Object Replication in a Distributed Shared Memory System
A Bachelor’s Thesis

Bradford Larsen
brad.larsen@gmail.com

Advisors:
Dr. Ronald S. Veldema, Friedrich-Alexander Universität
Dr. Philip J. Hatcher, University of New Hampshire

November 19, 2007
A DSM is a distributed system with a single logical address space mapped onto the address spaces of the individual systems.
Distributed Shared Memory Systems

Motivation

DSMs

- provide a more familiar programming model (multithreaded; mutable state)
- hide communication protocol from the users—no message passing
- automatically take advantage of larger clusters (in theory)
Why Care about Distributed Systems?

- scientific computing problems
- sometimes single computers aren’t powerful enough
- they seem to be the future
  - multicore processors
  - multisocket computers
  - NUMA architectures
Why Care about Distributed Systems?

top500.org Statistic

The fastest 500 supercomputers are *all* distributed.
The Large Address Space Virtual Machine

LVM is

- a fine-grain software-implemented DSM system
- an incomplete implementation of Java
- research of Ronald Veldema in Germany
- designed for memory-intensive problems (far larger than can fit in core)
The Large Address Space Virtual Machine

LVM is

- a fine-grain software-implemented DSM system
- an incomplete implementation of Java
- research of Ronald Veldema in Germany
- designed for memory-intensive problems (far larger than can fit in core)
LVM offers

- ability to run unmodified Java code (recompile)
- optimizing compiler; minimizes heap allocation
- interpreter, JIT compiler, garbage collection
- specialized swapper (outperforms OS’s)
- compiler pragmas for optimization
- additional runtime library functions
The Large Address Space Virtual Machine

Some Implementation Details:
- written in C++ (as a “better C”)
- remote procedure calls for communication
- Python for boilerplate code generation
- OS-independent virtual memory system
- noncompliant Java bytecode (register machine)
- thread migration for access to remote data
- does no data caching
The Large Address Space Virtual Machine

Some Implementation Details:

- written in C++ (as a “better C”)
- remote procedure calls for communication
- Python for boilerplate code generation
- OS-independent virtual memory system
- noncompliant Java bytecode (register machine)
- thread migration for access to remote data
- does no data caching
Thread 0 references object B
Object B migrates to Process 0
Thread 0 can access Object B locally
Shared Data—No Caching

Data Migration

Object B migrates back when Thread 1 references it.
Thread Migration

Thread 0 references object B
Shared Data—No Caching

Thread Migration

Process 0

Process 1

Thread 1

Thread 0

A

B

Thread 0 migrates to Process 1
Thread Migration

Thread 0 can access Object B locally
Thread Migration

Thread 0 migrates back when A is referenced again
LVM Threads

- perform computation for Java Thread objects
- migrate upon remote object reference
- have virtual register file, stack, and program counter
- virtual hardware slows sequential performance some
- virtual hardware allows speedy migration
The Problem

With no data caching, ping-pong migration can occur. Not good for performance!
Example: Parallel Array Summator

Overview

- objective: compute an array’s sum in parallel (N times)
- use threads; assume shared memory
- top-level algorithm:
  1. give each thread a portion of array to sum
  2. threads sum their portion
  3. threads update shared sum value
Example: Parallel Array Summator
Essential Characteristics

silly example, but a representative one:
- shared data
- high read/write ratio
- relatively infrequent synchronization
Example: Parallel Array Summator

Entry Point

```java
public static void main(String[] args)
{
    // set num iterations, elements, threads
    parseArgs(args);

    // initialize shared array
    double[] arr = new double[numElems];
    ramp(0.0, 1.0, arr);

    // sum array numIters times
    for(long x = 0; x < numIters; x++)
        double sum = sumArray(arr);
}
```
Example: Parallel Array Summator

sumArray Algorithm:

1. figure out how much work each thread gets
2. create desired number of threads
3. assign work to threads
4. start all threads
5. wait for threads to terminate
6. return sum
Example: Parallel Array Summator

Thread Code

```java
public class ParSumThread implements Thread {
    MutableDouble sum; // shared; set in ctor
    int start, end;

    public void run() {
        double s = 0.0; // thread-local variable

        // do assigned work
        for (long x = this.start; x < this.end; x++)
            s += arr[x];

        // update shared sum value
        synchronized (sum) {
            sum.value += s;
        }
    }
}
```
The Problem: An Inefficiency in LVM

Ping-Pong Migration

Thread objects are distributed round-robin by LVM.
The Problem: An Inefficiency in LVM

Ping-Pong Migration

Thread objects and the array are (usually) on different processes.
The Problem: An Inefficiency in LVM
Ping-Pong Migration

Thread object accessed on each loop iteration (this.start and this.end).
The Problem: An Inefficiency in LVM

Thread Code

```java
class ParSumThread implements Thread{
    MutableDouble sum; // shared; set in ctor
    int start, end;

    public void run(){
        double s = 0.0; // thread-local variable

        // do assigned work
        for(long x = this.start; x < this.end; x++)
            s += arr[x];

        // update shared sum value
        synchronized(sum){ sum.value += s; }
    }
}
```
The Problem: An Inefficiency in LVM

Ping-Pong Migration

The array is accessed on each loop iteration.
The Problem: An Inefficiency in LVM

Thread Code

```java
public class ParSumThread implements Thread{
    MutableDouble sum; // shared; set in ctor
    int start, end;

    public void run(){
        double s = 0.0; // thread-local variable

        // do assigned work
        for(long x = this.start; x < this.end; x++)
            s += arr[x];

        // update shared sum value
        synchronized(sum){ sum.value += s; }
    }
}
```
The Problem: An Inefficiency in LVM

Ping-Pong Migration

So—two thread migrations per thread for each iteration! Excess communication overhead.
The Problem: An Inefficiency in LVM

Ping-Pong Migration

So—two thread migrations per thread for each iteration! Excess communication overhead.
The Problem: An Inefficiency in LVM

Ping-Pong Migration

So—two thread migrations per thread for each iteration! Excess communication overhead.
The Problem: An Inefficiency in LVM

Ping-Pong Migration

So—two thread migrations per thread for each iteration! Excess communication overhead.
The Problem: An Inefficiency in LVM

Ping-Pong Migration

Process 0 becomes a hotspot and performance becomes worse than serial.
Outline

The Problem Context

A Possible Solution
  Hypothesis
  Object Replication

The Object Replicator

Evaluation

Future Work and Conclusion
Hypothesis

LVM can possibly be made faster by adding object replication.
Data Replication

Basic idea: cache frequently accessed data throughout system. Several considerations:

- data should have high read/write ratio
- could manually copy to each process
- manual replication is rewritten for each application
- requires cache coherency protocol—tricky!
Data Replication in a Fine-Grain DSM

Object B is replicated
Data Replication in a Fine-Grain DSM

Thread 0 can access Replica B locally
Outline

The Problem Context

A Possible Solution

The Object Replicator
  Design
  Additions to LVM
  Changes to the Virtual Memory System
  Changes to the Garbage Collector
  Maintaining the Memory Model

Evaluation

Future Work and Conclusion
Object Replicator
Design Considerations

Major issues:

- centralized or distributed ownership?
- automatic or manual replication?
- identifying candidate objects
- coherency protocol
Object Replicator
Design Considerations

Lesser issues:

- memory usage
- whom to replicate to?
- data structure thread contention
- added overhead
Object Replicator

Design Choices

- centralized ownership (i.e. replicas are “lesser”)
- manual mechanism for simplicity (and lack of time)
- candidate objects identified by user with help of migration profiler
- objects are replicated to all LVM processes
- per-process memory limit for replicated objects
- lock-free hash table
public static void main(String[] args) {
    // set num iterations, elements, threads
    parseArgs(args);

    // initialize shared array
double[] arr = new double[numElems];
ramp(0.0, 1.0, arr);
lvm.runtime.forceReplication(arr);

    // sum array numIter times
    for(long x = 0; x < numIter; x++)
        double sum = sumArray(arr);
}
Storing Replicas

- still have one process that owns each object
- original and replicas semantically different
Storing Replicas

- still have one process that owns each object
- original and replicas semantically different
- owner process maintains (local addr, list of replicators) table
- per-process (remote addr, replica addr) table

LVM is heavily multithreaded

using Ronald’s lock-free hash table
Storing Replicas

- still have one process that owns each object
- original and replicas semantically different
- owner process maintains (local addr, list of replicators) table
- per-process (remote addr, replica addr) table
- LVM is heavily multithreaded
- using Ronald’s lock-free hash table
LVM’s Address Space

- 64-bit virtual address space
- address format:
  - machine rank: 8 bits
  - segment number: 24 bits
  - segment offset: 16 bits
  - flags and such: 16 bits
- segmented memory
- bounded core memory use—other segments are compressed and swapped
Decoding an Address

Pre-Replicator

```c
javaObject *
refToObjPtr(object_reference_t ref){
    if(ref.is_remote()){
        // continue protocol on remote machine
        migrate_thread(ref);
    }

    // swap in segment if needed
    Segment *s = get_segment(ref.seg_number)
    javaObject *q = s->data + ref.seg_offset;
    return q;
}
```
Decoding an Address

Post-Replicator

```java
javaObject *
refToObjPtr(object_reference_t ref){
  if(ref.is_remote()){
    // use replica if there is one
    if(have_local_replica(ref))
      return get_replica(ref);

    // continue protocol on remote machine
    migrate_thread(ref);
  }

  // swap in segment if needed
  Segment *s = get_segment(ref.seg_number)
  javaObject *q = s->data + ref.seg_offset;
  return q;
}
```
LVM Threads

Post-Replicator

- lazily pull their unsynchronized changes from other processes
- now “remember” what they have touched since synchronization
- have an added table of (address, rank) pairs
Object B is replicated on Process 0.
Thread 1 references A, thus migrating.
Thread 1 references B, writing to the replica.
Thread 1 references C, migrating away from its changes.
Thread 1 references B. It knows it made changes on Process 0.
The changes are pulled from Process 0.
LVM Threads
Lazy Migration of Unsynchronized Changes

![Diagram showing LVM Threads with Process 0 and Process 1, Thread 0 and Thread 1, and states of variables A and B]

B on Process 1 is now up to date and the reference continues.
LVM Threads
Lazy Migration of Unsynchronized Changes

- must ensure consistency of objects
- need to create twins of changed objects for diff calculation
- on object reference, the thread’s local changes table is checked
LVM’s Garbage Collector

Pre-Replicator

- stop-the-world mark-and-sweep collector
- only global collection—all processes collect at once
- must examine thread stacks for possible references (because of stack allocation)
LVM’s Garbage Collector Algorithm

Pre-Replicator

1. stop user threads; wait for all processes to begin GC
2. scan all thread stacks for things that look like references
3. add possible references on stacks to root set
4. starting from root set
   4.1 mark all reachable locally-owned objects
   4.2 tell owner about each locally-reachable remote-owned object ("mark this object")
5. wait until all LVM processes have finished marking
6. reclaim memory from unmarked objects
LVM’s Garbage Collector Algorithm

Post-Replicator

1. stop user threads; wait for all processes to begin GC
2. scan all thread stacks for things that look like references (now accounts for replicas)
3. add possible references on stacks to root set
4. starting from root set
   4.1 mark all reachable locally-owned objects
   4.2 if replicated, tell replicators to mark replicas
   4.3 tell owner about each locally-reachable remote-owned object (“mark this object”)
5. wait until all LVM processes have finished marking
6. remove table entries for unmarked replicas
7. reclaim memory from unmarked objects
The Java Memory Model

- some inter-thread actions:
  - monitor lock/unlock/wait
  - thread start/terminate
  - volatile read/write

- mandates “happens-before” relations on inter-thread actions

- and more!

- complicated!
LVM’s Implementation of JMM

Pre-Replicator

- monitors are distributed among processes
- threads migrate to process with appropriate monitor
- only one copy of objects, so all writes are immediately visible
LVM’s Implementation of JMM

Post-Replicator

- addition of an update synchronization protocol
- LVM thread flushes local changes upon certain inter-thread actions
  - monitor lock/unlock/wait
  - thread start/terminate
- volatile fields cannot be replicated
- all copies of touched objects are synchronized
- after synchronization, all copies have identical state
Thread 0 modified Object A when it was on the Most Recent Process.
Thread 0 performs a Java inter-thread action—initiator sends a message to Most Recent.
Most Recent diffs A against its twin, updates the twin, then sends the changes to Owner.
Owner applies the diff to A and A’, then forwards the diff to the other Replicators.
Each Replicator applies the diff then sends a response.
Once Owner receives a response from each Replicator, it sends completion message to Initiator.
The synchronization initiated by Thread 0 is complete and Thread 0 continues execution.
Outline

The Problem Context

A Possible Solution

The Object Replicator

Evaluation

Benchmark: Parallel Array Summation
Benchmark: JCheck Model Checker
Benchmark: Griso Subgraph Locator

Future Work and Conclusion
Evaluation

Parallel Array Summation Timings

Array Summation
65536 elements; 1 thread/process

Execution Time (s)

Number of Processes

No Replication
LVM Replication

7.51 7.24 4.1 2.08 1.07
729.96 700.52 792.13
7.51 4.1 2.08 1.07

1 2 4 8
Evaluation
JCheck Model Checker

- model checking is essentially an exhaustive search of possibilities
- used mostly when reliability is paramount (e.g. flood gate system)
- JCheck is a multithreaded Java model checker
- JCheck’s domain is instruction orderings of concurrent programs
model checking is essentially an exhaustive search of possibilities

used mostly when reliability is paramount (e.g. flood gate system)

JCheck is a multithreaded Java model checker

JCheck’s domain is instruction orderings of concurrent programs

model checking example: there are two orders in which one can take a shower and undress
Evaluation
JCheck Model Checker

JCheck: Ping-Pong Protocol

Number of Processes
Execution Time (s)

- No Replication
- LVM Replication
- Manual Replication
Evaluation

Griso Subgraph Locator

Isomorphic graphs are those that are structurally the same.

Griso finds isomorphic subgraphs.
Evaluation
Griso Subgraph Locator

Griso

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10000</td>
</tr>
<tr>
<td>2</td>
<td>3000</td>
</tr>
<tr>
<td>4</td>
<td>5000</td>
</tr>
<tr>
<td>8</td>
<td>8000</td>
</tr>
</tbody>
</table>

LVM Replication
Manual Replication
Outline

The Problem Context

A Possible Solution

The Object Replicator

Evaluation

Future Work and Conclusion
Future Work

- automatic replication via heuristics
- take advantage of locality of reference when replicating
- smarter synchronization protocol
- verify memory model correctness
- system evaluation through common, public benchmarks
- revise and resubmit conference paper
Conclusion

- object replication typically increases performance in LVM
- the object replicator is somewhat slower than manual replication
- LVM replicator much more general than manual replication
- LVM replicator fairly transparent to user
- not immediately clear that Java Memory Model properly implemented
For More Info

- Resources on the Java Memory Model
  http://www.cs.umd.edu/~pugh/java/memoryModel/
- The IROP Grant
  http://www.unh.edu/undergrad-research/irop.html
- Ronald Veldema’s Page
  http://www2.informatik.uni-erlangen.de/Personen/veldema/?language=en
- Send me an email for a copy of our paper:
  brad.larsen@gmail.com