

Jackal, A Compiler Based Implementation of Java for Clusters of Workstations

R. Veldema R.A.F. Bhoedjang H.E. Bal

Dept. of Mathematics and Computer Science,
Vrije Universiteit, Amsterdam, The Netherlands

Keywords: software distributed shared memory , optimizing compilers, access checks

Abstract

This paper describes the design of Jackal, a compiler-driven distributed shared memory implementation of the Java programming language. Our goal is to efficiently execute (unmodified) multithreaded Java programs on a cluster of workstations. Jackal consists of a native Java compiler and a runtime system that implements a distributed shared memory protocol for variable sized memory regions. The Jackal compiler stores Java objects in shared regions and augments the programs it compiles with access checks. These access checks drive the memory consistency protocol. The Jackal compiler implements several optimizations to reduce the overhead of these software access checks. The main contributions of this paper are: techniques to allow a (Java) compiler to target a fine-grained all-software DSM using access-checks and compiler optimizations to achieve good efficiency.

1 Introduction

The performance and usability of current software Distributed Shared Memory (DSMs) systems is unsatisfactory for a number of reasons. Some systems do not perform well because no program level information is available (e.g., TreadMarks [9]). Others require the programmer to annotate the program to indicate where and how shared data are accessed, which is an awkward and error prone job. Yet other DSM systems translate a shared memory program into a message passing program. High performance Fortran (HPF), for example, uses a shared memory programming model, and lets the compiler generate message passing code. Again this requires the user to annotate the program, this time to aid the compiler's optimizer.

With other software DSMs, accesses to shared data somehow must be delimited, which implies that the programmer must decide in an early stage of program development which data structures will be shared. Object-based DSMs, such as Orca [2], CRL [7] and Jade [13] require the programmer to explicitly code which data items are to be shared and to encapsulate their uses by special function calls or annotations (e.g., *start-read-data* and *end-read-data* in CRL [7]).

A promising approach is *fine-grained* DSM, which provides a global (flat) address space, implemented using a DSM protocol that manages small regions in the same way that a multiprocessor manages cache lines. Fine-grained DSMs suffer much less from false sharing than page-based DSMs, because the unit of coherence is much smaller. The price paid for this is that the access checks (which test whether a region of shared data is available on the local machine) must be done in software, whereas with page-based DSMs the checks are done by the hardware MMU. With most current fine-grain DSMs (e.g., Shasta [14]), the access checks are inserted into the executable code using a binary-rewriter, making such DSMs difficult to port.

The goal of our work is to build a fine-grained DSM system for Java. Unlike earlier fine-grained DSMs, we let the (Java) compiler generate the access checks. The compiler uses information about the source programs to reduce the overhead of access checks as much as possible. The result will be that we can run unmodified multithreaded Java programs in parallel on a distributed-memory systems (e.g., a cluster of workstations), provided we have the source code (not just the byte code) available.

2 Related work

Java supports Remote Method Invocation (RMI), which can be successfully used for parallel programming on distributed shared memory machines [10]. RMI, however, offers a distributed memory programming model which is harder to use than the multi-threaded model.

The goal of the Java/DSM [17] project is closest to ours. Java/DSM uses the standard SUN JDK [1] and TreadMarks [9] to implement software distributed shared memory. TreadMarks [9] is a page-based software DSM that uses the MMU of the processor to maintain page level coherency. This introduces false sharing and expensive traps to the operating system. TreadMarks allows (C/C++) programs to run in parallel with minor changes only. Several other Java systems are also designed to run multithreaded programs unmodified on distributed-memory systems [12, 11]. Our work differs by using a fine-grained DSM system to manage shared data.

Shasta [14] and Sirocco [5] are fine grained DSMs that instrument an executable with access checks around every pointer access (except those that can be easily avoided such as stack and global variable accesses). Both systems lack global program level information which restricts their capacity to reduce access check overheads. Our system is compiler based and thus has no such restrictions.

CRL [7] requires the user to manually insert access checks around accesses to shared data in the source program, which can be a labour-intensive job. On the other hand, CRL's performance can be quite good if the programmer minimizes the number of access checks without unnecessarily reducing concurrency. The challenge for a compiler driven approach is to perform such optimizations automatically.

3 Memory management

Our system provides a distributed implementation of the multithreaded Java programming model, which was designed for shared-memory machines. The system deals with Java objects, regions, and pages. The object is the (only) abstraction seen by the programmer. A region is our unit of coherence. Processors can cache regions in their local memory. The DSM protocol transmits regions over the network and takes care of keeping regions consistent. Regions can have different sizes. The compiler decides how many regions are used for a given object and what their sizes are. Finally, the system also deals with pages, which are used to provide the illusion of a single (flat) address space. Below, we

discuss the management of objects, pages, and regions in more detail.

3.1 Object management

Java objects are stored in a global virtual address space. We use real, physical, pointers into this address space to avoid (expensive) software indirections. The implementation of the global address space is described in Section 3.2. An object is split into one or more regions (the unit of coherency in our system). A small object might thus be stored in a single region, and a large object (array) might be split across multiple regions. Every region has a header, region headers are discussed in section 3.3. Region granularity is decided by the compiler, based on access patterns and object size.

There are two advantages to the division of objects into multiple regions. First, we can largely avoid false sharing by making the regions in which object partitions are stored sufficiently small. Second, we can reduce network traffic by transferring regions instead of whole objects when some part of an object is accessed by a remote process. As with other fine-grained DSMs, this approach will result in a larger number of (smaller) messages than with page-based DSMs. Modern high-speed networks, however, efficiently support such fine-grained communication.

3.2 Address space management

Our system implements a global shared address space on a distributed-memory system. To ensure that a call to a memory allocation primitive on one machine delivers an address that is not used on other machines, each machine owns a disjunct part of the global virtual address space, from which it allocates memory. An example is given in Figure 1. If an object is created in the virtual memory space assigned to a certain machine, this machine is said to be the *home node* for the object and its constituent regions. This allows us to quickly determine the home node of a given object and saves us communication costs at every memory allocation to reserve global address space. One problem with this memory layout is that, with 32-bit addresses, we quickly run out of address space. If every machine is given 64 Mbytes of home memory the number of machines that fit into the address space is $2^{32}/2^{26} = 2^6 = 64$ machines. However, this solution still scales well enough to be able to tackle large problem sizes and simplifies our design considerably. Solutions to this and other problems have been proposed in [16]. A simpler solution, however, would be to use 64 bit machines.

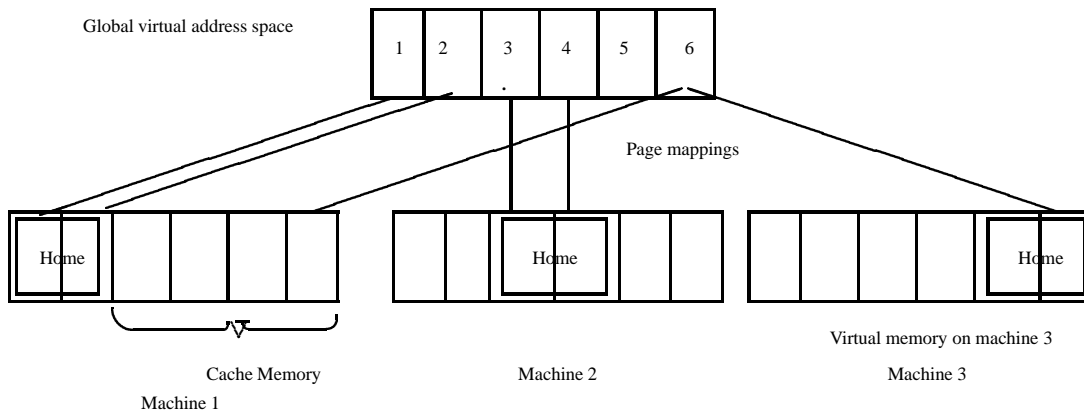


Figure 1: Global memory layout for the 3 machine case.

To ensure that there is always space for a region at its home node, half of the physical memory of a node is reserved for storing global pages owned by that machine. The other half of physical memory is used to map in virtual memory pages owned by other processors (the address space not marked “home” in Figure 1). This part of the physical memory thus acts as a region cache. At present processors are not allowed to allocate more memory than they can store in the home part of their physical memory. As a result there always will be physical memory available for a given region when it has to be flushed out of a region cache.

The first time a processor uses a global virtual address for which it is not the home node, it will get a segmentation fault. The machine will then check the validity of the faulting address, map the page pointed to in and zero it. Thereafter, execution continues from where the segmentation fault occurred. The software access check (where the segmentation fault occurred) will see the region header zeroed, which indicates that the region pointed to is not cached locally. The machine will therefore contact the home node of the region. The home node will retrieve the region using our DSM protocol.

If a machine runs out of physical pages when trying to map in a page (a segmentation fault occurred, a foreign address was used), the garbage collector is run, so pages that no longer contain regions are evicted.¹ If there are still not enough pages for the request, pages with currently unused regions are evicted and flushed back to their home nodes and the pages are unmapped. If later the page is reused, a segmentation fault occurs which will bring in the page, just as described above.

¹An alternative would be to swap pages to disk. On high-speed networks, however, flushing to the home node is faster than writing to disk. Half of the physical memory of each node is always mapped in, so pages can always be flushed to the home node.

3.3 Region management

With fine-grained DSMs, the accesses to shared data have to be checked in software. Accesses to shared data are surrounded by either a pair of (start_read, end_read) calls or by a pair of (start_write, end_write) calls. Multiple processes can simultaneously execute a (start_read, end_read) action on a given region, but only one process can execute a (start_write, end_write) action. The calls also execute DSM protocol code that make sure the region being accessed is available locally after a start operation.

We use the same DSM protocol as CRL and similar access checks. The DSM protocol is a home node, directory based invalidation protocol. We, like CRL, employ a sequential memory consistency model. Exact information about the coherency protocol can be found in [6]. The communication substrate used is LFC [3] on Myrinet [4] running on a cluster of workstations (Pentium Pro’s at 200 MHz).

The region’s state information is stored in front of the object (see Figure 2), so it can be found quickly. Next, pointers to all region pointers are stored in a list. As an example, Figure 2 shows a 2.5 Kbyte object that is split up into two regions of 1 Kbyte and one region of 512 bytes. All access checks (start_write, start_read, end_read, end_write) thus require two parameters: the start of the object and the address to be read from/written to. If the granularity of subdivision of an object into regions is not already known, the home node is asked for it. We can then locate the correct header using a simple calculation.

As an example, when a processor wants to read from the object of Figure 2 at position 1.5K, first a call is made to start_read(object start, address of read). In start_read we then locate the correct region header and invoke the “start read” state handler for that region. The

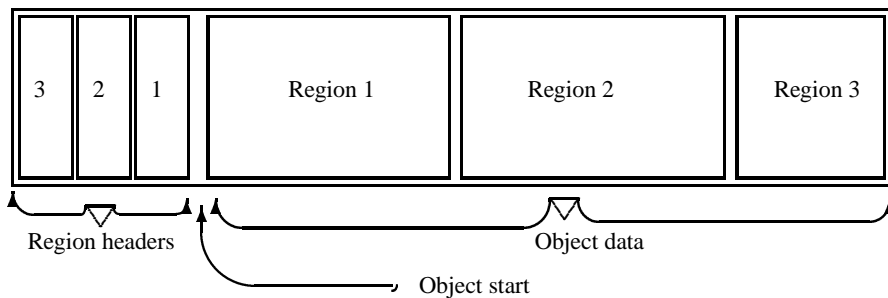


Figure 2: Layout in memory of a 2.5 Kbyte object.

DSM then fetches a copy of that region and stores it at the appropriate address. When the action is finished with the region, it calls `end_read` to release the region.

4 Compiler support

The compiler instruments every reference to global data with calls to `start-read`, `end-read`, `start-write` and `end-write`. A simple example is shown in Figure 3. Here the write access to “*this*” (i.e., the current object) needs to be instrumented with calls to `start_write` and `end_write`. As can be seen in this small example, access check overhead can be quite large for simple methods.

An important goal of the compiler is to reduce the overhead of the software access checks as much as possible. However, there are several caveats. Minimizing the number of access checks may not always result in the best performance, since it can reduce the amount of parallelism available in a method. Also, Java promotes the use of many small methods, where little optimization can be performed. This may be different for other languages (e.g., Fortran and C).

At present we perform two optimizations to reduce the number of access checks: we pull access checks out of loops and remove `end-start` pairs on the same region.

Pulling access checks out of loops is implemented by using a form of DU (Definition-Use) chains, the effects of which are shown in Figure 4. If a definition of a variable (i.e., a place in the program where the variable obtains a value) is outside the loop, we can safely pull the access-check out of the loop. If the definition is inside the loop, the access check must also be left inside the loop (potentially another object is instantiated and assigned to that variable).

Note, that when an array access check is pulled out of a loop, the compiler must determine which regions of the array are accessed and insert code around the loop to pull those regions to the local machine. If compiler analysis fails, either the entire object is pulled in (all regions of the array are subsequently locked for read/write) or the access check is left inside the loop.

```
// Sample Java code to be instrumented:
class A {
    int a;
    void foo() {
        a = 2;
    }
}
// And in i386 assembler:
A__foo0:
    pushl %ebp           ; save frame pointer
    movl %esp,%ebp      ; set new frame pointer
    pushl %esi           ; save register esi
    movl 8(%ebp),%esi    ; register esi := this
.L22:
    pushl %esi
    pushl %esi
    call start_write    ; start_write(this, this)
    movl $2,24(%esi)    ; this->a := 2
    pushl %esi
    pushl %esi
    call end_write      ; end_write(this, this)
    addl $16,%esp       ; clean up arguments
.L23:
    popl %esi           ; restore register esi
    leave               ; restore stack frame
    ret                 ; return to caller
```

Figure 3: Example Java code and the corresponding instrumented assembly.

```
s = set of pointers in loop that are never
    changed in loop
for each p in s do
    if p never used for a write in loop then
        output_start_read(p),
        tag_outputted_for_read(p)
    else    output_start_write(p)

compiled code for the loop ...

for each p in s do
    if tagged_for_read(p) then
        output_end_read(p)
    else output_end_write(p)
```

Figure 4: Code generation for loops.

```

class TestArray {
int [] a = new int[1000000];
int SumAndAdd(int lb, int ub) {
\\ compiler inserts start_write_entire_array(a);
    int sum = 0;
    for (int i=Map(lb);i<Map(ub);i++)
        sum += a[i];
    for (int i=Map(lb);i<Map(ub);i++)
        a[i] += sum;
\\ compiler inserts end_write_entire_array(a);
    }
}

```

Figure 5: Combining access checks is not always advantageous.

Another important optimization is read/write combining. This means that we can, if one statement has a *start-read/end-read* on x , and the next a *start-write/end-write*, combine them into a single *start-write, end-write*. The same holds for two consecutive *start-read/end-read* or *start-write/end-write* blocks for the same region, which we can combine into a single read or write block. This optimization is not always advantageous since it may at times reduce concurrency. Consider the example in Figure 5, where the access checks on the shared array 'a' can be pulled out from both loops and combined to result in a *single start-write, end-write* pair. Because, in this example, it is impossible to statically determine the array sections accessed the entire array is locked for write. Currently our solution is that access checks that originate from a loop are not combined with other access checks.

Sometimes access check lifting is illegal. Consider the Java code in Figure 6. The problem is that a write access check behaves like a mutex. When write access checks are combined with normal (Java) locks, deadlocks can be introduced by too much code lifting. In this example, if the access check on 's' is pulled out of the synchronized statement and out of the loop (FooOptimized), and in another piece of code (FooUnoptimized) the situation is reversed, deadlock might occur as follows. First one machine acquires a lock on 'lock' and another machine acquires the write access lock on 's'. Next they both attempt to acquire the other lock/write access lock and deadlock has occurred. The solution to this problem is to forbid access check lifting over synchronized statements.

Accesses to the stack are always local and are never instrumented. Global variables (static fields in Java) are distributed round robin using their memory addresses as indication of machine address. Every access to a global variable is instrumented with calls to $\{read,write\}\text{-global_variable}$. At present, global variables are not cached, but stored on a single machine.

```

// This is thread is run on multiple
//machines concurrently
Class ShowMe extends Thread {
static Object lock = new Object();
Stats s = Common.getStatsObject();
void FooUnoptimized() {
    for (int i=0;i<N; i++) {
        int tmp = calc();
        synchronized (lock) {
// compiler inserts:    start_write(s, s);
            s.sum += tmp;
// compiler inserts:    end_write(s, s);
        }
    }
}
void FooOptimized() {
// (optimized) compiler inserts start_write(s, s);
for (int i=0;i<N; i++) {
    int tmp = calc();
    synchronized (lock) {
        s.sum += tmp;
    }
}
// (optimized) compiler inserts: end_write(s, s);
}
}

```

Figure 6: Pulling access checks out of synchronized blocks makes for illegal code.

5 Java-specific issues

In building a Java DSM several Java-specific issues have to be addressed, including distributed garbage collection and exception handling.

5.1 Garbage collection

In Java, garbage collection (GC) is part of the language specification and our software DSM thus has support for it built in. The garbage collector used is a blocking, parallel version of mark-and-sweep. The problem with distributed garbage collection is that a machine may be running several threads which may all be updating objects, adding and removing references.

Our garbage collection algorithm is shown in Figure 7. A GC phase is only triggered if a node's local memory is full (malloc fails for the local heap). When this occurs, execution of all threads on all machines is stopped. After all machines have acknowledged that they have indeed stopped (a barrier is reached), all machines simultaneously start the garbage collection process. Other distributed shared memory algorithms can be found in [15] and [8].

Note that we speed up the garbage collection process by checking the cache of globally cached memory so we can reduce the size of the deferred list.

If a pointer to global memory is used and the memory is not mapped in yet (a segmentation fault oc-

```

step 1, stop execution of all threads on all machines
step 2, all incoming messages are queued
    2a, wait for all machines to reach step 2a (barrier)
step 3, every node, in parallel, starts marking its
    root-set (even in globally cached memory)
    The root set for every node is {global pointer variables +
    local thread-stacks}

    3a) if a pointer 'p' points to an address owned by cpu proc
        and its page is mapped and the region is valid,
        append p to scan_list[proc] and 'mark' the object p points to.

    3b) for proc != myproc do
        send(proc, scan_list[proc]);

    3c) redo step 3a, for every pointer on the deferred list either
        received from others or from scan_list[my_proc]

step 4a, wait for all machines to reach step 4a (barrier)
    4b, unmarked objects can now be deleted on all machines.
    4c, when a machine locally removes an object p, it appends
        p to a the remove_list.
    4d, now that local GC is complete, tell all machines to zero
        all region headers on remove_list (broadcasts remove_list)
step 5a, wait for machines to finish step 4 (barrier).
    5b, restart all threads and dequeue all queued messages.

```

Figure 7: Distributed garbage collection algorithm.

curred), we have to map the memory in and zero it. If, however, we no longer have any physical pages left (mmap fails), we have to free pages. Freeing a page, however, is legal only if it is unused (no used regions reside on it). We can create unused regions by starting a global GC phase. The basic algorithm is shown in Figure 8.

5.2 Exception handling support

Exceptions in Java can transfer control out of the current method, to the nearest catch statement in the call chain. If an exception is thrown in between a *start_read* and an *end_read*, then the *end_read* will not be executed. Our solution to this problem is to maintain a stack of stack pointers with every region. When executing a *start_read* or a *start_write*, the current stack pointer is pushed on the region's stack pointer stack. While unwinding the stack to find a catch block for the exception, a check is made to see if the unwind might cross a *start-end* block. This is done by checking all stack-pointer stack tops of all regions. If so, the *end_read* or *end_write* operation is called, which will pop an entry off the stack-pointer stack. An example of this behavior is shown in Figure 9. Here the throw from 'foo' will, before jumping to the catch block in bar, call *end_read(x)*. To be able to find the regions used, a list of regions is maintained (both for regions that are created locally,

and those that are cached).

6 Status and future work

The Jackal compiler is reasonably stable and a distribution of the compiler is planned for Spring 1999. We have implemented a prototype DSM system and the compiler support for it. We already are able to run several application programs on top of our DSM protocol using a 128 node PentiumPro/Myrinet cluster as testbed (i.e. without language support). We plan to enhance the compiler's optimizer to further reduce the number of access checks and to make the access checks themselves cheaper. Other work in progress is to enable the use of the hardware MMU in coordination with the compiler to completely remove access checks where advantageous, to let the compiler decide region sizes per object/array, and to implement region prefetching. We also plan to do more array access analysis to check how many regions should be retrieved when pulling an access check out of the loop (the current implementation fetches the entire object).

7 Conclusion

A (Java) compiler can successfully target a CRL-like DSM and optimize programs to improve their performance. A proof of concept is given for the Java

```
* If a node's global memory cache is full
(no more physical pages available, to
map the memory in)

step 1, start a GC phase.
step 2 find pages whose contents are regions
       which are no longer in use, these can
       be unmapped.
step 3, flush old pages back to the home node
       if more space is needed.
```

Figure 8: Running out of pages

```
class TestExceptions {
int x;
void foo() { throw new Exception(); }
void faa() {
//      start_read(this_pointer),
//      generated by the compiler
int b = x; // results in the start/end
//      read of the 'this' pointer.
    foo();
//      end_read(this_pointer),
//      also generated by the compiler
}
void bar() {
    try { faa(); }
    catch (Exception e) {
// this will catch the exception from foo()
    }
}
}
```

Figure 9: Exceptions and regions

programming language. Some obstacles were encountered in fully supporting the Java language (exceptions, locks), but these can be satisfactorily solved without programmer intervention.

The result of this work will be a Java compiler that can run multithreaded Java programs (from source code) in parallel on a distributed-memory machine, without any modifications.

References

- [1] Download site for SUN-JDK. <http://java.sun.com>.
- [2] H.E. Bal, R.A.F. Bhoedjang, R. 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] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.
- [4] 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.
- [5] Mark D. Hill Ioannis Schoinas, Babak Falsafi and David A. Wood James R. Larus. Sirocco: Cost-effective fine-grain distributed shared memory. pages 40–49, Paris, France, October 1998. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT).
- [6] K.L. Johnson. *High-Performance All-Software Distributed Shared Memory*. PhD thesis, Laboratory for Computer Science, MIT, Cambridge, MA, December 1995. Technical Report MIT/LCS/TR-674.
- [7] K.L. Johnson, M.F. Kaashoek, and D.A. Wal-lach. 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.
- [8] Mark Stuart Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, Austin, TX, December 1997.
- [9] 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.
- [10] J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, and A. Plaat. An Efficient Implementation of Java’s Remote Method Invocation. In *ACM SIG-PLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.
- [11] M. W. Macbeth, K. A. McGuigan, and Philip J. Hatcher. Executing Java Threads in Parallel in a Distributed-Memory Environment. In *Proc. CAS-CON’98*, pages 40–54, Missisauga, ON, 1998. Published by IBM Canada and the National Research Council of Canada.
- [12] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, pages 1225–1242, November 1997.
- [13] M.C. Rinard, D.J. Scales, and M.S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26(6):28–38, June 1993.
- [14] 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.
- [15] Kenjiro Taura and Akinori 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*, pages 18–21, Las Vegas, NV, June 1997.
- [16] Paul R. Wilson and Sheetal V. Kakkad. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Addresses on Standard Hardware. In *Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992. IEEE Press.
- [17] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. In *ACM 1997 PPOPP Workshop on Java for Science and Engineering Computation*, June 1997. http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/finalps/17_yu.ps.