Supporting Huge Address Spaces in a Virtual Machine for Java on a Cluster

Ronald Veldema and Michael Philippsen

University of Erlangen-Nuremberg, Computer Science Department 2,
Martensstr. 3 • 91058 Erlangen • Germany
{veldema, philippsen}@cs.fau.de

Abstract. To solve problems that require far more memory than a single machine can supply, data can be swapped to disk in some manner, it can be compressed, and/or the memory of multiple parallel machines can be used to provide enough memory and storage space. Instead of implementing either functionality anew and specific for each application, or instead of relying on the operating system's swapping algorithms (which are inflexible, not algorithm-aware, and often limited in their fixed storage capacity), our solution is a Large Virtual Machine (LVM) that transparently provides a large address space to applications and that is more flexible and efficient than operating system approaches.
LVM is a virtual machine for Java that is designed to support large address spaces for billions of objects. It swaps objects out to disk, compresses objects where needed, and uses multiple parallel machines in a Distributed Shared Memory (DSM) setting. The latter is the main focus of this paper. Allocation and collection performance is similar to well-known JVMs if no swapping is needed. With swapping and clustering, we are able to create a list containing 1.2×10^8 elements far faster than other JVMs. LVM's swapping is up to 10 times faster than OS-level swapping. A swap-aware GC algorithm helps by a factor of 3.

1 Introduction

There are problems that require extremely large numbers of objects (hundreds of gigabytes to terabyte(s)) and that are as such not bound by processor speed, but rather by the amount of available memory. Examples are simulations with large numbers of 'units', e.g., either molecular or fluid particles [16]; combinatorial search problems, e.g., finding the most frequent sub-graph in a set of other graphs, which requires to store all the graphs already processed; model checkers, which run a program on top of a (simulated) non-deterministic Turing machine (NDTM) and for each non-deterministic choice, the NDTM creates a copy of the simulated machine to explore both choices to check that no illegal program states can occur. Memory requirements for all of the above range from multiples of hundreds of gigabytes to a terabyte and above.

Since it is too costly or impossible to plug in enough memory into a single machine, programmers squeeze their code and rely on the operating system's swapping. Both of which is suboptimal. Reimplementing data structures and algorithms to reduce memory consumption takes time that is better spent implementing functionality and ensuring program correctness. Also, the operating system's virtual address space implementation

not only does not know what data is truly most recently used, but also the amount of virtual memory available (including swap space) is fixed and limited. Extending the amount of swap space is a tedious task for the system administrator and permanently reduces the amount of disk space available to the user. Finally, few operating systems compress swap space or exploit the aggregate memory and swap space available in a cluster.

Our LVM (Large address space Virtual Machine) swaps objects to disk in compressed format and provides a simple distributed shared address space to use all of a cluster's memory and disk space. While this functionality could also be implemented by the programmer, a virtual machine solution provides a separation of concerns. The programmer can concentrate on the correctness and efficiency of the application code instead of optimizing the low-level address space consumption. Also, address space optimizations for one particular program are often useless for the next program whereas a large address space virtual machine can be reused. As we need to modify basic VM data structures, LVM is written from scratch.

In sections 2.1 and 2.2, we describe the virtual address space. LVM's object management is described in Sections 2.3 and 2.5. In Section 2.4 we describe our optimized class library. Section 3 and 4 present performance numbers and cover related work.

2 LVM Implementation

Our compiler frontend [18] generates a register-based intermediate representation which is similar to LLVM [12]. This is fed into LVM that employs both an interpreter and a Just-In-Time compiler (JIT) to execute the code. The code is first interpreted, and if found important enough, it is compiled to native code. To ensure portability, our JIT is very simple: we compile a LVM-function first to C-code and from there to a shared library that is dynamically linked while the program is running.

We chose to use our own register-based intermediate for LVM instead of standard Java bytecodes to easily experiment with language extensions, annotations, compiler optimizations, etc. without being encumbered by Java's bytecode verification, conversion from a stack machine to a register machine, etc.

In LVM we focus on memory-conserving compiler optimization. For example, LVM performs escape analysis [7, 13, 19] and allocates objects that do not escape the allocating function/thread on the stack instead of the garbage collected heap to reduce pressure on the garbage collector.

2.1 Implementing the Address Space

The main problem with implementing a huge, distributed address space is addressing objects flexibly and efficiently. Implementing an object reference as a direct memory pointer is inflexible because it does not allow objects to easily move in memory and because it provides little information for analysis. On the other hand, a reference should be small, since there usually are many references that need to be kept in memory and are manipulated often. For performance, they should not be larger than the operand width of machine instructions. For these reasons, we employ 64 bit references to encode an

object's location in memory in a cluster. Every access to an object thus first needs to decode the object reference to retrieve a local object pointer. While this translation costs at run time, it allows us to access an address space that spans multiple machines. This indirection scheme is fully Java compatible as references are transparent in Java.

Every cluster node's local address space is divided into segments (of a megabyte). Objects are allocated inside such segments. If an (array-) object larger than a single segment is required, a segment is allocated that is large enough to hold it. Otherwise, arrays are treated as ordinary objects. Note that segment size is a trade-off between false sharing (swapping in a segment may also swap in unused data) and disk bandwidth.

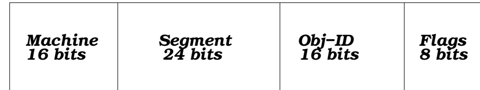

Fig. 1. Reference layout.

An object reference is structured as shown in Fig. 1. The machine field encodes the number of the cluster node on which the object is allocated, the segment field indicates which segment on that machine it resides, and the object-id fields gives the offset (in multiples of 32 bytes) at which the object is found in the segment.

```
class Data {
    int value;
    Data() {
        value = 12345;
    }
    void foo() {}

    public static
    void main(String args[]) {
        Data d = new Data();
        d.foo();
    }
}
```

```
Data_Data__:
%R0_64 = call refToObjectPtr(%R2_64)
('i', (%R0_64 + 12L) ) = 12345

Data_main___3Ljava_lang_String_2:
%R2_64 = vtable_Data
%R0_64 = call new_object(%R2_64)
%R2_64 = %R0_64
%R0_64 = call Data_Data__(%R2_64)
local('l', _14E,-8 ) = %R0_64
%R2_64 = %R0_64
%R0_64 = call refToObjectPtr (%R2_64)
%R0_64 = *('l', (%R0_64 + 0) )
%R0_64 = *('l', (%R0_64 + 104) )
%R2_64 = local('l', _14E, - 8)
%R0_64 = indirect_call %R0_64(%R2_64)
```

Fig. 2. Java example and resulting LVM code.

Because references are no direct memory addresses they need to be decoded. Fig. 2 shows a snippet of Java code and its corresponding LVM code where this is required. The latter is simplified, as we have switched off escape analysis, type inference, method inlining, and the removal of superfluous reference decoding.

In *main()*, first a new *Data* instance is created. This results in the invocation of *new_object* (which returns a *reference*). The *reference* is passed to the constructor in argument register *%R2_64*. The constructor uses *refToObjectPtr* to decode the reference to a physical object pointer. Afterwards it performs the assignment. The invocation of foo() in main() requires the *vtable* that is located in the first 8 bytes of the object. So again, the reference is decoded to a physical pointer by *refToObjectPtr*. The *vtable* is accessed, the method pointer is extracted, and foo() is invoked, passing the *reference* to *this* in %R2_64. With compiler optimization enabled, the superfluous calls to *refToObjectPtr* within a basic block are eliminated.

```
javaObject *
refToObjectPtr(object_reference_t ref) {
   if (gc_requested) gc_barrier_enter();
   if (ref == 0) throw_null_pointer_exception();
   test_dsm(ref);
  Segment*s = locate_segment(ref.seg_number)
   s->update_timestamp();
   javaObject *q = s->data + (32 *
         ref.get_seg_index());
   return q;
}

Segment*
locate_segment(int index) {
   Segment *s = &seg_arr[index];
   if (s->is_swapped_out()) {
         s->swap_in();
         in_core_segments++;
         if (in_core_segments > THRESHOLD)
               swap_out_oldest_segment();
   }
   return s;
}

Segment    seg_arr[MAX_SEGMENTS_PER_MACHINE];
```

Fig. 3. Decoding a reference to an object.

More interesting is how address encoding and decoding works in the presence of LVM's garbage collection in a distributed cluster environment, in the presence of multiple threads, and when swapping is integrated.

Fig. 3 shows the pseudo code for *refToObjectPtr*. Because of its ubiquity, *refToObjectPtr* is also used as the GC barrier. Whenever the garbage collector needs to wait for all threads to stop, all threads are gathered in gc_barrier_enter();

If it is not a null-pointer that has to be decoded, the current thread will either stay at the current cluster node if *test_dsm* determines that the reference addresses a local object. Otherwise, if it detects a remote access, the DSM system is called to migrate the currently executing thread to the cluster node that holds the addressed object.

Finally, the segment in which the object resides is retrieved. If necessary, the segment is swapped in from disk and decompressed. The offset within the segment is used to compute the object's address.

Whenever the limit of the number of in-memory segments has been reached, the oldest segment in memory is compressed[1] and swapped out. This ensures that the operating system's swapping mechanism is never triggered, as LVM will never use more memory

[1] For compression we use the LZO library since it combines high compression speed with reasonable compression ratios [1].

than core memory. LVM speeds up swapping by delaying all swap-out operations until the next swap-in operation. Which segment is least recently used is determined by a logical clock that is set by *update_timestamp()* upon each access. *Update_timestamp()* increments a global variable and sets the segment's timestamp to it. This only takes a few machine instructions. To further increase performance, LVM can be directed to swap-out a number of its oldest segments instead of just one segment when a memory shortage is encountered. The result is that most disk-IO can be performed in parallel. Note that after the operating system's page level swapping loads a page, the OS does not track individual page hits. In contrast, LVM knows exactly which segments have been used last.

To ensure safe multi-threaded access to the segments, segment access needs to be protected by a lock. However, most segments cannot be candidates for swapping because they are too new. For such segments, LVM bypasses the lock for performance reasons and updates the timestamp with an atomic increment.

2.2 DSM support

Where most DSM systems fetch remote data whenever a non-local access occurs, LVM relies solely on thread migration. Upon detecting a non-local data access, the thread (in its entirety) migrates to the machine that hosts the data to be accessed.

We employ this strategy for two reasons. First, all DSM protocols that fetch data for their operation (lazy, entry, scope consistency protocols, etc.), all require caching of objects and/or maintenance of copies for later diffing to find local changes. Also, they need some extra memory to store administrative data per page/object (for example, which machine has a copy, and in which access mode). These memory overheads impact memory usage and are unacceptable for our target applications.

Secondly, we can assume that any non-trivial parallel application will touch large amounts of shared data. If the size of the data is in the range of terabytes, the bandwidth requirements for achieving good speedup will be extremely high. This again means that traditional DSM protocols will mostly be a no-go for our target applications.

Hence, conceptually a call of *test_dsm* returns at a different node if migration is necessary. Of course, the performance of thread migration itself is crucial in this approach. We found that the key is a slightly verbose, machine independent stack-frame and call stack format. First, we use a separate call stack that is independent of the C call stack. Second, both the JIT and the interpreter maintain the same (machine independent) stack frame formats. Whenever the intermediate code writes to a 'virtual register', instead of writing to a physical register, it writes to a thread-local variable. While this slows down sequential code, it allows very fast thread migration as stack frames do not need to be analyzed to locate live registers/variables; stack-frames can be copied between cluster nodes verbatim. The complete call stack is kept in a migration-friendly format for efficiency (at the cost in baseline-performance). A stack frame itself consists of a return address, a parameter block, and a local variable block. The return address is a tuple {function *prev_function, int prev_insn_in_func, int prev_frame_offset}.

Thread migration traverses the stack using the prev_frame_offset links. For each activation, a translation table entry of the form {prev_function->name, offset_in_stack} is added and sent with the stack to the receiver. The receiver uses the translation table to

plug in new function addresses (as the receiver might have allocated functions at different addresses). Migration therefore takes a stack traversal at both sender and receiver with an additional hash look up per stack frame at the receiver to find function addresses for given function names.

To support efficient stack allocation of objects (escape analysis) under thread migration, we maintain a separate per-thread stack using a mark-release algorithm. Management of the non-escaped object stack is then as follows. At function entry, we record the top-of-the-stack pointer. Each non-escaped object allocation bumps the top-of-the-stack pointer to allocate memory. At function exit, the top-of-the-stack is restored, thereby freeing all objects pushed while the function was running. Of course, the compiler only generates code for the above if a function actually allocates an object on the stack.

We maintain a separate data structure for non-escaped objects for two reasons. First, it is difficult to allocate objects directly on a thread's call stack, because after a thread has moved, the call stack will likely be at a different address and also the stack-allocated objects. Any references to the object would need to be corrected to point to the new address. Second, the garbage collector needs to be able to determine if a value found on the stack is a reference or not, even if it is to a stack-allocated object. For this purpose, each run of the GC quickly builds a per-thread bitmap. An enabled bit here says that the address in the thread-local stack starts an object. Building the bitmap is easy as all non-escaped objects are allocated in one single stack data structure, allocated one after the other.

At thread migration, the stack-allocated objects are transferred along with the call stack of the thread. However, at the remote machine, each stack-allocated object will have an invalid method table pointer (which would be at a different address in each LVM instance). For each stack-allocated object, the sender of the stack therefore sends along a type descriptor of the object. The receiving machine uses the type descriptor to patch in the new machine-local method table references.

For speeding thread migration, we maintain both a thread pool of operating system threads and a pool of LVM-thread objects. When an LVM-thread migrates away, the LVM-thread object is put into an object pool and the operating system thread that executes the thread's instructions performs a longjmp back to its start routine where it waits for its reactivation. When an LVM-thread migrates to a machine, we thus only need to pick a preallocated LVM-thread object (which includes its call stack and thread-local heap), initialize it with the migrated LVM-thread's data, and activate a thread from the thread pool of operating system threads. Maintaining an object pool saves us the operating system interaction to allocate enough memory.

In addition to accessing remote objects, there are two other language features that require DSM support. First, to maintain Java's global variables, every write to a global variable is broadcast to all machines. A read of a global variable is therefore a purely local operation. Second, a distributed locking scheme is needed to support Java's 'synchronized' functionality. Each wait, lock, and unlock causes a message to be sent to the owner of the object on which the operation was called. The caller then waits for an acknowledge message. This acknowledgement is sent after the lock-owning machine has successfully executed the lock, unlock, or wait.

2.3 Object Allocation Strategies

LVM implements Java's automatic memory management. It tries to allocate objects in the following order: (1) try first to allocate the object in an in-core memory segment. If that fails due to lack of memory capacity, (2) try to allocate the object on a remote node of the cluster. If the cluster's core memories are full as well, (3) continue locally and try to allocate by swapping out some old segment. Only if the swap space is full as well, (4) a garbage collection is triggered to free local core memory. In short, LVM tries the cheapest allocation method first and proceeds to the most expensive one. Note that phases (2) and (3) can be reordered for a different allocation scenario.

If a program needs arrays larger than a single machine's memory, our HugeArray class should be used that internally fragments an array.

Because lack of object locality causes excessive thread migration, we allow the programmer to suggest object co-location. We do so by extending the semantics of *new* to express that the new object is best located near to or far away from another object. Since in general, establishing optimal co-allocation is very hard to perform by static compiler analysis, we chose to offer this optional annotation scheme to specify locality.

The syntax for our (optional) directive is:

- **new** */*$ close_to(ref) $*/* Type
- **new** */*$ far_from(ref) $*/* Type

where 'ref' is a reference to a previously allocated object. The directive is enclosed in Java style comments so that the code still compiles correctly when a standard Java compiler is used. We implement *close_to* by first trying to allocate the object on the same segment (potentially swapping it in). If that fails, LVM tries to at least allocate it on the same cluster node. With *far_from*, we explicitly try not to allocate the object on the same segment. However, we make no special effort to allocate it on a different cluster node. This allows the allocating machine to fill up first, plus it may reduce thread migration.

Close_to can also be used for maintaining load-balancing by the programmer forcing object allocation close-to its thread-objects (which are allocated round-robin by LVM).

2.4 Reducing Thread Migrations

There are a number of simple Java constructs that can potentially cause excessive thread migrations. See, for example, the code in Fig. 4. If the arrays 'a' and 'b' are allocated on two different cluster nodes, each array element comparison will cause two thread migrations (once to the machine holding 'a' and once for going back to access 'b').

For this reason, we provide a small class library containing elemental operations on arrays. To be exact, we provide methods for fast addition, subtraction, multiplication, and division of two arrays. In addition, Java's class library already offers *java.util. Arrays.equals()* and *java.lang.System.arraycopy()* to compare and copy two arrays. LVM's optimized methods test if both arrays are local, and if so, they do a local operation. If one of the two arrays is remote and the other one is local, the local array is sent to the remote cluster node which then executes the operation locally. This reduces the communication load to a total of two messages instead of $2{\cdot}N$ messages for an N

```
boolean equal_arrays(int[] a, int[] b) {
    for (int i=0; i<a.length; i++)
        if (a[i] != b[i]) return false;
    return true;
}
```

Fig. 4. Comparing two arrays.

element array. To reduce the load on the heap, LVM does not allocate the remote copy on the garbage-collected heap, but instead it is allocated on the system heap. This reduces the pressure on LVM's garbage collector. Note, that because we allocate objects on the system's heap we bypass LVM's swapping mechanism as well. For this reason we reserve a bit of the system's memory for this purpose in advance.

The same problem occurs when copying a graph of objects or when comparing two object graphs for equality if the objects are spread across the cluster. LVM solves both problems by means of a multi-machine object serialization. Object serialization is the process of converting a graph of connected objects into a byte array. Deserialization is the inverse operation. Multi-machine object serialization is specifically built to deal with object graphs that are potentially distributed across multiple cluster nodes. It serializes as many objects on a single machine as possible. It keeps already serialized objects in a hash table to guard against cyclic referencing of objects. Whenever a cycle is detected, a reference to the already serialized object is put into the byte array instead of the object's data. Whenever no more references to local objects can be serialized, the multi-machine serialization process continues on the first machine that holds a remote reference. To detect cycles that span machine boundaries, the hash table is sent along. Note that this scheme relies on LVM's property that references are cluster-wide valid.

Only when used for cloning of an object graph, the deserialization creates the object graph on the LVM heap. Otherwise, when serialization is used for testing equality of object graphs, the object graph is deserialized to the system heap using the system's malloc instead of LVM's garbage-collected heap.

2.5 Distributed Garbage Collection

Java prescribes the use of a garbage collector to automatically remove objects that are no longer reachable. Unfortunately, most of the (local or distributed) garbage collection schemes proposed in the past have high memory overheads. Since LVM must conserve memory whenever it can, the number of choices for designing LVM's GC are limited.

We preferred a distributed mark-and-sweep collector over a copying collector (generational or otherwise) since the latter waste half of the memory which is intolerable given our project's goal of an efficient huge object space (in our benchmarks, intra-segment free-list fragmentation is no problem). Moreover, unlike some distributed garbage collector schemes, we do not separate into local and a global garbage collection phases, again due to memory concerns: to support machine-local GC's, a machine must keep track of incoming references, which can grow to a large set. Also, the gains compared to only using a global GC are low [17]. Hence, LVM starts a garbage collection

phase whenever a cluster node hits its local heap usage boundary. It then requests a GC thread to be started on every cluster node.

Instead of marking the objects themselves, mark-and-sweep collectors can also use *mark-bitmaps* to store the marks. In addition, we use an *allocated-bitmap* to mark a location as allocated when a *new* is executed. Only the bit for the start address is set. The garbage collector can efficiently check that an object reference is valid by testing a single bit in the *allocated-bitmap*. During the sweep phase, an object is quickly determined to be garbage if the corresponding bit in the *mark-bitmap* is unset. Because we allocate objects in 32 byte increments, we require a 4 Kbyte bit array to cover a 1MByte segment.

Naive collectors are costly if they cross high latency network boundaries too often (going to another cluster node, swapping a segment in/out). LVM uses a number of optimizations to keep these costs down. First, to reduce the amount of GC-induced swapping, as many in-core references as possible are marked before any objects are marked that are known to be swapped out. For this reason, we maintain two to-do lists: one list *Core* for in-core objects to be marked, and one set *Swap* for swapped-out objects to be marked. Second, we sort the references in the *Swap* set based on the reference's segment before starting the mark phase for the referred-to objects. This ensures that objects on the same segment are marked together, hence swapping is further reduced. To reduce the cost of sorting the *Swap* set, we implement the *Swap* set as a hash table of buckets. Only the individual buckets then need to be sorted. We will hereafter call a GC using swap sets *'lazy swap GC'* in the measurements.

Third, when a remote reference is seen, it is buffered till either the local machine has no more local marking to do, or the buffer is full (max. 1024 references per buffer). To ensure a level of flow-control, only one outstanding mark-buffer is allowed per target machine.

After the mark phase has finished, every machine independently sweeps its local memory. Segments that were left untouched during marking are freed in one go. Segments that are only partially filled have their free lists rebuilt.[2]

3 Performance

To demonstrate LVM's effectiveness we first need to show that it is competitive with a standard JVM for small memory demands and that it outperforms the OS swapping algorithms for larger memory footprints.

We measure on two different machines (as our cluster's policy does not allow long running jobs). For the micro-benchmarks, we use two 2 GHz Athlon machines equipped with 2 gigabyte RAM each. For the application benchmarks, we use a cluster of Intel machines with 3 GHz Woodcrest CPUs. Each machine is equipped with a SATA disk with at least 80 GByte free space. All machines are equipped with both 10 GBit Infiniband and 1 GBit Ethernet. In all cases, LVM is configured to use at most 1.7 gigabyte RAM per machine for storing Java objects and arrays. This leaves 300 megabyte for

[2] Instead of using physical pointers that become invalid when a segment is swapped in at a different memory address, LVM implements the free list as offsets from the start of the segment to the next free space within the segment.

the operating system, networking software (communication buffers), the LVM garbage collector, the JIT-ed code, and the interpreter's data.

3.1 Micro Benchmarks

To measure the performance of object allocation and object access, we allocate (see Fig. 5) and traverse (see Fig. 6) linked lists of increasing lengths. To be exact, we start the program, create and traverse a list in a loop (10 iterations), and exit the program. After a list summation, each list becomes garbage. The VM is restarted for each new list length.

We perform the same test both with SUN's JDK 1.6 and LVM. Starting at 2 Gbyte SUN's VM relies on the operating system's swapping mechanism whereas LVM already starts to swap at 1.7 GByte. LVM outperforms SUN's JVM in both list creation and list traversal as soon as swapping is needed. We stop measuring JDK's performance at lists with 7.8×10^7 elements due to the excessive time needed.

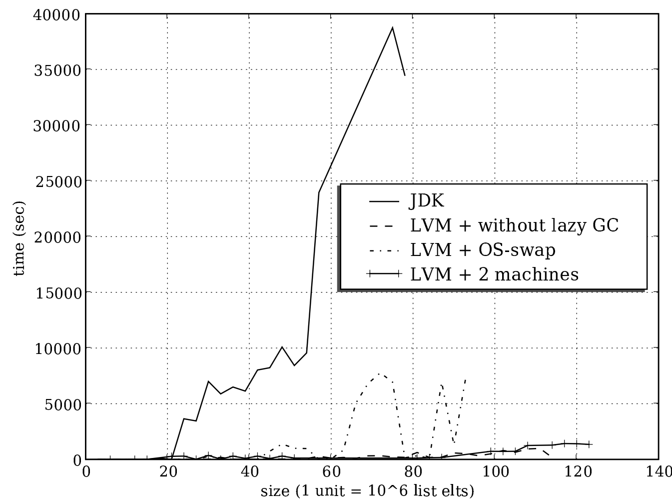

Fig. 5. LinkedList Creation.

To show that LVM-directed swapping is much more efficient than OS-level swapping we disabled LVM's swapping module (see LVM+OS-swap numbers) and instead relied on the OS's virtual memory implementation. Note that in OS-LVM, the code still contains calls to *refToObjectPtr*. It is interesting to see that with OS-level swapping the system becomes very unresponsive as soon as the OS starts to compete for memory against the JVM or the LVM-OS version. This competition also impacts messaging speed as I/O buffers compete for memory as well. When LVM's internal swapping is

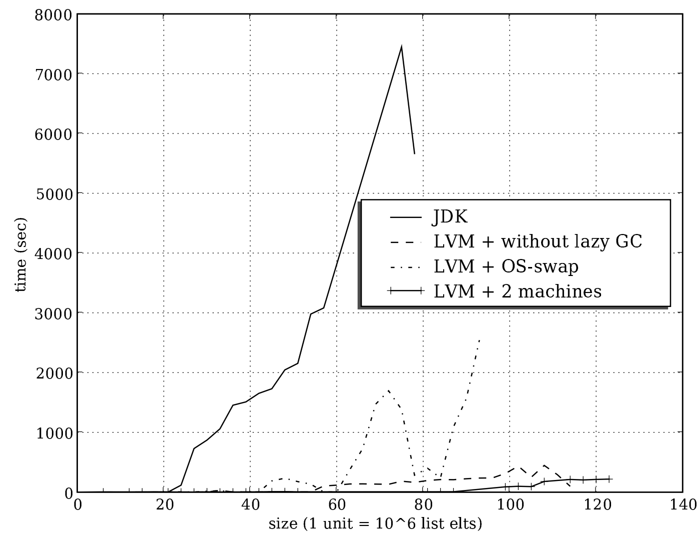

Fig. 6. LinkedList Traversal.

enabled, OS performance does not suffer because enough memory is always reserved for it.

The irregularities in the results are caused by the GC. For example, if many GC passes occur when the lists are almost completely constructed, a lot of memory must be scanned, the reverse when lists are still small. The irregularities are thus a harmonic of both the heuristics LVM uses to decide when to collect garbage and the list sizes.

The builtin-swap version of LVM is slightly slower in object allocation due to the extra code needed (2 if statements) on the fast path to test for the need to swap. It is clear that data compression is not a bottleneck (LZO compresses at 100 MB/s and decompress at 310 MB/s on the 2Ghz Athlons). A 1 MByte LVM segment is, in the list benchmark, on average compressed to a 360 KByte file on disk (approx 36%) which reduces disk-I/O time and frees disk space. Using two machines, we first fill one machine's core memory, then the other's. The speedup is not caused by parallelism, as there is still only one active thread in the cluster which migrates to the machine with free memory. The List/Node/Data constructors as well as all get/set methods are all inlined. Hence, method call overhead is thus not an issue in this benchmark.

LVM's performance greatly depends on the overhead of thread migration. Here, our implementation is extremely fast. A one-way thread migration takes just 54.9 μs over gigabit Ethernet. With Infiniband, the latency of thread migration drops to 19.9 μs. These times include thread exit, start, message transfer, and stack patching.

3.2 Application Benchmarks

JCheck is a **model checker** for a simplified Java dialect called Tapir. The model-checker tests all possible interleavings of thread-executions to find program bugs. Each state consists of a simulated heap and simulated threads. A Tapir program is translated into a simple bytecode format. At each point in the Tapir program where a context switch may occur, the Tapir compiler inserts a context switch bytecode instruction. Bytecode fragments delimited by context switch instructions are then emitted as separate Java functions.

To reduce the search space, each thread maintains a hash table of the states it has already seen. Before proceeding with a new state, a thread checks in the hash table if that state has already been visited. JCheck gives each thread its own private hash table to reduce synchronization costs. Once a new state is found, a thread publishes the new state by adding a reference to it to all the other thread's hash tables. As each thread maintains its own hash table, memory usage increases with cluster size.

For our LVM test, we wrote a simple Tapir program in which two processes alternatingly send an RPC to each other (which includes message delays and object-allocations). Note that this is almost the smallest problem to limit execution time (using SUN JDK, JCheck requires more than a day).

Table 1. JCheck results

	1 machine	2 machines	4 machines	8 machines
Time, no-lazy-GC (seconds)	8830.2	3728.2	429.8	1553.6
Time lazy-gc (seconds)	3803.7	1077.9	412.7	1483.1
Avg. Heap (MByte total in cluster)	10196	9152	10857	24152
Avg. #thread migr. per machine ($\times$ thousand)	—	69.7	111.2	212.3

The memory requirements (see Table 1) are extreme due to the number of states that need to be explored and the corresponding hash tables for them. To reduce the number of thread migrations, JCheck heavily uses the optimized arraycopy, treeCopy, and treeEquals methods (see Section 2.4). Thread migration mostly happens whenever a thread attempts to publish a new state in hash tables belonging to other threads.

Lazy swap GC is a big gain for JCheck; GC is three times faster with it which shows most clearly on the 1 machine measurements (where run time is only 2.3 times faster as GC is only a portion of the run time). When using more than one machine, preformance is greatly influenced by the speed in which the thread's hash tables are kept up-to-date to allow pruning of the search space. With eight machines (threads), this becomes hard. The increased heap usage with eight machines is caused by missing search-space pruning opportunities (and thereby lowering speedup). Swap compression allows a 1 MB segment to be compressed to a 65 KB file on average.

The **Griso sub-graph locator** finds occurrences of a graph P in another graph K. Since nodes and edges may be rotated, a complicated graph isomorphism test is needed. The algorithm first creates a set of permutations of P with the outgoing edges of each node permuted to create a set. This set is reduced by only allowing canonical forms of

the graphs into the set (while also converting K to its canonical form). The set is then partitioned into N parts, so that each of N worker threads can locate embeddings of a permutation of P in K.

Table 2. Griso results

	1 machine	2 machines	4 machines	8 machines
Time (seconds)	274752	176400	29871	6962
Avg. Heap (MByte total in cluster)	15531	14838	15472	14912
Avg. #thread migrations per machine	—	34644978	25242531	12315766

Memory consumption (see Table 2) is large since all canonical forms of the permutations of P need to be stored (again excluding a standard JVM). Fortunately, with increasing numbers of machines, the memory requirements per machine drop such that with the graph sizes that we have chosen, with 8 machines the graphs almost fit in the cluster's memory (1864 MByte per node, with an LVM limit of 1700 MByte causes 164 MByte worth of graphs on average that need to be swapped). This results in the superlinear speedup seen when going from 4 to 8 machines. Unfortunately, Griso can take very little advantage of the class-libraries provided. This causes the high thread-migrations counts.

Besides using a lot of objects, Griso also creates a lot of garbage. On 8 machines, 839 seconds is spent on garbage collection using lazy-swap-GC in 50 GC passes. Each GC pass takes about 16 seconds. Without lazy-swap-GC, this increases to 950 seconds total for GC, or 116 seconds longer. Each GC pass freed about a gigabyte of memory in mostly small arrays used to hold references to graph nodes (for cycle testing in graphs).

4 Related Work

LVM implements a host of techniques to increase available memory and performance. For each of these we will give a few entry points to the related work.

Operating systems. Swapping and compression of swap space is a technique usually associated with operating system implementations and out-of-core applications. In [8] Linux was adapted to compress MMU pages before swapping them out to disk. A small cache is used to store pages being compressed. In [5, 15] Linux is adapted to divide memory into two parts. One holds compressed pages, the other uncompressed. Instead of swapping out an uncompressed page, the system first attempts to compress the page and to place it into the compressed memory area. That avoids many disk-IO operations. Our system is implemented on top of an OS. Hence, LVM is not restricted to the MMU's 4K page sizes, it is portable, and allows for multiple techniques to reduce memory pressure (escape analysis, lazy GC swapping, etc.).

Operating systems can also increase a process's available memory by remote swapping or remote paging. One approach is described in [10]. Here, a special I/O device 'nswap' is registered in the kernel. Related to this is [9], where the lowest-level page-manager in an operating system is made cluster-aware. Both approaches perform remote

paging to allow idle nodes to cache pages of heavily loaded nodes to decrease reliance on slow disks for swapping. In contrast, LVM starts to remote-allocate objects once local memory is full and performs thread migration to access them afterwards.

Distributed garbage collection. A modified Linux notifies the Jikes RVM in [11] that a page is about to be swapped out. Whenever this happens, the GC creates a list of outgoing references from that page. Objects on that page are then part of the root-set for the GC's marking phase. All references to objects on swapped-out pages are ignored in subsequent collections. In contrast to LVM, RVM is restricted to a single machine and to the size of the OS level virtual address space. Closest to our distributed garbage collector is the system of [17]. However, it targets ABCL/f instead of Java, uses a traditional data fetching DSM system (with the associated memory overheads discussed above), and assumes that all data fits into core memory.

Locality directives. Related to our locality based directives is ccmalloc [6], an alternative to malloc that allows to allocate something close to some other ccmalloced block of memory. However, the authors' goal is cache optimization instead of swap optimization. Moreover, they target C instead of Java.

Out-of-core & DSM. There have been a number of Java-based DSM systems. An overview of DSM systems can be found in [14]. We will, however, concentrate on out-of-core in combination with DSM.

The interaction between, out-of-core applications, compilers, and execution on a DSM system is investigated in [2]. The authors perform source code analysis to add an inspector-executor style parallelization method. An inspector finds probable data usages and hands these over to the executor for the execution of the program. We perform no source code analysis to detect parallelism but rely on Java threads explicitly created by the programmer. Moreover, our different DSM style that relies on thread migration instead of data fetching is advantageous for memory-greedy applications. The compiler analysis for out-of-core applications in [3] inserts prefetch instructions to fetch array data from disk. The techniques described here are orthogonal to LVM: instead of inserting prefetch instructions, the LVM front-end compiler could try to call *refToObjectPtr* as early as possible.

LOTS [4] is closest to LVM. It is also a DSM that can swap out objects to disk. However, the mechanisms and techniques are quite different. LVM compresses data on disk while LOTS does not. Furthermore, LOTS can only use a third of the available memory/disk space for storing objects. LVM uses a virtual machine approach, while LOTS is provided as a C++ library. LOTS is an object-based DSM that migrates data and that therefore pays the memory penalty for storing proxy objects, diffs, and twins of pages. These overheads sum up significantly so that LOTS cannot support large address spaces, especially when large numbers of small objects are used. LVM uses thread migration and has no per-object DSM overheads. Finally, LVM manages standard Java code and migrates threads through a cluster automatically. LOTS requires manually inserted acquire and release statements to control data consistency and to use the C++ library constructs provided.

References

1. http://www.oberhumer.com/opensource/lzo/.

2. P. Brezany, A.N. Choudhary, and M. Dang. Parallelization of irregular out-of-core applications for distributed-memory systems. In *Proc. of HPCN Europe '97*, pages 811–820, Amsterdam., Apr. 1997.
3. A.D. Brown, T.C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM Trans. Comput. Syst.*, 19(2):111–170, 2001.
4. 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.
5. Irina Chihaia and Thomas Gross. An analytical model for software-only main memory compression. In *WMPI '04: Proc. of the 3rd workshop on Memory performance issues*, pages 107–113, Munich, Germany, June 2004.
6. T.M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proc. of the ACM SIGPLAN 1999 Conf. on Programming Language Design and Implementation*, pages 1–12, Atlanta, GA, May 1999.
7. J.D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff. Escape Analysis For Java. In *Proc. of the 1999 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, Denver, CO, Nov. 1999.
8. T. Cortes, Y. Becerra, and R. Cervera. Swap compression: resurrecting old ideas. *Software, Practice and Experience*, 30(5):567–587, 2000.
9. M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. *SIGOPS Oper. Syst. Rev.*, 29(5):201–212, 1995.
10. S. Finney, K. Ganchev, M. Klock, T. Newhall, and M. Spiegel. The NSWAP module for network swap. *Journal of Computing Sciences in Colleges*, 18(5):274–275, 2003.
11. M. Hertz, Y. Feng, and E.D. Berger. Garbage collection without paging. In *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '05)*, pages 143–153, Chicago, IL, 2005.
12. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2004 Intl. Symp. on Code Generation and Optimization (CGO'04)*, pages 75–85, Palo Alto, CA, Mar. 2004.
13. Kyungwoo Lee, Xing Fang, and Samuel P. Midkiff. Practical escape analyses: how good are they? In *Proc. of the 3rd Int'l Conf. on Virtual Execution Environments (VEE '07)*, pages 180–190, San Diego, CA, 2007.
14. 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.
15. Luigi Rizzo. A very fast algorithm for RAM compression. *SIGOPS Oper. Syst. Rev.*, 31(2):36–45, 1997.
16. R. Ryne, S. Habib, J. Qiang, K. Ko, Z. Li, B. McCandless, W. Mi, C. Ng, M. Saparov, V. Srinivas, Y. Sun, X. Zhan, V. Decyk, , and G. Golub. The US DOE Grand Challenge in computational accelerator physics. In *Proc. Linear Accelerator Conference (LINAC98)*, pages 603–+, Chicago, Aug. 1998.
17. K. Taura and A. Yonezawa. An Effective Garbage Collection Strategy for Parallel Programming Languages on Large Scale Distributed-Memory Machines. In *6th Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 18–21, Las Vegas, NV, June 1997.
18. R. Veldema, R.F.H. Hofman, R.A.F. Bhoedjang, C.J.H. Jacobs, and H.E. Bal. Source-level global optimizations for fine-grain distributed shared memory systems. In *8th Symp. on Principles and Practices of Parallel Programming (PPoPP)*, pages 83–92, Snowbird, Utah, June 2001.
19. J. Whaley and M. Rinard. Compositional Pointer And Escape Analysis For Java Programs. In *Proc. of the 1999 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 187–206, Denver, CO, Nov. 1999.