Source-Level Global Optimizations for Fine-Grain Distributed Shared Memory Systems

R. Veldema[†] R.F.H. Hofman[†] R.A.F. Bhoedjang[*] C.J.H. Jacobs[†] H.E. Bal[†]

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

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

ABSTRACT

This paper describes and evaluates the use of aggressive static analysis in Jackal, a fine-grain Distributed Shared Memory (DSM) system for Java. Jackal uses an optimizing, *source-level* compiler rather than the binary rewriting techniques employed by most other fine-grain DSM systems. Source-level analysis makes existing access-check optimizations (e.g., access-check batching) more effective and enables two novel fine-grain DSM optimizations: *object-graph aggregation* and *automatic computation migration*.

The compiler detects situations where an access to a *root object* is followed by accesses to subobjects. Jackal attempts to aggregate all access checks on objects in such *object graphs* into a single check on the graph's root object. If this check fails, the entire graph is fetched. Object-graph aggregation can reduce the number of network roundtrips and, since it is an advanced form of access-check batching, improves sequential performance.

Computation migration (or function shipping) is used to optimize critical sections in which a single processor owns both the shared data that is accessed and the lock that protects the data. It is usually more efficient to execute such critical sections on the processor that holds the lock and the data than to incur multiple roundtrips for acquiring the lock, fetching the data, writing the data back, and releasing the lock. Jackal's compiler detects such critical sections and optimizes them by generating single-roundtrip *computation-migration* code rather than standard data-shipping code.

Jackal's optimizations improve both sequential and parallel application performance. On average, sequential execution times of instrumented, optimized programs are within 10% of those of uninstrumented programs. Application speedups usually improve significantly and several Jackal applications perform as well as hand-optimized message-passing programs.

1. INTRODUCTION

Software fine-grain Distributed Shared-Memory (DSM) systems store shared data in small memory regions that are managed independently by a software cache coherence protocol. This approach avoids false sharing, a common problem in DSM systems, but introduces overhead in the form of *software access checks* that detect absent or invalid cache entries. Most current fine-grain DSM systems use binary rewriting tools to insert these checks [14, 25, 26]. Since the overhead of such checks is considerable, these systems analyze the binary code they rewrite and optimize by combining and removing checks.

This paper investigates an alternative approach that we have used in the implementation of Jackal, a fine-grain DSM system for the Java programming language. Jackal uses extensive *source-level* static analysis to optimize performance. This approach has two advantages: it makes existing access-check optimizations more effective and it enables new optimizations.

Existing access-check optimizations become more effective because Jackal's source-level compiler can differentiate between types, can compute the set of called functions at each call site, and because analysis and optimization can take place before complex pointer arithmetic is introduced by the compiler's backend optimizer. Jackal's global analyzer computes the possible targets of virtual function calls and can therefore expose more code to access-check optimizations that are based on common-subexpression elimination and redundancy analysis. In addition, the compiler uses array-aggregation analysis, an extended form of access-check batching [25] that aims to combine checks on individual array elements into a single check for the entire array. This can improve sequential performance substantially and additionally reduces the number of network roundtrips required to access array elements.

Jackal's source-level analysis also enables two new fine-grain DSM optimizations: *object-graph aggregation* and *automatic computation migration*. Object-graph aggregation is the pointer equivalent of array aggregation. Through static interprocedural data access analysis, the Jackal compiler detects situations where an access to some object (called the *root object*) is always followed by accesses to subobjects. In that case, the system views the root object and the subobjects as an *object graph*. Jackal attempts to aggregate all access checks on objects in such a graph into a single access check on the graph's root object. If this check fails, the entire object graph is fetched, which can reduce the number of network roundtrips. Object-graph aggregation, therefore, has the potential to improve both sequential and parallel performance.

Computation migration is used to optimize certain common types of critical sections. Java knows no explicit, programmer-specified association between data and the lock that protects it, although each Java object has a lock associated with it. It is, however, common to protect object data with the same object's lock. A barrier object, for example, might consist of two counters, and be accessed by synchronized methods that lock the barrier object and update a counter. When such an object is updated in a critical section, the updating thread typically incurs multiple network roundtrip delays, because it must acquire the lock, fetch the data, write the data back, and release the lock. With Jackal's computation-migration optimization, only a single roundtrip is needed to execute this type of critical section. Instead of executing the normal protocols for acquiring a lock and faulting in the object data, the thread that executes the synchronized code conceptually migrates to the processor that owns the object, executes the critical section, and returns. In reality, Jackal's compiler generates a separate function for the synchronized block of code. This function is invoked as a Remote Procedure Call (RPC). The compiler determines which live variables must be packed into the RPC request to build a correct execution environment.

To assess the impact of Jackal's analyses and optimizations, we compare, on the identical hardware, the performance of Jackal applications with the performance of equivalent message-passing applications on a high-performance RMI implementation. Using message passing, the programmer has almost complete control over the communication, so RMI applications must be considered hand-optimized in comparison with the compiler-optimized Jackal applications. Our performance results are encouraging. On average, the sequential performance of our applications is within 10% of sequential code without access checks. The use of computation migration, array aggregation, and object-graph aggregation always improves the parallel performance of Jackal programs, usually significantly, by reducing the number of messages they send. Two out of four Jackal programs perform as well as, or better than the equivalent RMI applications.

Our contributions are:

- We show that existing optimizations for fine-grain DSM systems can be implemented more aggressively using global, source-level analysis.
- We discuss a compiler optimization to aggregate access checks for object graphs; this improves sequential performance, and reduces network roundtrips.
- We discuss a compiler optimization to selectively do computation migration; this may reduce network roundtrips.
  Both these optimizations depend on source-level code information and global and interprocedural analysis.
- We present a performance analysis of the resulting system for some applications and compare them against a highly optimized message-passing implementation [20] (RMI) on the same platform.

The paper is structured as follows. Section 2 summarizes Java's memory model. Section 3 describes Jackal and its implementation. Section 4 discusses the application of static analyses to existing fine-grain DSM optimizations. Sections 5 and 6 describe Jackal's object-graph and computation-migration optimizations, respectively. Section 7 studies Jackal's performance on a Myrinet-based cluster. Section 8 discusses related work. Section 9 concludes the paper.

2. JAVA'S MEMORY MODEL

We briefly summarize Java's memory model; for a detailed description we refer to the language specification [10] and Pugh's critique of the memory model [22].

The memory model allows each Java thread to cache variables in its *working memory*. A thread's working memory must (conceptually) be flushed to *main memory* at each synchronization point. A synchronization point is a lock (unlock) operation that corresponds to the entry (exit) of a synchronized block of code. Lock and unlock operations must flush a thread's working memory, but an implementation is allowed to flush sooner, even after every write operation.

In contrast with entry consistency [3], Java's memory model does not couple locks to specific objects or fields. In particular, different fields of one object may be protected by different locks, so that those fields can be updated concurrently without introducing race conditions. This programming model is more flexible than entry consistency, but makes it more difficult for a DSM implementation to combine synchronization and cache-coherence traffic. Competing memory models, such as HLRC as implemented in Treadmarks [17] combine synchronization messages with *write notices*, but still incur multiple roundtrips for simple critical sections.

3. IMPLEMENTATION

Jackal consists of an optimizing Java compiler and a runtime system (RTS). The compiler translates Java sources into Intel x86 code rather than Java bytecode. The compiler generates software access checks; the optimizations to reduce the number and cost of these checks are the main topic of this paper.

The runtime system distributes Java threads in a round-robin fashion over idle processors and implements Jackal's cache-coherence protocol. While this protocol is described in [23], we review it here for self-containedness.

3.1 Coherence Protocol and Access Checks

Jackal's coherence protocol allows processors to cache a *region* created on another processor. A region is either a complete Java object or a slice of an array. To reduce false sharing, arrays are partitioned into contiguous but independent 256-byte regions. Since we believe false sharing within objects is uncommon, objects are never stored in multiple regions. The processor that allocates a region is called the region's *home node* and always provides storage for the region. (In terms of Java's memory model, a region's home node provides that region's *main memory*.)

Each region occupies a virtual-address range in a single, shared address space. A region's home node and the caching processors all store their copy of a particular region at the same virtual address. The shared address space is divided into P disjoint parts, where P equals the number of processors. Each processor allocates memory only in its own partition, so that computing a region's home node from its virtual address amounts to one divide operation.

Jackal employs an invalidation-based, multiple-writer protocol that combines features of HLRC [32] and TreadMarks [17]. Jackal does not use a single-writer protocol because it would force the compiler to mark the *end* of each read/write operation, which reduces the opportunity to lift access checks. In addition, end markers increase sequential overhead. As in HLRC and TreadMarks, modifications are flushed to a home node and twinning and diffing is used to allow concurrent writes to shared data.

The run-time data structures related to the coherence protocol are shown in Figure 1. All threads on one processor share one copy of a cached region. Java's memory model, however, requires that each thread maintain its own cache-state vector for each region. Each thread therefore maintains a *present* and a *dirty* bitmap, each of which contains one bit per 64 bytes of heap. Objects are 64-byte aligned to map a single object to a single bit in the bitmap. To reduce memory usage, pages for these bitmaps are allocated lazily.

The present bit in a thread T's bitmap indicates whether T retrieved an up-to-date copy of region R from R's home node. A dirty bit in thread T's bitmap indicates whether T wrote to region R since it fetched R from its home node.

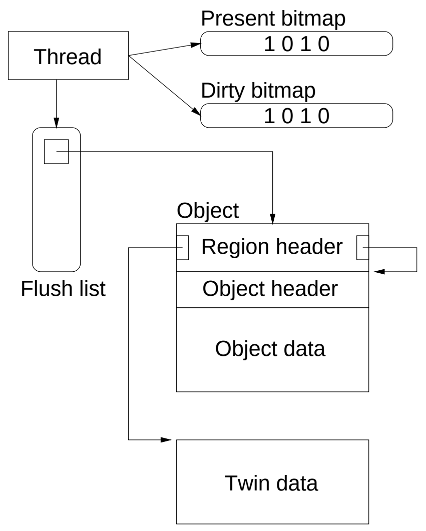

Figure 1: Runtime system data structures.

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

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

The protocol described so far invalidates and possibly flushes *all* data in a thread's working memory at each synchronization point. Since this potentially leads to much interprocessor communication, our implementation uses an optimized version of this protocol that still adheres to the memory model. In particular, it is not necessary to invalidate and flush a region that is accessed by only a single processor or that is only read [19]. Jackal's lazy flushing is evaluated and described in detail in [23].

3.2 Synchronization

Logically, each Java object contains a lock and a condition variable. Since threads can access objects from different processors, Jackal provides distributed synchronization protocols. Briefly, an object's home node acts as the object's lock manager. To acquire a lock, a thread sends a lock request message to the lock manager and waits. If necessary, the lock manager creates a thread that waits until the lock is released. When the lock is available, the manager replies with a grant message, otherwise a thread needs to be created to wait for the lock to be released. To unlock, the lock holder sends a unlock message to the home node.

```
if (condition) {
   read-check(a)
} else {
   write-check(a)
}
write-check(a)
```

Figure 2: Access check removal

4. CONVENTIONAL OPTIMIZATIONS

To improve performance, Jackal compiler removes superfluous access checks, aggregates regions and objects and co-allocates threads and regions. Although array aggregation and fully redundant access-check removal are not new, the versions implemented in Jackal are generally stronger than those implemented in for example Shasta [25], due to the heavy use of interprocedural and global analysis.

4.1 Removing Access Checks

Jackal's compiler frontend adds access checks to all heap accesses. Since these access checks add considerable runtime overhead, the compiler's backend optimization passes try to remove as many checks as possible. An access check consists of six x86 instructions. If two access checks are sufficiently close to each other to allow common subexpression elimination, one or two instructions can be eliminated (parts of the address calculations).

Since it is a source-level compiler, Jackal can perform interprocedural and global program analysis. Systems based on binary rewriting cannot, in general, perform interprocedural analysis: they cannot handle indirect calls (which includes the virtual function calls found in C++ and Java) or indirect jumps (introduced by switch statements compiled to jump tables). The frontend of Jackal's compiler, in contrast, can determine sets of virtual-function call targets (by selecting all methods from all derived classes that implement some method prototype) and maintain label lists for switch statements. This information is passed on to the compiler backend which uses it to remove access checks.

The compiler's backend uses forward interprocedural data-flow analysis over a program's control-flow graph (CFG) to remove access checks. An access check for address *x* at program point *p* can be removed if *x* has already been checked on *all* paths that reach *p*, but only if none of these paths contains a synchronization statement. Figure 2 illustrates this redundancy analysis. First, a forward data-flow pass discovers that all access checks in the example operate on the same object *a*. Since *a* is checked on all paths that lead to it, the third check is redundant. The *read-check* is replaced by a *write-check* to preserve correctness.

Note that the current implementation needs full redundancy for check elimination. Partial redundancy elimination (PRE / LCM [18]) would reduce the number of checks even further. In the example, PRE could potentially remove all access checks and place a single access check in front of the if statement. The difficulty in implementing PRE for access check elimination lies in lifting the address calculations (the 'a' in the example) while not violating Java's precise exception model.

The analysis outlined above is complicated by Java's ability to extend classes at runtime by loading bytecode from network or file. The Jackal RTS supports compilation and loading of bytecode at runtime [20]. If dynamic class loading is allowed, the compiler may inspect potential callees at a call site only if the complete set of callees can be determined at compile time. (This is the case if a static, final or private function is called.) Otherwise, the compiler conservatively assumes that at least one of the callees contains a synchronization point.

To increase the effectiveness of interprocedural analysis, the Jackal compiler can be instructed to make a *closed-world assumption*, which means that dynamic class loading is guaranteed not to occur in the program that is being compiled. If the compiler is allowed to make this assumption, function inlining becomes more effective as well. Access-check removal benefits from inlining, because it makes more code (including access checks) available to the optimizer.

The compiler can also use call-graph information to remove access checks to the *this* pointer. When all calls to a function originate from either a constructor or a *Thread.run*() method and the compiler is allowed to make the closed-world assumption, then we can safely remove all checks on *this* (until a synchronization point is seen), because we know that we are executing this function on the home node.

The compiler (by default) also does escape analysis [6]. Escape analysis detects which objects will never escape the thread that created them (or their creating function's life time). When such an object has been detected, *all* checks to the object can be removed and the object can be allocated on the stack. To generalize our escape analysis, we do not actually allocate the objects on the call stack but maintain a separate object stack per thread. This allows objects created in constructors to outlive their creating constructor by not restoring the object stack at a constructor's exit.

When such an object is passed to another method and that call site is the only location that method can be called from, (again making a closed world assumption), access checks to that parameter are removed as well (recursively). In the future, we intend to make the analysis stronger still, by specializing called functions when escaped objects are passed.

4.2 Array Aggregation

If array elements are accessed in a loop, the access checks to array elements may sometimes be lifted out of the loop, and be replaced by an aggregate array slice access check before the loop. The benefits are improved sequential run time and increased data throughput by allowing streaming of multiple array regions.

Array access check lifting is impossible if the loop is unsafe in that it contains synchronized blocks or calls to methods that (may) contain synchronized blocks. Interprocedural analysis is important to detect safe methods.

If a loop is safe and the compiler finds an access check to an array element inside a loop, the relation between the indexing expression and the loop control variables is analyzed. If the check does not depend on the loop control variables, it is lifted from the loop.

If the array indexing expression is loop dependent, we try and lift the array check. Loop dependence is defined here as dependent on a loop induction variable which is a tuple {variable, start value, step, multiplicand, modulo} as used in strength reduction. This allows complicated strides as the loop induction variable tuple is passed whole to the runtime system. If the bounds of the index can be precomputed as a function the loop control expressions, an aggregate check to fetch the required array slice is inserted before the loop, and the access check within the loop are removed.

If the index expression is not loop independent but the index bounds cannot be calculated, a call to the RTS is inserted before the array to request the whole array.

A trade-off is that when the rule to determine array aggregation misfires, parts of the array that are not used are also mapped in. This might for example happen when the array access check inside the loop is located inside an "if" statement. Also, when the loop iterates only a few times (say 2 or 3 times) over an array which has

```
class Nested {
  int x;
}

class Outer {
  Nested n = new Nested();
  int y;

  int get() {
    return y + n.x;
  }

  static public void main(String[] arg) {
    Outer o = new Outer();
    int x = o.get();
  }
}
```

Figure 3: Object-graph aggregation example.

already been mapped, the costs of calling the runtime system may be larger than with the inlined assembler check.

5. OBJECT-GRAPH AGGREGATION

Object-graph aggregation is the pointer equivalent of array aggregation. A faulted region frequently contains references to objects that are subsequently accessed, causing more cache misses. The compiler detects such access patterns, and replaces access checks on the individual objects in the *object graph* with a single access check on the graph's *root object*. If the check fails, the entire graph is retrieved. Like array aggregation, this improves sequential and parallel performance by, respectively, eliminating access checks and reducing the number of roundtrips required to fault in related objects.

Figure 3 illustrates this idea. The compiler detects that the invocation of method *get()* on object *o* involves not only an access check on *o*, but also on *o.n*. An object graph is identified, rooted at *o*, that has an edge to one other node, the allocation of the *Nested* object. In *main()*, the access check on *o* is replaced with a aggregate access check that validates both *o* and *o.n*. The access check on *o.n* in *Outer.get()* is removed.

To detect related objects, the compiler creates a heap approximation [9, 28] in which objects are identified by their allocation site. Since the algorithm cannot distinguish between different run-time objects that are created at the same allocation site, we will speak of *allocations*, which represent all objects allocated from that site. The algorithm produces a list of allocations, and, for each reference field of the corresponding objects, a list of allocations that the reference points to.

The algorithm consists of an outer iteration that stops when the allocation list and reference lists no longer change. Within an iteration, the code for all available methods is traversed. For each register or memory reference in the instruction stream, a list is maintained of allocations that it may point to. When an allocation site is encountered for the first time, it is registered in the allocation list. The reference that results from any allocation is propagated through the instruction stream, following assignments, and through method calls, following parameter passing. When an assignment of an allocation *A* to a reference field *f* of an object corresponding to allocation *B* is encountered, *A* is added to the reference list of *B.f*. Arrays of objects are special. If the compiler cannot compute the value of an index expression for such an array, then no distinction is made between different elements of the arrays: if any element refers to an allocation, all elements are considered to refer to it. When the algorithm stabilizes, each variable and memory reference has been associated with a set of allocations that it may point to.

A heap approximation is a graph whose nodes are the allocations, and whose edges represent a reference from a field of the source allocation to the target allocation. In the graph, we search for allocations that are the only entry point to a directed acyclic subgraph: there should be no other paths in the code by which any allocation in the DAG is accessed. When such a subgraph is detected, access checks to its interior nodes can be removed and the whole subgraph may be transferred when an access check to its root fails.

To be a candidate for aggregation, the subgraph must not contain any 'forbidden' objects; this includes *Thread* objects and objects on which a *lock*, *unlock*, *wait* or *notify* call is invoked. Aggregating objects in the latter (synchronization) category would cause thrashing.

Since DAGs may contain nested DAGs, a trade-off emerges in the marking of object graphs. We must choose an ideal object-graph depth. If the object graph is too deep, large portions of the object graph may be left unused after they have been faulted in. If the threshold is set too low, unnecessary cache misses will occur. Currently, Jackal requires the programmer to set the object graph threshold at compile time.

Another trade-off occurs when an object graph is only modified at its leaves; without object-graph aggregation only those leaf objects would be flushed, while the interior nodes would be in read-only mode, allowing lazy flushing. With object-graph aggregation the entire object graph will be marked dirty, causing the entire object graph to be flushed and re-fetched. This extra communication must be traded against the gains in sequential code speed and bulk data transfer.

6. COMPUTATION MIGRATION

Computation migration [13] is a mechanism that moves part or all of a computation and its state to the data used by that computation. This may be more efficient than moving multiple data objects to the computation or than repeatedly moving the same data object to the computation, which occurs typically with e.g. barrier objects. Jackal uses computation migration in two ways: to combine data and synchronization traffic and to co-locate the execution of a thread constructor with the data created by that constructor.

Java programmers do not indicate which locks protect which data items. This makes it difficult to combine data and synchronization traffic: Jackal may have to communicate multiple times to acquire a lock, to access the data protected by the lock, and to release the lock. Frequently, however, the data stored in a shared object is protected by that object's lock. Figure 4 illustrates this case. Recall that the home node of an object acts as the manager of the lock that is part of the object (see Section 3.2). Without computation migration, the execution of *inc()* therefore results in three network roundtrips to *x*'s home node if the caller of *inc()* is not on *x*'s home node: one roundtrip to acquire *x*'s lock, one to fetch *x*'s data, and one to flush the modified data, and a final message to release *x*'s lock.

Jackal's compiler detects and optimizes synchronized blocks such as the one in Figure 4. Under the conditions described below, the compiler will generate a separate function for such a synchronized block. This new function will be run on the synchronization object's home node and will be invoked as a Remote Procedure Call (RPC) by the thread that calls *inc()*. Stack variables in the calling code that are live just before the RPC call is issued are copied into the RPC's request message and act as parameters to the new

```
int inc(AtomicInt x, int delta) {
  int old;

  // migrate following block
  synchronized(x) {
    old = x.count;
    x.count += delta;
  }
  return old;
}
```

Figure 4: Computation migration example.

function. The compiler identifies these live variables automatically. Any stack variables that are still live at the end of the migrated computation (i.e., *old*) are marshaled into the reply that is sent to the caller. In this scheme, the execution of function *inc()* requires only a single roundtrip to the home node. The request message contains the value of parameter *delta*, which is live in *inc()*. At the home node, the lock is acquired, *x.count* is incremented, and the lock is released.

As an extra optimization, the compiler checks whether any object pointers are used after the synchronized statement. If the node that ran the migrated computation is also the home node for any of these objects (or object graphs), then it will piggyback those objects or object graphs on the reply. This optimizes common patterns such as the retrieval of an object or an object graph from a shared data structure.

The computation-migration optimization is triggered only when the code inside the synchronized block can be (conservatively) proved to terminate in finite time. The execution time of the synchronized block is estimated by counting instructions and considering loop nesting. If a synchronized block is deemed to be long-running, the optimization will not be triggered, to preserve load balance.

Jackal's compiler applies computation migration also *Thread*-object constructors. Such constructors are often all executed by a program's main thread to create additional worker threads. It is frequently the case that the data accessed by a worker thread is allocated in the worker thread's constructor. If these constructors all run on the same processor, all data objects created will have that processor as their home node, even though the new threads will be run on other processors. The compiler therefore replaces *Thread*-object allocations and constructor calls with an RTS call that ships the constructor to run at the same processor that the thread will run on. This ensures that the new thread and all objects it creates reside on the same processor. This will save mapping in data that typically belongs with the thread. Co-allocating threads and objects in this way is difficult for a binary instrumentor, because it requires knowledge of the Java class and interface hierarchy.

7. PERFORMANCE

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

Jackal was configured so that each processor has a maximum of 32 Mbyte of local heap and 32 Mbyte of cache for caching regions allocated by other processors. In all of the performance evaluations below, the compiler has been instructed to make a closed world assumption.

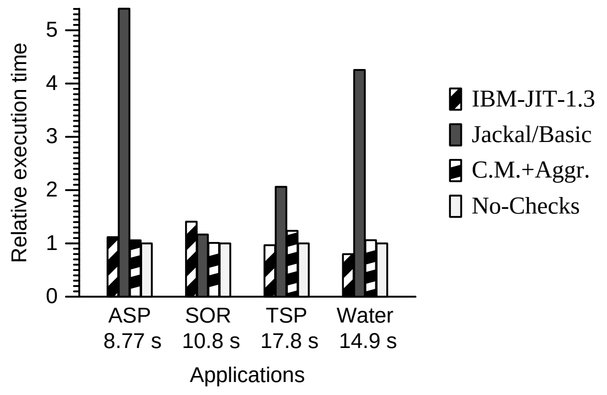

Figure 5: Relative sequential execution times.

7.1 Microbenchmarks

We measured the overhead of access checks by timing the traversal of a 10,000-element linked list on a single processor both with and without access checks. This benchmark performs one access check per list element. On average, each access check costs 0.049 μs, or approximately 10 cycles.

To determine the latency of object transfer, we measured the time needed for one processor to traverse a linked list of empty elements created by another processor. The measured time includes the roundtrip time to the home node and the costs for page mapping, adding regions to the traversing thread's flush list, and locking. We measured an average latency of 35 μs per node of the linked list.

The throughput obtained by transferring a large array of integers is 5.1 Mbyte/s. This throughput is independent of array size, because arrays are retrieved region by region. If array aggregation is enabled, the array is transferred in units of 768 bytes (to fit in one 1024 byte LFC packet), which results in a throughput of 24 Mbyte/s.

We also measured the impact of object-graph aggregation by microbenchmark tests. A sequential program without access checks traverses an array of 1000 binary trees, each consisting of a root and two empty subtrees, in 0.95 μs per tree. With unoptimized checks, traversal takes 139 μs per tree; this time also includes the time to register the objects in the thread's flush list. With object-graph aggregation enabled, traversal takes 46 μs per tree: access checks to the subtrees are removed. In a parallel test program, one machine creates the tree, the other traverses it. With unoptimized checks, this costs 274 μs per tree; three roundtrips per tree are necessary. Enabling object-graph aggregation reduces this to 124 μs per tree; only one roundtrip per tree is necessary.

7.2 Application Suite

Our application suite consists of four multithreaded Java programs: ASP, SOR, TSP, and Water. Besides the multithreaded, shared-memory versions of these programs, we also wrote equivalent message-passing versions of these programs. These programs all use our fast Remote Method Invocation (RMI) to communicate explicitly. Section 7.4 compares the performance of the multithreaded programs running on Jackal with the performance of these RMI programs.

7.3 Sequential Application Performance

Figure 5 shows the relative sequential performance of all applications. All applications were compiled and run with Jackal's native Java compiler and with version 1.3 of IBM's Just-In-Time (JIT) compiler. To factor out JIT compilation time, each application iterates four times. We assume that all compilation takes place during the first iteration and report the average execution time of the last three iterations.

For Jackal, we compiled each program in three ways:

- with access checks, without compile-time access-check optimizations (Jackal/Basic)
- with access checks, with all compile-time access-check optimizations (Jackal/C.M.+Aggregation)
- without access checks (Jackal/No-Checks)

The figure shows the execution time of all configurations relative to Jackal/No-Checks. The absolute execution times in seconds of this version are given on the horizontal axis.

In general, Jackal's compiler generates good sequential code. Version 1.3 of IBM's JIT compiler is the fastest JIT compiler currently available [7, 27]. With the exception of Water, Jackal/No-Checks outperforms the JIT compiler.

Adding access checks without optimization increases the sequential execution times by 222% on average. Jackal's access-check optimizations reduce this overhead on average to 9%. In the case of ASP and SOR, access-check overhead is almost entirely eliminated by array check aggregation. The inner loop of both algorithms traverses and updates complete matrix rows. This is detected by the compiler, which lifts all access checks out of the inner loop. The overhead of the remaining, lifted access checks is negligible.

In the case of TSP, unoptimized Jackal performance is bad because the majority of the work is done in a small recursive function that (initially) contains seven access checks. The object-graph aggregation optimization discovers that all these checks can be removed from the recursive function. The checks can either be removed because they check the *this* pointer, or they can be lifted to the initial caller of the recursive function and combined into two object-graph aggregate checks for the partial path and for the city distance table. The recursive function is a virtual method, so a binary rewriter cannot remove the access checks.

In Water, the data structure that is operated on in the inner loop is a two-dimensional array, which in the unoptimized version requires twice four array access checks and ten array element checks for each inner loop iteration. Together, array aggregation and object-graph aggregation reduce this to one check within the inner loop and one aggregate outside the inner loop.

7.4 Parallel Application Performance

This section compares the application performance of various Jackal configurations and an equivalent, hand-optimized RMI program. The RMI programs use a highly optimized RMI implementation [20] and run on the same hardware and communication platform (LFC) as Jackal. An RMI roundtrip on this platform costs 38 μs, maximum array throughput is 54 MByte/s. Both the Jackal and the RMI programs were compiled using Jackal's Java compiler. RMI gives the programmer almost full control over the communication that takes place. In particular, the programmer can combine data and synchronization messages, transfer entire arrays in a single roundtrip, and transfer multiple objects in a single roundtrip. In Jackal, these optimizations can be performed only after the compiler has discovered the opportunity of performing them.

The data set for each application is small. Fine-grained applications show RTS and access check overhead much more clearly than coarse-grained applications, which communicate infrequently. The differences for the various optimizations come out markedly; also, the comparison with the RMI implementation becomes extremely competitive, since the RMI implementations must be considered hand-tuned.

Below we discuss the performance of each application using Fig-

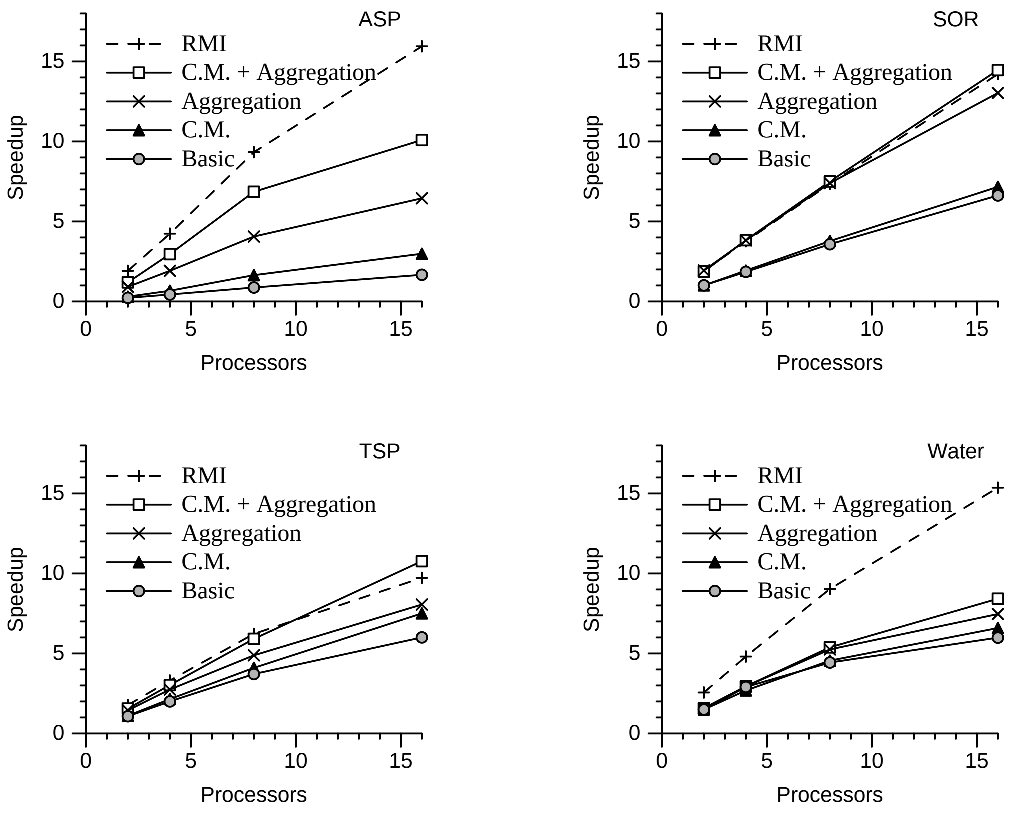

Figure 6: Jackal and RMI speedups.

ure 6 and Figure 7. Figure 6 shows speedups for the RMI programs and for four Jackal configurations. All speedups are relative to the sequential Jackal/No-Checks execution time. We separately consider the effect of two optimizations: computation migration (synchronized blocks) and aggregation (arrays and object graphs). Computation migration of *Thread* constructors is always enabled except in the Basic configuration, since Jackal's performance severely degrades without it.

Figure 7 shows the message counts and network data volumes on 16 processors.

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

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

Jackal's computation migration optimization significantly increases speedup since it allows the combining of data with the synchronization messages. Fetching a broadcast row now costs only one round-trip. With array aggregation enabled, the compiler detects that the inner loop operates on whole rows, so the access checks in

Jackal optimization	ASP	SOR	TSP	Water
Basic	29.0	20.9	28.3	338.8
Computation Migration	15.3	19.9	16.5	329.9
Aggregation	27.3	1.8	19.8	13.4
C.M. + Aggregation	15.3	1.0	8.4	12.2
RMI	15.1	2.3	4.5	15.5

Jackal vs. RMI data message counts * 1000

Jackal optimization	ASP	SOR	TSP	Water
Basic	39.3	4.6	40.9	32.8
Computation Migration	0.1	3.1	7.0	12.7
Aggregation	36.1	5.4	41.0	22.1
C.M. + Aggregation	0.1	4.1	5.0	14.9

Jackal control message counts * 1000; RMI has no control messages

Jackal optimization	ASP	SOR	TSP	Water
Basic	15.9	2.4	0.5	3.9
Computation Migration	14.3	2.3	0.3	3.7
Aggregation	15.7	2.6	0.4	5.2
C.M. + Aggregation	14.3	2.5	0.2	5.1
RMI	14.4	2.4	0.2	4.7

Jackal vs. RMI data volume in MByte

Figure 7: Jackal and RMI communication on 16 CPUs

the inner loop are replaced by one array aggregate check before the inner loop. A completed row is faulted in at once for each outer loop iteration. The speedup of the RMI program remains better because it uses asynchronous forwarding of rows in its spanning-tree broadcast. An alternative RMI implementation with synchronous forwarding gave a speedup no better than the Jackal version.

The data volume and number of messages exchanged are comparable for the RMI version and the Jackal version with computation migration and aggregation.

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

The Jackal version of SOR attains excellent speedup. This is entirely due to Jackal's array aggregate optimization. As in ASP, the Jackal compiler discovers that it can combine all access checks in SOR's innermost loop into a single check for the whole row. When this check fails, the entire row is sent to the requesting processor. As a result, Jackal achieves comparable speed-up, message count, and data exchange volume as the RMI implementation. The RMI message count consists of an empty request and a reply carrying the row; in Jackal, the request is counted as a control message, only the reply carrying the row is counted as a data message.

TSP. TSP solves the Traveling Salesman Problem for a 15-city input set. First, processor zero creates a distance table to hold the distances between each city and a list of job objects, each containing a partial path. Next, a worker thread on every processor tries to steal jobs and complete partial paths from the list. The cut-off bound is encapsulated in an object that contains the length of the shortest path discovered thus far. To avoid non-deterministic computation (which may give rise to superlinear speedup), the cut-of bound has been set to the actual minimum for this data set. Consequently, the speedup attained by all implementations is much lower than in the case of initially unbounded search, where linear speedup is typical.

The speedup numbers for the configurations with object-graph aggregation disabled suffer from inferior sequential code, see Section 7.2. Without computation migration, faulting in a new partial path takes more data messages (for the Job object and the partial path contained in it) and many more control messages: some for requesting the regions, and some for synchronization control. With all optimizations enabled, a partial path is obtained with one roundtrip, where the migration reply carries the object graph for the partial path. The RMI program and the optimized Jackal programs transmit approximately the same amount of data.

Water. Water is a Java port of the Water-n-squared application from the Splash benchmark suite [29]. The program simulates a collection of 343 water molecules. Each processor is assigned a partition of the molecule set and communicates with other processors to compute inter-molecule forces.

Most communication in Water stems from read and write misses on *Molecule* objects and the sub-objects referenced by them: force, acceleration, and position vectors for each atom, which are stored in separate arrays and are written only by their owner thread.

The RMI version requests the slices of the Molecule array it needs in each outer iteration in one RMI from each other processor. In the unoptimized Jackal version, the objects that make up the Molecules are faulted in one at a time inside the inner loop. Consequently, the unoptimized Jackal program makes many more roundtrips than the RMI program to retrieve all molecule data.

Jackal's loop analysis and heap analysis succeed in determining the access patterns, so that slices of Molecule arrays are obtained with one request for each other processor. However, each object that makes up a Molecule must be managed separately upon arrival, flush and protocol change. Moreover, changes to and from read-only state of the Molecules are frequent during the iterations, and this gives rise to a number of protocol messages that is proportional to the number of objects that make up the Molecules.

In the future, Jackal's compiler will perform object inlining analysis which would transform each Molecule into one object in stead of four. To assess the performance impact, we did the same transformation by hand. The speedup on 16 processors then becomes 11, and the amount of protocol messages is divided by four. The speedup is still less than RMI's, which must be attributed to more procotol overhead. We do not show this improved performance in Figure 6, since we explicitly want to avoid application-level tuning for Jackal.

8. RELATED WORK

Most DSM systems are either page-based [17, 19] or object-based [2, 3, 15]. Jackal manages pages to implement a shared address space in which regions are stored. For cache coherence, however, Jackal uses small, software-managed regions rather than pages and therefore largely avoids the false-sharing problems of page-based DSM systems. Like page-based DSMs supporting release consistency, we use twinning and diffing, albeit not over pages but over single objects.

Shasta [25] and Sirocco [14] use binary rewriters to insert and optimize software access checks. Such binary rewriters have, in general, less information than a source-level compiler. In particular, interprocedural analysis in the presence of virtual-function calls is difficult or impossible for binary rewriters. Jackal uses such interprocedural analysis to optimize access checks. In addition, Jackal uses type information to find groups of related objects. It would be interesting to compare the performance of both approaches. Unfortunately, such comparisons are, at present, difficult, due to the differences in the programming languages, the memory models, and the hardware platforms.

Some DSM systems have successfully used compiler-generated data-access hints. Dwarkadas et al.[8] describe a system that uses regular-section analysis [11] for explicitly parallel (Fortran) programs [8]. Their compiler analysis identifies producer-consumer relationships; these are then used to generate hints that allow the underlying TreadMarks DSM system to aggregate data and synchronization messages. The analysis, however, applies only to arrays, whereas Jackal also supports heap-allocated structures.

Object-based systems like CRL [15], Midway [3], and Orca [2] might seem a natural way to implement distributed shared memory for an object-oriented language like Java. These systems, however, use a form of entry consistency [3] and therefore couple locks to objects. In Java, each object conceptually contains a lock, but this lock can protect any piece of shared data, not just the data stored in the object that contains the lock. Since this is a very loose association between lock and object, Jackal has to work harder to combine data and synchronization traffic because no programmer assistance is available. Implementing computation migration requires compiler support, both to detect opportunities and implement computation migration.

Compiler-supported computation migration is used in Prelude [13] and Olden [24]. (Like Jackal, both these systems use compiler support to find the live variables that must be shipped to the remote processor.) Both Prelude and Olden use computation migration to enhance locality by eliminating multiple roundtrips to remote data objects (using programmer annotations). In Jackal, caching and aggregation are completely compiler detected and enforced.

Compiler supported object inlining achieves the same effect as our tree prefetching optimization, although our technique is more widely applicable as it allows aliasing between objects and variable sized arrays. Object inlining has been explored for example in [16].

MCRL [12] uses computation migration in a DSM system. Like CRL, it requires the user to encapsulate reads and writes to shared data between calls to the RTS. Read operations run on the local processor; write operations run on the home node of the data. CRL [15] automatically combines synchronization and data transfer because it uses an entry-consistent memory model.

Several other DSM systems focus on Java. Java/DSM [31] implements a JVM on top of TreadMarks [17] and is therefore page-based. Jackal, in contrast, uses software access checks and a small coherence unit. DOSA [30], derived from Treadmarks, is an approach to build a DSM on top of modern object-oriented languages. It does so by using their strict type systems to implement memory consistency. They add a layer of indirection around object references to allow objects to move addresses upon receiving a write fault. However, it implements little compiler based optimization.

Hyperion [21] adds access checks to Java bytecodes. In contrast with Jackal, Shasta, and Sirocco, however, Hyperion is sensitive to false sharing, because it caches all shared Java objects, including arrays, in their entirety.

cJVM, like Jackal, strives to provide a complete single system image [1] using transparent RPC to access remove objects. It does so by creating a cluster-aware JVM using transparent RPCs to access remote objects where Jackal uses object caching by default. cJVM differs from our system in that is performs no object-graph aggregation nor does it implement a tradeoff between data and function shipping.

9. CONCLUSION

We have described and evaluated aggressive, source-level compiler optimizations for a fine-grain DSM system. Two classes of optimizations are key: aggregation optimizations and computation-migration optimizations. Aggregation optimizations aggregate access checks on multiple, related data objects — arrays or object graphs — into a single check that faults in all related objects in a single cache-protocol transaction (typically one roundtrip to a home node). Aggregation optimizations reduce the average access-check overhead in the sequential executions of our Java programs to less than 10%. At the same time, these optimizations reduce the number of roundtrips required to retrieve up-to-date copies of shared regions, and therefore also improve parallel-application performance. Computation migration is used to combine synchronization and data traffic, as in entry-consistent DSMs, but without requiring an entry-consistent programming model.

Both optimizations rely heavily on extensive static analyses, including interprocedural analysis, heap approximation, escape analysis, etc. We have implemented these analyses and the optimizations enabled by them in a DSM for the Java programming language, but most of the analyses and optimizations are general and can be applied to other languages. The analyses, however, cannot easily be performed by a binary rewriter, because they rely type information, the ability to calculate the set of called functions at each call site, and the absence of complex pointer arithmetic.

Two out of four of our optimized applications perform as well as, or better than, equivalent, hand-coded message-passing programs and *all* optimized applications perform significantly better than versions that do not employ aggregation or computation migration.

10. REFERENCES

[1] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *Proc. of the 1999 Int. Conf. on Parallel Processing,* Aizu, Japan, Sept. 1999.

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

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

[4] R. Bhoedjang, K. Verstoep, T. Rühl, and H. Bal. Evaluating Design Alternatives for Reliable Communication on High-Speed Networks. In *ASPLOS*, pages 71–81, Nov. 2000.

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

[6] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *OOPSLA'99*, pages 1–19, Denver, CO, Nov. 1999.

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

[8] S. Dwarkadas, A. Cox, and W. Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *ASPLOS'96*, Oct. 1996.

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

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

[11] P. Havlak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, 1991.

[12] W. Hsieh, M. Kaashoek, and W. Weihl. Dynamic Computation Migration in DSM Systems. In *Supercomputing '96*, Pitssburgh, PA, Nov. 1996.

[13] W. Hsieh, P. Wang, and W. Weihl. Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems. In *PPoPP'93*, pages 239–248, May 1993.

[14] M. H. I. Schoinas, B. Falsafi and D. W. J.R. Larus. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *PACT*, pages 40–49, Paris, France, Oct. 1998.

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

[16] A. A. C. Julian Dolby. An automatic object inlining optimization and its evaluation. In *PLDI 2000*, pages 345–357, Vancouver, BC, Canada, June 2000.

[17] P. Keleher, A. 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, Jan. 1994.

[18] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion.

ACM SIGPLAN Notices, 27(7):224–234, 1992.

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

[20] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *PPoPP*, pages 173–182, May 1999.

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

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

[23] R. Veldema, R. Hofman, R. Bhoedjang and H.E. Bal. Runtime-optimizations for a Java DSM. In *ACM 2001 Java Grande Conference*, pages 89–98, San Francisco, CA, June 2001.

[24] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Trans. on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

[25] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *ASPLOS*, pages 174–185, Oct. 1996.

[26] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain Access Control for Distributed Shared Memory. In *ASPLOS*, pages 297–306, Oct. 1994.

[27] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[28] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *OOPSLA(14)*, pages 187–206, Denver, CO, Nov. 1999.

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

[30] Y.C. Hu, W. Yu, D. Wallach, A.L. Cox, W. Zwaenepoel. Run-time Support for Distributed Sharing in Typed Languages. In *Proceedings of Languages, Compilers, and Runtimes for Scalable Computing*, May 2000.

[31] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, Nov. 1997.

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