as a unit; and object trees can be transfered as a single unit.

In certain circumstances, Jackal's compiler is also able to identify these optimizations [24]; however, the programmer has no opportunity to fine-tune them, since he completely depends on the automatic optimization passes of the compiler.

Below, we discuss the performance of each application. All speedups are relative to the sequential Jackal program compiled without access checks.

We vary RTS optimizations by successively allowing more cases of lazy flushing:

- *basic*: no lazy flushing
- *home-only*
- *home-only* and *read-only*
- *home-only*, *read-only* and *exclusive*

Compiler optimizations are all enabled, except for computation migration, which is toggled to allow comparison of RTS optimizations with compiler optimizations. We toggle only one of the compiler optimizations because switching off many of the compiler optimizations (access check lifting, escape analysis, etc) severely impairs sequential performance, which makes performance evaluation useless. Computation migration has no impact on sequential performance.

To access the impact of RTS vs. compiler optimizations, we present two sequences of measurements: in the first sequence, we start with *basic*, then computation migration is enabled, then the series of lazy flushing states is successively enabled. In the second sequence of measurements, first all states of lazy flushing are successively enabled, and finally computation migration is enabled. If lazy flushing has a far larger impact on performance than the compiler optimization, these two sequences will resemble each other in their performance data. If, however, compiler optimizations are more important, the sequences will differ in their performance data.

Fig. 5 shows the relative data message counts, control message counts (which includes lock and unlock messages) and network data volumes for all application variants on 16 processors. The RMI data is used to normalize the statistics.

ASP. The All-pairs Shortest Paths (ASP) program computes the shortest path between any two nodes in a 500-node graph. Each processor is the home node for a contiguous block of rows of the graph's shared distance matrix. In iteration *k*, all threads (one per processor) read row *k* of the matrix and use it to update their own rows.

The communication pattern of ASP is a series of broadcasts from each processor in turn. Both the RMI and the Jackal program implement the broadcast with a spanning tree. A spanning tree is used for the shared-memory (Jackal) implementation to avoid contention on the data of the broadcast source. The RMI implementation integrates synchronization with the data messages and uses only one message (and an empty reply) to forward a row to a child in the tree. This message is sent asynchronously by a special forwarder thread on each node to avoid latencies on the critical path.

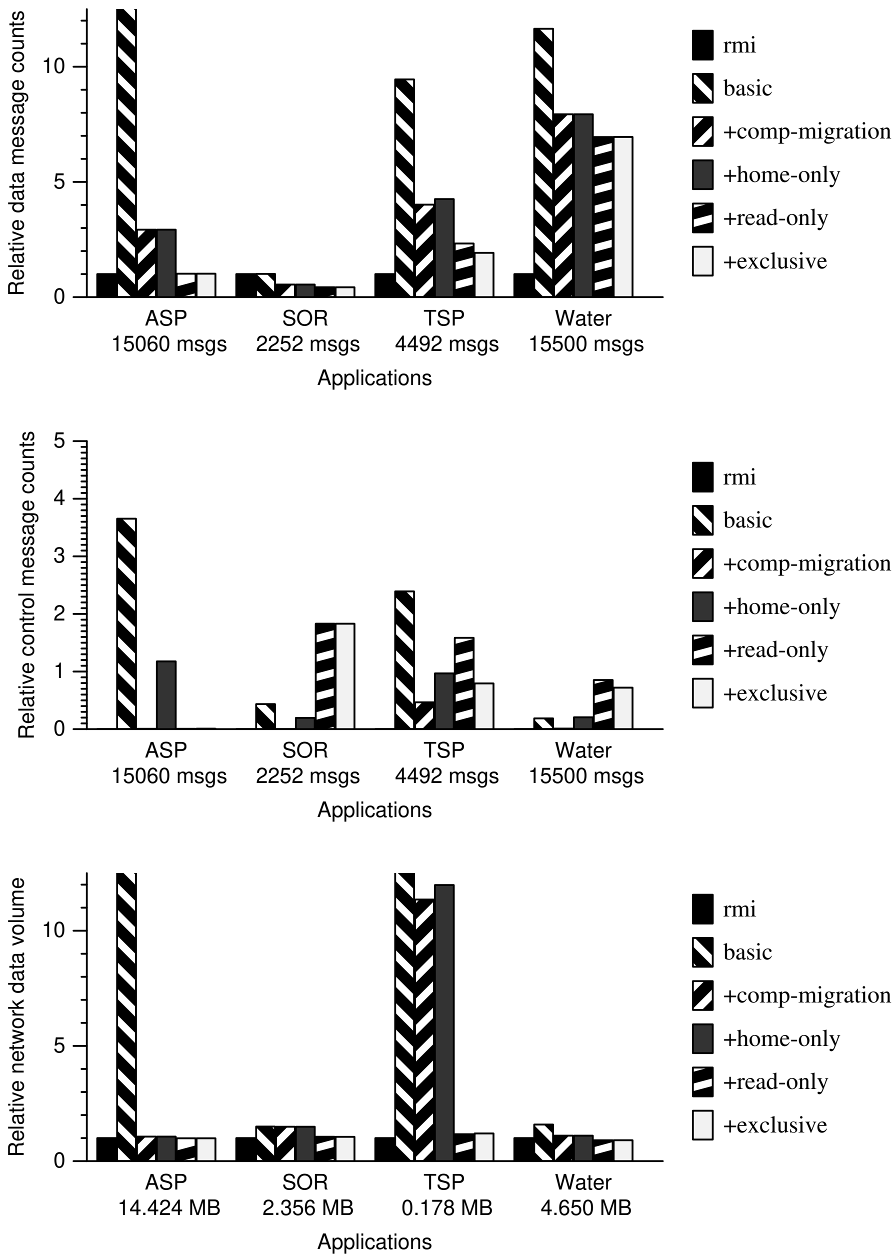

Figure 5: Message counts and data volume for Jackal, relative to RMI. In the top graph, data messages are counted. The numbers under the X axis are the message counts for RMI. In the middle graph, control messages are counted; these are normalized with respect to RMI *data* messages, since control messages do not occur for RMI. In the bottom graph, data volume is presented; only the Java application data is counted, message headers are ignored. The numbers under the X axis are the RMI data volumes.

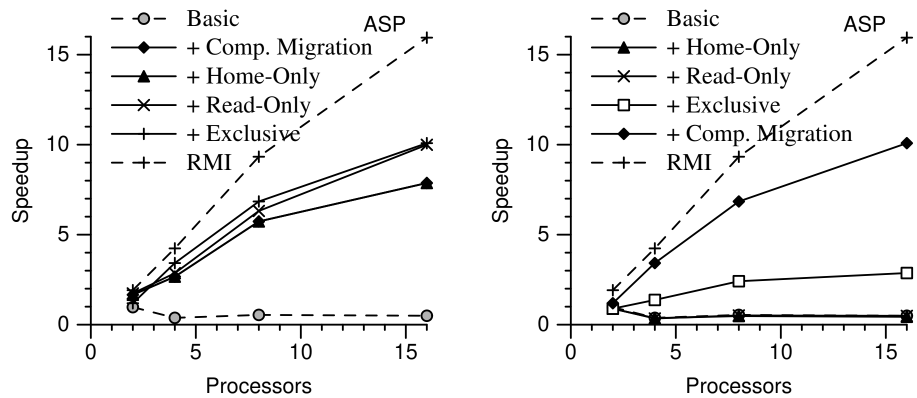

Figure 6: Speedup for ASP. Optimizations are enabled cumulatively, in the legenda from top to bottom. Left, the compiler optimization (computation migration) is enabled first, then the RTS optimizations. Right, the RTS optimizations are enabled first, finally the compiler optimization.

In the compiler-optimized Jackal version (with computation migration and array access check lifting enabled), transmission of a broadcast row is reduced to only one round-trip. The speedup of the RMI program remains better because it uses asynchronous forwarding of rows in its spanning-tree broadcast. An alternative RMI implementation with synchronous forwarding gives the same speedup as the Jackal version.

As appears from Fig. 6, the performance of ASP without optimizations is bad indeed. This is because ASP allocates its data sets in its thread constructors; without thread constructor migration, machine 0 is the home node for all data. Even with all runtime optimizations enabled, speedup is low (at most 2 on 16 processors), since machine 0 must service all data and control messages; see Fig. 5. Performance becomes reasonable only when the thread constructor is migrated and at least *read-only* flushing is enabled.

SOR. Successive over-relaxation (SOR) is a well-known iterative method for solving discretized Laplace equations on a grid. The program uses one thread per processor; each thread operates on a number of contiguous rows of the matrix. In each iteration, the thread that owns matrix partition t accesses (and caches) the last row of partition $t-1$ and the first row of partition $t+1$. We ran SOR with a 2050×2050 (16 Mbyte) matrix.

The Jackal version of SOR attains excellent speedup (see Fig. 7). This is entirely due to those Jackal compiler optimizations we did not vary: the compiler determines that it can combine all access checks in SOR's innermost loop into a single check for all of a row's elements. The entire row is streamed to the requesting processor after one request. In the Jackal version of SOR, the data set is *not* allocated in the constructor of the worker-thread objects, but in their *run*() method, which is not executed until the thread executes on its target processor. Data is written only by home nodes; neighbor rows are only read. This makes the DSM access patterns already optimal even before lazy flushing is applied. Since data is allocated from the *run*() method, computation migration brings no improvement either.

TSP. TSP solves the well-known Traveling Salesman Problem (TSP) for a 15-city input set. First, processor zero creates a list of partial paths and a distance table between each city. Next, a worker thread on every processor tries to steal and complete partial paths from the shared, centralized, job queue. The cut-off bound is encapsulated in an object that contains the length of the shortest path discovered thus far. To avoid non-deterministic computation (which may give rise to superlinear speedup), the cut-off bound has been set to the actual minimum for this data set.

Communication in TSP stems from accessing the centralized job queue, from flushing the current partial path, and from reading the minimum object. The RMI program and the optimized Jackal programs transmit approximately the same amount of data.

The performance differences caused by the various optimizations are small but telling (see Fig. 8). A leap in performance occurs when computation migration is switched on, and the run-time optimizations add a smaller improvement. TSP is the one application where support of the *exclusive* state offers discernible improvement. Partial paths are handed out in write mode, and the thread that evaluates the partial path is the only sharer of that path. After its evaluation, the path is susceptible to lazy flushing only if *exclusive* state is enabled. *Read-only* mode gives rise to improvement because the distance table that describes the city topography is read-only. This also appears clearly from the message statistics in Fig. 5. When *read-only* lazy flushing is enabled, the data communication volume is decreased by an order of magnitude.

Water. Water is a Java port of the Water-n-squared application from the Splash benchmark suite [25]. The program simulates a collection of 343 water molecules. Each processor is assigned a partition of the molecule set and communicates with other processors to compute intermolecule forces.

Most communication in Water stems from read misses on *Molecule* objects and the subobjects referenced by them (position vectors of the molecule). A molecule's force, acceleration, and higher order vectors are stored in separate arrays, which are written only by their owner thread.

Unlike the RMI version, the individual molecules are trans-

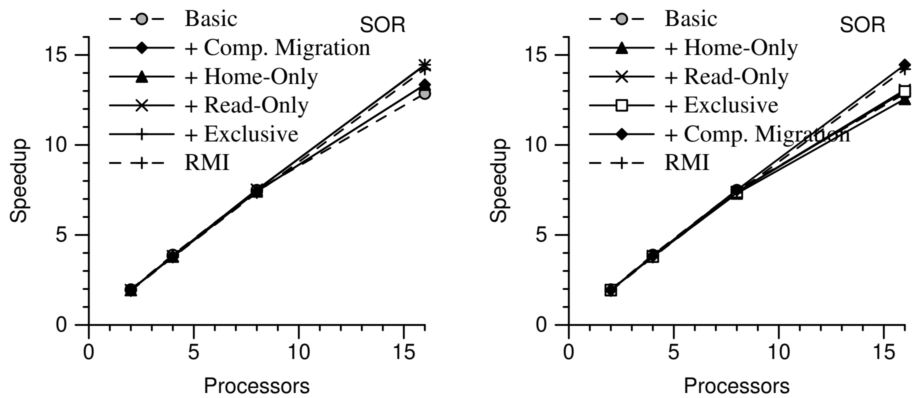

Figure 7: Speedup for SOR. See the ASP speedup graph for explanations.

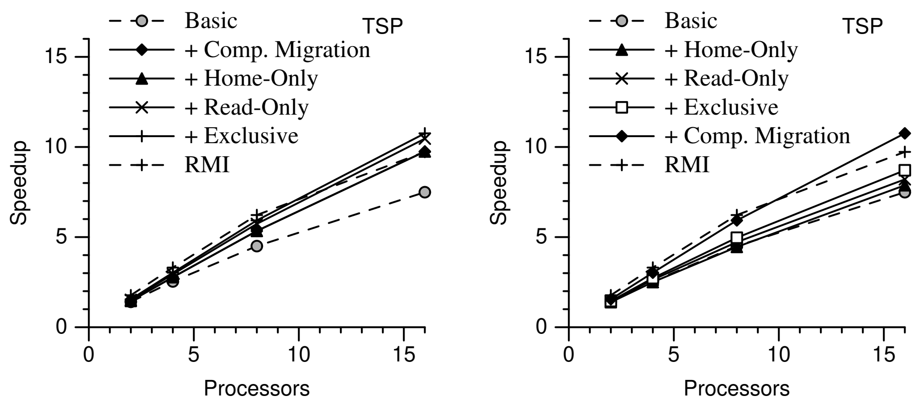

Figure 8: Speedup for TSP. See the ASP speedup graph for explanations.

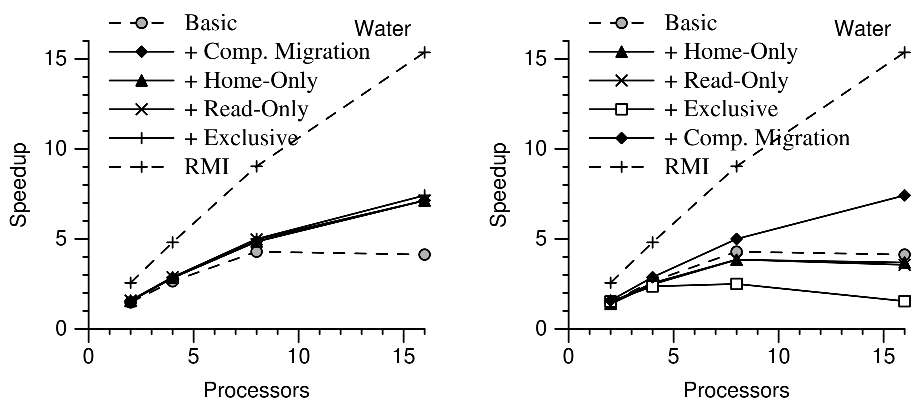

Figure 9: Speedup for Water. See the ASP speedup graph for explanations.

ferred one at a time. Consequently, the Jackal program makes many more roundtrips than the RMI program. In the future, we intend to extend Jackal's compiler with analysis to allow fetching of the entire sub-array of molecules at once; this would enable bulk communication for Water's *Molecule* objects.

As in ASP and TSP, the major performance improvements stem from the compiler optimizations; again, the run-time optimizations do add significantly, but without compiler optimizations the performance is bad indeed. Without compiler optimizations, lazy flushing causes a performance deterioration compared to the basic version. This may be attributed to the extra overhead described in Section 4.2. Enabling of *exclusive* mode in the right-hand graph of Fig. 9 causes a further performance decrease. The reason is that part of the shared data are allocated from the thread constructor. These data are written by their owner thread, but read by all other threads. Without computation migration, the home node for all these data is processor 0, which is swamped with state control traffic, as depicted in Fig. 4.

5.3 Discussion and future work

From the performance data presented above, a clear conclusion can be drawn. Turning on computation migration presents a major boost in performance (except for SOR, which gives good speedup in all versions). Enabling all lazy flushing optimizations, but disabling computation migration, does not yield even reasonable performance for ASP and Water. This is mainly due to the fact that these applications allocate data from the thread constructor, which is a natural thing to do for a Java program. Disabling of further compiler optimizations would make the resulting performance much less good, since sequential performance is impaired.

However, for all applications except SOR, the runtime optimizations on top of the compiler optimizations yield discernible improvements. The smallest improvement seems to be gained from *exclusive* state. On the eye, this state seems a sophisticated optimization that covers many important cases. However, its benefits are already reaped by thread constructor migration and *home-only* state: nearly always, thread constructor migration causes a region that is candidate for *exclusive* state to lie at its home node.

A fact that cannot be read directly from the graphs is that the total time spent in twinning, patching and diffing of objects is negligible in the optimized application runs. Data that is written is usually only written by a single owner, and thread constructor migration ensures that the owner is the home node. The exception is TSP, but there the partial paths that are actually modified by the worker threads are handed out in *exclusive* mode, which obviates the need for flushing and hence twin creation, diffing and patching.

One area for future work is dynamic migration of an object's home node. All control messages would be handled by the new home node, and twinning is unnecessary at the new home node. Possibly, this would make *exclusive* lazy flushing and thread constructor migration redundant. The protocol required for home node migration seems less complicated than the *exclusive* state protocol. Currently, the application programmer must be quite concerned on which machine data is allocated, since having it at the wrong home node brings large performance penalties. This is a valid concern not only for DSM machines, since large shared memory machines also have a home node concept. However, home node migration would probably make allocation considerations superfluous.

6. RELATED WORK

Most DSM systems are either page-based [15, 18, 17] or object-based [2, 3, 14] while discarding transparency. Jackal manages pages to implement a shared address space in which regions are stored. This allows shared data to be named by virtual addresses to avoid software address translation. For cache coherence, however, Jackal uses small, software-managed regions rather than pages and therefore largely avoids the false-sharing problems of page-based DSM systems. Like page-based DSMs supporting release consistency, we use twinning and diffing, albeit not over pages but over objects.

Treadmarks and CVM are both page-based systems that use some form of lazy release consistency (LRC). LRC, like our lazy flushing optimization, postpones writting updates to their home nodes. LRC waits until an acquire is made. Then the new accessor synchronizes with the previous releaser of the lock associated with the data. This allows many state changes to be piggybacked upon synchonization messages. Jackal asynchronously updates region states to support lazy flushing.

CRL [14] is an object based DSM that requires the programmer to annotate his (C) source code with start-read/write and end-read/write calls around accesses to shared regions, so the region to be accessed is locally available. Unlike Jackal, that implements the Java memory model, CRL implements a single writer protocol with sequential consistency. Regions are locally cached until another machine requires the same object, performing some lazy flushing at each end-read/write.

MCRL [11] is an object-based system derived from CRL that implements computation migration. Write operations are shipped to the region's creating machine, read operations are performed locally. Unlike Jackal, however, it does so unconditionally using some heuristics.

Hyperion [20] rewrites Java byte code to C and instruments the code with access checks. Hyperion caches all shared Java objects, including arrays, in their entirety and is therefore sensitive to false sharing. It does not employ any form of lazy flushing.

Fine-grained DSM systems largely avoid false sharing by using a small unit of cache coherence and software access checks. Shasta [22] uses a binary rewriter to add access checks to an existing executable. All implement some form of lazy flushing to record when a processor is exclusively using a region.

7. CONCLUSION

We have described optimizations for the Jackal RTS. Jackal is a DSM system for Java that consists of an optimizing compiler and a runtime system; we refer to [24] for a description of the system, including compiler optimizations.

We found that the RTS optimizations described in this paper are necessary to gain good performance, but only in conjunction with compiler optimizations. If only one of the compiler optimizations (computation migration) is switched off, performance becomes bad for three of the four applications.

When both compiler and runtime optimizations are enabled, our four Java applications attain reasonable to good performance compared to well-tuned and equivalent RMI applications. This is the more significant since small data sets were used, to better bring out performance differences.

8. REFERENCES

[1] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *In Proceedings of the Sixth International Static Analysis Symposium, September 1999*.

[2] H.E. Bal, R.A.F. Bhoedjang, R.F.H. Hofman, C. Jacobs, K.G. Langendoen, T. Rühl, and M.F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Trans. on Computer Systems*, 16(1):1–40, February 1998.

[3] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int. Computer Conference*, pages 528–537, Los Alamitos, CA, February 1993.

[4] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[5] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[6] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 1–19, Denver, CO, November 1999.

[7] O.P. Doederlein. The Java Performance Report — Part II. Online at http://www.javalobby.org/features/jpr/.

[8] 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 Int. Symp. on Computer Architecture*, pages 2–14, Seattle, WA, May 1990.

[9] R. Ghiya and L. Hendren. Putting Pointer Analysis to Work. In *Symp. on Principles of Programming Languages*, pages 121–133, San Diego, CA, January 1998.

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

[11] W.C. Hsieh, M.F. Kaashoek, and W.E. Weihl. Dynamic Computation Migration in DSM Systems.

[12] W.C. Hsieh, P. Wang, and W.E. Weihl. Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems. pages 239–248.

[13] M.D. Hill I. Schoinas, B. Falsafi and D.A. Wood J.R. Larus. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 40–49, Paris, France, October 1998.

[14] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the 15th Symp. on Operating Systems Principles*, pages 213–226, Copper Mountain, CO, December 1995.

[15] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 Usenix Conf.*, pages 115–131, San Francisco, CA, January 1994.

[16] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Int. Symp. on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[17] P. Keleher and C. Tseng. Enhancing software dsm for compiler parallelized applications. In *In Proceedings of the 11th International Parallel Processing Symposium*, April 1997.

[18] L.I. Kontothanassis and M.L. Scott. Using Memory-Mapped Network Interface to Improve the Performance of Distributed Shared Memory. In *Proc. of the 2nd Int. Symp. on High-Performance Computer Architecture*, pages 166–177, San Jose, CA, February 1996.

[19] J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *7th Symp. on Principles and Practice of Parallel Programming*, pages 173–182, Atlanta, GA, May 1999.

[20] M. W. Macbeth, K. A. McGuigan, and Philip J. Hatcher. Executing Java Threads in Parallel in a Distributed-Memory Environment. In *Proc. CASCON'98*, pages 40–54, Missisauga, ON, Canada, 1998.

[21] W. Pugh. Fixing the Java Memory Model. In *ACM 1999 Java Grande Conference*, pages 89–98, San Francisco, CA, June 1999.

[22] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, MA, October 1996.

[23] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000. Online at http://www.research.ibm.com/journal/sj39-1.html.

[24] R. Veldema, R.F.H. Hofman, C. Jacobs, R.A.F. Bhoedjang, and H.E. Bal. Jackal, A Compiler-Supported Distributed Shared Memory Implementation of Java. submitted for publication, http://www.cs.vu.nl/ rveldema/jackal-2001.ps, 2001.

[25] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Int. Symp. on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[26] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release-Consistency Protocols for Shared Virtual Memory Systems. In *2nd USENIX Symp. on Operating Systems Design and Implementation*, pages 75–88, Seattle, WA, October 1996.