A DSM PROTOCOL AWARE OF BOTH THREAD MIGRATION AND MEMORY CONSTRAINTS

Ronald Veldema[1], Bradford Larsen[2], Michael Philippsen [1]
[1]University of Erlangen-Nuremberg, Computer Science Department,
Programming Systems Group, Martensstr. 3, 91058 Erlangen, Germany
[2]University of New Hampshire, Computer Science Department
email: {veldema, philippsen}@cs.fau.de, brad.larsen@gmail.com

ABSTRACT
A DSM protocol ensures that a thread can access data allocated on another machine using some consistency protocol. The consistency protocol can either replicate the data and unify replica changes periodically or the thread, upon remote access, can migrate to the machine that hosts the data and access the data there. There is a performance trade-off between these extremes. Data replication suffers from a high memory overhead as every replicated object or page consumes memory on each machine. On the other hand, it is as bad to migrate threads upon each remote access since repeated accesses to the same distributed data set will cause repeated network communication whereas replication will incur this only once (at the cost of increased administration overhead to manage the replicas).

We propose a hybrid protocol that uses selective replication with thread migration as its default. Even in the presence of extreme memory pressure and thread-migrations, our protocol reaches or exceeds the performance that can be achieved by means of manual replication and explicit changes of the application's code.

KEY WORDS
DSM, protocol, virtual machine.

1 Introduction

There are many problems that require a large memory, larger than a single machine's core memory or even a whole cluster's combined core memories. To name some examples: combinatorial search problems, problems that use large graphs, particle simulations with large numbers of particles, etc.

The DSM protocol presented here *solely* addresses these classes of problem sizes where swapping is *always* needed. It is implemented as an extension of LVM [8], a virtual machine for Java that adds a distributed shared memory. LVM supports these large problem sizes efficiently by implementing its own swapping of objects to disk instead of relying on the operating system. In a cluster context, each machine adds its memory and disk space to the available global memory. Thread migration is used to access remote objects. However, to avoid excessive thread migration selective object replication is needed. This paper presents such a protocol that limits object replication to curb both memory usage and the amount of thread migration required.

We use thread migration by default for two reasons. First, all DSM protocols that fetch data for their operation—whether using lazy-, release-, entry-, or scope-consistency protocols—*require* copies of data for calculating local changes. In effect this at least halves available application memory. Second, we cannot afford to fetch huge numbers of objects locally.

Contributions in this paper are: how to allow both replication and thread migration in the same protocol efficiently, how to limit replication to a fixed pool of memory, and a simple heuristics to decide which objects to replicate. Note that no other DSM protocol that we know of fully addresses the problem of maintaining memory consistency of shared data under thread migration.

2 Related Work

Most DSMs (including the ones mentioned here) assume that applications completely fit into memory and that sufficient memory is available to keep two or more copies of all data. Our protocol only replicates a small part of the data. No other system performs our lazy diff-pulling on thread migration or limits the amount of memory available for replication.

Some page-based DSM systems, e.g. the Coherent Virtual machine (CVM) [7] and Millipede [4], can improve performance by selectively applying thread migration. In general, if a given page is written often enough, thread migration is applied (where we do thread migration by default and replicate objects with a lazy self-consistency protocol). Unlike with LVM, with Treadmarks/CVM/Millipede, the available memory does not grow when machines are added.

CRL [5] is a DSM library (for C) that provides an API for wrapping regions of memory to shared-objects (start-read/write(X), end-read/write(X)). Upon start-read/write(X), the region X is mapped locally. MCRL [3] extends CRL with thread migration. A start-write(X) now causes migration of the current activation record to the machine that hosts X. Under some heuristics, some reads cause computation migration as well. Overall, we differ from MCRL's protocols in various ways: we per-

form thread migration by default and only optionally replicate. Also, we use an update protocol rather than a caching protocol. Neither CRL nor MCRL are memory-aware and both are implemented as libraries where we use a VM approach that is transparent to the programmer.

Jessica2 [10] is a JVM that allows both data caching and thread migration. Data caching is used for remote object access. A migration policy allows the home of an object (i.e. where its meta-data is maintained) to change. The decision is based on a comparison of machine access ratios. Thread migration is used here to allow a programmer to implement application level load-balancing where we use thread migration to manage remote access and use data replication to reduce communication load. Also, Jessica2 is not memory-aware, whereas LVM is.

An overview of DSM systems can be found in [6]. We will focus on out-of-core combined with DSM.

LOTS [2] is closest to LVM. It is also a DSM that can swap out objects to disk. However, its mechanisms are very different. Furthermore, LOTS can only use a third of the available memory/disk space for storing objects (due to its traditional DSM that requires diffs/twins) so that no large numbers of objects can be used. Also, LVM uses thread migration and has no per-object DSM overheads except for objects replicated with the memory bounded replication facility introduced in this paper, which is specifically designed to use only a fixed, small amount of memory. Finally, LOTS requires manually inserted acquire and release statements to control data consistency and to use the C++ library constructs provided. All of this is generated by the compiler in our approach.

3 LVM

The effectiveness of our previous LVM prototype is shown in [8]. That prototype relied solely on thread migration for remote object access. I.e., whenever a remote object is accessed, the remote reference is examined to see which machine owns the remote object and the thread is sent to that machine to access the object locally.

Each cluster node's local address space is divided into 1 MB segments. In an object reference, we encode the machine number, the segment number, and the offset within the segment as shown in Fig. 1.

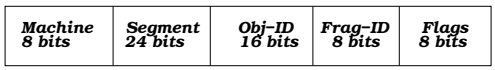

Figure 1. Reference layout.

Because references are not direct memory addresses they must be decoded: Before each reference usage there is a call to *refToObjectPtr* that, given a reference, returns the memory address in the local machine. There is of course a compiler pass that eliminates superfluous repeated calls to *refToObjectPtr* within basic blocks.

```
javaObject* refToObjectPtr( object_reference_t ref) {
  if (is_remote(ref)) {
    if is_locally_replicated(ref)) ref = get_replica(ref);
    else migrate_thread(ref, get_machine(ref)); }
  Segment*s = locate_segment(ref.seg_number);
  javaObject *q = s->data + (32 * get_seg_index(ref));
  return q; }
Segment* locate_segment(int index) {
  Segment *s = &seg_arr[index];
  ...swap-in-if-needed(s);
  ...swap-out-some-olds-if-needed();
  return s; }
Segment seg_arr[MAX_SEGMENTS_PER_MACHINE];
```

Figure 2. Decoding a reference to an object.

The cost for *refToObjectPtr* (Fig. 2) is non-trivial, but not extreme. The important part is the call to *is_locally_replicated(ref)* which ensures replica use. If no local replica is available, thread migration is performed. Except for this new line, the code is the same as that of [8] and needs not be understood further.

4 Replication Protocol

If an object causes excessive migration and the read-to-write-ratio is high, higher application performance and better load balance can be achieved by replication. Our replication protocol is correct with respect to the Java Memory Model (JMM) in that we guarantee JMM semantics for properly synchronized programs but not for incorrectly synchronized programs.

Our replication protocol is aware of memory constraints and is able to perform well with thread migration. The protocol limits the maximal amount of memory that is used for replicated objects, which are managed under an update protocol—changes to an object are synchronized with the copies of that object. Of course replication consumes memory for replicas and twins, that are necessary for later calculating what changes have been made. But by only allowing a limited amount of memory for replication, this concern is greatly alleviated.

To enable use of the replication protocol the *forceReplicate* library function must be invoked on an object. In response the protocol then replicates the object on all machines. This method can either be inserted into the program by the programmer, or it is automatically called by our replication heuristics (Section 4.3). If a machine does not have enough memory left for replication purposes, a machine can deny a replication request. The cluster node that owns the object keeps track of which other nodes have successfully created a replica of that object.

To implement replication, each LVM instance maintains a hash table of references to object/replica pairs (diffs are thus per object). As shown in Fig 2, every time a remote reference is accessed, *refToObjectPtr* searches the hash for

a local replica. If none is found *migrate_thread* is called.

For performance reasons and to allow the programmer a modicum of control over the replication protocol, replicability must be annotated to a class type explicitly:

```
//#pragma replicable
class Data {
  int value;
  Data() { RuntimeSystem.forceReplicate(this); }
}
```

We only generate the call to *is_locally_replicated(ref)* in *refToObjectPtr* for such annotated types. This increases performance of non-replicated object access as it avoids a useless hash lookup and results in smaller code. Because Java array types cannot be given names (cf. typedef in C), a use-case distinction of different arrays is hard. All array types are therefore implicitly replicable. The general rule for adding the replication annotation is to do so only for objects that are relatively seldom allocated.

To allow polymorphism for annotations, we use static type analysis in cooperation with two replication pragma usage rules. First, for any type X marked replicable, all sub-classes of X are automatically replicable. Second, only classes that directly inherit from java.lang.Object can be marked replicable.

4.1 Replication Protocol Implementation

The replication protocol must interact with three pre-existing subsystems: the thread migrator, the synchronization subsystem, and the garbage collector. We discuss these in turn.

Replication must interact with thread migration because to implement the semantics of the Java memory model, changes to objects need to be made available to the modifying thread after its own migration. Consider a scenario where a thread changes replicated object P on machine 0, then migrates to machine 1 where it accesses a object Q. If it then again uses P but this time on machine 1, the thread should see the change that it made earlier when is was still running on machine 0.

To implement this, we could require that threads eagerly pull their changes to replicated objects upon migration. Since this would transmit the full pool of replicated objects on each thread migration, we instead pull these changes lazily. Each thread maintains a table of object reference/node number pairs, where the node number is that of the machine that holds the most recent copy of the object. When a thread accesses an object that it has previously modified at a different machine, it pulls the diff from the node with the most recent copy, applies the diff locally, and updates the table to indicate that the current machine now has the most recent copy. This is illustrated in Fig. 3. We see LVM running on two cluster nodes. Machine 0 has the original of object *A* and machine 1 has a replica. Since thread 0 has modified *A* (1), the original *A* is out-of-date. When thread 0 modifies the replica of *A*, it adds the entry {*Reference of A, 1*} to its local changes table. Next, thread 0 migrates to machine 0 (2). There it references *A* (3). Because there is a local version of *A*, migration is unnecessary. Instead, the thread consults its migrated local changes table, finds the {*Reference of A, 1*} entry, and pulls the changes from machine 1 (4). Once the changes have been applied to the local copy, which in this case is *A's* original version, thread 0 can finally access *A* (5).

Replication also affects the synchronization protocol as the latter is tightly coupled to Java's memory model. Multiple copies of an object must be kept in sync. Our solution is to let a thread publish its changes whenever it executes a monitor operation, i.e. upon *lock*, *unlock*, *wait*, *notify*, *notifyAll*. Upon any of these operations, the executing thread flushes its local changes table, causing all copies of that object to be synchronized. For *properly synchronized programs*, these updates are atomic since shared variable access *requires* lock/unlock for mutual exclusion from the programmer.

To save space we cannot go through the protocol line-by-line, instead we discuss it globally: for each entry in the thread's local changes table, the thread tells the machine with the most recent version to send its changes to the object's owner. The object's owner applies the diff to the original and forwards the diff to every other machine with a replica of that object. Each replicator next applies the diffs to its local copies. Since at this point, all copies of the object have an identical state, the object's owner can tell the initiating machine that synchronization is complete.

Finally, to free unused replicas and twins, the replication mechanism must also interface with the garbage collector. The logic is straightforward: replicas can be collected, whenever the 'original' is collected. Thus, whenever, the 'original' is marked, we send messages to mark its replicas to also keep them alive. During the GC sweep phase, unmarked replicas/twins/objects are disposed of.

4.2 Volatile Variables

If in Java a field is marked 'volatile', it is guaranteed that first, no other access is reordered over the volatile access and that, second, the cache is flushed after the manipulation of the volatile field. We implement the first requirement in the compiler by tagging instructions in our intermediate language as 'volatile'. This causes the instruction to be treated the same as calls to *lock* and *unlock*. The second requirement is ensured by letting the compiler generate a call to *Thread::flush()*. Hence, a volatile access has the same (protocol-level) effect as *lock* or *unlock*.

4.3 Automatic Replication

Intuitively, replication of an object is a good idea if the number of reads is high and the number of modifications is low. However, it can be hard for the programmer to correctly detect objects with such a favorable read-write ratio and to insert *forceReplication* calls to the appropriate objects. We therefore provide a simple heuristics that auto-

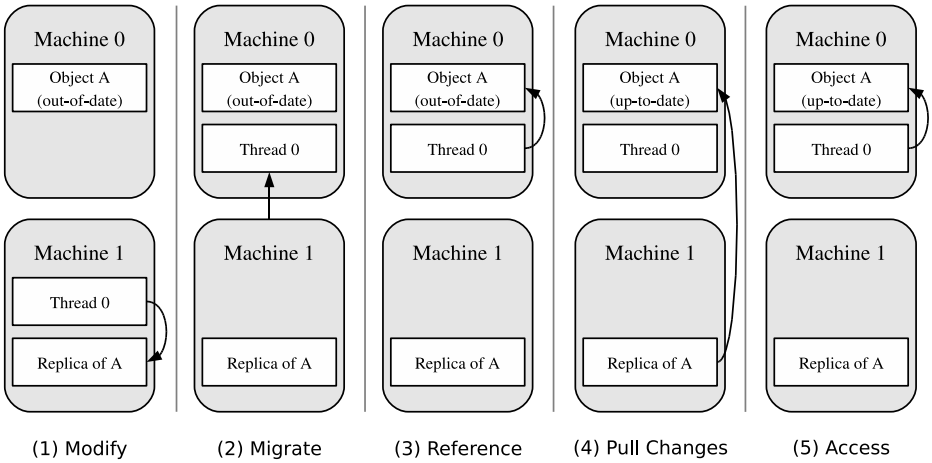

Figure 3. Lazily pulling thread-local changes.

matically calls *forceReplication* once an object has caused a threshold number of thread migrations of a suitable read-/write ratio (each thread migration is caused by either reading or writing a field). However, we still require potentially replicable objects to be marked *replicable* so that for most objects, the costly replica hash-lookup can be avoided. Currently, we require a 4:1 read to write ratio with at least a hundred migrations caused by a read.

This simple heuristic may fail in several cases. First, to keep memory consumption low, the hash table needs to be fixed to a small size. Otherwise, if every Java reference that causes replication would be recorded the hash would grow huge (especially given LVM's goal of a large object-space). The small size causes us to overwrite hash entries in case of key collisions. A too high number of key collisions can effectively disable replication.

The second problem is caused by an increment of a wrong counter. Consider a read-caused migration followed by a write access. At the target machine, the thread immediately after arrival modifies another object that lives on the target machine. Since the read counter is incremented but the write counter is not, the replication engine might incorrectly assume that replication is a good idea. Here is a (simple) example: **tmp = obj1.field; tmp = tmp + 1; obj1.field = tmp;** When execution migrates to the machine that holds 'obj1', the counters will record this migration to be caused by a read. The counters will ignore the subsequent (local) write operation. There are many variations on this theme.

Finally, the heuristics do not monitor the frequency of synchronization actions nor their costs. If synchronization is needed frequently, fewer objects should be replicated. Also, larger objects should be replicated less aggressively for the same reason. Ideally, the observed frequency should be dynamically/adaptively applied to the read/write heuristic we already use. Such heuristics are however delicate.

The programmer can combat these effects in two ways. First, one can disable automatic replication if the heuristics turn out to be too simplistic, or second, one can experiment with different applications of the *replicable* pragma.

5 Benchmarks

Our cluster nodes are quad-core Xeon "Woodcrest" processors running at 3.0 GHz with 4 MB Shared Level 2 Cache per dual core and 8 GB of RAM. Although the machines have 8 GB RAM, we restrict LVM to 1.7 GB to artificially increase memory presure. The Infiniband interconnect can communicate with 10 GBit/s bandwidth per link and direction. LVM, internally uses MPI for communications. We use Intel MPI 3.1.038 over Infiniband for our measurements. Standard JVM measurements are performed on a standalone machine equipped with only 2 GB of physical RAM. This ensures that LVM and JDK run with the same memory presures. We use Sun JDK 1.6 with standard options which is referred to as "JDK" below.

For each of our three applications we consider four versions. In **no-repl** we have no replication at all. In **manual-repl** we did not use replication annotations but instead for each object that needs replication, we allocate a per-thread copy manually by changing the application codes. Third, **force-repl** uses the no-repl code versions but adds *forceReplication* and the *replicable* pragma. Finally, **auto-repl** adds the *replicable* pragma only and lets the auto-replication heuristics select objects to replicate. All times reported are in seconds and are wall times, the mean numbers show the per-machine numbers. We do not compare against raw MPI implementations of these benchmarks for two reasons: the parallelism and programming models are different, and they would rely on the operating system's swapping mechanism.

Ocean. Ocean is a Java port of the corresponding Splash2 code [9]. Ocean studies large scale water movements in an ocean based on eddy and boundary water currents. It is implemented using a red-black Gaus-Seidel multigrid equa-

Table 1. Ocean results.

Mean Thread Migration Count

Machines	1	2	4	8
manual-repl	n/a	1939	33952	28529
force-repl	n/a	2473	62423	30466
auto-repl	n/a	2666	36906	23081

Mean Max Heap Size (MB)

Machines	1	2	4	8
manual-repl	4087	2831	1278	655
force-repl	4087	2040	1174	562
auto-repl	4087	2040	1174	671

Wall Time (seconds)

Machines	1	2	4	8
manual-repl	3172	1390	688	345
force-repl	3336	1403	695	358
auto-repl	3423	1388	683	339
JDK	64041	n/a	n/a	n/a

tion solver [1].

Ocean's main data-structure are a number of 4D arrays of doubles. Only the outer arrays are modified, the inner dimensions are ideal for replication. This is especially suited as the outer dimension(s) will occupy most memory. The relevant statistics are shown in Table 1. Unfortunately, the no-repl version is so slow that it does not run under the time limits imposed by our cluster's fair use policies.

For all versions, we can see that speedup is good. Ocean's auto-repl version wins over the manual-repl version (for 2 or more machines) because its hard to find and copy the correct data structures in the manual-repl version. In the force-repl version we replicated any replicable object. Over those objects, the auto-repl version replicated some objects that were hard to find manually. Both auto-repl and force-repl we gave 256 MB of replication memory, which is small relative to the heap size. The heap size for manual-repl is slightly larger as it replicates some objects unnecessarily (see two machine case).

JCheck. JCheck is a model checker for programs written in a Java dialect. Starting from some initial state, it tries all possible thread interleavings (the state-space) to find reachable error states. Each state consists of at least 14 objects including a large array.

Each machine stores a copy of the hash table of states already tried to avoid duplicate parallel searches. These duplicated hash tables cause (application level) loss of parallelism as each machine's hash needs to be kept reasonably in sync with the other's. In JCheck it pays to replicate the often-accessed, read-only control structures. Also, to reduce the number of thread migrations, all versions of JCheck heavily use the optimized *arraycopy*, *treeCopy*, and *treeEquals* methods (see [8]). The results are given in Table 2.

Table 2. JCheck Results.

Mean Thread Migration Count

Machines	1	2	4	8
no-repl	n/a	962109	631524	513232
manual-repl	n/a	44234	91266	38582
force-repl	n/a	30865	40773	169388
auto-repl	n/a	32408	30869	66202

Mean Max Heap Size (MB)

Machines	1	2	4	8
no-repl	10442	4962	4895	2592
manual-repl	10455	7654	3963	2828
force-repl	10443	5270	5932	3276
auto-repl	10442	4926	4895	2592

Wall Time (seconds)

Machines	1	2	4	8
no-repl	2498	2808	1934	1406
manual-repl	2297	938	404	996
force-repl	2866	1072	1787	370
auto-repl	2760	927	976	208
JDK	37515	n/a	n/a	n/a

Where manual-repl takes under an hour (2297s) on one machine, JDK (using OS-swapping) takes over 10 hours (37515s) on that machine. For either VM, time is mostly dominated by swapping. The overhead of doing the replica-lookup per object access can be clearly seen in JCheck: manual-repl has no lookups where force-repl/auto-repl do. This causes the difference of 2297s vs. 2866s using only one machine. With 8 machines, search space pruning becomes hard as threads can't synchronize their hash tables fast enough. This causes the slow down of manual-repl when going from 4 to 8 machines.

The force-repl and auto-repl versions win over manual-repl in JCheck because with manual-repl object equality of two replicated objects must be implemented by comparing object contents. Instead, with force/auto-repl a reference comparison of replicas suffices. The overall winner is auto-repl with 208s on 8 machines. Even though our replication heuristics is very simple, it is competitive compared to force-repl and manual-repl by being more aggressive in replication. Note that the force/auto-repl versions required far fewer code changes than the manual-repl version and that we only allow 1 MB of memory for replication in JCheck.

Griso. The Griso Subgraph Locator finds occurrences of a (sub) graph P in a (super) graph K. Due to potentially rotated nodes/edges this requires costly graph isomorphism tests. Memory consumption is large since all canonical forms of the permutations of P need to be stored. Fortunately, the memory consumption scales with the number of available machines. The results are shown in Table 3.

JDK performance (389629s) suffers from the semi-random memory accesses that do not affect LVM as its al-

Table 3. Griso Results.

Mean Thread Migration Count ($\times 1000$)

Machines	1	2	4	8
no-repl	n/a	34644	25242	12366
manual-repl	n/a	83	294	506
force-repl	n/a	9	19	15
auto-repl	n/a	96	6480	295

Mean Max Heap Size (MB)

Machines	1	2	4	8
no-repl	7413	3715	1864	932
manual-repl	7384	3700	1856	934
force-repl	7384	3700	1856	934
auto-repl	7384	3697	1854	930

Wall Time (seconds)

Machines	1	2	4	8
no-repl	10121	176400	54600	21180
manual-repl	10121	2427	847	406
force-repl	9760	2963	745	764
auto-repl	11637	4896	3608	797
JDK	389629	n/a	n/a	n/a

location regime automatically puts related nodes/edges on the same segment.

The results clearly show that some version of replication is absolutely necessary. Although our thread migration is fast, performing billions of migrations is deadly. We allow the protocol maximally 256 MB of memory

Note that with few machines, the speed in which the auto-repl version 'learns' which things to replicate (the nodes and edges of the super graph) is slow, which is seen in the high thread migration counts. Only when using 4 – 8 machines sufficient data is gathered for the heuristic to work. Heap usage for all protocol versions are about the same. Manual-repl wins with respect to speedup because it has no administrative data to maintain per replica.

6 Conclusions

We have described a DSM protocol that by default uses thread migration and applies selective object replication to remove excessive thread migrations. A surprising result is that with only little memory set aside for replication (1MB for JCheck, 256 MB for Ocean and Griso), the number of thread migrations already shrinks to reasonable levels. Both automatic and forced replication also nicely alleviate the programmer from the task of manual replica management. A simple heuristics to decide what to replicate by counting how many migrations were caused by read and write accesses has shown to be very effective. It is either more aggressive in how soon to replicate or it finds objects that benefit from replication and that the programmer has overlooked. By annotating which types are candidates for replication, replica statistics management is easier (and therefore the heuristics can be too). Overall, the heap sizes are about the same.

References

[1] Achi Brandt. Multi-Level Adaptive Solutions to Boundary-Value Problems. *Math. Comp.*, 31(138):333–390, April 1977.

[2] B.W.L. Cheun, C.L. Wang, and F.C.M. Lau. LOTS: A Software DSM Supporting Large Object Space. In *Proc. Cluster 2004*, pages 225–234, San Diego, CA, Sep. 2004.

[3] W.C-Yi Hsieh, M.F. Kaashoek, and W.E. Weihl. Dynamic Computation Migration in DSM Systems. In *Proc. of Supercomputing '96*, pages 44–54, Nov. 1996.

[4] A. Itzkovitz and A. Schuster. MultiView and Millipage - Fine-Grain Sharing in Page-Based DSMs. In *Proc. of the 5th USENIX Symp. on Operating Systems Design and Implementation (OSDI '99)*, pages 215–228, New Orleans, LA, Feb. 1999.

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

[6] J. Protic, M. Tomasevic, and V. Milutinovic. A survey of distributed shared memory systems. In *Proc. 28th Hawaii Intl. Conf. on System Sciences (HICSS'95)*, pages 74–84, Jan. 1995.

[7] Kritchalach Thitikamol and Pete Keleher. Thread migration and communication minimization in DSM systems. *Proc. of the IEEE, Special Issue on Distributed Shared Memory Systems*, 87(3):487–497, March 1999.

[8] Ronald Veldema and Michael Philippsen. Supporting Huge Address Spaces in a Virtual Machine for Java on a Cluster. In *Languages and Compilers for Parallel Computing (LCPC) 2007*, Urbana, IL, Oct. 2007.

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

[10] W. Zhu, W. Fang, C.L. Wang, and F.C.M. Lau. A New Transparent Java Thread Migration System Using Just-in-Time Recompilation. In *The 16th IASTED Intl. Conf. on Parallel and Distributed Computing and Systems (PDCS 2004)*, pages 766–771, MIT Cambridge, MA, Nov. 2004.