Compiler and Runtime Optimizations for Fine-Grained Distributed Shared Memory Systems

VRIJE UNIVERSITEIT

Compiler and Runtime Optimizations for Fine-Grained Distributed Shared Memory Systems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op 2 oktober 2003 om 13.45 uur
in het auditorium van de universiteit,
De Boelelaan 1105

door

Ronald Sander Veldema

geboren te Amsterdam

promotor: prof.dr.ir. H.E. Bal

Acknowledgements

First acknowledgments go to Rob van Nieuwpoort for his work on the initial RMI implementation and to Jason Maassen for getting the first implementation of the garbage collector working for the RMI runtime system. Other people I would like to acknowledge are Ceriel Jacobs for his help with debugging LASM and many other things and Rutger Hofman for his help fixing and molding the Jackal runtime system and for pinning down some bugs in the compiler. Many thanks go to Henri Bal and Dick Grune, my supervisors, for helping me through my years of AIO-ship, getting this thesis out of the door and especially Henri Bal for acquiring the DAS (and its successor DAS2) at the VU. A special thanks goes out to Raoul Bhoedjang for helping me with the initial stages of AIO-ship. Finally, I would like to thank the other members of my reading committee (Phil Hatcher, Luciana Arantes, Henk Sips, Raoul Bhoedjang and Dick Grune) and Rutger and Ceriel for their useful comments on this thesis.

This work is supported in part by a USF grant from the Vrije Universiteit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Goals of This Thesis

The goals of this thesis are to study compiler and runtime optimizations that allow multithreaded shared-memory Java programs to execute efficiently and unmodified on a distributed-memory machine (e.g., a cluster of workstations). We study the performance impact of several new optimizations in the context of Jackal, an efficient runtime system implementing a Distributed Shared Memory (DSM) system and a compiler that aggressively re-arranges both code and data to increase efficiency.

This thesis focuses on three components: Java, compilers, and DSM systems. Although Java semantics are assumed for many optimizations described here, most of the techniques are language-independent. All relevant parts of the aforementioned components will be discussed individually. Knowledge is assumed of basic multithreaded programming techniques, but this topic will also be briefly introduced. Basic compiler technology, such as data flow techniques, will also be briefly touched upon.

Because a basic understanding of the underlying runtime system is needed to understand the performance impact of certain compiler optimizations, a global overview of the runtime system is given as well.

1.2 Motivation

Java is a language and runtime system for creating portable programs. Although initially envisioned as a language for Internet computing, Java is also used extensively for general and scientific programming. Scientific problems are often very resource (computational or memory) intensive, requiring parallel or distributed execution to make their computations feasible at all in practice. To address this requirement, Java defines standards for both parallel and distributed execution.

Machines to run parallel applications on come in two varieties: using shared or distributed memory. Shared memory machines are relatively easy to program, as the machine itself will manage all communication between the individual processors. The added complexity also makes these machines difficult to build, but the resulting machine typically shows good speedup with increasing numbers of processors. Unfortunately, the size of bus-based shared memory machines is usually limited to a few tens of processors because of hardware scaling problems (with each processor added the bus-bandwidth has to increase). NUMA (Non-Uniform Memory Access) shared-memory machines scale to larger numbers of processors, but these machines use special hardware, making them expensive.

Distributed machines can be very cheap. They can, for example, consist of off-the-shelf commodity PCs connected by fast networks. Unlike shared memory machines, they require the application to handle and optimize all communication between processors. This is a difficult and error prone task, as the programmer has to consider many details. For example, the programmer has to think of

- categorizing data as local to a machine or global for the system,
- maintaining coherence between cached copies of data on different machines,
- FIFO-ness of message queues,
- multithreading or state machines to handle incoming messages,
- message upcall handlers,
- synchronization messages and data messages,
- flow-control,
- message combining,
- etc.

To reduce this complexity, Distributed Shared Memory (DSM) systems provide a simulation of a shared memory machine on top of a distributed machine. Such software thus provides the application programmer with all the ease of programming of a shared memory machine while executing on top of a cheaper distributed machine. All of the above problems associated with a distributed machine are then taken care of by the system.

Software DSM systems come in two basic varieties: coarse grained and fine grained DSMs. Coarse grained systems transfer large quantities of data over the network whenever a processor detects an access to data that is not locally available (fault detection). The size of a chunk of data generally equals the processor's hardware page size which is typically in the range of one to sixteen kilobytes. Usually there is no longer any relation between the programmer's code and data structures and those of the underlying DSM system. Both operate independently of each other.

Software fine-grained DSMs operate on small quantities of data such as single cache lines or single programmer defined data structures. Fine-grained DSMs generally require no processor support to function but instead require programmer or compiler support to detect non-local data accesses. The advantages of using a fine-grained DSM are that the programmer or compiler has more control over the system's behavior by stating the action to be taken at each remote data access.

1.3 Java's Parallel-Programming Support

An important advantage of Java is that the behavior of memory is explicitly specified by the Java Language Specification (JLS[21]). For example, the JLS specifies what the exact semantics of a for loop are, when a variable can be considered to be initialized and when not, and most importantly for this thesis, it specifies the exact semantics of how memory behaves both in the context of a single thread and when multiple threads are accessing memory.

The Java environment contains two mechanisms for parallel execution: Remote Method Invocation (RMI) and Java threads. RMI forces the programmer to explicitly code transfers of information from one machine to another machine by means of explicit remote procedure calls. This forces the programmer to do all the work of keeping the data between machines coherent if some distributed data structure is to be maintained. Using threads on the other hand, all communication is implicit as shared data can be kept in main memory and the compiler, runtime and processor(s) cooperate to keep main memory consistent.

Although the programmer has to do extra work to create an RMI program compared to a multithreaded program, he gains the ability to tune the performance of the parallel program. For example, RMI allows a programmer to tune program performance by creating smart encodings of the data transmitted over the network and to limit the number of times threads communicate by doing as much as possible with a single RMI.

To illustrate the advantages and disadvantages of threads and RMI over each other we will use the examples in Figure 1.1 and Figure 1.2 for the remainder of this introduction. The problem is that of a counter that is to be increased by multiple processors. Figure 1.1 shows the RMI solution, while Figure 1.2 shows the multithreaded solution.

As can be seen, the RMI program is larger than the multithreaded program, initialization alone requires a number of support classes which increases complexity. Also, fault detection is explicit in the use of the multiple try and catch statements and exception specifications that some methods contain (the *"throws RemoteException"* lines). The multithreaded program does not contain any code to handle faulty networks and machines. If any machine in the cluster were to fail, the program would be forced to stop. The RMI program could conceivably handle such situations.

Another potential disadvantage of an RMI program is its static program startup: the user of the program has to manually start the program at multiple processors. This makes it difficult to move objects from one machine to another and reduces location transparency of a program.

To start up the RMI version of the counter program, the Server object's main function has to be invoked at the machine denoted by the *HOST_URL* variable. Afterwards, the Client's main function has to be invoked at some number of machines. Program startup in the multithreaded case is much simpler: the main method of the *MyThread* object is started at some processor in the system (decided by for example the operating system or a runtime system). Additional threads are then started by creating a number of instances of the *MyThread* class and invoking their start methods. The Java runtime system will then create new threads and invoke the run method inside the new thread of control.

The RMI program also explicitly states processor identity; see the lines concerning *"HOST_ID"*. The multithreaded program simply starts virtual processors at will and has no knowledge of the processor executed on.

Most importantly, however, message sends are explicit in RMI: *"c.inc()"* will be invoked at the machine hosting *"c"*, a message *will* be sent at that method invocation.

```
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

interface CounterInterface extends Remote {
   public int inc() throws RemoteException;
}
class CounterImpl extends UnicastRemoteObject
       implements CounterInterface {
  int x;
  CounterImpl() throws RemoteException {
  }
  synchronized public int inc() throws RemoteException {
    x++;
    return x;
  }
}
class Client {
  public static void main(String arg[]) {
    try { CounterInterface c = (CounterInterface)
              Naming.lookup(Server.HOST_URL);
          while (true) {
             System.out.println("counter = " + c.inc() );
          }
    } catch (Exception e) {
      System.out.println("Error: " + e);
    }
  }
}
class Server extends LocateRegistry {
  final static String HOST_URL = "rmi://my_host/cnt";
  public static void main(String arg[]) {
    try { createRegistry(Registry.REGISTRY_PORT);
          CounterImpl c = new CounterImpl();
          Naming.bind(HOST_URL, c);
          Thread.sleep(10000);
     } catch (Exception e) {
       System.out.println("Error: " + e);
     }
  }
}
```

Figure 1.1: RMI example.

```
class Counter {
  int x;
  synchronized int inc() {
       x++;
       return x;
  }
}

class MyThread extends Thread {
  Counter c;
  MyThread(Counter c) { this.c = c;  }
  public void run() {
       while (true) {
           System.out.println("counter = " + c.inc() );
       }
  }
  public static void main(String arg[]) {
       Counter c = new Counter();
       int nr_of_threads = Integer.parseInt(arg[0]);
       for (int i=0;i<nr_of_threads;i++) {
           MyThread mt = new MyThread(c);
           mt.start();
       }
  }
}
```

Figure 1.2: Multiple threads adding to a single counter.

Also, for RMI methods the calling convention changes from call-by-reference in normal methods to call-by-value in RMI methods. The multithreaded program contains no such restrictions: communication is completely hidden from the programmer and is performed by memory accesses and (eventually) performed as a side effect of the *synchronized* keyword as prescribed by the Java Memory Model (JMM). The actual communication behavior (in terms of fields of objects sent back and forth to main memory) of the multithreaded program is therefore unpredictable and can be factors higher or lower than that of the RMI program. This makes performance predictions difficult on a distributed-memory machine and thus makes a multithreaded program more difficult to optimize by hand.

To support atomic operations, Java supports the synchronized block which acts as a monitor. A synchronized block accepts a single object reference parameter with an associated lock structure. Before entering a synchronized block, a thread must acquire the lock. When leaving the synchronized block, the thread releases the lock. While one thread has the lock, other threads trying to acquire the lock are blocked until that thread releases the lock. A shorthand notation for putting an entire method's body in a synchronized block is by tagging the method using the synchronized keyword creating a synchronized method.

In a multithreaded program a thread often needs to wait for some condition to occur, for example, when all threads have reached a certain state or an important state change has occured (i.e. shared variable update). To support this behavior without the thread having to actively poll for a condition to occur, Java supports the *wait*, *notify* and *notifyAll* methods available in each object. *Wait* waits for another thread to signal the object waited on using either *notify* or *notifyAll*. The difference between using *wait* and actively watching a condition is that *wait* does not consume processor cycles. It operates in conjunction with the threads-package to suspend the current thread.

1.4 Jackal: A High Level Description

To enable parallel execution on a distributed machine such as a cluster of workstations, we have built a special compiler and associated runtime system (RTS) named *Jackal*. *Jackal* translates Java source code into an executable program suitable for distributed execution. Jackal is a compiler and runtime system that together create a fine-grained, compiler supported DSM with object caching as its primary method of implementation.

Jackal takes advantage of the freedom the Java language offers to a compiler in the translation process. Before each object access, the Jackal compiler adds code (*access check*) that checks if the object is available locally. If not, the object is fetched from the remote machine holding the "master" copy. After the invoking thread has made its modifications to the object, the object is merged with the master copy. A thread signals to the runtime system that it has finished making its modifications by acquiring or releasing a lock. The merge is performed by applying the differences between the modified copy and an extra copy to the master copy. The extra copy of the object is made when the object first arrives at the modifying machine.

Using the system as described above we can already execute unmodified Java programs on a cluster of workstations. However, without proper additional optimization, the performance achieved will be inferior compared to hand optimized distributed (message passing) programs because of the overhead added by insertion of the access checks and because only a single object will be transferred over the network at a time. Therefore, compiler optimizations are needed to reduce the number of access checks executed and to perform prefetching and message combining.

Jackal employs two optimizations to reduce the number of access checks performed and two mechanisms to perform prefetching. Prefetching is used to fetch multiple objects at a time (see Section 1.6). Care must be taken, however, that only those objects are prefetched that will actually be used in the future. If unneeded objects are prefetched bandwidth will be wasted; if too few objects are prefetched, performance will be inferior.

A general problem Jackal's compiler has is that it will often have to make decisions based on static analysis using incomplete information about runtime values of variables and sizes of data structures. The compiler therefore relies on heuristics to guess whether an optimization will be advantageous. This thesis introduces several such heuristics.

The runtime system attempts to further optimize performance by caching data as much as possible by carefully recording which machines in the cluster have cached an object and in which state (read or write) it currently is in. In general, if there is only one accessing machine of an object or all machines are only reading data from an object, the object can remain cached (see Chapter 3).

Because Jackal's compiler uses Java and Java requires the use of a garbage collector, Jackal's runtime system contains a garbage collector (GC). The garbage collector is different from a normal single machine GC as it has to deal with references located at one machine pointing to objects located at another machine.

1.5 General Related Work

There exist several projects that try to make Java more acceptable for scientific computing. In this section we will discuss the most important of these.

Hyperion [41] is closest in idea to Jackal, the system proposed in this thesis. Hyperion, like Jackal, presents the programmer with a distributed shared memory view of the cluster. The Hyperion compiler compiles bytecode to C and passes that on to the systems' C compiler for further compilation to machine code. Whenever a getfield or putfield opcode is encountered in a bytecode file, special code is emitted before the actual object access to ensure that the object is really available. If the object is not available, it is fetched over the network.

cJVM [3], like Jackal, provides a distributed shared memory view of a cluster of computers but goes a step further by attempting to provide a full single system image: I/O is also transparent over the cluster.

Titanium [64], provides a dialect of Java by extending the Java language with immutable classes and using an explicit SPMD model for parallel execution. All references are by default global: they may point to objects on either the local machine or on another machine. The programmer can mark some objects *local* for efficiency. To ensure good sequential performance bytecodes are translated to C for further compilation.

The Do! [39] project consists of frameworks for both shared memory and distributed-memory programming and a preprocessor to replace usages of the shared memory framework by the framework for distributed memory. Programmers write their application as 'plugins' to the shared memory framework whereafter the preprocessor can enable distributed execution by replacing the shared memory framework.

Ajents [28] is a library of Java classes for easy distributed programming. Using the Ajents library the tasks of creating and managing remote objects becomes easier. For example, there are methods to create objects on other machines and call methods upon them afterwards.

JavaSpaces [15] is a library layered on top of RMI to allow distributed persistence and slightly easier programming for certain types of applications. Using JavaSpaces one can add an object to a service whereafter other machines can fetch the object using a form of query.

JavaParty [47] is a project that seeks to provide a partial distributed shared memory: only classes that have been marked with a JavaParty introduced *remote* keyword are remotely accessible. JavaParty is implemented as a source to source translator and thus relies on the efficiency of underlying JVM for performance.

Spar [51] extends the Java language with task and data parallel concepts to allow parallel execution. However, Spar is intended for data parallel applications only. Spar translates Spar/Java source code to C code to ensure good performance.

1.6 Contributions

The main contributions made in this thesis are:

- a technique to combine two arbitrary objects into a larger combined object. This allows larger combined objects to be sent over the network instead of one object at a time and at the same time reduces memory management overheads;

- a technique to, at compile time, recognize static DAG (Directed Acyclic Graphs) data structures inside heap data structures to allow whole DAGs to be sent over the network instead of one object at a time. This technique also reduces the costs of program instrumentation;

- a technique to increase garbage collector efficiency by using cached copies of objects;

- a technique for reducing the costs of thread synchronization by allowing the synchronization code to run at the machine maintaining the lock object;

The first two contributions have the effect of prefetching: multiple objects are in effect transferred over the network with a single network message. The optimizations differ in implementation and side effects. The first optimization physically merges two objects to a new larger object while the second optimization merely informs the runtime system that the group of related objects are to be sent using single network messages. By performing prefetching, we increase performance by removing network latencies that would have been incurred in the future.

The third contribution optimizes the performance of the Global Garbage Collector (GGC) by assuming that cached objects are still reasonably up-to-date. Normally when performing global garbage collection, the distributed object graph of live objects is traversed and the objects found during the traversal are marked. References to objects whose master copy is located at another machine have a message sent to them asking them to continue the marking process there.

By assuming that cached objects are still reasonably valid, we allow the GGC to reduce the number of messages sent because we no longer need to immediately send a message to the machine holding the master copy (by virtue of employing a multiple-writer protocol for objects). For correctness, the references found in the cached copies of objects are not only locally followed but are also buffered and sent to the machine holding the master copy, because it is still possible that the cached copy differs from the master copy. The advantage is that the references found can be buffered and sent in one bulk message to the home nodes of the cached copies.

The fourth contribution is a technique that allows the compiler to transform certain pieces of code into function shipping code (remote procedure call) instead of the default data shipping (object caching) approach. This optimizes performance by allowing some objects to remain cached (instead of needing them to be sent to the remote thread). This optimization is especially effective when objects can be piggybacked on top of the remote procedure call and multiple objects can remain cached.

Finally, our last contribution is an evaluation of all the optimizations proposed: for three out of four applications with some optimization enabled, Java's RMI performance is reached or nearly reached. We also compare Jackal's performance against that of CRL: a DSM system that allows many of Jackal's optimizations to be performed by hand. For some applications RMI's and CRL's performance is reached, for others the speedup achieved by Jackal lies within 5–35% of the RMI or CRL speedup.

1.7 Organization of This Thesis

The global structure of this thesis is to first examine Java's memory model and other distributed shared memory (DSM) systems. This is followed by examining Jackal's operation without any optimization applied, followed by each of the optimizations applied (a chapter each). This thesis then concludes with a performance evaluation and some conclusions about the final system.

In detail, Chapter 2 gives an in-depth overview of the Java memory model, its implications and its relations to other memory models. The Java Memory Model (JMM) specifies how multithreaded Java programs should behave in terms of when one thread can see a modification made to an object by another thread. We also examine the different kinds of DSM systems currently in existence and place our Jackal system in context. Chapter 3 examines Jackal's operation without any optimizations applied. Chapter 4 examines Jackal's compiler, its internal organization and some basic optimization strategies to increase both normal and DSM program execution efficiency. Chapter 5 examines Jackal's garbage collector. Chapter 6 explains how Jackal's compiler recognizes 'object DAGs' and how that information is applied to increase DSM program efficiency. Chapter 7 examines how Jackal's compiler optimizes synchronized blocks of code by program transformations. In Chapter 8 objects are combined to increase DSM and garbage collector performance. Chapter 9 examines the performance of the resulting system and Chapter 10 concludes by drawing some conclusions.

Chapter 2

Introduction to the (Revised) Java Memory Model

One of the primary concerns in Java's design is that a program written in Java should be able to run on a variety of hardware architectures ranging from small embedded devices to large multi-processor machines. This means that the contract between a thread's local memory and global main memory has to be specified accurately while still allowing enough flexibility for optimizations either by the compiler or the architecture.

In the Java Memory Model (JMM) threads are independent virtual processors that can be dynamically allocated under programmer control. Each thread has access to main memory, which is defined as the location where all objects are initially allocated. Each thread also has access to a private cache (working memory) where it may cache data from main memory at its own discretion. The first problem to be addressed is to specify exactly when (if at all) other threads observe changes made by a given thread. On small scale machines such as single-processor PCs, any update made by one thread is immediately observable by all other threads, as threads share all physical resources. On large scale Symmetric MultiProcessors (SMPs) a delay might be observable because a processor may decide to delay the update by waiting for a programmer defined thread synchronization event and only then making the changes visible to the other threads (lazy memory consistency). The latter solution was taken by Sun to allow large machines to run Java efficiently (see Chapter 17 of the Java Language Specification, JLS [21]).

Because the original Java Memory Model (JMM) contained certain unintended side effects and is unnecessarily strict in when threads can observe changes made by other threads in ill synchronized programs [50], the JMM will in the future be deprecated in favor of Java Specification Request 133 (JSR-133) [1]. The original Java memory model is specified using five entities: threads, main memory (with operations load and store), object fields (variables), working memory (with operations read and write) and locks. JSR-133 does not retain the concept of a single main memory but instead uses a more distributed approach that allows the use of hierarchical caches. Jackal implements JSR-

[1] http://www.jcp.org/en/jsr/detail?id=133

133.

In JSR-133 an object is a set of variables (fields) of a certain type, for example integer and floating point. Each field of an object may have some restrictions imposed upon it concerning cachability (a field of an object may be declared volatile). Whenever a thread reads or writes an object field, the field is conceptually copied to the thread's private cache, called *working memory* (which may translate to a thread private hardware cache or to physical processor registers). The variable will remain there, hidden from all other threads, until the thread is instructed to "flush" its working memory to main memory (at each synchronizing operation). At this time, modified variables are copied back to main memory and the cached values are discarded; read-only cached copies are immediately discarded. As a special case, the JMM specifies that at an unlock operation, no discarding is needed, which allows a thread to reuse a value obtained from main memory in the synchronized block without having to contact main memory again. When not using proper synchronization around shared data that is written to, updates to main memory may occur in a "surprising" manner although the writing thread itself will see the updates occur normally.

To allow the programmer to ensure that multiple threads do not concurrently read and write the same variable, the lock and unlock operations implement monitors (besides their action to flush working memory). This means that once a thread has entered a critical section bounded by a *lock(X), unlock(X)* pair (effectively becoming the owner of X), no other threads can enter a critical section guarded by *X* until the owning thread has executed a sufficient number of *unlock(X)'s* to relinquish ownership of *X*.

2.1 Relation to other Memory Models

We first give some definitions. A memory model is what the underlying memory subsystem guarantees to the user program in terms of ordering and timing of observable changes to memory by one thread when changes are made by another thread and by itself. The granularity of addressable units is not a fixed quantity in the discussion below. The Java memory model describes thread actions in terms of actions on individual variables that may be integers, doubles or references, whereas other memory models either specify these properties per individual byte (processor memory model) or per memory management page (DSM memory model). A memory management page often ranges from one to sixteen kilobytes.

The JMM is a *programming-language* memory model while most other memory models are *processor* memory models (scope consistency, lazy release consistency, etc). A programming language memory model is defined at a higher level than a processor memory model, because it deals with programming language storage concepts (objects, fields, variables) rather than processor or operating system storage concepts (bytes, pages). It is up to the language implementor to implement the memory model on top of an existing processor memory model. A difference with some other memory models is that the JMM does not associate data with locks. Entering a monitor does not only guarantee that the locking object is up-to-date with respect to other thread's modifications but all other objects as well.

There are essentially two forms of memory models, each with some variations: strongly and weakly consistent memory models. A strongly consistent memory model guarantees that updates to memory by one thread are (almost) immediately observable by other threads. A weakly consistent memory model guarantees that upon some later (processor/programmer specified) point in the execution of the program, other threads or main memory will be brought up-to-date. Variations on weak consistency memory models are made by allowing partial or total ordering of updates with the extra option of how much to delay the update to main memory.

A memory model (such as the JMM) is implemented by some coherence protocol. A coherence protocol specifies what action will be taken at which time. Two basic types of coherence protocols currently exist: single and multiple-writer protocols. A single-writer coherence protocol allows only a single writer at a time to access a given object. To detect that a write operation has finished, a clearly defined end-write operation has to exist after which another thread can start to write the object. The single-writer protocol allows some flexibility as to how and when reader threads can access the object while a writer is busy. Some implementations allow concurrent readers, a single reader or mutual exclusion between readers and writers (possible because the writer has to explicitly terminate its write operation).

Multiple-writer protocols allow multiple writer threads to concurrently modify objects. The objects' values are reconciled at a synchronizing operation. Like single-writer protocols, multiple-writer protocols allow some flexibility as to how and when reader threads can access the object while a writer is busy. Because multiple-writer protocols do not require the implementor to signal the termination of write operations, no mutual exclusion between readers and writers' read and write actions is possible.

We now look at some existing memory consistency models: sequential consistency, release consistency and its derivative scope consistency and finally how the Java memory model compares to the above.

Sequential consistency [37] is the strictest memory model by putting the most restrictions on the implementors. It requires all observable changes to memory to be made as if the store ordering were some sequential execution of some threads. This allows prefetching but disallows store re-ordering. This requires the use of a single-writer protocol to ensure that serializability of operations is maintained.

Release consistency[33] allows stores (from cache to main memory) to be delayed until a synchronizing action is performed. Release consistency comes in two basic forms, Eager Release consistency and Lazy Release consistency.

Eager Release consistency specifies that stores can be delayed until a release action (unlock) and allows reads to bypass pending stores (but not over a synchronizing action). Nothing happens when an acquire operation (lock) is executed. When a release action is encountered, all delayed stores are applied at once, reducing the number of interactions with main-memory.

Lazy Release consistency[33, 66] uses the lock acquire operation instead of the release operation to synchronize with other threads. It then determines what needs to be written back to main memory by looking at what other threads have read and written. The underlying protocol then flushes data only where needed and potentially only to those threads requiring that data. The release operation is a completely thread-local operation requiring no interaction with main memory.

Scope consistency [26] sends only the updates made within one (dynamic) scope to other threads. Updates made while in an outer scope remain cached. Again, the implementation is allowed to send updates to a variable early and an implementation is also allowed to prefetch over a scope entry point.

Java supports a weakly consistent memory implementable with either a single-writer protocol or a multiple-writer protocol and allows out-of-order stores (a flush of working memory prescribes no order to the resulting writes to main memory). Reads and writes within a single thread are totally ordered which means that within a single thread, the result of a write operation is immediately visible to the next read operation within that same thread. The JMM supports a multiple-writer protocol because multiple threads can write to the same variable simultaneously, updates are potentially delayed up to a synchronization point thus allowing weakly consistent memory, while within a thread's working memory stores are strongly ordered (the writes to working memory must be observed as if they were executed sequentially).

2.2 The Java Memory Model Specification

The old JMM, as defined by Sun, describes the interaction that threads have with memory using seven actions: *read*, *use*, *write*, *store*, *flush* and the thread synchronizing actions *lock* and *unlock*.

- When a thread *T* wants to *use* a variable *V*, it must first create a copy of *V* in its working memory using the *read* operation if *V* is not already available in working memory; its state in *T's* working memory is set to *read-only*.
- When a thread *T* wants to *write* variable *V*, a *write* action occurs creating a copy of *V* in *T's* working memory. Eventually the variable will be *stored* in main memory. While not yet stored in main memory, the variable will be marked *dirty* in working memory.
- When a thread *T* wants to write a variable *V* that was already available in *T's* working memory in *read-only* mode, it is marked *dirty* in *T's* working memory without fetching a new copy of *V* from main memory beforehand.
- When a *lock* operation is executed by some thread *T*, the values of all dirty variables in *T's* working memory are stored in main memory and all variables in *T's* working memory are discarded.
- When an *unlock* operation is executed by some thread *T*, the values of all dirty variables in *T's* working memory are stored in main memory and their states are changed to *read-only*.

An example of how the JMM works in practice is shown in Figure 2.1. To implement the assignment in *foo()*, first *src* is copied from main memory to the cache. Next, a copy of *dest* is created in the cache and the value of *src* in the cache is written to it. During the assignment dest is marked dirty in the cache. At the end of *foo*, *dest* is stored in main memory overwriting any existing value already there. *Dest* will no longer be marked dirty afterwards.

```
class JMM {
    int dest, src;
    synchronized void foo() {
      dest = src;
    }
}
```

Figure 2.1: Example JMM functionality.

A thread *T* need not wait until a synchronizing action to write a *dirty* variable back to main memory. Similarly, a thread need not wait for a synchronization action to remove a *read-only* variable from *T's* working memory.

The JMM also does not require that a variable *V* be copied to working memory immediately preceding the first use or store of *V*. A variable is allowed to be copied earlier, allowing prefetching to *T's* cache. Such prefetching must, however, respect synchronizing actions. However, if an implementation of the JMM can prove that prefetching *V* over a thread synchronization operation will not change observable behavior, the implementation is allowed to perform the prefetch. This might for example occur when it can be proven that no other thread will access that object during the synchronization action or that all threads are only reading from an object. The same holds for flushing variables to main memory; if the implementation of the JMM can prove that not flushing a certain variable *V* will not produce any observable change in program behavior, the variable *V* does not have to be flushed at that synchronizing operation.

Sun created the initial Java Memory Model (JMM) specification, which while widely praised for its effort, contained some critical inefficiencies and unintended side effects [49]. This has been corrected in JSR-133 (Java Specification Request), which is intended to become the replacement for the current JMM in the near future. In this thesis, we will assume the specifications given in JSR-133 for this and any further discussions. From now on, JMM will refer to JSR-133.

JSR-133, while retaining the basic concepts of the original JMM, such as thread level caches with flushing bound to the synchronized statement, releases some of its strictness such as the serializability of operations imposed by the original JMM. JSR-133 also enforces some stricter rules upon fields that are marked volatile, final and static to conform more to the programmer's expectations while trying to make the model easier to understand (one of the most frequent objections against the original JMM). Most importantly, however, JSR-133 no longer maintains the idea of a single main memory where each thread can observe an update made instantaneously but instead states which updates are no longer visible. This allows some threads to observe updates to main memory made by other threads earlier than others.

JSR-133 operates using only five entities: *threads*, *object fields*, their values at a certain point in a program, a *GUID* (ordered Global Unique IDentifier for each dynamic overwrite of a variable) and an overwrite operation when a write overwrites a field, thus creating a new GUID (in JSR-133 terms, a *write* is a tuple of value, GUID and field).

We first provide some definitions. Each thread in the JSR-133 memory model maintains a set *AllWrites* that denotes all writes to all variables from all threads. *AllWrites(V)* denotes the subset of *AllWrites* to only the variable *V*. Each thread *T* also maintains a set, *Overwritten(T)*, which denotes the set of writes thread *T* has executed. A thread cannot see any overwritten values. *Previous(T)* is the set of writes thread *T* has already seen or knows have occurred. This implies that a thread should never be able to observe a value in *Previous(T)* since it contains the writes a thread knows have been overwritten. When an object is created, its variables (meaning its contained fields) are added to AllWrites with their default values and new GUID's. To model the set of overwrites main memory has seen, main memory also has a set *Overwritten(M)* and accompanying set *Previous(M)* to model the previous and overwritten values all threads have written and overwritten by flushing data to main memory.

When a thread T now reads a value *V*, it must by definition come from *AllWrites(V)* as a value must result from either a write or the field's default value. More specifically, we can subtract the set of overwritten values thread T has done itself which results in *AllWrites(V)* − *Overwritten(T)*.

Whenever a synchronization action occurs, main memory *M* and thread *T's* cache are synchronized causing the set *Overwritten(M)* to be added to *Overwritten(T)*, the set *Previous(M)* to be added to *Previous(T)* and vice versa to main memory.

In a properly synchronized program, this ensures that at all times the size of the set *AllWrites(V)* − *Overwritten(V)* is one, meaning that there is only one value to be read for a variable *V*. Under a non properly synchronized program this set may be larger than one, meaning that an arbitrarily old not yet overwritten value may be read (from *Previous(T)*).

The example shown in Figure 2.1 now works as follows. *Src* will contain either its default value (0 as defined by the Java language specification) or it will have been written to by either this thread or some other thread causing the write to be in the *AllWrites* set and optionally the *Previous* and *Overwritten* sets associated with main memory if that other thread has executed a synchronization statement. Upon writing to *dest*, the old value is added to the *Previous(T)* and *Overwritten(T)* sets. When executing the synchronization statement, the per thread sets *Previous(T)* and *Overwritten(T)* are added to the same sets associated with main memory. When another thread now wants to read dest, it can no longer read a write of *dest* from the overwritten set associated with main memory. If there were other statements in the synchronized block in *foo*, they would not be allowed to see any writes to dest in *Previous(T)*.

2.2.1 JMM Incompatibility in Jackal

Jackal does not comply with the original Java Memory Model because threads on a single machine can see each others' updates earlier than they can see the updates from threads on other machines. Jackal only violates the JMM when the programs are not properly synchronized. Figure 2.2 illustrates this problem where all threads are concurrently accessing each other's objects without using synchronization. This example is due to Luc Bouge and has been kindly pointed out to us by Phil Hatcher. It traces both Jackal's and Hyperion's [41] implementation on an "improperly" synchronized program. The trace corresponds to a Java program running with four threads and two processors.

Each column represents a thread (there are two threads per machine), creating a form of a hierarchical cache. The threads on one machine will see updates from each other

	Home(x)		Home(y)	
Time	Thread 0	Thread 1	Thread 2	Thread 3
0	put y (0)		put x (0)	
1		get y (0)		get x (0)
2		put x (1)		put y (1)
3	get x (1)		get y (1)	
4	flush		flush	
5	get x (0)		get y (0)	

Figure 2.2: JMM violation by Jackal, initially x=-1, y=-1.

earlier than threads on other machines. The value between parentheses is the result of the put or get. The problem is that local threads ”see” the local update of a cached value ”early”, before the lock operation forces the value to be sent back to the home node. In this example at t=1, thread 1 observes y=0 while y at home is still set to -1. This violates the JMM, which disallows a thread to see an update made by another thread until the update has been performed on the master copy.

This example violates the original JMM because the original JMM specifies only one main memory. This means that whenever one thread flushes its cache to main memory *all* other threads should observe the new value. Pugh’s proposal for the new JMM does not specify that there be only one main memory, but instead specifies that the values a thread can see for a certain variable be chosen from all non-overwritten writes performed anywhere.

If the programs were “properly” synchronized, this problem would not occur (all accesses to shared data, x and y in this example, would be delimited by locks). It is the local threads’ executing unsynchronized accesses to the variables that causes the problem.

2.3 Extent Of Flexibility of JSR-133

The JMM exactly defines what is allowed and what is disallowed inside a Java memory model implementation, but to comply with Java’s main goal of being able to run on both small and large machines it is quite flexible in what it allows.

Prefetching, i.e. a processor fetching data before the data is needed (reducing the need for a processor to wait for data to arrive at its core), is explicitly allowed. However, such prefetching is not allowed to reach beyond a synchronization barrier. Also, data may be loaded speculatively; when an object’s field is accessed, its neighboring fields may be fetched (and potentially used later on) along with the requested object field. This may occur when a processor prefetches data in cached lines of some fixed size, for example, four, eight or sixteen integer values.

Also, a JMM implementation is allowed to write data back to main memory before a synchronization action has occurred. This corresponds to the case where a processor with limited cache space runs out of room to cache new data and is thus forced to write old cached data back to main memory.

```
class T {
  int a;
  int []b = new int[5];
  void foo() {
    a     = 1;
    b[10] = 2; // throws an exception
  }
}
```

Figure 2.3: Reordering statements not always legal.

There is also a subtle interaction between exceptions and the Java memory model. Consider the example shown in Figure 2.3. When *foo* executes the assignment to *b[10]*, an ArrayIndexOutOfBounds exception occurs. The assignment to field *a should* be made visible in main memory even when the exception is not caught, which would cause termination of the thread with an uncaught exception error. This example also shows a weakness in the JMM; it disallows all kinds of field assignment reordering whenever a statement might throw an exception. When a JVM reorders the assignments to *a* and *b[]* in this example the different results are clearly visible to the programmer and thus incorrect unless the JVM can prove that the access to the *b* array will not throw an exception, which it can't in this example.

Aside from the reordering problem in the example, working memory is not guaranteed to be flushed *at all* in case of an uncaught exception, if the thread throwing the exception never executes a synchronization action. Actual implementations of the JMM, such as the standard, shared memory JVMs from Sun, IBM and others will however flush the working memory of a thread in case of an uncaught exception. Jackal will also flush a thread's working memory in case of an uncaught exception.

2.4 Implementation Strategies: Page-Based Vs. Object-Based DSM Systems

To implement the Java Memory Model on a cluster of workstations, we need to create an illusion of a large shared memory on top of a distributed memory: Distributed Shared Memory (DSM). DSM systems come in several varieties. At one extreme they are implemented completely in hardware, using special memory controllers and fast specialized networks. At the other extreme they are implemented completely in software using inexpensive machines connected by an ordinary network. This thesis focuses on the latter approach. There are three ways to build a distributed shared memory (DSM) system in software:

- runtime-system centric, using the processor's memory management unit (MMU) to detect remote memory usage, creating a *coarse-grained* DSM. Usually the runtime system, upon detecting a remote memory access, will request the data to be brought to the requesting machine. The Runtime System (RTS) manipulates entire pages, so this approach is also known as page-based DSM.

- programming-language centric, using special programming language features to determine when a remote machine's memory is to be accessed. Usually this means marking functions or methods of a language to be invoked remotely (RPC) using function shipping to allow the function call to be shipped to the data accessed. The RTS now manipulates user-defined objects, so this approach is known as object-based DSM.
- compiler-technology centric, using the compiler to add code to a program to detect remote machine memory accesses. This usually involves shipping the data to the machine requesting memory. This approach divides the global address space of a DSM into small, say 32 to 128 byte chunks, called cache-lines (not to be confused with a processor's cache-lines) creating a *fine-grained* DSM.

There are several hybrids between these three approaches, for example, a mix of a page-based and cache-line based DSM has been attempted in Multiview [27]. cJVM [3] mixes elements of an object and cache line based DSM by defaulting to RPC but at runtime trying to optimize by using object caching and equating the object to a cache-line.

By default, Jackal acts as a cache-line based DSM. As an optimization, Jackal tries to use function shipping whenever possible.

2.4.1 Page-based DSMs

Page-based DSMs use the memory management unit (MMU) of a processor to trap accesses to hardware pages that are not locally available. If a page is absent, the processor will invoke an error handling routine implemented by the DSM. The DSM then determines which other machine in the cluster has the correct page and requests it. The machine holding the page receives the request and sends the page to the requestor after doing some administrative actions to denote that there is now a copy of that page (based on the memory coherence protocol used by the DSM). The requestor will then receive that page and map it in at the correct location in its memory space. This approach has been taken in Treadmarks[32], CVM [31] and other systems.

This approach is transparent to the programming language used and thus is unlikely to take advantage of its properties. The basic unit of data transfer in these systems is the hardware page which can range from a single kilobyte to four or sixteen kilobytes. Such a large granularity can cause performance problems because the entire page will be transmitted, potentially overloading the network. However, modern page-based DSM systems compress these pages by only transferring the updates made to a page.

The second problem with page-based DSMs is that several unrelated objects might be located on a single page, giving rise to false sharing. Consider the following example. We have a page containing objects X and Y located on processor A. Processor B wants to update X while processor C wants to update Y. In a simple implementation, processor B and C will both need to have copies made of the page and write their results back to A as the protocol does not know that B and C will access disjoint data. This will cause much communication (thrashing) to occur if either or both processors frequently write such a variable with either processor performing much synchronization. If the DSM implementation did know the individual processor's data usage patterns, the page would not have to be transferred back to A as X and Y could be privately owned by B and C.

```
class LinkedList {
    int value;
    LinkedList Next;
}

remote class Remote {
    int data;
    int foo(LinkedList l)
    { if (l == null) return data;
      return data + foo(l.Next); }
}

class pseudo_rpc {
    Remote o;
    LinkedList  l = new LinkedList();
    int foo() {
       return o.foo(l);
    }
}
```

Figure 2.4: Java Party RMI code.

2.4.2 Object-based DSM using RPC

Object-based DSMs provide the illusion of a shared address space by encapsulating each shared data access inside a method (remote procedure invocation or RPC). All method invocations are then caught by the compiler and runtime system to be run on the processor that currently owns the object the method is invoked upon. To glue it all together there is usually some scheduler in the runtime system to enable remote object and thread allocation. This approach usually uses objects and array (slices) as units of data transfer. Systems that implement this model include Orca [4], JavaParty [47], and others.

The problems when using RPC are numerous: data caching is difficult, arguments to method invocations need to be handled specially and usually the semantics of accessing parameters of RPC methods are different from accessing parameters of local methods.

To illustrate these problems, consider the JavaParty example in Figure 2.4. Here a class *Remote* has been declared remote causing all methods to be invoked on the machine where the *Remote* object has been allocated.

The *Remote.foo()* method has a linked-list as a formal parameter and reads a single field of the remote object and returns it. Since the LinkedList is a normal object, it will be *serialized* (marshaled, see [44]) to a byte stream and sent over the network. After deserialization by the receiver, a new copy of the linked-list will have been created and the *foo* method is invoked. The field of the remote object is read and returned. The return value is again serialized and the return value is sent back to the machine where the RMI originated from. At this point the invoking machine can continue execution.

There are several efficiency problems with this approach. First, it is expensive to serialize the entire linked-list when only a small part of it is used. Second, serialization requires objects to be introspected to locate field and method information (this has been solved in [40]). Third, when the remote object is only read from, it can be cached or replicated on the machines that access it. A naive implementation without any caching suffers from unnecessary network transfers, especially when most remote data accesses can be expected to be read accesses. This problem has been solved in for example Orca [4], which replicates objects for this purpose.

2.4.3 Fine-grained DSMs

Fine-grained DSMs [25, 54] divide the global address space into many small fixed or variable sized chunks and use software access checks to trap accesses to nonlocal data in the same manner as the MMU was used in page-based DSMs. This enables sub-page units of coherence, for example, 32 or 64 byte pages (cache lines). An access check in these systems is just a series of ordinary machine instructions to test the pointer about to be used. This has the obvious problem of adding overhead and thus increasing execution time. Generally the overhead in most of these systems ranges from 5% to 30% but can peak to very high overheads (more than 260%, see [54]). In this thesis we will discuss several techniques to reduce this overhead. Current reordering, super-scalar processors usually have high memory latencies allowing the execution of the access check itself to be at least partially hidden.

There are a number of ways to add access checks to a program: using instrumentation of existing executables; instrumentation of source code; or using the compiler to add access checks as part of the normal code generation process when creating code from its input.

True cache line based DSMs are often implemented using binary re-writers. They take an existing multithreaded executable, disassemble it, add some code before each memory usage and recreate the executable. This has the advantage that little programmer effort is required to make a parallel program that works on a cluster of workstations but these systems have all the problems associated with binary re-writers. For example, it is hard (but not impossible) to implement large scale transformations of data and functions and it is difficult to make portable solutions. Examples are Shasta [54] and Sirocco [25].

From a performance standpoint, there are a number of obstacles to overcome to enable large data transfers (which are often the most efficient in modern networks). For example, whenever a linked-list is traversed, the linked-list will be fetched one element at a time. For a linked-list of N elements at-least $2 \times N$ messages will be sent where ideally only enough messages need to be sent to hold all linked-list elements and an initial linked-list request message.

Other systems use compilers, translators or preprocessors instead of binary re-writers to add access checks to a program. These have the advantage of being more portable. Another advantage is the availability of source code level information which enables more optimizations and simpler implementations of them.

The system (Jackal) described in this thesis falls into this category; it uses a compiler to add access checks to a program and implements aggressive optimizations to reduce the number of messages sent over the network and tries to generate RPC code where that is profitable.

2.5 Summary

Summarizing, the Java memory model specifies how and when threads can communicate values to each other. This is specified by giving each thread a private memory that it operates upon and a global memory where computed values are stored in and retrieved from.

Fields that have been written to must be flushed from the cache between the store and the next synchronization action (be it a lock or unlock). A locking operation, besides flushing working memory, guarantees mutual exclusion between different threads.

Because of problems with the current JMM, we will assume JSR-133, the revised Java Memory Model, throughout this thesis. It solves the problems found thus far in the old JMM (the double-checking paradigm, volatile variables, serializability of operations, single main memory, and others) and still allows each thread to cache and prefetch object fields.

A major difference between the JMM and some other memory coherence protocols is that it does not associate the data protected by a lock to the lock itself. The JMM most closely relates to scope consistency because of the use of synchronized *blocks* in the Java programming language and lazy release consistency because updates to main memory are delayed until a synchronization action occurs.

To implement the JMM on a large distributed-memory machine we have a number of alternatives: use expensive hardware and standard software or to use complex software approaches combined with standard hardware. The hardware approach can achieve high performance at high cost while the pure software approach can sometimes come close to the performance of the pure hardware approach at the cost of more complex software.

The software approaches all vary in where in the software development process the program is made aware of distributed memory. A binary rewriter allows a program to be distributed after it is finished, while a translator takes some source code and transforms it to produce new source code. The system's compiler is then used to produce a final executable program. The translator and compiler can be integrated into one larger compiler as Jackal does, allowing for more low-level optimizations. The extra optimization opportunities stem from the inability of the traditional compiler to take advantage of the semantics of an access check. Finally, the programming language itself can impose restrictions on which operations are allowed to transparently support distributed execution. Essentially, all shared memory accesses are encapsulated by method calls, and the methods are transparently shipped to the location of the data accessed (like Orca and other systems).

Chapter 3

The Basic Jackal System

3.1 Introduction

Jackal consists of a compiler and runtime system that together provide an object-based DSM. An object-based DSM implies that Jackal transfers objects rather than entire hardware pages or individual object fields over the network.

The compiler (see Chapter 4) translates Java code to its own internal intermediate code which is then handed to the compiler's back-end, which performs all optimizations. After the optimization passes are finished, the code is transformed to the target machine's assembly code format and passed to the system's assembler and linker. Jackal thus uses a native compiler rather than a JIT (Just In Time) compiler or byte-code interpreter. The runtime system implements all of object allocation, communication, garbage collection and the DSM coherence protocol.

The way Jackal implements the DSM is by using software access checks. An access check determines, using a small number of (ordinary) machine instructions, if an object or array about to be accessed is available in the correct state for the access to be performed (read from or write to an object). If the object is not present or not in the correct read or write mode, the access check calls the runtime system to retrieve the object. Access checks are automatically generated by the compiler's front-end before each object and array access.

Because the JMM allows multiple threads to modify the same variable using cached copies, Jackal continues this behavior using a multiple-writer protocol. Any concurrent modifications to the same object are merged by choosing one over the other (which update prevails is not specified either in the JMM or by Jackal).

We will first discuss the basic unoptimized operation of a Jackal program where no access checks will be moved, removed, or combined by the compiler nor are objects prefetched. Also, no runtime optimizations are done. This basic system is a straightforward implementation of the Java Memory Model on a distributed-memory system (a cluster of workstations). It will be used in subsequent chapters to study various performance optimizations.

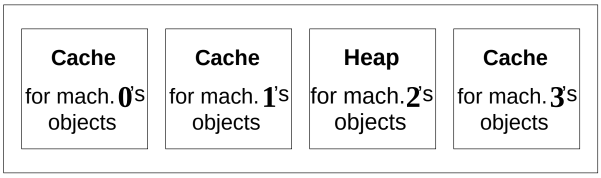


Figure 3.1: Virtual address-space organization of processor 2.

3.2 Global Address-Space Layout

As Jackal uses a *home node* for each object to maintain the master copy of the object, a simple method is required to compute the initial machine's address given only the object's address. The problem is complicated if the remote copy has not yet been initialized (the global address space is initially zeroed). To simplify the computation of the initial home node, Jackal partitions the global address space. The global address-space layout dictates the address range machine *P* is allowed to allocate its objects in. A cached copy from a remote machine is allocated at the same virtual address as at its home node. Jackal partitions the virtual address space of the machine into *N* parts of *M* megabytes, where *N* is the number of machines participating in the Jackal cluster and *M* a constant that is chosen when the compiler and runtime system are compiled. The initial home node of an object can now be computed by taking its address and dividing it by *M*. After the initial home node has been located, the *home_node* field in the ObjectHeader (see Figure 3.2) of the cached copy is used instead to support home-node migration.

Figure 3.1 shows Jackal's basic address-space layout for machine 2 when using four machines in a cluster. Machine 2's reserved address space is used to allocate its own objects in (its heap), while the rest of the global address space is used to cache objects from the other three machines.

3.3 Basic operation

A Jackal program will during its execution perform the following steps:

- processor allocation, program startup and thread creation;
- accessing remote objects;
- flushing remote objects back to their home machines.

In the following subsections we will first globally examine these steps.

3.3.1 Basic Operation - Program Startup

Before program startup, a number of processors is reserved exclusively for running the Jackal program. Then, the program is started on them simultaneously. The processors synchronize, initializing internal data structures and initializing the network communication software. The user's code is started by executing the *main* method of the program's startup object at a single machine in the network. Other machines in the Jackal cluster are left idle, waiting for threads to be started on them.

The user's code might then do its own initialization etc, and create its thread objects to do the actual work. Next, the user code starts these threads by invoking the *start* method of the created thread objects. The Jackal runtime system intercepts these calls to *Thread.start* and starts those threads at idle machines in the network (in a round-robin fashion). The threads start running in their *run()* method at the remote machine. Inevitably they will access some object fields, either from local objects or from other machines' objects. When the object access occurs, the access check generated by the Jackal compiler will invoke the RunTime System (RTS) to retrieve the data.

3.3.2 Basic Operation - Accessing Remote Objects

When a remote object (an object *allocated* at another machine) is not locally available upon encountering an access check, Jackal's runtime system needs to retrieve the object from elsewhere. Read availability is checked by performing a bit test on a thread's *present bitmap*, write availability is checked by examining the *dirty bitmap* (see Figure 3.2). Availability is computed by taking the address of an object and computing a bit address from it. The bit address is used to index one of the above bitmaps. Access detection is thus managed per thread, although all threads on a single machine access the same cached copy.

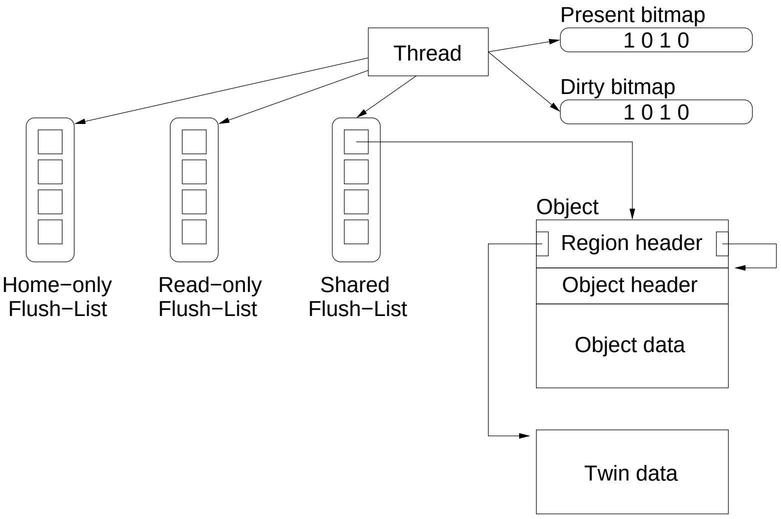
Jackal uses a home-node concept, which means that each object has an associated *home node*, which is some machine in the network that currently maintains all meta-information about the object. This information consists of: which machines are currently reading and writing the object; object locking information (to support Java's synchronized keyword); and how many times the object has migrated its home node.

Since in Java arrays can be arbitrarily large and there may be an arbitrary number of threads reading and writing elements of the array, Jackal splits arrays into several, fixed size, chunks called regions. Thus to support automatic array partitioning, memory is managed per region: a region is either a whole object or an array partition. This solves two problems at once. Firstly, when thread *T* operates on array segment *X*, only segment *X* has to be transferred over the network, potentially reducing the amount of network bandwidth required. Secondly, when multiple threads access disjoint segments of the array, there is a higher probability that the individual pieces may be cached.

3.3.3 Basic Operation - Flushing Cached Objects

Once a thread has finished a computation it often needs to flush working memory to publish the computation's results. Implementing a per thread working memory requires Jackal to record which objects a thread has cached from other machines so that it can flush any modifications to the home-node copy.

Each thread maintains several lists of region references, to allow a thread to flush its cached dirty regions. A list of such regions is called a *flush list*. The flush lists are shown at the top of Figure 3.2. Whenever thread *T* caches object *O* from machine *X*, *O* is added to one of *T*'s flush lists associated with machine *X*.



```
struct ObjectHeader {
 javaVTable       *vtable;     // points to a table of method pointers
 int32             object_status_bits; // status: stack-allocated, etc
 javaObjectHeader *next_object;
 int32             home_node;
}

struct RegionHeader {
 int8          region_status_bits; // status: home-only, shared, read-only
 int8          status_flip_count;  // home-migration counter
 ObjectHeader *start_of_object_pointer;
 void         *twin_data;
 int64         reader_bitmap; // who have it for read.
 int64         writer_bitmap; // who have it for write.
}
```

Figure 3.2: Threads, objects and flush lists.

```
1 class Data {
2    int field;
3 };
4
5 class MyThread extends Thread {
6    Data d;
7    static void foo(Data d) {
8       d.field = 42;
9    }
10   public void run() {
11      // Code
12      foo( d );
13      // Code
14   }
15 };
```

Figure 3.3: Accessing a field.

Within the basic Jackal system, each thread maintains three flush lists per processor in the cluster. One flush list records regions that no other machine has a copy of, one flush list records regions that are system wide only read from and another records regions that are shared between processors where at least one has modified the region. When thread *T* executes a synchronization action, all flush lists of thread *T* are traversed, any modified objects are written back to their home nodes, and the lists are emptied.

3.4 Object Management: Individual Steps In Detail

3.4.1 Retrieving A Single Object

Here we will look in detail at what happens in the basic Jackal system when a thread wants to access a non-static field of an object. For this purpose, the example in Figure 3.3 will be followed during its execution.

First assume that an instance of MyThread has been allocated and its *run* method is executing on some machine in the Jackal cluster. The compiler will have detected the field access at line 8 and inserted an access check to object *'d'*, transforming the code to the code shown in Figure 3.4. The example is slightly misleading however, in that the source code is not transformed, but rather the access check is added in the output of the compiler's front-end.

At this point, let us have a closer look at the exact workings of the access check. Its pseudo-code is shown in Figure 3.5.

Every chunk (region) of 64 bytes of memory has, per thread, two bits associated with it: a readable and writable bit. These bits are maintained inside two large, per thread, bitmaps (see Figure 3.2). The access-check code checks these bits to determine if a thread already has permissions to read or write an object. We have opted for 64 byte chunks as it is the closest power of two above the smallest Java object size including all object and DSM region headers. If an object or array partition or Z bytes is cached, exactly enough

```
class Data {
  int field;
};

class MyThread {
  Data d;
  static void foo(Data d) {
     //<<<    write-object-access-check(d)   >>>
     d.field = 42;
  }
  public void run() {
     // Code
     foo( d );
     // Code
  }
};
```

Figure 3.4: Instrumented code of Figure 3.3.

```
temp         = object-ptr - base_of_object_heap
chunk        = temp  / size_of_chunk
bitmap_index = chunk / 8;
byte_index   = chunk % 8;
access_byte  = {read,write bitmap} [ bitmap_index ]
access_bit   = access_byte [ byte_index ]
if (access_bit is zero)  {
    call runtime-system(object-ptr)
}
```

Figure 3.5: Pseudo access-check code.

```
      movl 8(%ebp), %eax
      leal -1107296256(%eax), %eax
      shl 6, %eax
      movl %fs:(0), %ecx
      bt %eax,(%ecx)
      jc success
      call shm_write_object_eax
success:
      ...
```

Figure 3.6: Access-check code, actual x86 machine code.

bits are set in the bitmaps to cover the Z bytes.

First, as can be seen in the pseudo-code, the address is “normalized” by subtracting the starting address of the (global, cluster wide) Java object heap. This reduces the amount of space occupied by the read and write bitmaps.

Next the index into the bitmap is calculated and the correct “access bit” extracted. If this bit is zero, the object or array chunk is not present or not available in the right mode. In that case, the runtime system is invoked and the object is fetched over the network from its home node and mapped in the local address space, in the correct mode, at the local machine. The accessing thread’s bitmap is updated and execution continues. A hidden cost lies in mapping the object in the local address space which requires a page to be mapped at the local machine to store that cached copy. Because the number of pages storing cached copies of objects might eventually exceed the number of pages available in the system, the runtime system maintains a counter telling how many pages were mapped for storing cached objects. If this number exceeds some preset threshold a global GC is executed to remove stale cached object copies.

The access check shown in Figure 3.5 is optimized for uniprocessor machines where Jackal’s thread package will reset the two thread-global variables “read or write bitmap pointer” upon each user-level thread switch. In the actual implementation (see Figure 3.6) these pointers are located inside a thread-local segment at *%fs:(0)* and *%fs:(4)* to enable multi-processor execution (%fs identifies a segment of memory to use, the appended number is the offset into the segment). If the processor Jackal is targeting does not support thread-local segments (on for example PowerPC or *ARM* platforms) we suggest that these be located inside machine registers to ensure efficient execution.

Now, we get back to our running example of Figure 3.3. When the *run* method of class MyThread reaches the method invocation of *foo*, *foo* is invoked, and the access check is hit.

When the access check notices that the object is not locally available, it invokes the runtime system through the *shm_write_object* entry point. The runtime system computes the home node of the object (by dividing the address of the object by the amount of space allocated to each machine) and sends it a request message. The home node then responds by sending back a copy of the object. The requester, upon receipt of the object, maps the object into the local address space. If the object was requested for write access and other threads are also holding copies of the object, then the requestor creates a copy (*twin*). A

```
class X {
  static int field = 1;
}
class Y {
   void foo() {
       System.out.println("X.field="+X.field);
   }
}
```

Figure 3.7: Assigning to a field.

reference to the object is then added to one of the requesting thread's *flush lists*. The *twin* is required when the thread executes a flush operation to find the modifications made by a thread. The *flush lists* will at that point contain references to all objects cached by this thread. After the object is locally available, execution continues after the faulting access check.

The protocol that maintains the consistency between the cached copies and the home node also implements home-node migration and switches between read-only and shared states. The protocol's implementation is very similar to that of JUMP (see [8]).

3.4.2 Retrieving A Chunk Of An Array

Accessing an array element is similar to accessing an object field, except that the array is divided into 256 byte chunks and each chunk is individually managed.

The access-check code for arrays is therefore changed to not only take the address of the start of the array but also the address of the array element to be accessed. Also, the runtime system will, upon a failing access check, map only the offending array chunk. The runtime system entry point for an array access check therefore takes two parameters: the start of the array and the address of the array element to be accessed.

Java prescribes array index bounds checks, so besides mapping the accessed array chunk Jackal will also map the first array chunk that contains the array length, needed to implement these checks.

3.4.3 Static Initializers

According to the Java language specification (JLS), all static members of class *X* having initialization expressions need all these initializers to be executed upon first usage of class *X*. This behavior is observable as the initialization expressions might have side effects such as printing something on a console. Consider the example in Figure 3.7.

In this example, the assignment of '1' to *X.field* will only be performed once, when *X.field* is accessed. Besides this, the JLS also specifies that static initializers are only allowed to be initialized once. For an implementation this means that per class a boolean needs to be maintained to record if the static members have already been initialized. The JLS also enforces that when multiple threads concurrently access the object enforcing static field initialization, only one thread is allowed to do the initialization and the others are blocked until the first thread finishes the initialization code.

Jackal implements the JLS here using self-modifying code and by running the static initializers of a class only on CPU 0, where the static variables of all classes reside.

The exact workings using the *X.field* example above are as follows. For regular usage, the compiler generates the normal *X.field* access and records the address of the access in an access table *W*. Before any Java code has run, *W* is handed to the runtime system, which replaces all object access codes pointed to by *W* with a call to some generated stub code. The generated stubs call the runtime system, passing a pointer to the static initialization code and a pointer to a lock to keep other threads from executing a particular static initializer concurrently.

Once a thread's (say *T*'s) execution reaches the inserted call for static initializer *S*, the call instruction is executed and the runtime system (RTS) entered through the generated stub code. The RTS tests the status of the lock structure *L* to determine if static initializer *S* has already been run, in which case the runtime system is immediately exited and the original code, temporarily stored in table *W*, is restored atomically. If the lock *L* has already been acquired by some other thread *Q*, thread *Q* must still be busy running static initializer *S* and thread *T* waits for static initializer *S* to finish. If lock *L* has not yet been acquired, thread *T* acquires *L* and runs the static initializer *S*. Once *S* has run, the lock *L* is signaled causing any threads waiting for *S*'s completion to continue execution. At that point thread *T* atomically swaps the call to the runtime system with the original object access code and continues.

For Jackal execution, the runtime system implements an extra step. It executes an RPC to machine zero in the cluster to run the static initialization code there. Upon success, the RTS patches the code at both the remote and local machine.

3.4.4 Static, Final and Volatile Field Accesses

Volatile variables are object fields that should not be cached inside a thread's working memory. Additionally, JSR-133 requires that accessing a volatile variable acts like a synchronization action. Any cached values should thus be flushed upon accessing the volatile variable as if the volatile variable were enclosed in a synchronized block.

Implementing this behavior is straightforward. The compiler front-end, upon encountering a volatile variable access, marks the access as *volatile* and inserts a call to the runtime system to flush working memory. The compiler back-end ensures that no code or data accesses will be pulled over accesses to such variables by any optimization.

Static field accesses are variables that are always accessible, needing no enclosing object instance; they are in effect global variables accessible by any thread at any time. If the variable is also marked final, Java's language specification requires the variable to be initialized in each constructor of the declaring class or initialized directly at its declaration site. If initialized inline and the variable denotes a primitive type such as 'int' or 'double', the variable's value is substituted for the variable access by the Java front-end.

3.4.5 Flushing a Thread's Cache

Flushing a cached modified object entails copying a cached object back to its location at its home node or invalidating it in case the copy was in read mode. If the cached

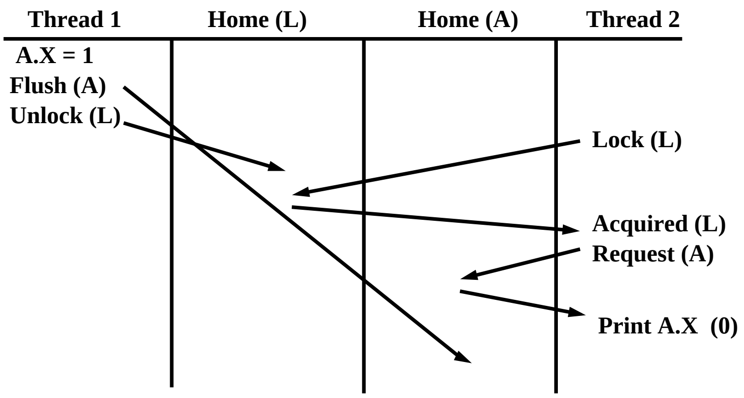


Figure 3.8: Flushing without acknowledgment.

copy was modified and needs to be copied back to the home node, the copy cannot be done verbatim as multiple threads may modify the object concurrently as Jackal employs a multiple-writer protocol. Consider an object *O* located at machine H containing two fields: *A* and *B*. *A* is changed by a thread at machine R1 while field *B* is changed by a thread at machine R2. First machine R1 overwrites the home-node copy with its data, next R2 overwrites *O* with its copy. The last overwrite would erroneously overwrite *A* as changed by machine R1.

To avoid this effect, only the fields changed by a thread are written to the home-node copy. Changed fields are found by comparing against a twin that is created when a thread has an object mapped for write access.

The compiler creates specialized *'diff'* and *'patch'* routines per class to speed-up the generation of diffs at runtime (the changes between a twin and its working copy) and the patching in of the differences at the home node. References to the generated routines are stored in each object's vtable (see the ObjectHeader). Because the generated routines have knowledge about object layout, they are able to skip compiler added field alignments and avoid object introspection. The runtime system contains diff/patch routines for each array type.

Flushes must be acknowledged from all machines participating in the flush to ensure that once a lock/unlock has succeeded all machines have up-to-date data. If we would not wait for acknowledgements, threads would be able to observe stale data while already inside a synchronization statement. An example of what could happen if flushes were not acknowledged is included in Figure 3.8. Here, *thread 1, home(L), home(A)* and *thread 2* are all located at different machines. Due to different network latencies *thread 1*'s flush and unlock messages are delivered out of order. This causes *thread 2*'s lock acquire to be processed before the modified *A* is processed at *home(A)*. Since messages from different machines are not ordered (only messages from a single machine are processed in FIFO order), the data request from *thread 2* is processed before the flushed *A* from *thread 1* has been delivered. *Thread 2* now erroneously prints the old value of *A.X* (0) instead of the value *thread 1* has written to it (1).

Having *thread 1* wait for an acknowledgment from *home(A)* ensures that *A*'s value is up-to-date at *home(A)* before *thread 2* can obtain *lock(L)*, thus ensuring that *thread 2* observes an up-to-date value for *A*.

3.4.6 Flushing Lazily

There are several cases where the RTS can reduce network traffic by keeping objects cached across synchronization actions. For this *lazy flushing* the home node must record which machines have read or write access to the object. Machine N is recorded as a reader or writer if the N-th bit is set in the reader_bitmap or writer_bitmap bitmap from the ObjectHeader (see Figure 3.2). The RTS recognizes the following cases:

- trivially, if the home node modifies an object, no network traffic is required when flushing an object;
- if there is only one user of an object, that object need not be flushed (in effect, the machine is the owner of that object);
- cached read-only objects that are not mapped for write access by any thread in the system do not require invalidation;
- a cached read-only object need only be invalidated at a synchronization point, it does not need to be returned to its home-node even if it is write-mapped elsewhere in the system;

To reduce network traffic as much as possible, Jackal implements *home migration*: whenever there is only a single machine that writes to an object, it is made the home node of the object. This home-node can easily determine that there is only a single writer left by counting the number of bits set in the ObjectHeader's writer_bitmap. Messages sent to the old home node are forwarded to the new home node. To reduce the costs incurred by the forwarding scheme, the machine where the request originated has its home-node identification for the object updated once the request is processed. Home-node migration is useful only when Jackal's default object placement at the creator machine turns out to be suboptimal.

The implementation of home-node migration is straightforward. The object's *home_node* field inside the *ObjectHeader* is overwritten with a new value. When a thread notices that an object's home node has changed, it moves the entries for that object's regions from its flush list for the original home to its flush list for the new home. To avoid thrashing, each object is allowed to migrate to a new home node only a fixed number of times. The migration counter is maintained in the home-node copy of the object. The number of home-migrations allowed is currently an arbitrary runtime constant.

Whenever a writer wants to write to an object where multiple read-only copies already exist over multiple machines, messages are sent to each reading machine to discard the read-only copy once a synchronization point is reached. The exact scenario is as follows. A machine requests a write access copy from the home node of the object. The home node examines the reader/writer bitmap located inside the DSM header of the object and notices that there are currently only readers of an object and sends those machines invalidation messages. The machines reading the object respond by removing lazy flushing from the cached object (the object is moved from the flush list for read-only objects to the flush list for shared objects). After all machines have been notified, a writable copy is given to the requesting machine and it is made the new home node.

The way we implement this is by realizing that the readers only need to know of the write after the write has been flushed; after that again, the application synchronization must ensure that the reader has actually flushed the (no longer read-only) region so it will be faulted in with the write in effect.

The idea behind this protocol is to push all protocol overhead outside the latency, and in many cases (e.g. the acknowledges for un-readonly messages) piggyback the protocol info.

3.5 Problems with the global address-space layout

In the current global address-space layout, each processor owns 64 megabytes of the address space to allocate its own objects in. This of course limits a single machine to allocate objects as it will never be able to allocate more than 64 megabytes of "live" objects at a time, even if that particular machine has much more physical main memory available.

Another problem with this address-space layout scheme is that it severely limits the number of machines that can participate in a Jackal cluster. Under RedHat Linux, versions 6.2 to 7.2 with our communication package added, the largest amount of linear memory available to Jackal is about one and a half gigabyte. Available linear memory is further limited by, for example, the kernel, shared libraries, and special devices (such as network cards) that map in data at fixed addresses. This problem is made worse by the inability to "predict" free address space ranges. We require this ability for optimized access checking and to avoid collisions between the global address space and, for example, a shared library or hardware device driver. In our current test-bed, the DAS [1], we are limited to a maximum of about 24 machines inside a cluster because of this problem. The number of machines that can participate in a Jackal cluster can be calculated using the formula: $\frac{1.5\ gigabytes free}{64\ megabytes\ \ per\ \ machine} = \frac{1.5 * 2^{30}}{2^{26}}$ machines = 24 machines

One obvious solution to this problem is to use a larger virtual address space, for example by using a 64-bit machine. But even on 64-bit machines such as an IA-64 or Compaq/DEC Alpha, the problem that no fixed address space range can be predicted or reserved remains.

A case can also be made for overlapping address space usages by the nodes in the cluster. For example, each machine will have a heap starting from address X to Y for allocating its objects and another address space range for cached objects, say from Y to Z. This, however, will require complicated and expensive pointer translation schemes or another indirection level for pointer accesses and for this reason we have discarded this possibility. This solution has been taken by, for example, Hyperion [41].

Another obvious solution is to decrease the amount of memory per node for its own object allocation pool to say 32 megabytes. While this would increase the size of the maximum Jackal cluster to about 48 machines, this solution will cause some programs to fail because not enough memory is available to them anymore.

Finally, the last problem with the global address-space layout described above is fragmentation. The space reserved for cached objects is often fragmented as it mirrors the heap of each machine. In the worst-case scenario, each of a large number of pages might host only a single cached object, which increases physical memory usage. Ideally, all cached objects should be compactly allocated in the address space allocated for caching remote objects. This would greatly reduce the number of physical pages used, and thus reduce the probability that a machine would be forced to swap out pages to disk. A fragmented address space also reduces the chance that a page hosting a remote object will be mapped.

3.6 Summary

Jackal's compiler translates Java code to machine code. The compiler adds *access-checks* before each object and array access to its output during translation. An access check checks if that thread has the object already mapped for read or write using a per thread bitmap. If the object is not already mapped, Jackal's runtime system is invoked to map the object.

If an object is not locally available, Jackal's runtime system will fetch the object from its *home node*. After mapping the object, a reference to the object is included in the requesting thread's *flush list*. Whenever a synchronization point is reached, the flush list of that thread is traversed and its objects are sent back to their home nodes.

Jackal attempts to reduce the number of objects that need to be flushed by performing both *lazy flushing* and *home-migration*. Lazy flushing allows data to remain cached whenever there is only one user of the object or the object is *read-only*. If there is only a single writer of an object, home migration moves the logical home of that object to this machine.

Chapter 4

The Compiler

4.1 Introduction

This chapter introduces the compiler used by Jackal, including its traditional program analysis and optimizations already described elsewhere.

In the Jackal system it is the compiler's job to translate Java source code into efficient executable machine code that can be transparently executed on a cluster. Figure 4.1 shows the entire compiler's pipeline. At the start of the pipeline, the programmer has created a Java source file (or Java bytecode file). The front-end parses the Java source code, checks its semantics according to the Java specifications and creates the LASM intermediate code (Liberally ASseMbly language, a compiler intermediate language of our own design) which is passed to the compiler's back-end. The compiler back-end runs a number of optimization passes over the intermediate code, using analysis over both the code from the input files and earlier compiled code in the intermediate code database.

The optimization passes are written in a modular fashion; each pass takes as input a set of LASM procedures and produces a set of optimized LASM procedures. Optimization modules can be individually enabled, disabled, reordered and repeated freely.

The ability to load and store intermediate code on disk allows the compiler to see implementations of functions defined in other source code files. This allows for aggressive interprocedural optimization across file boundaries. The focus of this chapter will be on the optimization passes in the LASM back-end and less on the front-end passes, which are based on traditional compiler technology.

4.2 Front-end Tasks

The front-end reads a Java source file from disk and creates parse tree using a parser generated by Jade [58]. Next, the front-end traverses the generated parse tree to check its correctness according to the Java Language Specification [21]. After all semantic tests have completed, it traverses the parse tree one final time to generate LASM code. When encountering a field or array access it also emits an access check. For example, in the

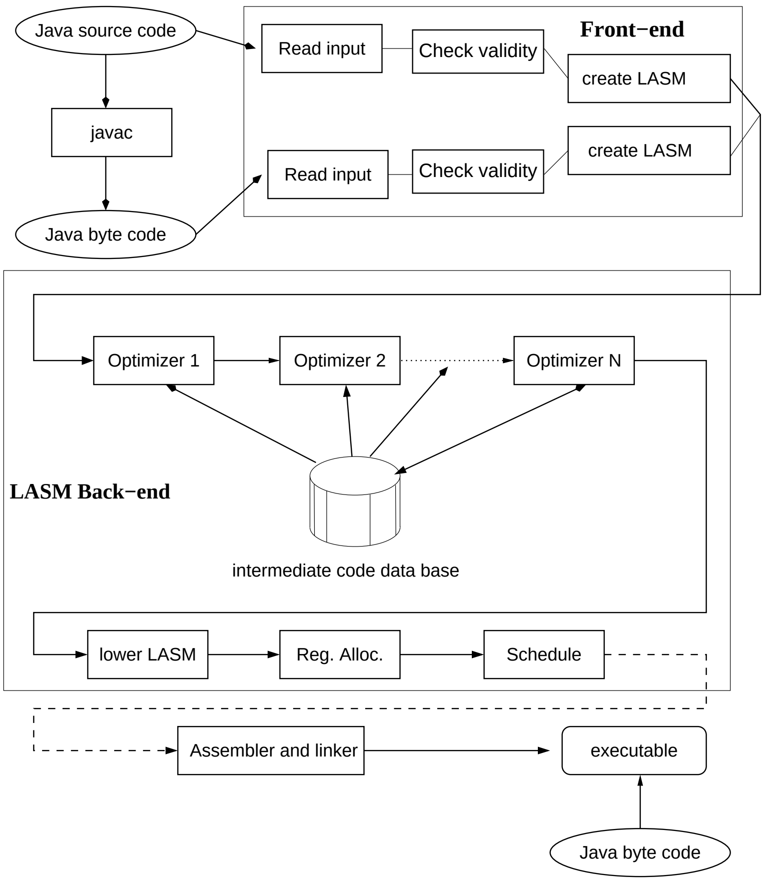


Figure 4.1: The compiler pipeline.

```
class Example {
    int a;
    void foo(int b) {
        // <check(this)>
        this.a = b;
    }
    public static void main(String args[]) {
        new Example().foo(12);
    }
}
```

Figure 4.2: Simple instrumentation example.

code in Figure 4.2, for each object access a check is added. Since the front-end does not maintain global information and optimizations on the front-end language (Java) are complex, no optimization takes place. Another reason for not optimizing in the front-end is to maintain modularity of design; optimization and code generation are completely separated and optimizations need not be re-created for each new front-end but can instead be reused.

The example in Figure 4.2 suggests that the checks are implemented in Java but they are not. They are generated into LASM code as each object (or array) access is encountered. At this stage, an access check is translated into a single LASM access-check instruction. This allows easy recognition, removal and insertion of access checks in the code. Later, when all access-check-related optimization passes have finished, they are rewritten into multiple LASM instructions which will then in turn be rewritten to the target machine code using the normal LASM code generation path.

4.3 Back-end Tasks

It is the task of the back-end to do all code optimizations for sequential and parallel performance. The back-end is centered around LASM, an intermediate language used throughout the compiler. LASM, like SUIF [2], has elements of a high, medium and low-level intermediate language inside a single intermediate language. This allows optimizations to be written once and applied to all three levels of abstraction. For example, object inlining (high-level) and instruction scheduling (low-level) are performed on the same intermediate code.

The Java front-end hands to the back-end a set of functions, each consisting of a list of instructions. The front-end also passes a set of class descriptors consisting of lists of typed object fields. Figure 4.3 shows what the back-end receives from the front-end for the example in Figure 4.2.

The *this* parameter is an implicit parameter required to implement Java's object-oriented features. It is of type *g*, meaning pointer. Compiler-introduced temporary variables (pseudo registers) are encoded by a '%' followed by a type letter (*g* for pointer, *i* for 32-bit integer, etc) followed by an identification number. There can be an arbitrary number of these inside a LASM function. A register allocation pass will map them to

```
FUNCTION jac_Example_foo__I(param('g', this+0,
                          param('i', b+0) )
debug_label 5:/home1/rveldema/tests/Code1.java
%g21704 =   param('g', this+0, , signed)
access check [%g21704 ,write ,object,
      complete-object, check-id:0]
%i21706 =   param('i', b+0, , signed)
(%g21704).Example.a  =  %i21706
END.

FUNCTION jac_Example_main___3Ljava_lang_String_2( param('g',
                    args+0, signed))
debug_label 7:/home1/rveldema/tests/Code1.java
%g21707 =   $vtable_Example
%g21709 =  direct_call new_object (%g21707), Example, 1
%i21711 =  12
 access check [%g21709 ,read ,object,
           complete-object, check-id:1]
%g21712 =   (%g21709).vtable
%g21713 =   *('g', (116 + %g21712) )
%g22076 = call_indirect %g21713 (%i21711, %g21709)
               call_target_list=<jac_Example_foo__I>
END.

DESCRIPTOR(class_Example) {
        inlined-class: class_java_lang_Object {
                        vtable *v;
                        int32 flags;
                        class_java_lang_Object *Next;
                        int64 readers;
                        int64 writers;
                       }
        int a;
}
```

Figure 4.3: Example 4.2 in LASM.

```
jac_Example_foo__I:
        pushl %ebp
        movl %esp,%ebp
        movl  8(%ebp), %edx
        leal -1308622848(%edx), %eax
        shrl $6, %eax
        movl %fs:(0), %ecx
        bt %eax, (%ecx); jc .L5833
        call shm_start_write_object_edx
.L5833:
        movl  12(%ebp), %ecx
        movl  %ecx, 28(%edx)
        popl %ebp
        ret

jac_Example_main___3Ljava_lang_String_2:
        pushl %ebp
        movl %esp,%ebp
        pushl $vtable_Example
        call new_object
        addl $4, %esp
        pushl $12
        movl 0(%eax), %ecx
        movl  124(%ecx), %ecx
        pushl %eax
        call *%ecx
        addl $8, %esp
        popl %ebp
        ret
```

Figure 4.4: Unoptimized, GNU style, x86 code generated for Figure 4.2.

either local variables or actual machine registers. The front-end may also add debugging instructions to the LASM code as depicted by the debug_label instructions. These can be passed to external debuggers.

Initially, all field accesses are symbolic. For example, the assignment to field *a* in the Example class is symbolically encoded (line 8 in Figure 4.3). After optimization, this code is transformed into a machine-dependent version where symbolic field accesses are rewritten to physical memory references, and the parameter accesses rewritten to the calling convention encoding dictated by the processor manufacturer's Application Binary Interface (ABI). After this transformation, the code is matched with a machine descriptor: all LASM instructions are rewritten to conform to one of the instructions from the machine descriptor and pseudo registers are rewritten to physical registers. After this final transformation and the final machine-dependent optimizations are applied and the code shown in Figure 4.4 is produced. The code is then handed to the target machine's assembler and linker which results in executable code.

4.3.1 LASM Implementation

LASM is completely implemented in C++ to enable easy extensibility and maintainability. The base for the intermediate code are the classes *LasmInstruction* and *LasmExpression*. *LasmInstructions* are the statements in our programs; they may contain several other instructions and expressions. An example is *LasmWhileInstruction* which inherits from *LasmInstruction*, contains a list of other *LasmInstructions* and a *LasmExpression* for its loop condition. When needed (for example when going to machine level), the *LasmWhileInstruction* is rewritten to a series of assignments and (conditional) jump expressions.

LasmInstructions such as labels and debug statements are not matched with the machine descriptor. *LasmExpressions* such as assignments, (un)conditional jumps and increments are matched with the machine descriptor. The rationale for this distinction is that statements, such as labels and debug statements are machine-independent while expressions such as assignments and such are machine-dependent. A machine might for example enforce that the right-hand side of an assignment always be a register; debug statements have no such restrictions. To this end, there is a *LasmArithmeticInstruction* which inherits from *LasmInstruction* and contains a reference to an expression that must be matched with the machine descriptor. Expressions include: access checks, bounds checks, arbitrarily complex assignments and others.

Each instruction and expression object has a read and write method to load and save code to and from disk to enable inter-file optimization. Each instruction and expression also has methods to check properties like whether a variable is used or killed and whether it uses memory or a specific register.

The front-end, besides supplying the instructions for each function, also supplies LASM with a set of class descriptors. Class descriptors consist of a set of properties and a list of object fields. Again, to enable cross-file optimizations, each class descriptor has a read and write method to load and save class descriptors from disk.

The interaction between expressions and class descriptors is as follows. Since the back-end needs to know for each heap access the exact (static) type of object being accessed there exists a class named *LasmObjectAccessExpression* in LASM which is to be used for implementing heap accesses. A *LasmObjectAccessExpression* takes a base address of an object, an optional array indexing expression and a symbolic object field, which is one of the field descriptors taken from the static type's class descriptor (the object being accessed). In the example in Figure 4.3, at line 8, the symbolic field access object representing the left-hand side of the assignment has a pointer to the field descriptor *Example.a* which again has a pointer to the class descriptor *Example*.

After optimization, offsets are assigned to class descriptor fields in the order in which they appear inside the class descriptor. Once the fields of the class descriptors have had their offsets assigned, the sizes of individual objects are known. Before this pass has run, optimizations are free to rearrange object fields inside class descriptors to suit their needs, for example to optimize caching behavior.

This approach to object-field offsetting enforces that additional data structures and code that needs to be generated by the front-end (e.g. data structures to support Java's introspection facilities), need to be generated using front-end installed call-backs. In the interface between front-end and LASM, the front-end can install a function that is called once for each class descriptor after all potential object layout transforming optimizations have run (and after object field offsetting).

After the object layout has been fixed, the code is rewritten, replacing symbolic field accesses by hard memory references. The field access at line 8 in Figure 4.3 would be changed to **('g', %g21704 + 20)* to denote that field *a* is located at offset 20 from (pointer) register *%g21704*, and that offset needs to be indirected to set the field's value.

Access checks are handed to LASM in the form of instantiations of class *LasmAccessCheckInstruction*. After all access-check-related optimizations have finished, the checks are replaced by ordinary LASM instructions and passed to remaining LASM optimization and code generation passes alongside with the other code. Having a *LasmAccessCheckExpression* class simplifies the process of removal, identification and insertions of access checks. Moving an access check is now as simple as removing a single object from a list and inserting it elsewhere, instead of having to transport a series of instructions.

4.4 Classic Optimizations

Classic optimizations are optimizations that improve sequential code efficiency. This includes removing redundant or duplicate calculations and replacing inefficient code sequences by more efficient sequences. Since Jackal's goal is to support efficient execution, sequential efficiency is important, especially because Jackal is targeted towards demanding applications such as massive scientific computations. Many of these optimizations are described in Muchnick's book on compiler optimization [45]. The most important classic optimizations that LASM implements are: global common subexpression elimination [45]; lazy code motion [34, 45]; loop-invariant code motion; loop unrolling; seek longest instructions; constant multiply/divide elimination; array index bounds check elimination; static initializer check elimination; register promotion; strength reduction; code straightening; jump chain removal; function inlining; and tail call elimination.

4.4.1 Method Inlining

An important early optimization worth special mention is method inlining. Method inlining replaces a call to a method by its definition. As the definition may reside in a different source code file, the LASM code to be invoked may have to be read from disk. Also, Java complicates method inlining by allowing both dynamic linking and binding. Dynamic linking and binding allows one to load bytecode files from disk into a running program, which reduces the number of assumptions that can be made by the compiler for optimization purposes. Dynamic binding reduces the compiler's opportunities for interprocedural optimizations as the compiler can almost never be completely certain about the identity of the method called.

Jackal implements dynamic linking by supplying two compilers; one for heavy optimization to be called before program execution and one lightweight compiler dedicated to bytecode compilation at runtime. The runtime bytecode compiler translates bytecode to C, which can then be compiled to executable code by the system's C compiler. The dynamic bytecode compiler is described in [40].

The limitations on method inlining imposed by dynamic binding and linking, can be reduced both by the application programmer and the compiler. The application programmer can annotate the code with *final*, *static* and *private*. *Final* allows the programmer to assert that a class will not be inherited from, not even in the presence of dynamic linking. When applied to a method, it states (and enforces) that the method will not be overridden. *Static* and *private* allow the programmer to ensure that only one version of a method will be called and that it will not be overridden. This approach has the problem that it requires programmer cooperation.

Alternatively, a compiler (but not Jackal) can add checks to the generated code or test that no dynamic loading will occur in the program being compiled or run to ensure that its assumptions are valid and only then apply its optimizations. However, this approach complicates the compiler to a certain degree and its success is not guaranteed, as the compiler must often make conservative assumptions to guarantee program semantics.

Finally, the compiler and programmer can cooperate by allowing the programmer to assert a *closed-world assumption*. A closed-world assumption implies that no dynamic linking will occur. This approach has been taken by Jackal. Once the compiler knows that no dynamic loading will occur, the front-end uses type inference to generate, for each call site, a list of functions that might be called at that point. LASM can then, for example, use that information to specialize and optimize those call sites. Enabling a closed-world assumption is implemented through a command-line flag to the compiler.

4.4.2 Loop-Invariant Checks and Array Prefetching

The optimization passes that perform loop-invariant code motion have been extended to lift access checks from loops. The rules for lifting an access check from a loop are, however, slightly stronger than for normal instructions: an access check can be lifted from a loop only when there is no synchronization in the loop. Otherwise updates to the lifted variable made by other threads would not become visible within the loop.

When a check to a normal object or to a fixed array element (for example, A[2]) is

```
int z = 1;
for (int i=5;i<390;i+=3)
{
        p1 =  z*mul1 + inc1;
        p2 = p1*mul2 + inc2;
        p3 = p2*mul3 + inc3;
        p4 = p3 % mod;
        A[p4] = i;
        z += 2;
}
```

Figure 4.5: Power of strength reduction.

lifted there is no problem: the check can simply be moved in front of the loop. However, when the check to be lifted checks an array element indexed by a loop variable we have the option of checking either the whole array or only the segment of the array actually accessed. LASM tries the latter by calculating the bounds of the array indexing expression and will insert a call to the runtime system to fetch the array segment before the loop. If the analysis fails, the programmer can request through a compiler switch that the whole array be prefetched or that the access check be left in place.

The compiler analysis for array prefetching is built upon strength reduction analysis [35, 45]. Strength reduction analysis provides the array prefetch analysis with a set of induction variables. An induction variable is a tuple *(variable, MUL, INC, DEP)* where the tuple describes an assignment: *variable = (DEP×MUL + INC)* inside the loop. DEP is another variable, MUL and INC are loop invariants.

The first step in the analysis is to calculate the set of induction variables *V* for a loop *P*. For each access check to an array element *A[X]* encountered in *P*, the set of induction variables *V* is searched for an induction variable *T* that describes *X*. If found, a call to *prefetch_read_array_segment* or *prefetch_write_array_segment* is inserted before the loop with *A*, *T*, the loop variable and loop bounds as arguments.

If *T* is not found, an assignment *X = E modulo C* is searched for. If the assignment is found, E is searched for in the set of induction variables. If *E* is found, the call to the runtime system is inserted before the loop, with *A*, *T*, *C*, the loop variable and loop bound as arguments. This enables the analysis to withstand a single modulo operation inside the array indexing expression.

The analysis is strong enough to recognize loop bounds for many polynomial expressions, as the example in Figure 4.5 shows. Since all *mul* and *inc* variables are loop invariant, the strength reduction analysis will yield the induction variable tuples in Figure 4.6.

The next step in the analysis attempts to do substitution in each tuple to remove inter-tuple dependencies. This creates the formula *p3 = mul3*mul2*mul1*z + mul3*mul1*2 + mul3*mul2*inc1 + mul3*inc2 + inc3*, which is only dependent on basic loop indirection variable 'z'. This step will yield the tuple in Figure 4.7 for p3. Variables that are passed to the runtime system are the tuple coefficients of *p3*, a reference to the array to be prefetched, the bounds of the loop and the value of *mod*. The resulting call to the runtime system will look like that shown in Figure 4.8. The pseudo-code for the array prefetching

```
(p3, mul3, inc3, p2)
(p2, mul2, inc2, p1)
(p1, mul1, inc1, z)
(z,     1,    2, z)
(i,     1,    3, i)
```

Figure 4.6: Loop induction variables.

```
{
  p4,
  mul1 * mul2 * mul3,
  mul3*mul1*2 + mul3*mul2*inc1 + mul3*inc2 + inc3,
  z,
  mod
}
```

Figure 4.7: Final loop induction variable for p4.

```
prefetch_write_array_segment(A,
// describe p4
                           mod,
                           mul1*mul2*mul3,
                           mul3*mul1*2 + mul3*mul2*inc1 + mul3*inc2 + inc3,
// 'z's start value
                           z,
// describe 'i' (start, end, step)
                           5,
                           390,
                           3);
```

Figure 4.8: Resulting call the RTS inserted into the code.

```
FOR EACH FUNCTION f IN program,
DO
    run loop-invariant code motion pass over f;

    FOR EACH LOOP L IN f
    DO
        IF (NO "call to lock(object)" IN L, recursively) THEN
        DO
           IV = gather_induction_vars(L);

           FOR EACH "access_check( A[ X ] )" IN L
           DO
                P = induction var for X IN IV
                IF (P EXISTS) THEN
                DO
                   INSERT BEFORE L, "array_prefetch(A, P, params(L), infinite)"
                ELSE
                    Z = "X = Y modulo G" IN L
                    IF (Z EXISTS and loop_invariant(G)) THEN
                    DO
                        P = induction var for Y IN IV
                        IF (P EXISTS) THEN
                        DO
                           INSERT BEFORE L, "array_prefetch(A, P, params(L), G)"
                           REMOVE "access_check( A[ X ] )" from L
                        DONE
                    DONE
                DONE
           DONE
        DONE
    DONE

    run loop-invariant code motion pass over f;

    replace "array_prefetch(X); array_prefetch(X)" by "array_prefetch(X)"
DONE
```

Figure 4.9: Pseudo-code for array prefetch analysis.

algorithm is included in Figure 4.9.

After the array prefetching analysis has run, array prefetch calls as inserted before the loop may be combined to form larger array prefetches. A simple example where this might occur is shown in Figure 4.10, as handed to the back-end by the front-end. Figure 4.11 shows the code after loop-invariant code motion. Figure 4.12 shows the code after the array prefetching code transformations. The inserted array prefetch code in the outer loop is trivially loop invariant and lifted, after which two identical array prefetches appear next to each other and one is removed. The final code is shown in Figure 4.13.

```
class Combinable {
    void foo(int A[]) {
        for (int i=0;i<10;i++) {
            for (int j=0;j<10;j++) {
                //access_check(A[ j ]);
                //access_check(A[ i ]);
                A[ j ] = A[ i ] + 1;
            }
        }
    }
}
```

Figure 4.10: Array prefetch optimization, starting point.

```
class Combinable {
    void foo(int A[]) {
        for (int i=0;i<10;i++) {
            //access_check(A[ i ]);
            for (int j=0;j<10;j++) {
                //access_check(A[ j ]);
                A[ j ] = A[ i ] + 1;
            }
        }
    }
}
```

Figure 4.11: Array prefetch optimization, after loop-invariant code motion.

```
class Combinable {
    void foo(int A[]) {
        //array_prefetch(A, 0, 10, infinite)
        for (int i=0;i<10;i++) {
            //array_prefetch(A, 0, 10, infinite);
            for (int j=0;j<10;j++) {
                A[ j ] = A[ i ] + 1;
            }
        }
    }
}
```

Figure 4.12: Array prefetch optimization, arrays prefetched.

```
class Combinable {
    void foo(int A[]) {
        //array_prefetch(A, 0, 10, infinite)
        for (int i=0;i<10;i++) {
            for (int j=0;j<10;j++) {
                A[ j ] = A[ i ] + 1;
            }
        }
    }
}
```

Figure 4.13: Array prefetch optimization, array prefetch optimized.

```
class List {
    int payload;
    List Next;
}


class Alias {
    List Head;

    void Add(List l) {
        l.Next = this.Head;
        this.Head = l;
    }

    public static void main(String args[]) {
      Alias A = new Alias(); // A=(1)
      List L1 = new List();  // L2=(2)
      List L2 = new List();  // L3=(3)
      A.Add(L1);
      A.Add(L2);
    }
}
```

Figure 4.14: Alias/heap graph example (1).

4.4.3 Alias Analysis

Alias analysis provides optimizers with information to determine if, at compile time, two object references possibly or certainly refer to the same object at runtime. LASM exploits Java's property that two objects allocated with *new* can never alias each other (*new* always returns unique results) and that all objects must be allocated either with *new Object* or *new Type[]*. The front-end calls only two allocation primitives: *new_object* and *new_array*. LASM optimizers can easily find these allocation points by searching for the proper call instructions. LASM then creates a graph which tells for each allocation point how it relates to all the others. For example, the graph depicted in Figure 4.15 is created for the Java code in Figure 4.14. A node in the heap graph denotes an allocation site. An edge

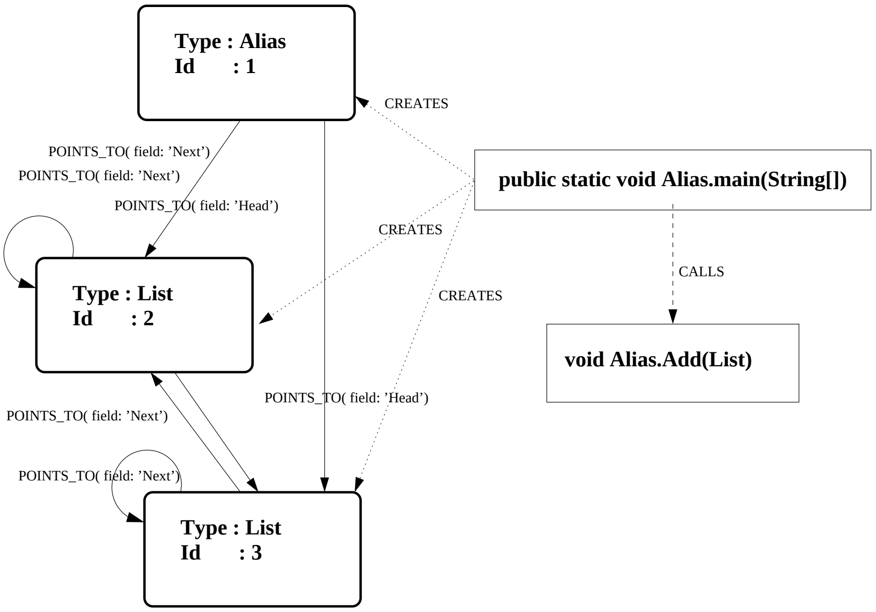

Type : Alias
Id : 1
CREATES
public static void Alias.main(String[])
POINTS_TO(field: 'Next')
POINTS_TO(field: 'Next')
POINTS_TO(field: 'Head')
CREATES
CALLS
Type : List
Id : 2
CREATES
void Alias.Add(List)
POINTS_TO(field: 'Head')
POINTS_TO(field: 'Next')
POINTS_TO(field: 'Next')
Type : List
Id : 3

Figure 4.15: The generated heap graph.

in the graph shows that a field of the object allocated at one allocation site may contain a reference to an object allocated at another allocation site. We first describe how this graph is created, then how it is used by the optimizers.

The algorithm is based on a data-flow algorithm that propagates information around the whole program. It has an algorithm setup phase to initialize the data flow, followed by iterative propagation passes that are run until no change occurs anymore (a fixpoint is reached).

The first step is to convert all routines to Single Static Assignment (*SSA*) form [13] to simplify the analysis. The next step in the analysis is to allocate nodes in the graph for each *new_array/new_object* in the program, and to assign a unique number (an allocation-identifier or allocation-id for short) to each of the graph nodes.

Each instruction is given an initially empty set of data flow items where each item is a tuple *(location-expression, allocation-id list)* (such a tuple can also be termed an aliasing-set). Such a tuple describes for each location-expression (which may be some register, memory reference, parameter or local variable) which allocation-ids (allocation sites) are possibly referenced. The tuple associated with the instruction that describes a new object or array invocation is initialized with the tuple *(return value register expression, (new object's allocation id))*. This step concludes the data flow initialization phase. The propagation phase of the algorithm propagates these allocation-id sets throughout the function's Control-Flow Graph (CFG).

Whenever an assignment $A = B$ is encountered, the tuples associated with B are searched for and new tuples for A are created with the same allocation-id list as B. Whenever a call is encountered, the tuples associated with its arguments are passed to all the called functions (there may be many called methods because of inheritance), and the tuples associated with the returned expressions of the called functions are added to the call instruction's tuple set (all tuples *X* associated with each of the expressions *Y* inside any *return Y* instructions in the called functions).

When no instruction in the program has anything more to contribute to any other instruction's set of tuples, the heap graph is constructed. The heap graph is created by finding assignments of the form *(A.B) = C*. Here the allocation-ids (which map equally to graph node numbers) associated with *C* and *A* are searched for in the set of data flow items associated with the assignment instruction. If both data flow items are found, edges in the graph are created from all allocations associated with A to all allocations associated with C. The edges are labeled with *".B"*.

Because a single data flow iteration described above may not be enough to propagate all data through the program to create a complete graph, the system terminates only when the graph cannot be made more precise and no data can be copied from one instruction to another.

Whenever we encounter an instruction *P* resembling *Z = A.B*, and a tuple has already been associated with *A*, we can use the heap graph constructed thus far to continue the data flow propagation process. In the tuple *(A, ID)*, for each allocation id in *ID*, we locate its associated heap node *W*. For each heap node *W* we find the *B* field and its associated aliasing allocations *Q*. Now we add for each *Q* new tuples (*A.B, Q*) and (*Z, Q*) in *P*'s aliasing set.

At the end of the analysis, each instruction will have associated with it a set of tuples *(location-expression, allocation-id list)* that tells unambiguously, flow-sensitively, which objects *location-expression* might point to. However, the information can become imprecise at control-flow joins, as the information from each control-flow branch is merged due to the data-flow process. For example, consider an if-then-else statement with an allocation statement in both the if and else part which assigns its result to *P*. After the if-then-else, control-flow merges and *P* will point to either of the two objects. Also, we cannot distinguish between the elements of dynamic data structures such as linked-lists, but can only determine that some object is part of a linked-list due to its properties in the heap graph.

Another problematic control-flow merge is the return statement. The return statement can be problematic if the application programmer has written his own allocation routines. All objects allocated in such functions will be summarized in one graph node while the objects allocated might be used for different purposes, whereas the analysis requires to keep them apart for good optimization.

SUIF [2] uses partial transfer functions to solve this problem, effectively cloning the programmer's allocation routines at each call site. Another option is to aggressively inline functions containing object allocations to truly create multiple object allocation sites. LASM currently uses a mixture of the two approaches by allowing both inlining and method cloning. Inlining increases code size while method cloning offers the possibility of merging similar clones afterwards.

To demonstrate what the analysis finally produces, let us examine the program in Figure 4.14. The pseudo-code for the heap analysis algorithm is shown in Figure 4.16.

```
OPERATOR A += B, WHERE A OF TYPE SET IS
     IF B NOT IN A THEN DO
        nothing_changed = false;
        ADD B TO SET A;

PROCEDURE initialize(program) IS
    INTEGER alloc_id = 0;
    FOR EACH FUNCTION F IN program DO
      convert F to SSA form;
      FOR EACH I = "P=new object" OR "P=new array" IN F DO
         alias(F, I, P) = {alloc_id};
         alloc_id = alloc_id + 1;

FUNCTION next(I) IS
         IF (I IS jump) RETURN JUMP_TARGETS(I)
         ELSE           RETURN I+1;

PROCEDURE local_propagate(program) IS
    FOR EACH FUNCTION F IN program DO, FOR EACH INSTRUCTION I IN F DO
                IF I == "R = call function(A,B,C,D, ...)" DO
                   FOR EACH PARAMETER P TO "function (A,B,C,D, ...)" DO
                       alias(function,0,formal(P)) += alias(F,I,P);
                   alias(F,I,R) += function.return_aliases;
                   alias(F,I+1,Z) += alias(F,I,Z);
                ELSE IF I == "return X" THEN
                   F.return_aliases += alias(F,I,X);
                ELSE
                  alias(F,next(I),Z) += alias(F,I,Z);

PROCEDURE try_to_complete_graph(program) IS
    FOR EACH FUNCTION F IN program DO, FOR EACH INSTRUCTION I IN F DO
            IF (I == "A.B = C") THEN
                FOR EACH X in alias(F, I, A) DO
                    FOR EACH Y in alias(F, I, C) DO
                        graph += {node(X at "B", Y)};

PROCEDURE global_propagate(program) IS
    FOR EACH FUNCTION F IN program DO, FOR EACH INSTRUCTION I IN F DO
            IF (I == "A = B.C") THEN
                FOR EACH X in alias(F, I, B) DO
                    FOR EACH node(X at B, Z) in graph DO
                        alias(F,I,A) += {Z};

PROCEDURE create_heap_graph(program) IS
        initialize(program);
        REPEAT
           nothing_changed = true;
           local_propagate(program);
           try_to_complete_graph(program);
           global_propagate(program);
        UNTIL nothing_changed == false;
```

Figure 4.16: Pseudo-code for heap graph creation.

```
class List {
    int payload;
    List Next;
}

class Alias {
    List Head;

    void Add(List l) {
          // this=(1), l=(2,3)
        l.Next = this.Head;
          // this=(1), l=(2,3)
        this.Head = l;
    }

    public static void main(String args[]) {
      Alias A = new Alias(); // A=(1)
      List L1 = new List();  // A=(1), L1=(2)
      List L2 = new List();  // A=(1), L1=(2), L2=(3)
      A.Add(L1);  // A=(1), L1=(2), L2=(3)
      A.Add(L2);  // A=(1), L1=(2), 12=(3)
    }
}
```

Figure 4.17: Alias/heap graph example (2).

The program is first translated into LASM code by the front-end. The back-end then detects the three object allocation sites (the two new *Lists* and the single new *Alias*) and assigns each a unique number *(1, 2 and 3)*. Those numbers are propagated throughout the program, flowing through the calls to *Alias.Add*. In *Alias.Add* the two assignments have associated with them the sets *(1)* and *(2,3)*. This results in the code in Figure 4.17.

The graph creation routines then look at the individual assignments in *Alias.Add* for uses of object fields and assignments to object fields. For example, the first assignment in *Alias.Add*: *l.Next = this.Head* dereferences *l* which has associated with it the set (2,3), coincidentally the same set as *this.Head*. Because we are assigning to an object field we are now able to add edges to the heap graph: all permutations from *(2,3)* to *(2,3)* (which includes the self edges for graph nodes *(2)* and *(3)*). The second assignment creates edges from *(1)* to *(2)* and *(3)*. Finally the analysis creates the graph shown in Figure 4.15. The dotted arrows are object allocation edges. A dashed arrow between two functions denotes an edge in the call graph. Solid arrows between two object allocations denote aliasing relations.

More precisely, a solid edge denotes that objects allocated from the edge-source allocation site contain a field which may point to objects allocated by the edge-target allocation site. Note the use of the word ’may’ because the actual assignment to the reference field of the source object may be conditional (located inside an *if* statement for example). The the graph does not include any quantity qualifiers for either edges or nodes as the compiler is unable to determine how many objects are allocated at a given allocation site.

The subsequent optimizers described below and in subsequent chapters (escape analysis, object inlining, DAG detection, etc.) use this graph to determine if one object will ever contain a pointer to another object and which objects a reference may ever point to.

4.4.4 Escape Analysis

Escape analysis looks at each object or array allocation to determine if the object created will be reachable by other threads. An object accessible by other threads is called *escaped*. An object *X* can escape a thread by storing a reference to *X* inside a global variable or by storing a reference to *X* inside an already escaping object. Objects that both do not escape their creating function and thread can be allocated on the (call) stack reducing the need for garbage collection. Once an object has been stack-allocated, other compiler optimizations might apply. For example, after stack allocation, the object's fields are available for copy propagation, constant folding, register promotion and others increasing sequential execution speed. Much work has been done on fast escape analysis and very aggressive escape analysis [11].

Also when an object is stack-allocated, all synchronization actions upon the objects can be removed. To preserve correctness, however, only the mutual exclusion effects of lock and unlock statements on stack-allocated objects are removed. Memory is still flushed at each lock and unlock statement. This behavior is introduced by JSR-133, because when no other threads can see the object, they will not be able to see any synchronization actions upon the object either.

Jackal's compiler starts the analysis by creating a heap graph. The heap graph nodes are then annotated with escape analysis information as follows. First, each assignment to a global variable (a static variable in Java terminology) in the program is located. Once such an assignment *global = var* has been found, *global* is located inside the aliasing list associated with the assignment instruction, yielding a set of object allocation id's. The allocation(s) associated with the allocation id's are marked "escaped" in the heap graph.

When all assignments to global variables have been processed, the graph is processed. Each object that inherits from *java.lang.Thread* or implements the *java.lang.Runnable* interface is marked escaped. Finally, each object that is pointed to by an escaping object is also marked escaped (with closure). Afterwards, all objects not marked escaped can be stack-allocated.

For generality of the implementation, the objects are by default not allocated on the call stack but on a separate stack, as the call stack is assumed small and fixed while nonescaping objects can be arbitrarily large. Also, with modern processors, small stack frames increase caching performance as the top few stack entries can be cached inside processor registers. The stack for nonescaping objects is managed independently of the call stack (*nonescaped-object-stack*) using a mark-release allocator. Upon entry of a procedure containing nonescaping objects, the position of the top of the escape-stack is recorded inside a local variable (eligible to register promotion). At procedure exit, the top of the stack is reset to the old value (mark-release allocation).

There is one problem with the approach outlined above; an object might not escape the thread of control but might escape the allocating procedure as shown in Figure 4.18. Here a user has created his own object allocation procedure *my_alloc*. The object created by the call to *my_alloc* inside *start* does not escape the thread calling *start* but cannot be

```
class Escape {
    void work() {
        // Code
    }
    Escape my_alloc() {
        return new Escape();
    }
    void start() {
        Escape p = my_alloc();
        p.work();
    }
}
```

Figure 4.18: Failing escape analysis.

stack-allocated because *start* would hold a reference to an object that has already been removed at the exit of *my_alloc*.

Our solution is to either clone or inline *my_alloc* if possible or to omit calls to record and reset the state of the nonescaped-object-stack from functions that contain nonescaped-object allocations. Currently, cloning and inlining of functions is aggressively performed during the heap graph creation process and calls to reset the state of the nonescaped-object-stack are only omitted from object constructors. However, in the latter case, the stack growth may be unbounded for some programs, as the compiler will in general not know how many times the call to *my_alloc* will be executed. This might happen, for example, if an object allocation is performed inside a loop.

A common programming idiom is to allocate subobjects of a parent object from the parent object's constructor. To still allow for nonescaped stack allocations of these subobjects, constructors never mark-release the nonescaped-object-stack. This enables (related) objects allocated inside a constructor to be allocated on the *escape-stack* even if they outlive the function (constructor) that created them.

4.4.5 Access-Check Elimination

The front-end adds access checks for all object accesses, regardless of whether the added check is redundant. This supports the modularity principle; the goals of generating (LASM) code and optimizing LASM code are neatly separated.

An access check *X* is redundant whenever it has already been performed on all paths in the control-flow graph (CFG) that can reach *X* without intervening synchronization. For example, in Figure 4.19 the third check is redundant. The check removal algorithm first converts each function to SSA form. Next, the compiler performs a complete backward search over the CFG to locate instances of *check(X)* to determine that indeed on all paths, object *X* has already been checked. The backward search is implemented by assigning to each label *L* a set of jump statements that can reach label *L*, and, when walking back, upon hitting the label recursively continue the walk at the jump statements.

Because the program semantics need to be observed when removing access checks, removing a write check requires us to change *all* preceding read checks to write checks.

```
class Example {
    class Data { int d; }
    int p;
    void foo(boolean c, Data d) {
        if (c) { // 1:check(d,write);
                 d.d--;
        } else { // 2:check(d,read);
                 p = d.d;
        }
        // 3:check(d,write);
        d.d *= 2;
    }
}
```

Figure 4.19: Access check elimination.

```
class Example {
    class Data { int d; }
    int p;
    void foo(boolean c, Data d) {
        if (c) { // 1:check(d,write);
                 d.d--;
        } else { // 2:check(d,read);
                 p = d.d;
        }
        // same conditioned if:
        if (c) { // 3:check(d,write);
                 d.d-=3;
        } else { // 4:check(d,read);
                 p += d.d;
        }

    }
}
```

Figure 4.20: Access check elimination, duplicate if conditions.

```
class Example {
    class Data { int d; }
    double p;
    void foo(boolean c, Data d) {
        if (c) { // 1:check(d,write);
            d.d--;
        } else { // 2:check(d,read);
            p = d.d * Math.cos(1.2);
        }
        // 3:check(d,write)
        d.d *= 2;
    }
}
```

Figure 4.21: Access check elimination, call to Math.cos.

See the example in Figure 4.19: when check *3* is omitted, check *2* needs to be changed to a write check to preserve program semantics. Thus when traversing the CFG backwards to eliminate a write check(X), any encountered read check(X) is turned into a write check(X). This approach may be too aggressive if the access check to be eliminated is executed conditionally. An example is shown in Figure 4.20 where an if statement is duplicated elsewhere. When traversing the CFG to eliminate access check *3*, access check *2* would be turned into a write check while the object itself could have remained in read mode, allowing for more runtime optimizations.

Access-check elimination becomes more complex in the presence of method invocations on a control-flow graph path. When a method invocation is encountered, the invoked method may be unknown, implement the access check that we are currently searching for or it may implement synchronization actions causing the thread's cache to be flushed. For this purpose, LASM performs a lookup on the invoked method(s). Each LASM method maintains a bit that shows if that method (or any of its called methods) contains some synchronization action. If set, the control-flow path under consideration is abandoned and the search fails. Currently no search is performed to see if the invoked method implements a check(X) when check(X) is being eliminated. Instead, we rely on inlining to expose more access checks to the optimizer.

Because not all the methods invoked are compiled using a LASM back-end (native methods and DSM support code), LASM maintains a list of externally compiled functions. Each list entry is a tuple of (flushes-memory, allows loop-invariant code motion, throws exceptions). Methods for which entries are maintained include *cos*, *sin*, *non_escape_new_object* and others that can be safely ignored by the access-check elimination algorithm. This allows code containing JNI[1] calls to be analyzed and optimized.

An example where this is useful, is shown in Figure 4.21. The call to Math.cos is implemented using the native *cos* from the platform Jackal is compiled for. Since *cos* does not implement any synchronization actions, *cos* can be safely ignored, which allows elimination of the third access check.

[1] Java Native Interface (JNI) is Java's standard interface to code written in other languages, most notably C and C++.

4.5 Summary

We have described Jackal's compiler and many of the optimizations. The compiler is able to perform cross-file optimizations using a data base of intermediate code. This allows for whole program optimization. The class libraries are pre-compiled but their intermediate LASM codes are kept for analysis.

We have described the algorithms to perform loop-invariant code motion on access checks. This allows Jackal to prefetch whole arrays or only the section of an array that is used inside a loop. A side effect of array prefetching is that code might execute faster because not each array element access needs to be access checked. One call is inserted in front of the loop to check all array elements at once. Array prefetching will fail if a loop contains synchronization code. Detecting such a case requires interprocedural analysis.

The escape analysis algorithm LASM implements has been briefly discussed. Escape analysis reduces pressure on the garbage collector but also allows access checks to nonescaping objects to be removed.

Finally, the access-check elimination optimization was described. Access-check elimination requires interprocedural analysis to determine that a call encountered upon a path in the CFG does not execute any synchronization statements.

Pulling access checks from loops has already been performed in Shasta [54]. but the method introduced in this chapter is more flexible. Escape analysis has been implemented in several earlier systems but none use escape analysis as a means to perform access check removal.

Chapter 5

The Garbage Collector

5.1 Introduction

To support Java's automatic memory management, Jackal implements a (cluster wide) Garbage Collector (GC). Because threads are allowed to exchange references, Jackal has to handle both intermachine and intramachine references. Whenever an object has been sent to another machine, it cannot be locally removed without querying the machine holding the copy if it has become garbage there as well. Also, the objects referenced by the transferred object (exported references) cannot be removed as at any time later in the execution of the program the remote machine can start using one of the contained references. Objects that have not been involved in any communication (have not been sent over the network themselves and no reference to them was placed in an object that has been sent over the network) can be removed without querying other machines.

The allocator is an efficient page based allocator. Small objects are allocated within a single page using a bump allocator. When a page is full, a new page is allocated. A page is currently four kilobytes. Large objects (larger than a single page) are allocated by allocating multiple pages. A consequence of this object allocation method is that objects cannot be freed individually, only whole pages can be de-allocated at a time.

We have chosen mark-and-sweep GCs for both the local and global GCs because mark-and-sweep collectors minimize execution times rather than pause times and have little space and time overhead when an application performs no allocations at runtime. Also, because of our target applications (scientific and generally compute-intensive applications), we wish to avoid any unnecessary pointer indirections or the costs of maintaining stack-maps that a generational/copying GC will require. Finally, because Jackal exchanges pointers to objects between machines, a copying GC will complicate Jackal's runtime system as pointers will have to be translated at message receipt and transmission.

To minimize GC related communication, Jackal implements both a local and a global garbage collector (GC). Each machine runs a local mark-and-sweep GC when it runs out of memory. A local GC does not involve any synchronization or communication with other machines, but will not free any object a remote machine might hold a reference to. This is vital, because we expect that most objects are not shared between machines and can be recycled without communication.

When too many locally allocated objects are (potentially) remotely reachable because a reference to them has been exported, local GCs will fail to reclaim enough memory. At that time a global GC is initiated; a global GC finds *all* garbage at the expense of communication and synchronization. Jackal's global GC algorithm involves all machines and performs a distributed mark-and-sweep.

The contribution made in this chapter is:

- a description of a new global garbage collection algorithm for this DSM system. The algorithm differs from other parallel garbage collection algorithms for Java, because it has to deal with replicated objects and partitioned arrays. This work has been published in [59].

5.2 Local Garbage Collection

Jackal's local garbage collector is a straightforward mark-and-sweep collector. A machine initiates a local GC when its heap reaches its maximum size. It then visits and marks all objects that are reachable from its root set (defined below) by using a to-do list of objects that are to be marked. The to-do list is implemented using the 'next_object' pointer field inside the object header (see the object header in Figure 3.2 in Section 3.3). In the mark phase, only references to local objects are followed; references to objects created by other machines are skipped. In the sweep phase, all unmarked objects created at the current machine are freed. If a local GC fails to free enough memory, a global GC is initiated. At present, Jackal initiates a global GC when less than a quarter of the local heap is free after a local GC or more than 1 Mbyte (arbitrary) of local objects is exported to other machines.

For local GCs, a machine's *root set* consists of:

- all global variables that are of a reference type (object or array);
- all references stored in thread stacks,;
- all exported references to local objects;
- and modified cached regions of all threads.

The last two items of the root set are what sets the local GC apart from a normal Java implementation's local GC. Exported references are added to the root set because Jackal must deal with remote machines that hold references to local objects for which no local references exist anymore: such an object must not be deleted by a local GC, because it may still be reachable from another machine's root set. To detect which objects are potentially referenced by remote machines, Jackal inspects all outgoing data messages for pointers to local objects. These pointers are then added to the local *export table*, which, in turn, is added to the local GC root set. To aid in deciding whether a local or global GC should be initiated, the export table records how many bytes the exported objects consume.

The second addition to the root set originates from local pointers that may have been inserted into a cached region. For example, a thread may have cached a region and written a pointer to a local object in it. If the region has not yet been flushed to the home node, the packet inspection routines have not been given a chance to add the local pointer to the export table. If we were to ignore cached regions, the local object would be incorrectly considered garbage.

A solution would be to always flush working memory before each garbage collection phase but aside from the performance degradation incurred, there is an associated correctness problem. We would be required to asynchronously flush the working memory of all threads. However, we cannot flush a thread's working memory at arbitrary times as then it would be possible to flush a threads working memory in between an access check and the actual object usage it protects. A modification could then be potentially lost as it would not be shipped to the home-node copy.

Since we are thus required to mark the regions recorded in the flush lists, the code to scan regions for references has to scan either single cached array partitions or whole cached objects. This sets our garbage collector apart from others (see Section 5.5 below) that do not have to deal with partitioned arrays.

To speed up packet and object inspection, the compiler emits an offset table for each object type so that pointers in an object or packet can be identified unambiguously. The compiler also emits a 'primitive' bit for each class that indicates whether or not an object or array can contain pointers. This way we can quickly identify objects that contain no pointers and have a fast way to locate pointers in other objects.

The pseudo-code for both the object allocator and local garbage collector is included in Figure 5.1. The method *find_all_objects_in_local_GC_root_set()* (not shown) will return the pointers in the root set as defined above in the form of a single linked list with each object linked through the *next_object* pointer allocated in each object header.

5.3 Global Garbage Collection

Global Garbage-Collection (GGC) phases are triggered for two reasons in our system: either the local garbage collector fails to free enough memory because the export table has grown large or an attempt is made to map too many objects, which causes the cached address space to become too large. The global garbage collector does not include the export tables in its root set and therefore objects that have at one time been exported can be deleted. The global GC, as a part of the sweep process, removes all exported, unreachable objects from the export tables from all machines. This allows the local GC to function again afterwards.

The cached address space may become too large when one machine attempts to read data from other machines. Since a single machine does not have enough main memory to hold the entire address space (as some main memory is reserved for other uses by the Jackal runtime system or it may simply be the case that the machines do not have much physical main memory available), a global GC will free memory that is currently unused.

Both cases are handled by our global GC algorithm. Once a machine has decided that a global GC is needed, it requests all other machines to participate in a global GC. Race conditions between multiple concurrent global GC requests are avoided using a centralized lock maintained by machine 0.

First each machine starts marking its local objects starting from its root set. In con-

```
static page *current_page = allocate page;

void *allocate_object(type t) {
   int bytes = size of type(t);
   ObjectHeader *new_obj = NULL;
   if (bytes > page size) {
       new_obj = allocate_enough_pages(bytes);
   } else {
       page *p = current_page;
       if (p->free < bytes)
           p = current_page = allocate_new_page();
       new_obj = p->current_ptr;
       p->current_ptr += bytes;
   }
   initialize_object(new_obj, t);
   return new_obj;
}

void local_GC() {
   stop_all_threads_on_this_machine();
   to_do = find_all_objects_in_local_GC_root_set();
   mark_all_pages_unmarked();
   while (not empty(to_do)) {
      ObjectHeader *p = take_one(to_do);
      if (object local to this machine(p) and
          object p not yet marked) {
          mark p and mark_page( object_to_page_nr(p) );
          for (each non NULL pointer Z in object p) {
              add Z to to_do
          }
      }
   }
   free_unmarked_pages();
   restart_all_threads_on_this_machine();
}
```

Figure 5.1: Allocator and Local GC Pseudo Code.

trast with a local GC, references to remote objects are followed. This is done by sending a *mark message* containing the remote object references to the object's home node (a mark message consists of a list of pointers to objects). When the home node receives a mark message, it extracts the pointers from the mark message and invokes the mark object routines on them. In the process it may generate more mark messages. To reduce communication overhead, remote references are buffered until a mark message is completely full (the mark message size is equal to the size of a packet of the underlying network layer). To handle home-node migration, the physical home node (the machine that originally created the object) is informed that the object is 'live' while the logical home node is marked and its containing references examined as *it* contains the up-to-date values of its fields. The mark phase of the global GC terminates when there are no more outstanding mark messages.

To determine that there are no more outstanding mark messages, each machine maintains a counter of outstanding mark messages. This counter is incremented when sending a mark message and decremented using acknowledgment messages sent by the receiver of the mark message. To improve performance mark messages are sent asynchronously and acknowledgments are only sent when there are no objects left to be marked.

After performing a mark phase, each processor performs a global reduce on the number of sent and processed messages over the entire system. Once there are no more unprocessed messages all processors begin with the sweep process.

After the mark phase, the global GC needs to clear dead cached objects. This is because the runtime system expects remote objects to be zeroed at the first use of the remote object (the runtime system uses a test against zero for the vtable field inside the object header to test whether or not the remote object's ObjectHeader and its DSM headers have already been initialized). Simply zeroing the vtable pointer inside all object headers is not sufficient, as after a global GC new objects may be allocated at different offsets than the old objects.

This scenario is exemplified in Figure 5.2. Here a large dead object is later overlayed by two new objects without the dead object being completely cleared. If the runtime system upon finding that the vtable fields in the ObjectHeaders of the two new objects are non-zero, had assumed that the object headers were already initialized, the system would crash as the object header's stale data would be illegally interpreted as valid headers.

To reduce the strain on the individual machine's virtual memory subsystem, specifically to reduce the number of pages mapped, Jackal attempts to not touch any pages *during* a global GC, that were not already mapped before the global GC. It attempts this by conservatively clearing objects. Since the information about where each remote object has been exported to has been lost at its home node (we only know that an object *has* been transferred), our only recourse is to broadcast the list of dead objects to each machine. Each machine will then clear each object on the lists received from each processor. However, we need to avoid clearing the dead objects at machines where the object has never been mapped. If we were to clear the object at machines it was not previously mapped at, we would increase (rather than reduce) the number of pages in use, negating the very objective we were initiating the global GC for.

To this end, before a machine removes any garbage objects, it traverses the list of (its own) dead objects and marks the pages they reside on dead inside a bitmap (128 K).

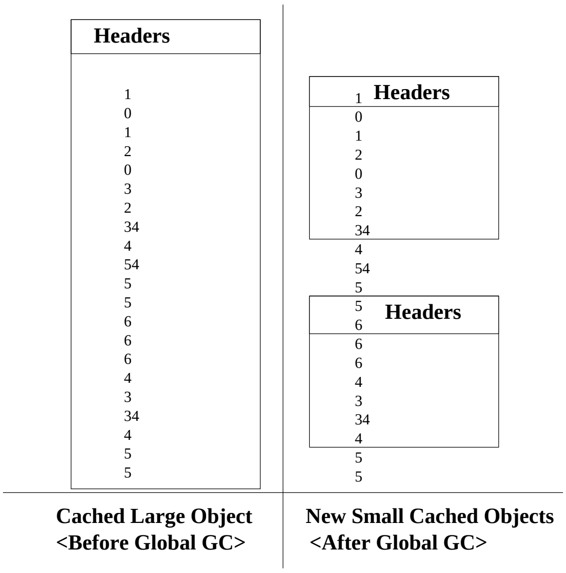


Figure 5.2: Overlaying a dead object with new objects without clearing.

After the GGC marking phase, each machine broadcasts its bitmap and merges it with the bitmaps received, creating a *global-garbage page-bitmap*. Afterwards each machine has a list of pages that are garbage. During normal operation, each time an object is locally mapped in, the pages containing the object are marked *mapped* inside the *accessed-bitmap*. In a GGC phase, after broadcasting both bitmaps, each machine broadcasts its complete list of pointers to objects that have at one time been sent over the network (determined using the *exported bit* set inside each communicated object) that have now become garbage. The garbage objects are removed from the exported hash and overwritten by zeroes if the underlying page has at one time been mapped (determined using the *accessed-bitmap*) and the vtable of the cached object is not zero. After the cached objects have been cleared, the *accessed-bitmap* is overwritten with the *global-garbage page-bitmap*.

Using both the *accessed-bitmap* and *global-garbage page-bitmaps* allows us to conservatively clear objects.

The pseudo-code for the global garbage collector is included in Figure 5.3.

5.4 Optimizations

Jackal performs an optimization to speed up global garbage collection. First, when marking, pointers stored in cached objects are followed: in effect, replicas of the home-node copies are used. This can speed up the marking process and reduce the number of messages sent but it will not reduce the number of pointers sent. Consider a linked list with nodes alternating on machine *X* and machine *Y* (see Figure 5.4). Both machines have recently traversed the list and it is thus completely available in memory on both machines.

```
void receive_mark_mesage(p) {
        add pointers in p to to_do;
}
void perform_global_GC_mark_phase() {
   mark_message[CPUS] = {};
   to_do = find_all_objects_in_glocal_GC_root_set();
   mark_all_pages_unmarked();
   while (not empty(to_do) or exist outstanding mark messages) {
      ObjectHeader *p = take_one(to_do);
      if (p == NULL or mark message[home(p)] is full) {
          if (mark message[home(p)] not empty)
              send mark_message[home(p)] to home(p)
          if (p == NULL)
             continue;
      }
      live_pages_map += page_nr_of(p)
      if (object local to this machine(p)) {
          if (object p not yet marked) {
              mark p and mark_page( object_to_page_nr(p) );
              for (each pointer Z in object p)
                  add Z to to_do
          }
      } else {
          // remote pointer:
          if (valid_object_header(p))
              mark p;
              for (each pointer Z in object p)
                  add Z to to_do
          mark_message[home(p)] += p;
      }
   }
}
void perform_global_GC_sweep_phase() {
   dead_pages_map = all pages - live pages_map;
   broadcast { dead objects, dead_pages_map }
   upon all broadcast receipts {
        clear all dead objects using all dead_pages_maps
   }
   remove dead objects from flush lists and exported hash
   free_unmarked_pages();
}
// machine zero upon receiving a global GC request:
void global_gc_request_at_machine_zero() {
    broadcast to all machines {
        stop_all_threads();              barrier();
        perform_global_GC_mark_phase();  barrier();
        perform_global_GC_sweep_phase(); barrier();
        restart_all_threads();
    }
}
// when a thread wants to initiate a global GC:
void start_global_GC() {
    send_global_GC_request_to_machine_zero();
}
```

Figure 5.3: Global GC pseudo code.

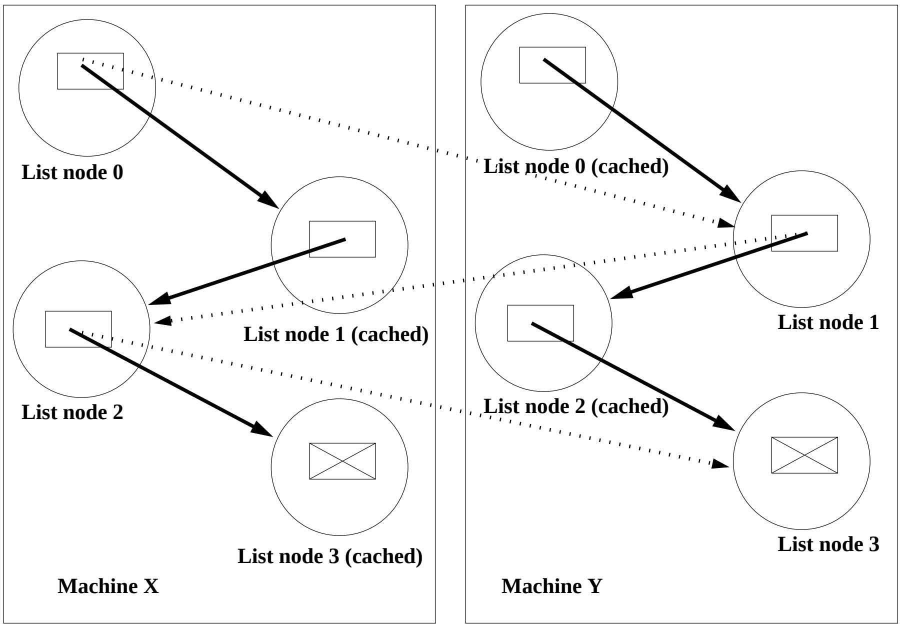


Figure 5.4: Optimization 1, garbage collecting a distributed list.

The mark phase can now alternate between the cached objects and its own heap, so it never needs to wait for information from other machines. Once both machines have received each other's mark message, they quickly discover that all references contained have already been marked and continue to the sweep phase. In this example, our GC algorithm would send two messages, independent of the length of the list.

A potential problem is that the cached copies might be stale: reference fields may have been overwritten by new references or *null*. Following these references might keep more objects live. We believe however that the reduction in the number of messages more than makes up for the possibility of a larger heap size.

We further optimize marking cached objects by avoiding page-mapping costs during garbage collection. When the garbage collector follows pointers into cached memory, it follows only pointers to mapped pages. Also, if a cached object or array spans multiple pages, we scan only the mapped pages. Testing whether a page is mapped is fast: the page map is stored in user space in the *accessed-bitmap*.

5.5 Related Work

Many garbage-collection papers focus on garbage collection in *distributed* rather than parallel systems. Plainfossé et al. give an overview of work in this area [48]. We focus on parallel systems.

With the parallel GC algorithm of Hughes [24], every machine can initiate a global

GC concurrently with other machines. Unlike our system, it assumes an unlimited number of pages to support the global address space and does not deal with software caches.

Some work has been done on garbage collection in DSMs. Kogan and Schuster use weighted reference counts per page [36]. Like most reference counting schemes, their scheme fails to reclaim cyclic structures that span multiple pages. We employ a mark-and-sweep garbage collector, which can free all cycles.

LEMMA [43] is a software DSM for ML with both local and global garbage collection. LEMMA uses a copying garbage collector. LEMMA assumes that the entire global address space fits into the memory of every machine. We do not make this assumption.

Taura and Yonezawa built a garbage collector for ABCL/f running on a cluster of workstations [56]. Their parallel garbage collector is based on the Boehm and Weiser collector [7], a reasonably efficient mark-and-sweep garbage collector. They do not handle caching.

Yu and Cox [65] built a garbage collector for parallel C and C++ programs that run on the Treadmarks DSM [32], which is a page-based implementation of lazy release consistency. Their garbage collector is also based on the Boehm and Weiser collector.

5.6 Summary

We have described Jackal's garbage collector and the optimizations it applies to increase program efficiency. The local garbage collector implements a simple, efficient mark-and-sweep collector that treats all exported references as a component of the root set. Whenever the local GC fails because too much of its local heap has been exported to other machines, a global GC is invoked. The global GC is also invoked if too many objects from other machines have been cached, causing the machine to run out of physical memory.

The global GC functions by sending mark messages to remote machines while disregarding the export table. An object is live if either there exists a reference within that machine (from a thread's stack, the cached objects or a global variable) or a mark message is received during the global mark phase. Otherwise that object can be removed. After a global GC, pages from the cached address space that are not referenced can be removed. To reduce the aggression the page unmapping process might otherwise exhibit, only pages that are not touched during the marking process are removed.

The performance of the global GC as a whole will be evaluated in Section 9.7.6 where a simple application is evaluated that generates much inter-processor garbage. The other applications that we have evaluated did not generate enough inter-processor garbage to allow a global-GC phase to occur.

Chapter 6

DAG Detection

6.1 Introduction

Jackal as described so far fetches one region (object or array partition) at a time. This causes a thread to be stalled many times waiting for objects to be fetched and generates many small messages which have a relatively high communication overhead. Therefore Jackal attempts to combine many accesses to fetch multiple objects at a time.

There are several ways to fetch multiple objects at a time; both at compile time and at runtime. Prefetching at runtime has several potential pitfalls. Firstly, the runtime system has to “guess” how much is to be prefetched for a given request, as it does not “know” how object *Y* will be used by requester *X*. If too many objects are prefetched, too much data is sent over the network, wasting precious memory, network and processing resources. If too few objects are prefetched, the full potential of prefetching is not used and the expected gains are not reached. Secondly, as the runtime system will have to keep track of past accesses and make estimates of future usages, it might be slow in adapting to changing access patterns in the application.

To overcome these problems Jackal uses compiler support to steer prefetching. Jackal’s compiler back-end will annotate the LASM code of a program, changing access checks to request not only the object *X* faulted on, but also any objects reachable from *X* if the compiler’s heuristics demand it. Because all data can afterwards be assumed available, the access checks to the prefetched objects can be removed. DAG detection therefore increases sequential efficiency as well as parallel efficiency.

An example of the code this optimization produces is shown in Figure 6.2. The original code is shown in Figure 6.1. Here the second access check to the inner dimension is removed because in the heap graph the 2D array is shown as a DAG (as all arrays are). The first access check is changed to fetch the entire DAG because the heap graph indicated that *a* will reference a DAG root.

The contribution made in this chapter is:

- a description on how to detect DAGs inside the heap approximation created as described in Section 4.4.3 and to transform the code to remove access checks to the

```
void foo(int [][]a) {
        // check(a, &a[3], read)
        int[]p = a[3];
        // check(p, &p[2], read)
        int t = p[2];
        print (t);
}
```

Figure 6.1: Before DAG detection.

```
void foo(int [][]a) {
        // check(a, whole_DAG, read)
        int[]p = a[3];
        int t = p[2];
        print (t);
}
```

Figure 6.2: After DAG detection.

interior nodes while fetching the entire DAGs upon each fault of a DAG root. This work has been published in [60].

6.2 Compiler Analysis

The compiler's support for Directed Acyclic Graph (DAG) analysis is divided into three passes. The first pass creates a heap (approximation) graph as described in Section 4.4.3. This analysis leaves the code in SSA form and leaves each instruction with a list of aliases, some of which describe the operands of the current instruction. The second pass analyzes the created heap graph to detect DAGs of a user-supplied maximum "weight". The maximum "weight" of a DAG is defined as the number of object allocation sites contained in the DAG. The maximum "weight" is supplied using a compiler command-line flag. The final pass in the analysis removes access checks to inner DAG nodes and changes the access checks to the DAG's root to fetch not only the checked object but the complete DAG of reachable objects.

DAG detection (pass 2) is performed by testing each node in the heap graph to see if it is a suitable DAG root. We start by testing each node with a traversal of the DAG, maintaining a set of visited nodes for cycle detection. If a node has already been visited, the node under analysis is not a DAG root and the next node in the graph is tried. If a DAG has been located in the heap graph, its weight is calculated by counting the number of reachable objects from its root. DAGs that are 'too heavy' are rejected. The result of this step in the analysis yields all DAGs up to a given 'weight'. When all DAGs have been identified in the heap graph, the code transformation phase is started.

The code transformation phase starts by visiting all access checks in the program and determining if the access check's object allocation site(s) is (are) *all* within a DAG. For each access check *X* checking reference *Y* for local accessibility, the expression comput-

```
synchronized void foo(Leaf X) {
  // << check(X,write) >>
  X.data = 1;
}

void zoo(DAG t, Leaf l) {
  // << check(t,read) >>
  foo(t.leaf);
  foo(l);
}
```

Figure 6.3: Example to demonstrate the need to examine all aliasing relationships.

```
class Data { int payload; }

class DAG {
    Data left  = new Data();
    Data right = new Data();
}


class use_after_sync {
    void foo(DAG a)
    {
        // check(a,read)
        Data d = a.left;
        synchronized (this) { // flush (a) && lock(this)
            // check(d,write)
            d.payload += 2;
        }
    }
}
```

Figure 6.4: Synchronization between root and leaf object usages.

ing *Y* is searched for in the aliasing sets belonging to the containing instruction. This yields a set of allocation sites that the checked object might have been allocated from. The resulting aliasing set is then used to find the associated heap nodes in the heap graph. When *all* heap nodes that the access check checks are found to be properly inside suitable DAGs, that particular check *X* can be removed. This guards against the example shown in Figure 6.3. In this example, foo's X parameter might be part of a DAG *or* a simple object allocated on its own elsewhere. We can therefore not remove the check to *X* in foo, as we are not sure that the checked reference will be properly inside a DAG.

The situation is more complicated if there are synchronization statements between the check to the DAG root and the checks to either intermediate or leaf objects. Consider the example in Figure 6.4. Let us assume that the DAG object is the root of a DAG and therefore the *Data* objects are leafs of the DAG. We cannot simply remove the *check(d,write)* as *d* might have been changed by another thread at another machine and those updates should

```
// read:
interior = root.interior;
// read:
leaf     = interior.leaf;
// write:
leaf.data = 3;
```

Figure 6.5: Bad read-write ratio's in DAG.

be visible in the synchronized block. Changing *check(a,read)* to *check(a,write,DAG)*, is not enough, we would still miss updates to d.payload.

Jackal solves this problem by supplying compiler analysis to determine if the DAG must be reloaded after the synchronized block has been entered or exited. The compiler analysis detects if any pointers to objects *Y* copied from a field of root *X* or its children are copied before a synchronization action and used afterwards. If so, the access check to the root is changed to *check(object,read or write,DAG-reload)* to reload the entire DAG after the synchronization action.

This approach has the penalty of reloading a (potentially) large DAG while only a small part of it might be used inside the synchronized block. However, we expect the gains more than make up for the penalty paid. Also, we do not observe the behavior of the program in Figure 6.4 in the applications described in this thesis. Programmers seem to write out the access path from DAG root to leaf for each usage instead of performing this slight optimization by hand (it can potentially reduce the number of field accesses).

Another problem that needs to be addressed, as shown in Figure 6.5, is that while the DAG root might be read-only, a leaf might be written to. Because the access check to the leaf has been removed, the entire DAG must be marked “modified” to capture the same program semantics under optimization. To implement the analysis, whenever a read access check to a DAG object is located, the CFG is traversed to locate write access checks to DAG interior nodes. Whenever a call is encountered inside the CFG, the CFGs of the called methods are traversed. Detecting if a write access check checks an interior DAG node is easy, using the heap graph to lookup aliasing relationships of any two access checks.

The pseudocode for the DAG detection and access check removal code is given in Figure 6.6.

6.3 Discussion

Jackal’s compiler has the ability to detect DAGs in a program’s heap graph. Suitable DAGs are chosen and the program code is changed to remove access checks to interior and leaf DAG nodes. The checks to the DAG’s root are changed so that the entire DAG is transferred over the network in case of an access check miss. When this optimization applies, savings of 50% or more can be obtained on program runtime (of course depending on the program’s use of data structures where this optimization applies). In Chapter 9, the performance of DAG detection will be evaluated for a number of applications.

```
forbidden_types = {java.lang.Thread, ...}

graph detect_suitable_DAGs(graph g) {
  graph r = {};
  for (each graph node p) {
        if (p is DAG root and weight(p) < maximum_DAG_weight) {
            if (contains no graph nodes forbiden to be in DAG(p))
                r += p;
        }
  }
  return r;
}
void mark_forbidden_graph_nodes(graph g) {
   for (each graph node p in g) {
       if (p allocated one of forbidden_types)
            mark DAG forbidden p
       }
   }
}
void DAG_detection() {
   program = to SSA form (program)
   graph g = create_heap_graph(program)
   // as a side effect of heap graph analysis,
   // each function now has for each reference used
   // a set of allocation ids (allocation id tells the compiler
   // where the object pointed to may have been created)
   g = mark_forbidden_graph_nodes(g);
   graph d1 = detect_suitable_DAGs(g);
   for (each function f in program) {
       for (each access check d checking reference x in f) {
           alloc_id_set s = get alloc_id set associated with x from f
           if (each graph node q in s checks root from d1) {
              change d to DAG root access check
              if (access check p checking reference y where y is
                 loaded from x, p is located inside synchronized block)
                   {
                     change p to reload DAG access check
                   }
              if (d is a read check
                 and exists access check w checking reference t
                 where t is (indirectly) loaded from x
                 and w is a write check) {
                   change d to a write check
              }
           } else if (each graph node q in s checks DAG interrior node from d1) {
              remote access check d
           }
       }
   }
}
```

Figure 6.6: Pseudocode for DAG detection.

Static DAGs detected by the compiler may seem restrictive as they do not include dynamic DAGs such as linked-lists and other statically self refering data-structures, but they do include some of the most interesting cases for check elimination. When applied to multidimensional arrays, DAG detection allows the removal of access checks to the lower dimensions. When an object contains references to objects that contain no references (leaf objects), the checks to the leaf objects are removed. Presumably most run-time heap accesses occur in leaf objects or in the lower dimensions of a multidimensional array.

An open problem in this chapter is finding optimal DAG weight values. However, most gains are to be expected with values ranging from 2 to 5 to limit the search space, which allows for a set of small test runs to find the optimum value for the application at hand.

A related problem with the algorithm as described above is that *all* recognized and suitable DAGs in the object graph are changed to DAGs regardless of optimality. The only trigger for the optimization is the user supplied maximum DAG weight. This may not be optimal for all programs, for example, in the following cases:

- programs that use deep DAGs and write only to the leaves; a large DAG weight should then be chosen to cover many leaves with a single DAG; the interior nodes are then marked “modified” although they will only be read from;
- programs that operate on multiple data structures each of which requires a different optimal DAG weight;
- programs using unbalanced DAGs.

These problems remain open in the current prototype.

Choosing too large a DAG causes problems if only a small portion of the DAG is written to because it reduces the runtime system’s opportunities to eliminate data transfers, as read-only data is handled especially (see Section 3.4.6). Consider a heap graph that consists of a large array that contains references to several objects where each in turn contains a pointer to a leaf object. If the array were chosen as DAG root, a write to one object in a DAG will cause all other objects in the DAG to be marked “modified” as well, potentially causing all of them to be flushed at a synchronization point. On the other hand, if the array were small and traversed often, the gains of the reduced access checks executed might negate the costs of such spurious data transfers.

Another open problem in our DAG detection algorithm lies in handling non-uniformly accessed DAGs where a part or a leaf is to be treated differently from the rest of the DAG for optimum performance. Consider a data structure where one of the leaf objects is written to but the rest of the DAG is only read from. Ideally, the compiler and runtime system would exclude the “strange” leaf by letting the compiler mark the leaf object and would not remove the access checks to the problematic leaf object.

Chapter 7

Computation migration

7.1 Introduction

As described thus far, when a thread starts execution on a machine it stays at that machine until it has finished. This may be detrimental to performance in a number of scenarios, for example, when a thread often executes synchronization blocks that contain nonlocal data accesses. Under this common scenario each iteration requires at least seven messages (see Figure 7.2) under normal non-optimized execution.

The following example illustrates the problem. Assume the requesting machine (*R*) wants to execute the program in Figure 7.1. Here object *L* is locked followed by a write to object *D* where both objects *D* and *L* are located on the same machine *H* (see Figure 7.2). The first action is to flush working memory, but for simplicity let us assume working memory to be empty. Next, a message (1) is sent by the requesting machine *R* to acquire *L*'s lock. The moment *L*'s lock has been acquired, a reply message (2) is sent to *R*. When *R* receives the reply, it will attempt to modify object *D*.

As a result, two more messages are exchanged to fetch object *D* (unless *D* has migrated in the meanwhile): a write request for object *D* (message 3) and a reply message (4) containing object *D*. After updating object *D*, *R* executes an unlock statement that causes object *D* to be flushed from working memory (message 5). Next, *R* sends an unlock message (6) to *H* and waits for an ack (7) from *H*.

We will introduce an optimization that may reduce the number of messages to two in the most favorable case.

```
// lock
synchronized(L) {
// modify D
  D.field = 42;
// unlock
}
```

Figure 7.1: Locking example.

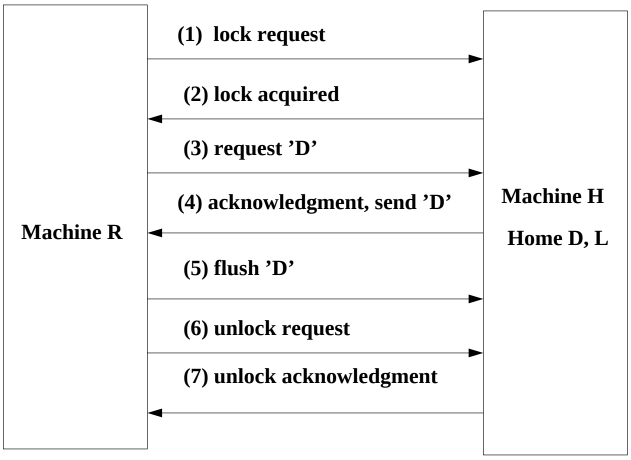


Figure 7.2: Messages sent during example 7.1.

The contributions made in this chapter (published in [60]) are:

- a description of how to *automatically* transform segments of code from object caching code to function-shipping style code;
- a description on how to combine the new function shipping code with automatic piggybacking of objects to reduce communication.

7.2 Remote Synchronization Invocation (RSI)

As the above description shows, synchronization on remote objects is very expensive. A more efficient strategy is to temporarily move the executing thread from machine *R* to machine *H* which hosts the lock object *L* when a synchronization statement is encountered. The number of messages can then be reduced drastically from seven to a minimum of two messages.

For this purpose, the compiler's optimizer implements a pass that cuts synchronization blocks out of functions and transform them into new functions (*function splicing*) suitable for remote invocation. The compiler recognizes calls to the runtime system to lock and unlock object *L*'s associated lock and moves the delimited code to a new function *F*. The code is then replaced by a call to the runtime system *do_RSI*, passing the address of *F*, any local variables that were live at the call to *lock*, and the address of object *L*. Any variables that are live at the statement after the call to *unlock* are return values of *F*.

The thread at *R* must flush its memory before its invocation of *F*: *F* contains a synchronized block. Equally, the thread that executed *F* at *H* must flush at its unlock call at the end of *F*. Flushed object messages can potentially be combined with the RSI request or reply.

An example is shown in Figure 7.3 with its transformed code in 7.4. At the call to

```
int foo(Object this) {
  int temp = 4;         // live = {this}
  call lock(this);      // live = {this, temp}
  this.field = 42;      // live = {this, temp}
  temp++;
  call unlock(this);    // live = {this, temp}
  return temp;          // live = {temp}
}
```

Figure 7.3: Function splicing, starting point.

```
int foo.splice0(Object this, int temp) {
  call lock(this);      // live = {this, temp}
  this.field = 42;      // live = {this, temp}
  temp++;
  call unlock(this);    // live = {this, temp}
  return temp;          // live = {temp}
}

int foo(Object this) {
  int temp = 4;         // live = {this}
  temp = do_RSI(HOME_NODE_OF(this),
                foo.splice0,
                this,
                temp);
  return temp;         // live = {temp}
}
```

Figure 7.4: Function splicing, transformed code.

lock, *this* and *temp* are live variables. They are copied to the parameter list of the new function *foo.splice0*. *Temp* is changed inside the synchronized function and thus needs to be shipped back: it is returned from the spliced function. The original code is replaced by the call to *do_RSI* with the live variables and the address of the spliced function appended to its parameter list. As a field of *this* is modified, subsequent reads to a field of *this* will require *this* to be re-fetched, which is handled lazily.

To preserve load balance the compiler rejects some function splicing candidates: when the contained code block contains more than a certain number of LASM instructions (1000 by default) or if there are more than two 'return values' of the synchronization block (local variables written inside the synchronized block and read afterwards). The compiler also disallows loops in the synchronized block as otherwise we assume the code to be long running and detrimental to load balancing. LASM functions support up to two return values; if there are more than two live variables no function splicing will be performed for that particular synchronization block. This limitation is caused by the lack of sufficient processor registers in the x86 architecture.

7.3 Message Combining

Many synchronized blocks serve a single purpose: for example to remove an element from a shared data structure or to add to it. Consider the removal of nodes from a linked list located at machine *0* by threads executing on machines *(1 ... N)*. As multiple threads may simultaneously access the linked list, the 'remove' methods of the list need to be synchronized and are turned into remote synchronization invocations. The same argumentation holds for adding elements to the linked list, the code of those methods also needs to be synchronized. Although the number of messages is reduced by performing computation migration, an extra message is still required to move the linked-list element to or from machine *0*.

The compiler is able to optimize both cases. When a reference is created, either by *new* or by a field access, and is dereferenced after the synchronized block, the reference is added to the parameter list of *'unlock'*, to allow the referenced object's data to be piggy-backed on the lock release message. This covers the case where an element is removed from the linked list and returned. In the example in Figure 7.3, the object denoted by *this* is modified in the synchronized block and can thus be piggy-backed on the RSI request. If *this* is subsequently read or written to, *this* would be piggy-backed on the RSI request. The *do_RSI* call in this case gets an extra parameter with an annotation indicating that the contents of the objects must be packaged along with the RSI reply. This information is conveyed to the runtime system by a parameter descriptor generated by the compiler.

Because any modified objects at the point of the RSI must be made available to the RSI's code at the remote machine, the runtime system will flush all working memory before performing the RSI. Objects which are thus already available at the other machine need not be piggybacked. Only objects exclusively owned by the RSI performing machine are piggybacked on the request message.

When a live reference variable at the call to 'object-lock' is stored into another object inside the synchronized block, that variable is appended to the parameter list of the call to object-unlock to allow it to be piggy-backed on the lock acquire message. This optimizes the case where an element is added to our linked-list example.

The compiler also checks if a DAG root (see Chapter 6) is returned or added to a data structure such as our linked list from an RSI (one of the live variables at a lock or unlock was a DAG root). In such a case the whole DAG is piggy-backed upon the lock acquire or release message. However, a problem occurs whenever an interior DAG node is accessed inside the function splicing candidate. Since all access checks were removed to such objects, the RSI would require those objects to be shipped along with the acquire / release message but this is difficult if the interior object is available but a pointer to the DAG root is unavailable. In that case, the block is rejected as an RSI candidate.

An example of this program behavior is shown in Figure 7.5. Here a sub-DAG is accessed inside a synchronized block but the DAG-root is no longer available. The access check removal code will however try and remove the access checks to the sub-DAG nodes inside the synchronized block. To allow this to work the DAG-root references would have to be shipped to the machine where the code of the RSI will be performed, however, this is not implemented.

Summarizing, only in the following cases will computation migration be triggered:

```
void zoo() {
    int [][][]dag = new int[N][M][L];
    int [][]subdag = dag[2];
    <other code>
    // 'subdag' is a sub-DAG of 'dag',
    // all access checks to it can
    // subsequently be removed.
    synchronized(this) {
        subdag[2][3] = 3;
    }
}
```

Figure 7.5: Accessing a Sub-DAG inside a synchronized block.

- there is a lock-unlock pair;
- the number of instructions in between is less than 1000;
- DAG detection is enabled and there is no access check to a DAG interior node with the DAG root access check outside the lock-unlock pair;
- there are no more than two live variables at the call to unlock.

7.4 Related Work

Several systems incorporate computation migration. Some systems employ function shipping (using computation migration to remotely execute a single function) to simulate a shared object space. Such systems include JavaParty [47] and Orca [4]).

Programs using JavaParty must annotate classes with a *remote* keyword to allow the programmer to indicate which functions should be remotely executed whereas Jackal performs this automatically.

Shared objects in Orca can be either replicated or stored on a single machine; operations on objects are performed using function shipping.

MCRL [23] is a DSM system where thread migration occurs whenever a write request is encountered. The control-flow then moves to the machine where the object is located. The decision to perform computation migration is performed at runtime. In contrast, Jackal performs all computation migration decisions at compile time under compiler control.

Olden [52] is a parallel extension of C to create SPMD programs. For each pointer dereference, the compiler uses a heuristic to decide between object caching and computation migration. When computation migration is selected, the compiler changes the code to capture the function's state at the point of the pointer dereference to allow the function to be restarted at a remote processor. Control flow resumes at the dereferencing processor when the function is exited.

Prelude is a language containing many mechanisms to allow parallel execution to occur, one of which is computation migration. Like Jackal, Prelude also performs its computation migration decisions at compile time using continuations for single stack frames as its mechanism to actually perform them.

Unlike the previous two systems, Jackal also makes the decision to perform computation migration at compile time but per synchronized block of code looking only at the complexity of the delimited code. Unlike the previous two systems, Jackal also performs piggy-backing of data about to be used.

7.5 Summary and Future Work

We have described Jackal's computation migration mechanism, which moves synchronized blocks of code to new functions (code which the back-end sees as code delimited by calls to *lock()* and *unlock()*) and performs function shipping for those pieces of code. The new functions are executed at the home node of the synchronized block's lock object.

On a simple benchmark, speedups improvements scale with the number of object accesses saved (as expected). The applicability of computation migration as proposed here is very good. The optimization is triggered for all applications. The profitability of the optimization is however often negated by the high costs of thread switching in the current prototype.

This optimization has the potential of reducing the number of messages from seven in a naive implementation to only two in the most optimized case.

Function splicing is, however, a very general technique which could be applied to all blocks of code that are communication-intensive to a single host. Jackal lacks the analysis to perform profitable function splicing in the general case. Instead Jackal uses the simple heuristic of only looking at synchronized blocks.

Chapter 8

Aggressive Object Combining

8.1 Introduction

Object-oriented languages like Java encourage programmers to create many small objects, both for concept abstractions and for data encapsulation. This creates two possible problems. First, in a distributed environment many small objects will be sent over the network instead of a smaller number of larger objects. This problem will be dealt with in section 8.5. The second problem is that it strains both the object allocator and the garbage collector unnecessarily.

Escape analysis[11] helps in this respect by allocating some objects on the stack, reducing garbage collection overhead. Escape analysis, however, will fail whenever the allocated objects escape the function or thread that allocated them or in programs where the aliasing relations are too difficult to analyze.

Object inlining [14, 19, 38] is another technique to reduce the overhead of object management. It inlines an object (*Y*) into another object (*X*) by replacing a pointer in *X* to *Y* (an indirection pointer) by the fields of *Y*. Object inlining reduces the number of objects allocated (and garbage-collected), sometimes reduces memory usage by eliminating object header information, and saves one pointer indirection in accessing the inlined fields. Initializing an inlined object is presumably faster than allocating a new object because there is no need to search for free space for the inlined object. This overhead has been effectively pushed to compile time. Another advantage is that with object inlining enabled, multiple objects can be zeroed with one single operation instead of one at a time. (We must clear objects upon their creation because Java specifies that newly created objects should have their fields set to zero). Together with sometimes better caching behavior all these advantages can result in substantial performance gains.

Unfortunately, traditional object inlining can be applied only in very specific cases, in which a container object contains a pointer to another object that is to be inlined. Traditional object inlining fails to inline aliased objects, which can be accessed from global variables or through other objects, because the transformation might violate program semantics. Likewise, it cannot handle reassignments (overwrites) of the pointer from the inlining object to the inlinee.

In this chapter, we introduce a new and more general optimization called *object combining*. This optimization aims at combining any set of objects, regardless of whether or not they point to each other. The only requirement is that a pointer to the first object be available at the allocation site of the second. The optimization generalizes object inlining in two ways:

- it can handle aliased objects and pointer overwrites, so it is more aggressive than object inlining;
- it does not just combine related objects (with a pointer between them), but it tries to find any collection of objects for which combining may be beneficial.

The key idea in allowing aliasing and pointer overwrites is to separate the transformations that object inlining does: our optimization combines the fields of objects, but it does not immediately eliminate the pointer indirection. A later pass tries to remove the pointer indirections (indirection pointers now point a little further into the same object) by replacing the indirection with direct accesses to the inlined data. We will show that leaving the pointer indirection in place allows far more general combining and inlining of objects. If object aliasing or pointer overwrites occur for inlined objects, our optimization detects the aliasing at runtime and uses either a runtime data transformation to preserve program semantics or foregoes object combining, creating new objects instead of reusing the combined object.

Because pointers to inlined objects can be freely ’leaked’, the inlined object retains the object header (consisting of a virtual method table and a flags field). The header information could potentially be removed if no references to inlined objects are ‘leaked’, but this optimization is not considered in this thesis.

In theory, our mechanisms can combine almost all objects, except for a few difficult cases discussed later. However, object combining also has a cost. If combined objects are often overwritten, the execution time overhead of the runtime transformations is prohibitive. Likewise, if objects with completely different life spans are combined into one object (which is allocated and freed by the memory manager in one pass), much unused memory will be left inside such a container object that cannot be garbage-collected. Therefore, it is important to identify those cases for which object combining is beneficial. We not only look at related objects (as object inlining does), but also at objects that are used together in the same functions, objects that are allocated in the same loop, and objects that belong to the same data structure, for example, a list. Also, we try to find objects with similar life spans using profiling information from previous runs, in addition to compiler analysis. In each specific case, we apply heuristics to determine if object combining is desirable.

The results show that the optimization reduces execution time by up to 34% for applications that create and use objects intensively.

The contributions made in this chapter are as follows (published in [61]):

- we show that separating object inlining from eliminating the indirection pointer allows a much broader class of objects to be combined while still achieving the same benefits;
- we describe different mechanisms that allow combining of difficult cases, such as aliased objects, arrays of objects, unrelated objects, and recursive data structures;
- we present heuristics based on static analysis and profiling techniques to guide object

combining decisions;

This chapter is structured as follows. In Section 8.2 we discuss the mechanisms for combining a given set of objects. In Section 8.3 we describe how we determine which objects will be combined. Pointer indirection elimination to inlined objects is discussed in Section 8.4. Section 8.6 relates the work presented here to previous work. Section 8.7 draws some conclusions.

8.2 Combining Objects

To combine two objects, the compiler adds one of them to the other object. For related objects, where one object *X* contains a pointer to another object *Y*, object *Y* is appended to object *X*; in all other cases, the choice is arbitrary. To inline an object *Y* into an object *X*, extra room is allocated inside *X*'s class descriptor to hold the inlined data and the allocation of *Y* is transformed from

Y = allocate object(type)

into

Y = initialize_inlined_object(type, X, &X→inlined_Y)

The latter code executes the same instructions that are needed for initializing objects allocated with *new* (inserting the method table pointer and initializing the flags field), but without having to search for space on the heap and potentially without having to zero it if the container object was already zeroed. This allows the creation of a 'fast code path' at each inlined object allocation site (12 or 15 instructions). Inlining the normal (heap-based) object allocation code is unprofitable due to its large size.

For object combining to work, the method that allocates *Y* now also must have a reference to *X* because *Y* is allocated inside an instance of *X*. The compiler therefore clones and patches the call chain from the method that allocates *X* leading to the method containing the above allocation of *Y* by adding the missing parameter to each. As the cloning process proceeds, pointers to *X* are made available to the allocation site of *Y*.

Because the size of an object must be completely known at its allocation site, there must be no subobjects of unknown size inside the container object. If *Y* is an array whose size cannot be determined by the compiler, the compiler will attempt to move the computation of the array size in front of the allocation of *X*. If successful, the allocation of *X* is patched to allocate *X* with *Y* attached to it with the correct size. Otherwise, object combining for *Y* inside *X* is abandoned.

The compiler can combine an arbitrary number of objects into one object by repeatedly combining two objects. Besides the object combining as described above, we use one other mechanism for combining multiple objects. For objects of one class that are allocated as part of the same dynamic data structure (e.g., a list), the system may use a slab with a certain number of preallocated objects. This effectively creates a custom slab-based allocator [18, 22] per allocation site (see Section 8.3.3).

The main problem with object combining is how to preserve the semantics of the original program. A difficult problem is what to do when a candidate object allocation site for object combining is potentially executed more than once for a single container object

```
1:   class Containee {
2:      int data;
3:   }
4:
5:   class Container {
6:      Containee field;
7:   }
8:
9:   Container Y = new Container();
10:
11:  for (int i=0;i<2;i++) {
12:      Y.field = new Containee();
13:      static_variable[i] = Y.field;
14:  }
15:
16:
17:  static_variable[0].data = 1;
18:  static_variable[1].data = 2;
19:
20:  println("V1="+static_variable[0].data);
21:  println("V2="+static_variable[1].data);
```

Figure 8.1: Source code; notice the overwrite to *Y.field* at line 12.

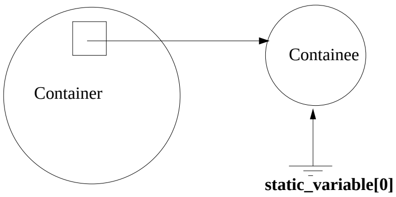


Figure 8.2: Memory layout for Figure 8.1, before the reassignment.

(which cannot be determined in general at compile time). Consider the program in Figure 8.1, which we will use as a running example in this section. At runtime, the memory layout at line 13 in the first loop iteration is shown in Figure 8.2. After the reassignment in the second loop iteration, the old object is reachable through *static_variable[0]* and the new object is reachable through the object reference inside the container object and through *static_variable[1]* (Figure 8.3).

A naive (incorrect) implementation of object combining will allocate a single containee object inside the container object and result in a memory layout as shown in Figure 8.4. This violates the original program semantics, as updates through the static variable will be visible at the (reused) combined object as well. The example will print *V1=2* and *V2=2* instead of the correct *V1=1* and *V2=2*.

Traditional object inlining algorithms therefore do not optimize the program of Figure 8.1.

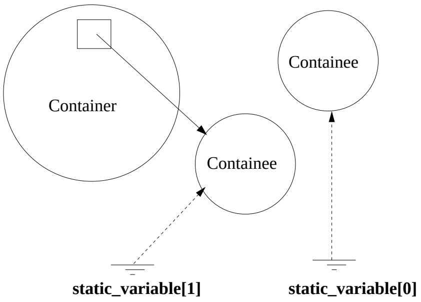


Figure 8.3: Memory layout for Figure 8.1, after the reassignment.

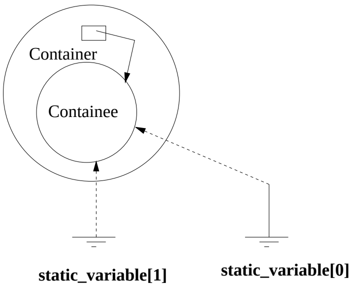


Figure 8.4: Memory layout for Figure 8.1, after the reassignment, with erroneous object combining.

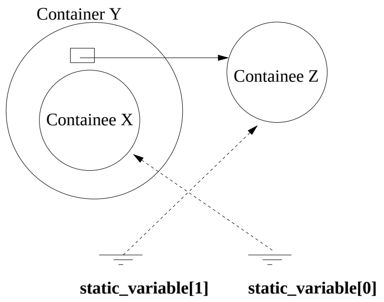


Figure 8.5: Memory layout corresponding to Figure 8.1, object combining enabled, reassignment solved with *Uninlining*.

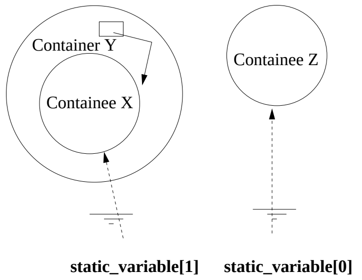


Figure 8.6: Memory layout corresponding to Figure 8.1, object combining enabled, reassignment solved with *Recursive Pointer Rewriting*.

We do not reject object combining candidates because an indirection pointer might also point to objects other than the object candidate under consideration. The fact that a reference-field *might* point to the object combining candidate is sufficient to trigger object combining.

Our scheme, instead, either provides runtime data transformations or solves the problem by undoing the object combining (at runtime) when this situation is detected. The runtime data transformation is expensive if many inline objects are re-allocated. Therefore we rely on heuristics (static analysis and profiling) to determine if the runtime data transformations are desirable or to disable object combining. In Figure 8.1, for example, combining would not be applied if the loop on line 11 were executed many times instead of twice. To allow combining for Figure 8.1, we use two different techniques, each using a mix of compiler support, execution profiling and runtime system support.

Technique 1: Uninlining

Uninlining creates a new object whenever re-allocation of a combined object *X* inside *Y* is attempted, effectively disabling object combining at runtime. This can easily be determined (at run-time) by detecting whether the inlined object *X* is already in use before trying to use the space it occupies in object *Y*. This can be done, for example, by examining a *flags* field that has been set by the first object allocation of *X*. At the reassignment, a new object is created and a reference to it is returned instead of one to the inlined object *X*. This has the effect of creating the memory layout of Figure 8.5. This solution is to be chosen if the program contains field reassignments. Even when aliasing requires a new object to be allocated, a single object allocation is still saved compared to not using object combining. The only extra costs incurred are that of checking a flag and an extra call to the heap-based object allocation code.

Technique 2: Recursive Pointer Rewriting

Recursive Pointer Rewriting is more complicated; whenever an inlined object *X* is already in use when trying to re-allocate it (which is determined exactly as in *Uninlining*) we allocate a new object, again just as for *Uninlining*. However, the original inlined object *X* is now moved to the new object, and its space is overwritten by the new inlined object. All references to *X* (and references to any inlined objects that *X* might contain) are rewritten to references to a newly allocated clone of object *X*. This recursive rewriting needs to find all references to *X* by scanning all reachable data in the program, so it is an expensive operation (similar to garbage collection). After rewriting, the inlined object *X* can be safely reused. This has the effect of creating the memory layout of Figure 8.6.

The advantage of using *Recursive Pointer Rewriting* over *Uninlining* is that the indirection pointer is guaranteed to *always* point to the inlined object, allowing the compiler to generate code to skip the use of the indirection pointer. Thus whenever the compiler detects an opportunity for indirection removal it enforces *Recursive Pointer Rewriting* for that inlined object. If no opportunity for indirection removal is detected, *Uninlining* will be used to resolve re-allocation aliases.

Because the recursive rewriting operation is expensive, this solution is to be chosen only if aliasing to *X* occurs infrequently, while there are many uses of it to mitigate any rewriting costs. This can, for example, be detected using a profiling run of an instrumented version of the program.

8.3 Finding Opportunities for Object Combining

Above we described general mechanisms for combining objects while preserving program semantics. Another issue is which objects should be combined. In theory, our mechanisms can combine almost any arbitrary collection of objects. However, object combining is disadvantageous if there are many aliases, or if the life spans of the objects are too different. Therefore, a more intelligent scheme is required. Our system looks for specific cases where object combining might be beneficial:

- reference-wise related objects, where one object contains a reference to another;
- objects that are unrelated but that are often used together;
- objects that are of the same type and part of the same data structure, such as a list;
- objects that, based on profiling information, are likely to have similar life spans.

The first case is the only one that is potentially covered by traditional object-inlining techniques. All cases can be regarded as triggers for our optimization; they are discussed in more detail in the subsections below.

For each object-combining opportunity that is identified, we use heuristics to determine if the optimization is indeed desirable. The heuristics are based on static analysis, and some also on profiling information from a previous run. After a profiling run has completed, the compiler can be instructed to restrict combining when too many re-allocations occur.

Searching for object-combining candidates is difficult in the presence of polymorphism as we would not want to reserve space inside a container object for a containee allocated inside function *F* when *F* is overridden in derived class *B*. When most of the time *B.F* is called instead of *A.F*, the reserved memory will be wasted most of the time. Java specifies that all calls should be dynamically dispatched, unless the invoked method is marked static, private or final. Our current implementation does not combine allocations if the container is located inside a caller and a potential containee inside a dynamically dispatched method. Instead, we rely on techniques to remove such calls using type-inference by employing the techniques proposed in [17, 55]. Also, our compiler can be instructed to make a closed-world assumption, in which case it will not support dynamic class loading. We will make this assumption in the rest of the chapter, as it allows for more effective, entire-program analysis.

8.3.1 Related Objects

Two objects are said to be related when one object potentially contains a pointer to another. This case is similar to the one optimized by traditional object inlining (except that our optimization is far more aggressive). To find these cases, the first step is to transform the code to SSA [12] form and create a heap approximation using techniques from [20]. SSA simplifies program analysis while the heap approximation provides the compiler with the sharing patterns between objects, allowing for more informed object combining decisions.

The next step is to locate all occurrences of object allocations:

X = allocate object(type)

or

X = allocate array(type,number of elements).

These are potential candidates to have objects inlined into. Next, the control-flow graph and call graph are traversed from those program locations to locate other occurrences of

Y = allocate object(type)

or

Y = allocate array(type,number of elements).

The heap approximation graph node associated with *X* is then located and examined to determine if *X* contains a pointer to *Y*. If it does, this implies that elsewhere in the program an assignment

```
    Containee X  = allocate Containee;
    Containee_Constructor(X);
    Container Y  = allocate Container;
    Container_Constructor(Y,X);

// is turned into:

    Container Y  = allocate Container;
    Containee X  = allocate Containee;
    Containee_Constructor(X);
    Container_Constructor(Y, X);
```

Figure 8.7: Object/array allocation site re-ordering.

X.field = Y

or

X[Z] = Y

has been found by the heap approximation construction algorithm. Next, we check if a pointer to *X* is always available at the allocation of *Y*. This is the case if either *Y* is located inside another function called from the function *X* is located in, *Y* and *X* are located inside the same basic block, or the basic block *X* is allocated in dominates the basic block *Y* is allocated in. If one of these criteria is satisfied, object Y can be inlined into object X, thus combining the two objects.

If the allocation site of the container and containee objects appear in the wrong lexical order, the compiler may fail to find this opportunity. Therefore, the compiler can reverse the ordering. The reordering heuristic tries to move an object allocation *X* back over another object allocation *Y*, if both allocations are located inside the same basic block and *X* might contain a reference to *Y*, as determined from the heap approximation. The constructor calls to *X* and *Y* as well as the allocation of the inlined object are left as they are. After this reordering, object combining can be applied. An example of this optimization is shown in Figure 8.7.

Because the elements inside an object-array (an array of objects instead of Java's array of references) must be uniform in type and size, the heap graph is also inspected to determine if array elements in the pointer array might point to different types or sizes of objects/arrays. If this is the case, the container array in question is left alone, foregoing all object combining for that array. For example, if the elements of a multidimensional array are not of the same size or cannot be determined to be so (the multidimensional array is not guaranteed to be rectangular), object combining is foregone. Container objects have no such restriction, unlike container arrays.

Finally, our implementation inlines at most one array with a dynamic size into each object to limit the complexity of addressing data inside the object.

```
class List {
   Node Head;

   void add(Node n) {
       n.Next = Head;
       Head = n;
   }
}
```

Figure 8.8: linked-list node addition.

8.3.2 Unrelated Objects

If two objects do not contain references which may point to the other object, they are said to be unrelated. In some cases, combining unrelated objects may increase performance much like combining related objects can. For example, if the objects have corresponding life spans it may be beneficial to combine them.

For unrelated objects such as:

A = new Data(); B = new Data();

our object combining heuristic determines if the two objects are always used together. This is the case if they are always dereferenced (or in Java terms: a field is accessed) together in each function of the program. If so, the two unrelated objects are combined.

8.3.3 Recursive Objects

The algorithms described thus far combine a fixed number of objects. Many programs, however, use recursive data structures like lists, which are inherently dynamic. In a heap approximation graph [20], all elements of such a recursive data structure are typically represented by a single node, so the compiler cannot distinguish them. Still, in many cases it may be beneficial to apply object combining to such data structures. Our optimization combines the data structures into multiple slabs (chunks), each of which has a fixed size.

We recognize this case by examining the heap graph [20], which represents a linked-list node insertion such as depicted in Figure 8.8 by a graph as depicted in Figure 8.9. Recognition is implemented by determining for each graph node if there are any self edges. If there are, the associated allocation site is transformed.

After these recursive data structures have been identified, the class descriptor and allocation site are patched to allocate these objects inside a *slab*. A slab contains space for a fixed number of objects and some slab administration (the current and maximal number of objects inside the slab) which is appended to the first object in the slab. This transformation turns recursive object allocations into allocations of small object arrays, which are filled each time a new object is allocated. Upon overflow, a new object array is allocated.

The program depicted in Figure 8.10 is transformed into that of Figure 8.11. The runtime side of recursive object combining records how many objects are already put inside a slab of a certain type. The static variable *slab_chunk* is used to maintain the

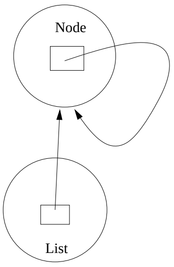


Figure 8.9: Heap approximation for a linked-list.

```
void fill() {
  List l = new List();
  for (int i=0;i<1000;i++) {
        Node n = new Node();
        l.add(n);
  }
}
```

Figure 8.10: Filling a linked list, initial situation.

```
void fill() {
  List l = new List();
  for (int i=0;i<1000;i++) {
        static Node *slab_chunk[100];
        Node n = slab_allocate(type_id Node,
                               &slab_chunk);
        l.add(n);
  }
}
```

Figure 8.11: Filling a linked list, after transformation.

```
class Sub { }
class Data {
    Sub s;
    Data() {
        s = new Sub();
    }
}
class CloneCtorNeeded {
   Data d1, d2;
// Note that we may need to do different
// things for the two different Data allocations.
   void foo() {
        // create one Data object:
        d1 = new Data();
        work(d1);
        // create many Data objects.
        for (int i=0;i<N;i++) {
           d2 = new Data();
           work(d2);
        }
   }
   void work(Data d) { ... }
}
```

Figure 8.12: Constructor cloning to increase analysis precision.

current slab such that the runtime system does not have to search for the slab of the correct type.

8.3.4 Determining Object-Combining Candidates using Profiling

Since the compiler cannot always determine the number of objects allocated at a given allocation site, our final recourse is to use profiling information from a run with an instrumented program to get estimates of these statistics. When later these statistics are fed back to the compiler, the compiler can reject object combining candidates or decide to slab-allocate them.

Jackal's object-combining profiling support inserts, at code generation time, a call to the runtime system before each allocation site, passing the name of the function in which the allocation occurs, and the sequence number of the allocation in that function, thus allowing the runtime system to maintain a count for each allocation site. After a run of the program, these numbers are written to disk. Using the resulting profile, the compiler will only combine objects that are allocated roughly the same number of times.

To increase the precision of the profiling information, object constructors containing other object allocations are separated by cloning them. This ensures that the profile will not aggregate the profiling information for all object allocation sites from object constructors. The profile will then correctly correlate object constructors and allocation sites. An example of this behavior is given in Figure 8.12. When the *Data* constructor is not cloned, the analysis will accumulate the information from both *Data* allocations and pass the accumulated data to both allocation sites and potentially make the wrong assumptions for either of the *Data* allocations.

The profiler might reveal object allocations that are executed often but are not subject to any of the optimizations described above. The profile-based compiler can be instructed to allocate the objects from such an allocation site in slabs. The implementation is exactly like that of recursive object combining. Summarizing, objects that are not recursively defined, are not stack-allocatable, cannot be combined with another object, and have an unknown number of allocations at compile time may be caught with this optimization.

8.4 Indirection Removal

Current (and future) machines show a widening gap between processor speed and both memory latency and throughput. Much research is being invested to reduce this gap by software and hardware techniques ([9, 10, 57]).

Whenever memory is accessed that is not already available in the processor's cache, processors typically stall for some cycles (7 to 20) until the requested memory has been fetched. To reduce the number of stalls, processors typically cache and prefetch memory, thus overlapping computation and memory latency. This technique fails, however, when the address of the memory accessed is not known in advance, as in the case of multiple indirections. Removal of the usages of the indirection pointer (termed indirection removal) helps here by exposing memory access patterns to the processor. Also, the pointers themselves are not loaded into the processor's cache which avoids cache pollution.

Another effect of indirection removal is the increased potential for compiler optimization, because the compiler can now "see" the data beyond the access. However, if the address calculation to address the inlined object is expensive or when register pressure is high, the removal of the indirection pointer can be detrimental to code quality.

Thus far in this chapter, objects have been combined but the inlined objects are still accessed through the indirection pointers, stalling the processor in case of a (first level) cache miss. Since our implementation of object combining imposes few restrictions on what may or may not be combined, the containee object can be directly accessed only when the indirection pointer is guaranteed to always point to a single containee object. Below we will enumerate the conditions that must be obeyed before an access to a containee object may use the containee object directly for indirection removal.

Firstly, indirection removal should only be performed if the replacing address calculation is not too expensive. Accesses to a containee object inside a container object are cheap, as they are always accessed at a fixed offset inside a container. For array accesses, the situation is different because a multiplication has to be performed to calculate the address of the containee. Of course, the cost of performing this multiplication depends on the architecture being compiled for. If indirections in object arrays are removed, the calculation becomes of the form:

```
(array.length * sizeof(reference)) +
(element_index * sizeof(inlined_object))
```

The calculation of *array.length*sizeof(reference)* is used to skip the indirection pointers inside the object array to get at the actual inlined objects. The multiplication by the

```
Inliner ex = new Inliner();
// compare 1
if (ex.data == null)
    System.out.println("ok");

Inlinee temp = ex.data = new Inlinee();
// compare 2
if (ex.data != null)
    System.out.println("ok");

ex.data = new Inlinee();
// compare 3
if (ex.data != temp)
    System.out.println("ok");

ex.data = null;
// compare 4
if (ex.data == null)
    System.out.println("ok");
```

Figure 8.13: Testing references in the presence of indirection pointer removal.

size of the inlined object is used to calculate the address of the correct object inside the object array. Since the *sizeof* expressions yield compile time constants, the multiplication is rewritten by the compiler to sequences of shifts, adds and subtracts.

If the size of the array is not known at compile time, accessing the *array.length* field requires an indirection which we were trying to avoid. Thus for arrays whose length is not known at compile time, no indirections are removed. Also, if the size of the inlined object is not a power of two, multiple shifts and adds are required to compute the multiplication which negates the advantage of indirection removal. A small amount of alignment is performed to allow this case to occur more often. To conclude, our implementation only removes the indirection to objects or arrays inside object arrays if both the size of the container array is known at compile time and the size of the inlined object or array is a power of two.

The second problem with indirection removal stems from reference comparisons. If the indirection pointer itself were removed from the inlining object, as traditional object inlining does, a different test is required for reference comparisons, for example, a test to determine if the (inlined) object has already been initialized. However, we allow almost arbitrary reassignments to the indirection pointer which forces use of the indirection pointer for reference comparisons. This allows the comparisons in the example in Figure 8.13 to execute as expected, regardless of whether technique 1 or 2 (see Section 8.2) was used to resolve aliasing to inlined objects. We cannot change the indirection pointer usages to direct uses of the inlined object nor can we test if an inlined object has already been initialized as, for example, the indirection may have been explicitly reset to null in the source program while external references to the inlined object may already exist. Also, the constructor of the inlined object may have thrown an exception, initializing only part of the object, thus not reaching the initialization of the indirection pointer. A test against

```
Inliner ex = new Inliner();
ex.data = new Inlinee();

// should be ok:
ex.data.foo();

ex.data = null;

// should throw null pointer exception:
ex.data.foo();
```

Figure 8.14: Accesses should still throw null pointer exceptions in the presence of indirection pointer removal.

```
int[][] arr = new int[N][];

for (int i=0;i<Z;i++)
  arr[i] = new int[M];

print( arr[K][L] );
```

Figure 8.15: Hard to test at compile time whether an access will throw a null pointer exception.

null should in such a case still return *true*.

The final problem with indirection removal stems from null pointer exceptions. If in the original program an indirection pointer is used that was set to null, the access should throw an instance of *java.lang.NullPointerException*. However, if we replace the indirection by an addition to get at the inlined object, no true memory access occurs and hence no null pointer exception will be thrown. This problem is demonstrated in the example in Figure 8.14.

It is not efficient to first test whether or not the indirection pointer is *null*. Doing so will require a comparison and a jump instruction, slowing down execution. Our only recourse is then to perform conservative, exhaustive, analysis at compile time to test whether or not a reference can be zero. In general, this is not possible, but in many cases this test can be successfully implemented by testing if the code conforms to a number of cases. To demonstrate a difficult (but reasonably common) case, consider the example in Figure 8.15. The compiler will combine all individual arrays over *M* into the array over *N* to create a single object.

However, when trying to remove the indirections over *K* in the print statement, the compiler has to prove that element *K* has been initialized. This requires proving that the loop over *Z* will fully initialize array *arr*. If this cannot be proven, the indirection pointer must be used for computing *A[K]*.

This test becomes increasingly difficult in case of combined objects instead of arrays, as the combined object's constructor might throw an exception. An example of this problem is shown in Figure 8.16. Here the inlinee's object constructor will throw a division-by-zero exception upon calculating 1/0. The exception will be caught and the assignment of the *inliner.inlinee* field will never happen and will still be *null*. Upon trying

```
// some class definitions
class Inlinee {
  int data;

  Inlinee(int n)
  {
    // div. by zero exception !
    data = 1 / n;
  }
}

class Inliner {
  Inlinee inlinee = null;
}

// create a container object
Inliner inliner = new Inliner();

try {
  // initialize the inlined object
  inliner.inlinee = new Inlinee(0);
} catch (Exception e) {
  //   . . .
}

// inliner.inlinee = null;

// should throw a null pointer exception as
// inliner.inlinee should still be null.
inliner.inlinee.data = 1;
```

Figure 8.16: Example: exception analysis is required for indirection removal.

the assignment in the last line of the example, a null pointer exception should be thrown, which would not happen if the indirection pointer were discarded.

A related problem is the resetting to *null* of the indirection pointer by user code, for example, if the line *inliner.inlinee = null;* were uncommented. The compiler will immediately forego indirection removal if assignments with a potential null reference are detected to the indirection pointer.

8.5 Managing Object Combining and Communication

In the previous sections we have described how combined objects can come in existence with the compiler always being optimistic: whenever an object combining possibility presents itself, it is taken by the compiler. However, distributed execution increases the complexity of the decision of whether or not one object should be combined with another. One problem that might occur is when an object which is often written to is combined with an object that is only read from. The read-only part of the container object is then unnecessarily transferred over the network, while it might otherwise have been cached. Another potential penalty is the increased probability of false sharing: if objects *A* and *B* are combined while machine 1 will only write to object *A* and machine 2 will only write to object *B*. The combined object will be *flushed* at each synchronization statement while it could have remained cached if object combining was not applied.

Object combining's effect of increasing message sizes can also be obtained using DAG detection (see Chapter 6), although object combining allows a larger class of objects to be packaged over the network, for example, objects that are part of a dynamic data structure (say a linked-list) or that are in the middle of some larger data structure. DAG detection necessarily operates only at the edges of such data structures and is therefore less applicable. However, object combining does not remove access checks to combined objects, an advantage DAG detection has over object combining.

Any access checks to combined objects remain as they are after object combining has occurred. The runtime system will detect such faults and translate them to access checks to the larger container object. The start_write or start_read object routines in the runtime system are thus changed to:

```
void start_read(object_ptr) or start_write(object_ptr) {
   if (object_ptr is a combined object) {
        call start_read or start_write (container of object_ptr);
   } else {
        retrieve(object);
        map(object);
   }
}
```

8.5.1 Combining Objects for Parallel Efficiency

As it is in general impossible to determine the sharing patterns of objects between threads, we have opted to determine what should be combined by using a profile-based feed-back compilation loop.

```
class Data {
  int field;
  Particle atom = new Particle();
}

class Container {
   Data [] d = new Data[10000];

   Container() {
     for (int i=0;i<10000;i++)
         d[i] = new Data();
   }
}

public static void main(String args[]) {
  Container c = new Container();
  new MyThread(c).start();
}

class MyThread extends Thread {
  Container c;
  MyThread(Container c) {
    this.c = c;
  }
  public void run() {
    for (int i=0;i<10;i++) {
      //   check(c)
      //   check(c.d)
      //   check(c.d[i])
      //   check(c.d[i].atom)

      c.d[i].field++;
      c.d[i].atom.compute();
    }
   //   flush_memory(); (thread exit flushes)
  }
}
```

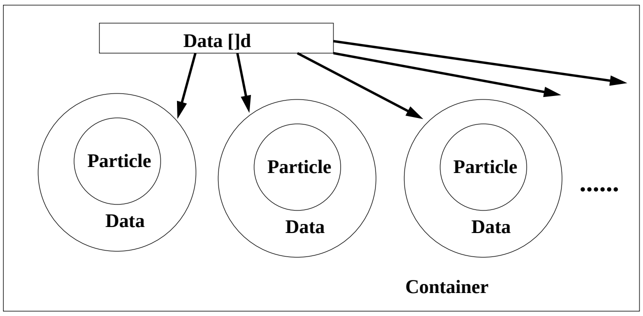


Figure 8.17: Optimistic combining failure.

To illustrate the problems that need to be tackled, let us examine the Java code in Figure 8.17. First, the compiler will notice that the Data object allocations in the Container constructor can be appended to the Data array to create an object array. Next, the whole object array can be appended to the *Container* object in the *main* method. Another object combining possibility lies in combining the *Particle* object with the *Data* object. The result of all these combinings is an object shown at the bottom of Figure 8.17. If our sole objective is memory performance, this situation is completely satisfactory. However, when executing the same code in a distributed fashion new problems arise.

Because subobjects are not individually managed by Jackal — only the container objects are — we introduce a potential false sharing problem. If the *run* method of the new thread is executed at another machine, the whole container object will be sent over the network repeatedly including its combined objects. Because a constituent of the combined object is changed, the whole combined object is marked changed and will be sent back to the home of the container object.

The compiler thus needs to decide that the particle should be appended to the data object, as they are always transferred together to the machine running *MyThread.run()*. Furthermore it should decide not to combine the *Container* object and its *Data* array because the *Data* array is read-only after its creation.

Jackal adapts to these effects by using profile-directed recompilation. In the first run of a program, a history is maintained per object allocation site that marks the number of times an object has been transferred over the network (by flushing or by data requests), how large allocated objects are on average, how many times the type has been instantiated, and finally, how many words are changed when comparing the twin and the working copy before sending the *diff* home.

The compiler will not combine two allocations *A* and *B* if:

- one is transferred more often than another (within some fixed range, 10% by default);
- one is created (far) more often than another, as we assume that for every *A* object transferred a different *B* instance will be transferred over the network;
- the combined object will become larger than some fixed value (512 bytes by default);
- one of the combining objects is an array that is on average larger than some fixed value (512 bytes by default);
- A has some fields changed where B has no changes made to its fields.

DSM object management for combined objects and arrays is maintained as follows: container *objects* are managed as single entities while container *arrays* are still split into several pieces to reduce the possibility of false sharing. Arrays embedded into normal objects will however not be split. Whenever the container object is sent over the network, the whole contained array is sent as well. The same holds for arrays embedded into arrays, the whole embedded array is sent over the network. It should be noted that in case of a container array (of arrays or objects) the first part of the array (containing the references pointing to the combined arrays/objects) is still split into 256 bytes parts as without object combining. Graphically, the memory layouts of Figure 8.18 results.

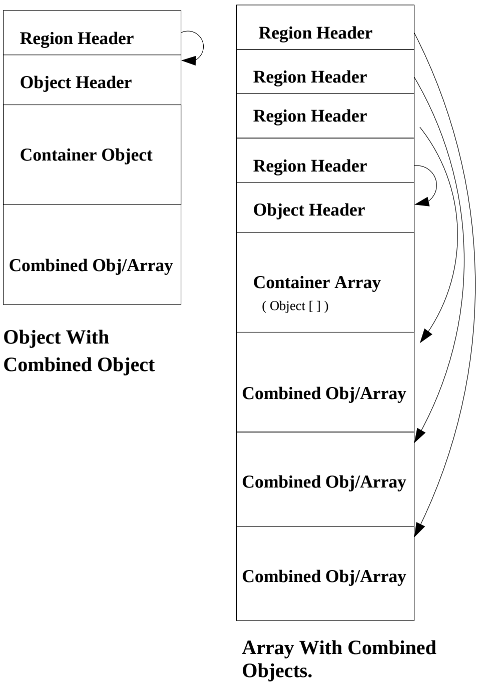


Figure 8.18: DSM object management for combined objects.

8.5.2 Profile Interpretation Problems

In the current implementation, several problems remain in handling the profiles created by the individual machines in the cluster. The first problem arises when a profiling run's input set does not touch all the code paths in a program. The optimizer will then act on incomplete information, potentially worsening performance.

Secondly, a problem arises when a profile run is made by, for example, a cluster of N machines. After a profiling run we will end up with N separate profiles on disk. The recompilation phase will read all N profiles and average them for each allocation site. By averaging all communication and allocation statistics several pieces of information are lost. Consider the following example: object combining candidates *A* and *B* are both transmitted an equal number of times but to different machines. Averaging the profile information will remove this information and result in the compiler not rejecting this object combining candidate.

Thirdly, performance may be highest when combining *A* and *B* for a number of phases in the program while not in others. Such communication behavior might happen if the same objects or types are used for different causes during a programs lifetime. A more dynamic object combining and uninlining scheme may be more beneficial.

Finally, a problem arises if *A* and *B* are transferred an equal number of times but are transmitted at widely different times in the program's execution. The transfers may even have been separated by synchronization statements. The profile will show that the objects have the same communication behaviors so the object combining candidate is not rejected by the compiler. A related problem is when the communication patterns change during a run of a program. For example, if object *A* is transmitted during the first half of a program's execution while *B* is transmitted during the second half of a program's execution. A more dynamic way of handling uninlining may be more appropriate for such programs.

8.6 Related Work

Dolby et al. [14] describe an object inlining technique which has been implemented in the Concert compiler [30] (for a dialect of C++). They replace the pointer to an object immediately by the fields of the object pointed to. Their system disallows pointer overwrites and requires the inlining object to point to the inlined object. Also, their system inlines object X into Y at the position of the pointer to X inside Y. This creates incompatible object layouts requiring complicated cloning of methods to solve problems, where some call sites might expect objects with inlined objects to be passed as parameters while others do not.

Faud et al. [38] also replace the pointer to an object by its data but lack the powerful cloning facilities that the Concert compiler provides. However, they can handle *some* pointer overwrites as they try to determine if the same semantics can be obtained from using deep copies instead of the pointer assignment at pointer overwrites for object inlined objects.

Ghemawat et al. [19] implemented object inlining purely based on type information and the field modifiers. For example, a field that is private inside a private class that is not overwritten can be object inlined. Their analysis, however, does not allow pointer overwrites and the inlining decision is made based on object type alone.

There is also some work on slab-based memory allocation [18]. Slab-based memory allocators attempt to combine objects inside a slab. A slab is allocated with enough room for a number of objects. An object allocation first attempts to allocate inside a slab and only upon failure allocates a new slab. Using this approach, whole slabs are allocated and freed which reduces heap contention. These approaches are similar to our recursive object combining technique from Section 8.3.3, except that we allow such slabs to be created automatically, completely under compiler and runtime control.

Much work has been done on escape analysis [5, 11, 46, 62]. Escape analysis attempts to allocate objects on the call stack and thus, like object combining, reduces object allocation and GC overheads. Some escape analysis implementations remove unnecessary synchronization statements. Escape analysis complements object combining; our system implements both techniques to reduce overheads introduced by an object-oriented programming style. We first try to allocate objects on the stack and only then try to inline an object into another object.

8.7 Summary and Future Work

We have described a new optimization, object combining, which reduces the memory management overhead in object-oriented languages like Java. The optimization performs far more aggressive object-inlining than has been explored thus far: it allows arbitrary pointer overwrites, allocation reordering, combining unrelated objects and using profiling to aid object combining decisions.

In [61] we examined the effects of object combining on a single processor only using the Manta system for six object intensive applications. The optimization reduces memory management time up to a factor 1.5. It reduces the execution time of three applications by 20% to 34%. However, these gains are for applications that heavily allocate objects during their runtime.

There are still some cases where our system fails to combine objects or arrays or where our system is overly aggressive in combining objects. It cannot yet inline multiple (compile-time) variable-sized arrays. In some cases, much unused space is left inside objects, resulting in a much larger heap usage. Better heuristics are needed to solve this problem, for example, using object life-time analysis [53]. This technique attempts to “guess” how long an object might reside on the heap. Whenever two objects have approximately the same “guessed” life times, they are candidates for object combining (even if they are unrelated). Currently, we use profiling to retrieve this information, but the techniques described in [53] might provide this information without profiling.

After object combining has occurred, the opportunities for field reordering [9, 10, 57] and object interleaving [57] increase as more fields are available to the optimizer. It would be interesting to see how much benefit could be obtained from such optimizations.

Finally, we have outlined how Jackal copes with too aggressive object combining decisions by performing profile-based recompilation. A combined object is managed as a single entity, which often causes inefficiencies because of ill-pairing. First a run of a program is made without any object combining enabled. This results in allocation and communication statistics for each object allocation site. At the recompilation phase, the compiler reads and merges the profiles from the separate processes and uses them to reject some object combining candidates.

Two problems remain in the current implementation: handling the profiling information gathered from multiple profiles and coping with dynamically changing communication behavior. The consequences of these limitations are discussed in Chapter 9 on the resulting system's performance.

Chapter 9

Performance

9.1 Introduction

In this chapter we will study the performance of Jackal's subsystems and the effects of the various optimizations. Specifically we will study the speedups, access-check overheads, and the performance of some applications.

To validate our results, we compare Jackal with both Manta's efficient RMI [40] and CRL [29]. RMI is Java's standard RPC mechanism and thus effectively allows the programmer to write hand-optimized message-passing code. Manta uses the same compiler as Jackal (with Jackal's mechanisms disabled). This allows applications to be compiled with the same compiler optimizations for both systems. CRL is a C-based DSM that allows a programmer to hand-optimize his program. The CRL applications were all compiled with a different compiler, namely GCC. For compatibility, we have ported CRL to use Myrinet's GM communication package for CRL's communication systems. The number of changes made to CRL are, however, small. They are restricted to employ different send and receive methods.

Linux Kernel Version	2.4.9
Total Memory	1 GByte
L1-Cache	256 Kbyte
Context Switch Time (wait-notify)	0.4 microseconds
GM roundtrip latency (64 bytes)	24 microseconds
GM roundtrip latency (256 bytes)	41 microseconds
GM bandwidth (256 bytes)	20 MByte

Table 9.1: Test setup data.

9.2 Test Setup

All tests were run on a cluster of 16 1GHz Pentium III machines. Although the machines are dual-processor machines and Jackal is SMP capable, Jackal is configured to use only one of the processors to ensure that the results are comparable to those of CRL, as CRL is not SMP capable. Each machine is equipped with 256 Kbytes of L1 cache and 1 Gbyte of main memory. The machines are all running Linux (2.4.9), connected by a Myrinet [6] network. We use GM, an efficient user-level communication system, as our communication layer. Figure 9.1 shows the relevant data for our test setup.

Jackal uses the standard Linux-threads implementation of pthreads, a POSIX standards API for threads programming. The Linux-threads package supports fully preemptive kernel threads. With Linux-threads a (Java level) context switch takes 0.4 microseconds using wait-notify.

Because GM does not support interrupts, we use a blocking kernel thread (which we shall name “GM-poll-thread” from here on) that waits inside the kernel for messages. Like polling, messages are delivered to Jackal’s runtime system by upcalls.

To reduce the need for (process level) thread switches, GM has been modified to allow the runtime system to poll for messages while the GM-poll-thread remains blocked inside the kernel. This allows most messages to be received and processed without kernel interaction by allowing the application to poll the network and subsequently extract and process pending messages. Whenever a network message has arrived and has been pending for more than 20 microseconds the GM-poll-thread will awaken and process the message. Polling is performed from the idle loop in each machine. As an optimization, the last thread that would suspend in condition synchronization polls the network in stead of using a seperate idle thread.

For a 256-byte message (coincides with the size of one of Jackal’s array partitions), the maximum attainable two-way throughput GM achieves is 20 Mbyte/s (empty request message and 256 bytes of return message payload). At that message size GM achieves a minimum roundtrip latency of approximately 41 microseconds. For smaller messages (64 bytes, which is about the minimum size of a data request sent by Jackal including all of Jackal’s protocol message headers), the latency GM achieves drops to 24 microseconds.

Jackal was configured so that each machine has a maximum of 64 Mbyte of local heap for allocating its own objects while the rest of physical memory is used to cache objects from other machines. If a machine fails to allocate a new object and more than 50 MBytes of its own heap has been exported to other machines (after which a local garbage collection pass cannot reclaim them), a global GC is initiated.

To reduce the number of combinations of the various optimizations proposed in this thesis, only a limited number of likely configurations are shown for each application. Normal compiler optimizations, escape analysis and basic access-check removal are always enabled (see Section 4.4.5). Additional combinations of Jackal’s optimizations are shown where some result is visible.

		#machines		
Program	Optimization	1	2	# messages
Linked-List (μs per list node)	seq	0.05		
Linked-List	par	0.55	29.0	400000
Linked-List	rec	0.6	0.9	3200
1 Kbyte, 1D Array (ms)	seq	4.0		
1 Kbyte, 1D Array	par	12.0	166.0	262144
1 Kbyte, 1D Array	arr	7.0	17.0	65536
Binary-Tree (μs per Binary-Tree)	seq	0.1		
Binary-Tree	par	2.0	98.0	60000
Binary-Tree	DAG	1.7	42.8	10000
Binary-Tree	inl	1.8	36.2	10000
Synchronized / no accesses (μs)	seq	1.0		
Synchronized / no accesses	par	1.0	39.6	400000
Synchronized / no accesses	rsi	1.6	75.3	200000
Synchronized / 5 Accesses (μs)	seq	2.1		
Synchronized / 5 Accesses	par	4.8	684.7	1400000
Synchronized / 5 Accesses	rsi	6.8	102.1	200000

seq = no access checks, par = with access checks,
arr = with array prefetching, DAG = with DAG detection,
inl = with object combining, rsi = with computation migration,
rec = with object combining for recursive types.

Table 9.2: Microbenchmark statistics. Times in μsec per element.

9.3 Microbenchmarks

To examine the performance of the low-level primitives that Jackal is built upon, we start the performance analysis with a number of microbenchmarks. Each of these microbenchmarks has been chosen to show either the base-line performance Jackal achieves or the effects of a specific compiler optimization:

- access-check overhead (with no access-check generated, with access-check removal, with access-check removal + DAG detection);
- bandwidth and null latency (for an array partition, a Linked-List node, and a Binary-Tree with and without object combining);
- null latency for synchronized blocks (with and without computation migration);
- gains of object combining (without Jackal overheads).

We start by examining the performance characteristics of performing Linked-List traversal when run with Jackal. Each Linked-List node contains a reference to the next list node and an integer to function as payload. First we measure access-check overhead by timing the traversal of a 200,000-element list on a single machine both with and without access checks. In this benchmark, one thread will create the entire Linked-List, while another will traverse it.

The relevant statistics are shown in Table 9.2. As can be seen in the table, the access-check overhead is roughly a factor of 10 which is not surprising, since without access checks the main loop is translated into only three instructions. With access checks enabled, one access check is added to the main loop and the loop size increases to eight instructions. Because the traversing thread will not have the Linked-List nodes mapped when it first encounters them, the access check will always fail, which involves the runtime system, so that the Linked-List node can be added to the thread's flush list. Because region states and flush lists need to be protected from concurrent modifications and accesses, a lock needs to be acquired in the runtime system.

The approximate cost of a single access check is 0.55 μs. If the Linked-List node is already mapped, the access check succeeds, so the runtime system is not entered and the access-check costs drop to 0.07μs per list node.

We measured the latency for transferring small objects in Jackal by examining the same Linked-List, but now running on 2 CPUs. At program startup, machine zero creates the Linked-List, and starts a thread. The Jackal runtime system then selects machine one to start the new thread upon, and the new thread computes how long the traversal of the list takes per Linked-List node in the loop. The actions taken by the runtime system are as follows: first an access check determines that a Linked-List node is not locally present and calls the runtime system. The runtime system requests that particular Linked-List node from machine zero. Machine zero replies by sending the Linked-List node to machine one. Because machine zero is now exporting references to local objects, the Linked-List node is scanned and the reference to the *next* Linked-List node it contains is added to the exported table to ensure that the garbage collector will not recycle the Linked-List and its nodes (see Chapter 5). Machine one copies the data out of the message and reactivates the requesting thread.

The measured time thus includes the roundtrip time to the home node and the costs for page mapping, packet inspection, adding regions to the traversing thread's cache, and locking. We measured a latency of 29.0 μs per node of the Linked-List.

This latency can be improved by prefetching some Linked-List nodes with each request. This is implemented by recognizing that the Linked-List nodes are recursive data-structures and are thus candidates for performing recursive object inlining (see Section 8.3.3). After allowing the compiler to detect this situation, the latency per Linked-List node drops to a mere 0.9 μs (packing 125 objects inside a slab, sending whole slabs at a time), because 1,588×2 request/reply messages are sent instead of the normal 200,000×2. This effect is clearly noticeable in the number of messages sent for this benchmark (a drop from 400000 to 3200 messages).

Because array indexing and transfer performance is very important to program efficiency for scientific programs, we will now examine the performance Jackal achieves when traversing large arrays. We will examine array performance using two benchmarks. The first benchmark consists of a single large array of bytes (1 MByte) whose elements are summed, while the second benchmark consists of a large array containing references to other (smaller) arrays.

The throughput obtained by transferring a single large array of bytes is 5.8 Mbyte/s between two machines. In this benchmark, one machine creates the array and another traverses it by reading the value of every sixteenth byte. Only every sixteenth byte is

```
class Leaf {
  int data;
}
class StaticDAG {
  Leaf t = new Leaf();
  Leaf t = new Leaf();
}
```

Figure 9.1: Static DAG structure.

read to reduce the access-check overheads on this benchmark. The costs of performing the access check, entering the runtime system and recording the individual regions of the array into the requesting thread's flush list are contributing to the lower bandwidth. Almost 14.2 Mbytes of GM's bandwidth is thus wasted ($20 - 5.8$). This loss is to be attributed to the transmission of extra message headers required to initialize the array and its associated region headers. For large arrays, throughput is independent of array size/type, because arrays are transmitted region by region. Per array region, the latency is 44.5 microseconds (3.5 microseconds over the base GM latency). Transmitting an array of (non-null) *pointers* requires packet inspection. This cost is clearly visible in the throughput, which drops to 3.9 Mbyte/s when performing the second benchmark.

The low throughput is due to the small region size. The throughput can be increased by letting the compiler generate array prefetching code (see Section 4.4.2). If this optimization is enabled, the array is streamed over the network in chunks to fit a GM packet (4 KBytes), resulting in a throughput of 58.9 Mbyte/s. The increased throughput is also noticeable in the number of messages sent: the message count drops from 262144 to 65536 messages. A decrease in the number of messages sent has the added benefit of reducing the number of interactions with GM.

To demonstrate the effectiveness of DAG detection (see Chapter 6), we will now examine its effects when traversing an array of 1,000 small DAGs (containing only a root, a left and a right node, see Figure 9.1).

DAG detection allows the removal of the access checks to the *Leaf* objects but the checks to the *StaticDAG* objects remain. The removal of the access checks explains the increase in performance in the one-processor case. The decrease in latency by performing DAG detection is good but not optimal, as the runtime system still has to perform the protocol actions for each object in the DAG (initialization and registering them with the flush lists).

The static DAG benchmark can be further optimized by enabling object combining, in which case all three objects comprising the DAG are merged. To ensure that no 'bad' combinings occur, we first perform a profiling run and feed that information back to the compiler. For all profiled applications we use the same input sets for the actual run as for the profiling run. This ensures that we are using the most accurate data for making object-combining decisions. This again results in a speedup, but will not increase single-processor performance as all access checks remain in place. Any performance gains (compared to performing no optimization) in the single-processor case are to be attributed to 'accidental' cache optimizations because of the different layouts of objects

```
// Synchronized / 0 Accesses
synchronized (LOCK) {
}
// Synchronized / 5 Accesses
synchronized (LOCK) {
    a1.field++;
    a2.field++;
    a3.field++;
    a4.field++;
    a5.field++;
}
```

Figure 9.2: Code of Synchronized Micro Benchmarks.

in memory and the reduction in the number of DSM administrative headers. Parallel performance is better than with DAG detection because of the reduction of DSM region management overheads.

The last benchmark in this section is the synchronization microbenchmark, which consists of a thread updating a counter protected by a synchronized block. The counter itself is embedded in a *Counter* object that has an *inc* method to perform the actual operation. The counter objects themselves are located at machine zero, machine one will be executing the synchronized block of code.

As mentioned in Chapter 7, Jackal can optimize synchronized blocks of code by splicing them off to create new functions that can be called remotely (computation migration). As Table 9.2 shows, the function splicing optimization makes (empty) synchronized blocks of code slightly more expensive if the lock object is already present at the local machine and impairs performance in the parallel case as well. This is due to thread switches that must be made at the receiver, because a new thread has to be started at the remote processor. Also it is not guaranteed that the new thread will immediately start its execution. Performance is slightly increased by employing a thread pool, which turns thread creations into notifications on the thread pool using wait-notify. This however still does not guarantee that the new thread will immediate start its execution.

The picture changes when multiple objects inside the synchronized block are accessed, see Figure 9.2. These objects will never have to be sent over the network, as would be the case without computation migration. The performance increase is linear with respect to the number of object transfers that have been saved; in the example five objects transfers have been saved (see table entries Synchronized + 5 Accesses.par versus Synchronized + 5 Accesses.rsi).

9.4 Object-Combining Performance

Because object combining is a new optimization with potentially profound effects on sequential performance, we will evaluate its effects separately. In particular, we will study the effects that object combining has on *sequential* Java programs in this section. To remove any effects that Jackal's runtime system might have in terms of extra overhead,

Compiler	ASP (sec)	SOR (sec)
IBM-JVM	15.61	8.85
Jackal	14.83	8.90

Table 9.3: Performance of Compiler Generated Code for Sample Sequential Programs.

the runtime system is stripped of all DSM code. There are thus no access checks, there is no flush list maintainance and there are no DSM headers allocated in front of each object.

To validate our results we compare against IBM's JVM 1.3.0, which contains an efficient compacting garbage collector and JIT. To reduce the effects of dynamic compilation, both tests are run twice within one application. The measurements shown are taken from the last run. IBM's JVM does not allow us to measure the amount of time occupied by object allocation, only the time by garbage collection.

Performance measurements in this section were all performed on a single, stand-alone 550 MHz Pentium III with 512 Mbytes of main memory and 512 Kbytes of cache running RedHat Linux 7.2.

9.4.1 Object-Combining Microbenchmarks

To give an indication of the general efficiency of Jackal, compared to that of IBM's JVM, we first present some microbenchmarks that are not memory-management intensive. This will place the rest of this section into its proper context. As each of the above compilers has its own back-end and sets of optimizations, the performance is shown for two programs that are not allocation-intensive (see Table 9.3). SOR (successive over-relaxation) takes the average of each point in a 2D matrix between itself and its neighbors. It stresses loop optimizations and array indexing. ASP (all pairs shortest path) finds the shortest paths between any two nodes in a 750 node graph. It is a multithreaded application that executes a synchronization statement after each graph node has been processed. As can be seen from the table, Jackal and the IBM JVM have roughly similar performance for these two applications.

Table 9.4 shows the performance for two microbenchmarks. The column marked *T* shows the elapsed time (in seconds) of the two applications with different optimization settings. *T+ind* denotes the results with indirection removal enabled. The column marked *ind* gives the number of dynamic indirections removed. This number is produced by compiling a separate instrumented version of a program with each removable indirection replaced by an increment of a global counter. The column marked *GC* gives the total amount of time spent in the garbage collector. Total time spent in the memory management code (GC + object allocation) is given separately in the column marked *alloc*. The last column marked *heapsz* is the maximum size of the heap during a run.

The *IBM* row shows the results for the IBM JVM. Since IBM's JIT does not perform object inlining and only the GC times are available to us, T+ind, ind and alloc columns are left empty. The row marked *none* shows the results with object combining disabled. The *all* row shows the results with all object combining options enabled. The *Related* row contains the results when only objects are combined where one (potentially) contains a

	T	**T+ind**	**ind**	**GC**	**alloc**	**heapsz**
LinkedList						
IBM	150.70			63.147	unknown	0.99
none	109.79		0	12.252	80.798	0.18
all	92.44	94.17	0	11.569	65.655	0.18
Related	154.23	150.03	0	12.301	80.574	0.15
no region	92.99	95.18	0	11.671	66.036	0.18
no recursive	94.71	93.14	0	11.561	65.633	0.18
no unrelated	95.19	93.65	0	11.756	66.024	0.18
no cloning	93.23	92.53	0	11.899	66.674	0.18
no profile	159.76	148.12	0	14.233	82.124	0.17
no reverse	***92.40***	94.70	0	13.045	68.333	0.18
Binary-Tree						
IBM	12.24			2.925	unknown	0.99
none	11.99		0	0.348	3.670	0.79
all	11.54	11.55	0	0.290	3.297	1.10
Related	12.64	12.62	0	0.343	3.661	0.76
no region	11.52	11.44	0	0.299	3.281	1.10
no recursive	11.56	11.46	0	0.294	3.311	1.10
no unrelated	11.61	11.45	0	0.289	3.292	1.10
no cloning	11.56	***11.43***	0	0.287	3.258	1.10
no profile	11.71	11.51	0	0.298	3.311	1.10
no reverse	11.57	11.46	0	0.283	3.302	1.10

All times in seconds, heap size in MBytes.

Table 9.4: Runtime statistics for LinkedList and Binary-Tree.

reference to another. The other rows contain the results when all optimizations are enabled but with the named optimization disabled. *No slab* disables profile-based slab allocation (for allocations that the profile indicated were allocated often). *No recursive* disables slab-based allocation for recursive object types. *No unrelated* disables the combining of objects that are always used together. *No cloning* does not search for object combining candidates inside called functions (only the constructor is examined). *No profile* does not use profiling information to reject an object combining opportunity (this version is thus more aggressive). *No reverse* does not change the order of allocations in a program. We have indicated the best result for Jackal in bold face for each application.

Microbenchmark *LinkedList* creates large numbers of linked lists of 100 elements and then removes all references to them. As can be seen in Table 9.4, *Related* worsens performance considerably because it decides to inline the list head into the list object. Unfortunately, new nodes are prepended to the list, so the runtime system discovers that it has to reallocate the head node each time a node is added using *UnInlining*. A profiling run discovers this problem and disregards the object combining candidate. There are no true data accesses in this benchmark so no indirections are removed. LinkedList does benefit from slab-based allocation because it reduces the number of times that memory needs to be zeroed. Also, with slab based allocation disabled, the recursion-based object allocation takes over and achieves the same result.

Benchmark *Binary-Tree* creates a (binary) tree of integer numbers and next reorganizes the tree by creating a new sub-tree until it is balanced. The results in Table 9.4 show that the performance of Jackal without object combining enabled is slightly better than that of the IBM JVM. The *Related* version finds no inlining possibilities (because there is only one object allocation site). It does however convert the code (unnecessarily it turns out) to SSA form and back, which explains the different result from the *none* version. The only object combining possibility is slab based, which is triggered both based on the profile and because the Binary-Tree objects are recursive.

We will now look at the performance for four applications.

9.4.2 IDA*

IDA* implements a 15-piece sliding-tile puzzle. To implement the solver, a multithreaded breadth-first search is implemented. There are N threads that pop a board from a shared job queue, shift the board's blank three ways and push the three new boards back onto the job queue. This process continues until the board that is searched for has been found or the search space is exhausted for the current search depth.

Object combining opportunities stem from creating a multidimensional array instead of the default array of references to integer arrays to implement the boards. The resulting multidimensional array can also be combined with the job object that also contains some integers (the previous position) to disallow the blank to be pushed back to its exact previous position.

Table 9.5 shows that all variants are faster than IBM's JVM even with object combining disabled. All object-combining versions make roughly the same object-combining decisions, and gain about 20% with respect to the *none* version. Although indirection removal removes 41 million pointer indirections, this is not reflected in the execution time,

	T	**T+ind**	**ind**	**GC**	**alloc**	**heapsz**
IDA						
IBM	31.35			5.417	unknown	1.87
none	17.32		0	1.595	8.419	4.75
all	14.12	14.29	41,449,143	1.400	5.735	4.68
Related	13.95	14.11	41,449,143	1.417	5.768	4.64
no region	14.00	14.17	41,449,143	1.391	5.758	4.68
no recursive	14.13	14.32	41,449,143	1.412	5.747	4.68
no unrelated	13.95	14.15	41,449,143	1.391	5.679	4.68
no cloning	14.15	14.34	41,449,143	1.406	5.757	4.68
no profile	14.12	***13.92***	41,449,143	1.413	5.763	4.68
no reverse	13.95	14.16	41,449,143	1.379	5.650	4.68
Sokoban						
IBM	7.28			0.952	unknown	63.9
none	6.83		0	1.666	2.991	97.6
all	6.07	5.59	39,827,623	1.097	1.829	97.6
Related	5.83	***5.49***	39,827,623	1.118	1.819	97.6
no region	6.06	5.57	39,827,623	1.122	1.820	97.6
no recursive	6.06	5.57	39,827,623	1.105	1.812	97.6
no unrelated	6.07	5.57	39,827,623	1.102	1.807	97.6
no cloning	6.05	5.59	39,827,623	1.119	1.854	97.6
no profile	6.06	5.56	39,827,623	1.114	1.835	97.6
no reverse	6.05	5.57	39,827,623	1.097	1.812	97.6

All times in seconds, heapsize in MBytes.

Table 9.5: Runtime statistics for IDA and Sokoban.

because of higher register pressure in performance critical code (an x86 processor only has few registers available).

9.4.3 Sokoban

Sokoban is a game in which a man has to push barrels randomly scattered throughout a 2D maze to the right of the maze. It is implemented using a “maze” object that contains the map of the maze. This map is implemented as a two-dimensional character array and a position vector. Additionally, there is a small table containing references to the last few boards generated. This table prevents the man from moving around in (small) circles.

The application uses dynamic programming: each time the man takes a step or moves a barrel, a reference to the new maze state is stored into a table. Before trying to take a step, the array of previously generated mazes is consulted to see if that maze has already been tried. This check reduces the search space enormously. Because of this optimization, escape analysis will see the recursively generated mazes potentially escape the thread of control and give up. Object combining will then take over, combining the 2D map and the man’s position vector. Because with object combining enabled the working set is larger than 64 MB, our maximum heap size had to be enlarged to 100MB for Sokoban.

Sokoban gains 20% from doing “Related” object combining and gains another 6% from indirection removal by eliminating 39 million dynamic pointer indirections.

9.4.4 Super-Optimizer

A super-optimizer [42] attempts to find better alternatives for a given instruction sequence by exhaustive search over all possible instruction sequences and operands. As the search space is exponential in terms of program length, this super-optimizer only searches for a maximum of a two-instruction replacement. The program is organized around two main functions: a producer recursively generates all permutations of up to two instructions and passes each to a consumer function to test them for possible equality in terms of results for a sequence of test inputs. When a possibly equal sequence of instructions is found, it is added to a queue of possible equals.

Each instruction sequence is a program object which contains the instruction sequence length and a reference to an array of instruction pointers. Each instruction is a triplet of result, left, and right operand objects. A typical instruction sequence consists of 10 objects. With object combining enabled, whole instructions can become single objects.

Generation of new instructions is implemented using:

```
Instruction in = new Instruction(type,
                     new Operand(type1, value1),
                     new Operand(type2, value2),
                     new Operand(type3, value3));
```

To inline all operands into the Instruction object, the instruction object needs to be allocated before creation of the Operand objects. The optimizer thus reorders the allocations (see Section 8.3.1), moving the Instruction allocation in front of the Operand allocations. This decreases execution time by a factor of 1.5 to 64.2 seconds (see the *no reverse* line in Table 9.6).

	T	**T+ind**	**ind**	**GC**	**alloc**	**heapsz**
SuperOpt.						
IBM	102.69			45.036	unknown	0.99
none	93.51		0	0.183	54.083	3.32
all	62.82	61.32	100,659,649	6.793	32.273	3.96
Related	70.73	71.32	218,047,264	5.452	36.045	3.92
no region	60.91	***60.14***	100,659,649	6.630	29.776	3.96
no recursive	62.18	62.03	100,659,649	6.830	32.156	3.87
no unrelated	65.74	65.37	239,050,453	6.816	32.784	3.96
no cloning	62.47	62.11	100,659,649	6.665	32.000	3.92
no profile	62.52	61.32	100,659,649	6.766	32.277	3.96
no reverse	65.21	62.79	100,659,649	6.787	32.043	3.81
Checkers						
IBM	20.898			6.550	unknown	0.99
none	8.61		0	0.612	3.328	0.89
all	8.56	8.72	20,485,593	0.559	3.329	1.0
Related	8.46	***8.38***	20,485,593	0.584	3.374	0.90
no region	8.63	8.70	20,485,593	0.581	3.359	0.95
no recursive	8.55	8.66	20,485,593	0.579	3.322	0.95
no unrelated	8.63	8.67	20,485,593	0.565	3.322	1.0
no cloning	8.58	8.65	20,485,237	0.553	3.321	1.0
no profile	8.64	8.65	20,485,593	0.576	3.362	1.0
no reverse	8.60	8.67	20,485,593	0.586	3.380	1.0

All times in seconds, heap size in MBytes.

Table 9.6: Runtime statistics for SuperOptimizer and Checkers.

Escape analysis is already very effective for the SuperOptimizer and remains so with object combining enabled. The SuperOptimizer already gains much from Related object combining (24%) but gains another 12% by doing some form of slab allocation because the number of zeroing operations decreases. Indirection removal also helps slightly by removing many dereferences to operand objects (1.2%).

When reordering (*no reverse*) is disabled, operands can no longer be combined with the instruction object. The next step is that our object combiner tries to combine them because they are always used together. It combines an Operand with another Operand and the result with an Instruction and achieves almost the same speedup.

9.4.5 Checkers

Checkers is an implementation of the two-player game of checkers played using alpha-beta search. This implementation is a straightforward Java conversion of a C version from MIT's CILK distribution (see [16]), with CILK's *spawn*, *sync* and *abort* statements replaced by a multithreaded scheme with a job queue of spawned explicit invocation-record objects. Object combining opportunities stem from combining the board's data (a one-dimensional integer array marking the positions of the pieces) into the board structure.

There is the possibility of slab allocation of new boards, as the profiled application suggests. The compiler indeed uses the profile to slab-allocate the board data, but the effects are negligible for all versions.

9.4.6 Object-Combining Benchmarks Summary

We now briefly summarize the lessons learned from the measurements. First of all, the measurements indicate that our basic Jackal system (without object combining) has a reasonably efficient memory-management implementation. The sequential speed of the original Jackal programs is at least as good as that of the IBM JVM; usually it is substantially better (up to 59%). As Jackal and the IBM JVM use different garbage-collection and memory-allocation techniques and memory-allocation statistics are not available from the IBM JVM, it is difficult to do a direct comparison of memory-management performance. The statistics that we have collected (about garbage-collection in Jackal and the IBM JVM, and object-allocation in Jackal), however, indicate that the differences are small. Therefore, the results obtained for object combining in Jackal are not biased by a slow memory allocator or a slow garbage collector.

Table 9.7 gives an overview of the performance gain of object combining, computed as the percentage of execution time saved by the best version of Jackal (i.e., shown in bold face in the application tables) and the version of Jackal without object combining. Also, the table indicates which optimizations help most.

The Related strategy of combining objects where one object contains a pointer to the other already improves performance substantially. This optimization is closest to the traditional object inlining optimization, except that our optimization is far more aggressive, as it uses two techniques (*UnInlining* and *Recursive Pointer Rewriting*) as a safety guard against semantical problems (see Section 8.2). Our optimization also searches for op-

Program	Gain	Most important optimizations
Binary-Tree	5%	Slab-allocator
Linked-List	16%	Slab-allocator
IDA	20%	Related
Sokoban	20%	Related + indirection removal
Super-optimizer	34%	Related + slab-allocator
Checkers	2%	Related + indirection removal

Table 9.7: Summary of the performance measurements.

portunities for combining objects that do not point to each other. Slab-based allocation improves performance for both benchmarks and one application.

The measurements also show that several other optimizations have less impact. Removing the indirection pointer from the inlining to the inlined object wins 6% for one application, but has little or no gain for the other applications. Finally, using profiling information from previous runs was useful in the Linked-List benchmark (to prevent the compiler from making poor decisions) but not in the applications that we considered. A problem we have not yet addressed in our work is how to select the best optimization strategy automatically. In general, the *Related* strategy works well for all applications; some (but not all) applications also profit from other optimizations.

9.5 Jackal Compared To CRL

CRL [29], like Jackal, is a complete software DSM system. Like Jackal, CRL uses software access checks to indicate when an application starts to read or write to a region but instead of employing a compiler to add the access checks to a program, CRL requires the application programmer to insert and optimize away the access checks. Also, unlike Jackal, CRL requires the application programmer to terminate his read or write annotations with matching ’end-read’ or ’end-write’ calls. This ensures sequential consistency with a single-writer protocol. Because CRL does not use Java but C, and because C allows a user to perform object combining by hand (which is natural to do in C), the CRL performance shows the maximum performance that a software Java DSM can achieve by prefetching and/or rewriting the user’s data structures.

At the protocol level, Jackal has a slight advantage over CRL: Jackal has adaptive home migration for its regions whereas CRL uses a fixed home node. Jackal thus in principle can significantly reduce the number of messages; however, in most programs the processor that is about to operate on a certain object also creates that object, thus becoming its home-node, which negates any advantages of floating versus fixed home-node schemes for these objects.

Another advantage Jackal might have over CRL (performance wise) is the ability to split arrays into multiple regions automatically. CRL’s syntax and API make this difficult to do for the application programmer. However, in the applications tested, arrays are almost always read or written in their entirety (without false sharing), which negates any

Jackal, CRL and RMI data message counts × 1000					
Jackal optimization	TSP	ASP	Water	BarnesHut	SOR
Basic	15.25	31.04	366.00	269.43	10.37
Computation Migration	**9.27**	**15.03**	363.02	293.01	9.38
DAG Detection	12.82	15.07	**238.78**	262.67	10.33
Object Combining	15.33	29.73	363.99	**147.04**	10.19
Array Prefetching	14.99	29.47	364.61	292.84	**0.80**
RMI	0.59	1.89	0.47	-	0.17
CRL	4.55	34.21	333.96	288.60	1.44

Table 9.8: Jackal, CRL and RMI data message counts on 16 CPUs.

Jackal, CRL control message counts × 1000; RMI has none					
Jackal optimization	TSP	ASP	Water	BarnesHut	SOR
Basic	63.69	151.59	369.81	387.41	12.64
Computation Migration	**21.52**	**45.12**	366.56	416.09	10.83
DAG Detection	53.54	136.54	**240.98**	384.05	12.30
Object Combining	62.35	145.94	367.98	**254.72**	12.43
Array Prefetching	64.08	148.33	368.06	415.75	**2.93**
CRL	3.50	21.40	178.51	120.64	0.30

Table 9.9: Jackal, CRL and RMI control message counts on 16 CPUs.

gains to be obtained from array splitting. The arrays that are shared between threads are almost always read-only arrays partitioned to the number of running threads and containing pointers to the data structures that *are* modified by the application threads.

On average, CRL is faster than Jackal because the C compiler used (GCC) generates more efficient code and because of the "perfect" object inlining decisions made by the application programmer, which reduces the number of uses of indirection pointers. Also CRL in general has better speedups than Jackal. These are in general also due to the better object inlining decisions and the absence of any patching and diffing that Jackal is required to implement for each object that is transferred.

CRL achieves a latency slightly better than Jackal's 29 microseconds for retrieving an empty object: 27 microseconds. For a 256 byte region, CRL achieves a 38.7 microsecond latency.

9.6 Jackal Compared To Manta RMI

To evaluate the relative performance of Jackal, we also compare its efficiency to that of Manta's efficient RMI implementation [40]. RMI is the standard implementation of remote procedure calls (RPC) in Java (see Chapter 1). RMI has several advantages over Jackal, first of all because the programmer has complete control over nearly all aspects of communication (when, where and how much). An RMI program explicitly states whether object *X* should be sent to machine *Y* and if any reachable objects from *X* should be sent

Jackal, CRL vs. RMI data volume in MByte					
Jackal optimization	TSP	ASP	Water	BarnesHut	SOR
Basic	0.13	4.59	1.93	0.24	0.20
Computation Migration	**0.04**	**4.19**	1.96	0.25	**0.18**
DAG Detection	0.11	1055.53	**1.87**	**0.23**	0.20
Object Combining	5.67	4.57	2.27	0.25	0.19
Array Prefetching	0.13	4.57	1.92	0.25	0.18
RMI	0.04	3.65	0.79	-	0.15
CRL	0.3	5.1	7.1	0.8	0.15

Table 9.10: Jackal, CRL and RMI data volume on 16 CPUs.

along with it. Manta optimizes standard RMI by employing compiler-generated serialization code (code that transforms a runtime object graph into an array of bytes) and by using a different wire protocol than prescribed by Sun.

Manta's RMI implementation achieves a latency slightly worse than Jackal's 29 microseconds for performing a null RMI (RMI without parameters and return value): 34.6 microseconds. When performing an RMI with 256 bytes of a byte array as return value, RMI's performance drops to 92 microseconds.

9.7 Application Performance: Jackal, RMI and CRL

Tables 9.8, 9.9 and 9.10 show the number of bytes sent over the network averaged over all CPUs for RMI, CRL and various Jackal compiled versions. These statistics will be discussed below for each of the applications. There is no equivalent Barnes Hut program available to us.

The tables with parallel timings are presented with each application.

9.7.1 TSP

TSP solves the Traveling Salesman Problem for a 15-city input set. First, machine 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 machine tries to steal jobs and to 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 super-linear speedup), the cut-off bound has been set to the actual minimum for this input problem. Consequently, the speedup obtained by all implementations is much lower than in the case of initially unbounded search.

The performance numbers for the various optimizations for TSP are shown in Table 9.11. In the case of TSP, unoptimized Jackal performance is poor because the majority of the work is done in a small recursive function that (initially) contains seven access checks (see *calculateSubPath* in Figure 9.3). The DAG detection optimization discovers that most of 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

```
class Job {
  int []partial_path;
}

class JobQueue {
  Job[] jobs;
  int size;
}

class Minimum {
  int minimum;
}

class DistanceTable {
  int [][]table;
}

void calculateSubPath(int hops, int length, int[] path) {
  int me, dist;
  // 1 --- read-check (m)
  if (length >= m.minimum) return;
  // 2 --- read-check (this)
  int nc = nrCities;
  if (hops + 1 == nc) {
    m.set( length );
    return;
  }
  // Path really is a partial route.
  // Call calculateSubPath recursively for each sub-path.
  // 3 --- read check path[hops]
  me = path[hops];
  for(int i=0; i<nc; i++) {
    //TSP.present inlined: read-check path[i] in loop over nc
    if (! TSP.present(i, hops, path)) {
      // 4 --- write-check path[hops+1]
      path[hops+1] = i;
      // 5 --- read-check distanceTable
      // 6 --- read-check distanceTable.table[me]
      // 7 --- read-check distanceTable.table[me][i]
      dist = distanceTable.table[me][i];
      calculateSubPath(hops+1, length+dist, path);
      }
   }
}
```

Figure 9.3: TSP: *unoptimized* main compute code and data structures.

	#machines				
TSP	1	2	4	8	16
IBM JVM	20.99				
CRL	9.06	4.59	2.32	1.18	0.66
RMI	14.57	7.56	3.88	2.14	1.41
NoChecks	10.80				
Basic	23.47	12.17	6.28	3.29	1.96
ArrayPrefetch	60.09	30.50	15.42	7.88	4.30
DAG-Detection	16.61	8.68	4.53	2.47	**1.59**
ObjectCombining	25.23	15.31	10.08	7.50	7.21
+ profile usage	25.23	15.44	10.14	7.41	6.85
ComputationMigration	24.95	12.92	6.69	3.47	1.95
+ ArrayPrefetching	55.34	28.03	14.26	7.24	4.03
+ piggy-backing	24.96	12.85	6.67	3.48	1.94
+ piggy-backing + array-prefetching	55.53	28.11	14.28	7.30	4.01

Table 9.11: TSP: CRL, RMI and Jackal timings (seconds).

caller of the recursive function and combined into two access checks to check availability of entire object graphs for the partial path and for the city distance table.

The speedups for the configurations with DAG detection disabled therefore suffer from inferior sequential code. When enabling computation migration combined with piggybacking, faulting in a new partial path takes fewer data messages (for the Job object and the partial path contained in it) and many fewer control messages: some for requesting the regions, and some for synchronization control. Performance still lags behind because of the high costs of switching to a new thread for executing the migrated code (due to heavy-weight kernel threads). With all optimizations enabled, a partial path is obtained with one roundtrip, where the migration reply carries the object graph for the partial path. Then the RMI program and the optimized Jackal programs transmit approximately the same amount of data.

Object combining when applied optimistically worsens performance because it combines objects too aggressively. For example, it will combine the job-queue object with the array of jobs contained, and the array of jobs with the job objects themselves. This behavior is slightly improved with the use of the profile but the compiler still makes roughly the same decisions; the minimum object is no longer object combined.

Array prefetching actually slows down performance in TSP because the call to the runtime system to perform array prefetching is inserted into the (recursive) critical path of TSP. Furthermore the array prefetch inserted is superfluous as it is often performed on the same read-only data (the table holding the distance between the various cities).

The CRL version of TSP has better performance because the programmer has already inlined the partial path in the job object and has moved the start-read from inside the partial path expansion function to its caller. This is legal as the programmer *knows* that the combined path will not be modified by other threads while inside *calculateSubPath*.

	#machines				
ASP	1	2	4	8	16
IBM JVM	28.41				
CRL	16.85	8.62	4.46	2.38	1.16
RMI	21.04	10.78	5.73	3.34	2.15
No Checks	22.04				
Basic	47.61	27.33	14.71	8.23	4.62
ArrayPrefetch	20.10	12.96	7.30	4.49	2.80
ComputationMigration	36.72	18.83	9.78	5.36	3.23
+ ArrayPrefetching	19.96	12.78	6.15	3.44	2.30
+ piggy-backing + array-prefetching	20.19	12.83	6.15	3.25	**2.03**
DAG-Detection	18.34	35.15	77.06	127.99	149.26
ObjectCombining	36.66	19.01	10.09	5.70	3.37

Table 9.12: ASP: CRL, RMI and Jackal timings (seconds).

9.7.2 ASP

The All-pairs Shortest Paths (ASP) program computes the shortest path between any two nodes in a 1000-node graph. Each machine is the home node for a contiguous block of rows of the graph's shared distance matrix. In iteration *k*, all threads —one per machine— 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 machine in turn (each thread inserts a reference into every other thread's local matrix of distances which causes broadcast-like behavior at the network level). 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 spanning tree. This message is sent asynchronously by a special forwarder thread on each node to avoid latencies on the critical path.

For Jackal, this type of spanning tree is superior to the spanning tree used for the RMI implementation: the latter would involve, besides thread-spawning overhead for computation migration, two network roundtrips per iteration on the critical path. This scenario turned out to ruin performance. The current spanning-tree implementation also hampers performance, because the receivers in the spanning tree lag further behind in each iteration: fetching a row involves interrupting the parent, spawning a computation-migration thread, often invoking a wait(), and wakeup. Because the parent usually has two children, the costs are doubled to two interrupts and up to four thread switches per hop in the spanning tree. For the leaves in the spanning tree, this delay amounts to ca. 600 μs per interation, which explains the performance difference between CRL and Jackal. The speedup of the RMI program suffers, like Jackal, from the high costs of thread switching.

It should be noted here that the implementation of the spanning tree is completely implemented in Java. There are no native method calls to specialized low-level network

```
public class MoleculeEnsemble {
    public double[][][][] forces;
}
```

Figure 9.4: Water molecule encoding.

primitives to implement the broadcast.

Jackal's computation migration optimization significantly increases speedup (see Table 9.12) since it allows the combining of data with the synchronization messages. Fetching a broadcast row with computation migration combined with piggybacking costs now only one round-trip.

With array aggregation enabled, the compiler detects that the inner loop operates on whole rows, so the access checks in 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 versions with DAG detection enabled fetch the entire distance table matrix each iteration instead of only the changed column. This needlessly causes much read-only data to be exchanged over the network. The object-combining algorithm fails to combine the rows of the matrix into a multi-dimensional array because some rows of the distance table matrix are overwritten each broadcast and thus no object combining occurs in ASP. Object combining thus achieves performance comparable with the unoptimized version. The difference in the sequential code speed and that with object combining enabled on a single processor lies in the code transforms performed in preparation for object combining (cloning object constructors, transforming the code to SSA form and back) which in case of ASP has a positive effect on caching behavior.

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

9.7.3 Water

Water is a Java conversion of the corresponding SPLASH program written in C. Water performs an (N-Square) N-body simulation of a number of moving water molecules. Communication is required to recompute the interactions between the molecules after the water molecules have moved inside an iteration.

Work is divided by subdividing the number of molecules amongst a number of worker threads. After each thread has finished computing new directions and accelerations for its own molecules, it publishes those results for the other threads by entering a special summation/barrier class which besides its thread synchronization function simultaneously exchanges its updates with other threads. After the barrier each thread will have its cached molecules invalidated and request new molecule data.

A water molecule is coded as shown in Figure 9.4. The forces array holds the forces, directions and accelerations for for all the water molecule's atoms. The compiler is able to determine the length of each of the arrays and is able to determine that none of the arrays will be null at runtime.

The heap graph analysis confirms that the individual sub-arrays inside the molecule

	#machines				
Water	1	2	4	8	16
IBM JVM	31.0				
Pure C	18.92				
CRL	23.41	13.07	7.04	4.00	2.55
RMI	20.26	10.32	5.38	2.71	1.52
No Checks	24.88				
No Escape-Analysis	28.73				
No Access-Check opt.	55.00				
Basic	38.49	21.16	11.29	6.24	4.20
ArrayPrefetch	25.29	14.26	7.80	4.49	3.23
ComputationMigration	25.57	14.62	8.58	6.09	8.38
DAG-Detection	24.37	13.40	7.22	4.22	3.03
+ DAG Detection (2)	24.36	13.43	7.24	4.32	3.05
+ DAG Detection (4)	24.55	13.36	7.08	4.05	2.79
+ DAG Detection (5)	24.54	13.41	7.26	4.58	5.16
+ DAG Detection (6)	24.66	13.49	7.25	4.64	5.34
+ Array-of-DAG Detection (3)	23.87	13.16	7.10	4.01	3.03
+ Array-of-DAG Detection (4)	25.95	14.60	7.70	4.51	**2.61**
+ Array-of-DAG Detection (5)	25.22	14.24	7.68	4.89	5.32
+ Array-of-Dag Detection (6)	25.51	14.26	7.54	5.04	5.23
ObjectCombining	26.41	14.73	7.94	4.68	3.32

Table 9.13: SPLASH-Water: CRL, RMI and Jackal timings (seconds).

are not aliased and form a static DAG. Indeed, as expected the most effective optimization for water is DAG detection (see Table 9.13) where the access checks to the molecules position, acceleration etc. vectors are removed and whole *MoleculeEnsembles* are transferred over the network including the *forces* array. Performance can be optimized by lowering the DAG-weight to forego the network transfers of some of the read-only inner dimensions of the *MoleculeEnsemble*. Higher DAG weights do not truly aversely affect performance (see the versions of water compiled with with DAG Detection (X) in Table 9.13).

The CRL version of Water has all array dimensions flattened to create a single region: C supports true multidimensional arrays, whereas Java supports arrays of arrays only. This means that there are *no* pointers to lower dimensions that need to be transferred over the network. The CRL version of water also suffers from significant instrumentation overhead (see *CRL* vs *Pure C*) even though the programmer of Water has made some effort to reduce this overhead. The RMI version of Water solves the problem of the read-only dimensions by recreating the whole array at the receiving side (at the cost of potentially more garbage collection).

The Jackal version without any optimization performs *very* poorly. The major overheads stem from the allocation of a very large number of temporary matrices and vector objects that escape analysis is able to eliminate successfully (see *No Escape-Analysis* vs. *No Checks*). Another major source of overhead is the cost of the access checks which the standard access-check optimizers are very successful in eliminating (see *No Access-Check opt.* vs. *Basic* on 1 machine). With all access-check optimizers enabled, the access-check overheads are reduced to 1.2% and is on-par with the performance of CRL on a single machine.

Object combining can also be triggered for water, which results in the transfer of whole *MoleculeEnsembles* over the network. Unlike DAG detection, no access checks are removed and sequential execution speed stays the same. Because of a reduction in the number of regions that must be managed by the runtime system, performing a flush of working memory is less expensive.

Computation migration is triggered for a number of small synchronized blocks of code inside the summation/barrier class. With computation migration enabled, transfers of the summation object are eliminated; instead RPCs are performed to machine 0. This summation object is also at times used as a barrier for all threads. Each time the barrier is hit, an integer inside the barrier, denoting the number of waiting threads at the barrier, is increased. This causes the summation object to be marked dirty which in turn causes many protocol actions when computation migration is disabled.

Array prefetching is able to remove large numbers of access checks inside some critical (but simple) loops, increasing sequential performance. The arrays themselves are however small, usually with 3 to 8 elements that fit inside a single region. Array prefetching thus has little effect on parallel performance.

9.7.4 Barnes-Hut

Barnes-Hut is an *O(N log N) N*-body simulation taken from the SPLASH benchmark suite [63]. Unfortunately, no equivalent RMI version of Barnes-Hut is available to us.

BarnesHut	#machines 1	2	4	8	16
IBM JVM	5.81				
Pure C	2.11				
CRL	5.04	3.04	2.31	1.62	1.10
No Checks	3.94				
No Access-Check opt.	9.77				
No Escape-Analysis	25.35				
Basic	5.14	4.81	3.99	2.89	2.21
Array-Prefetch	4.63	4.52	3.83	2.85	2.18
DAG-Detection	4.62	4.41	3.65	2.59	1.99
Computation-Migration	5.09	4.81	4.10	3.00	2.55
Object-Combining	4.58	9.74	6.19	5.51	5.54
+ profile usage	4.68	4.14	2.84	1.99	**1.62**
+ profile usage + Computation-Migration	5.74	5.02	2.93	2.60	1.96

Table 9.14: SPLASH-Barnes-Hut: CRL and Jackal timings (seconds).

The Jackal and CRL versions however are directly comparable.

Barnes-Hut is coded in an object-oriented fashion using a class hierarchy to code its oct-tree of bodies. The most important classes are: *Node*, *Cell*, *Body* and *Leaf* (see Figure 9.5). The classes *Cell*, *Body* and *Leaf* all inherit from class *Node*. Instances of *Cell* represent the internal nodes inside the oct-tree. *Leaf* instances hold eight references to *Body* class instances where each *Body* contains references to both a velocity and acceleration vector object.

The program operates in two phases. During the first phase, the *maketree* phase, each thread aids in the creation of a global oct-tree of bodies. Barnes-Hut performs its own memory management by taking inserted bodies from a global array and returning the bodies to it afterwards. This reduces the strain on the memory manager. To ensure that each body is treated by one thread at a time some form of synchronization is required. In the next phase, the *compute* phase, the forces and interactions between the bodies are computed. These two steps are repeated for a fixed number of times. The last two iterations are timed.

For each of the systems we investigated (Jackal, C/CRL and the IBM JVM), Barnes-Hut shows large differences in sequential execution times; the compilers are able to generate comparatively efficient code (see Table 9.14) but each has a different code generator and input language (C, Java or Java bytecode). The pure C version of BarnesHut is however appreciatively faster than both the *IBM JVM*, *C/CRL* and *Jackal/No-Checks* versions. The difference in execution times between the CRL and Jackal versions without instrumentation is to be found in the extra memory indirections and in the cast checks Java mandates to cast from *Node* objects to *Cell*, *Body* and *Leaf* objects. Another overhead compared to CRL is thread synchronization; CRL is single-threaded, and thus has no thread synchronization overhead.

The access-check overhead for the BarnesHut application is reasonable (23.3% overhead, certainly when compared to the overhead of CRL). The standard access-check removal optimizations are extremely effective, as can clearly be seen when comparing the *No Access-Check opt.* version with the *Basic* version. An optimization that is also very important for sequential performance is escape analysis. During its computation phases Barnes-Hut allocates a large number of temporary vector objects to hold positions, directions and accelerations of bodies. With escape analysis enabled no garbage collections occur during the applications execution.

The major constituent of Jackal's parallel overhead compared to CRL stems from the high cost of thread synchronization. The CRL version synchronizes by using a *start_write* annotation which excludes all other processors from accessing that region until the accompanying *end_write* annotation has been executed. In this way no additional locking mechanism is required for CRL programs; CRL effectively combines synchronization with data accesses. The Jackal version, however, uses Java's method of thread synchronization to implement the locking of bodies. Each synchronize operation requires two flushes of working memory in addition to the costs of Jackal's heavy-weight synchronization protocol. Flushing a thread's working memory requires looking at *all* bodies whereas CRL requires looking at the region that is locked. Lazy flushing as Jackal implements it reduces the number of objects that need to be checked during a flush as some objects will be considered local but the number of objects is still far larger than with CRL.

A more detailed timing run of both the CRL and Jackal versions shows that both spend about the same time in the *compute* phase. All the performance difference between CRL and Jackal is due to the *maketree* phase because of its heavy use of thread synchronization. On 16 machines, the *maketree* phase takes about 0.09 seconds for the CRL version, and 0.45 seconds for the Jackal version with profiled object inlining enabled. Of these 0.45 seconds, 0.11 seconds are spent in waiting for objects, 0.18 seconds are spent in flushes, and 0.05 seconds are spent in lock acquiring. Thus, about half of the Jackal overhead is due to the Java Memory Model, which requires flushes of working memory.

Object combining without using profiling information worsens performance because optimistic object combining is too aggressive. By employing the profile Jackal is able to optimize the application's communication behavior by combining the vectors inside the *Body* object (see Figure 9.5) with the body itself, and combining the *Cell* reference arrays inside Cell objects with the *Cell* object. Object combining is also triggered for the *Body* reference array with the *Leaf* object that contains a reference to it.

The same effect can be achieved by enabling DAG detection but this solution suffers from the overheads of maintaining the DSM header administration for the sub-DAGs. This overhead is eliminated when using object combining at the expense of no reduction in access checks. Furthermore, although the compiler is able to detect several DAGs in the heap graph, the access checks cannot always be removed or combined to create DAG prefetches, as the number of casts from *InternalCell* to either *Leaf* or *Cell* creates aliasing which the compiler currently cannot overcome.

Finally, enabling computation migration again worsens performance because of the high costs of thread creation and thread switching. Array prefetching helps sequential performance a little by removing access checks to the various arrays of bodies, leafs and cells during the *maketree* phase.

```
class Node {
  Vector pos = new Vector();
  double mass;
  int cost, level, child;
  Cell parent;
}
class Body extends Node {
  Vector vel = new Vector();
  Vector acc = new Vector();
  double potential;
}
class InternalCell extends Node{
  InternalCell next, prev;
}
class Cell extends InternalCell {
  InternalCell [] body_table = new InternalCell[8];
}
class Leaf extends InternalCell{
  Body [] body_table = new Body[8];
}
```

Figure 9.5: Barnes-Hut data structures.

	#machines				
SOR	1	2	4	8	16
IBM JVM	3.41				
CRL	1.78	0.91	0.47	0.24	0.13
RMI	2.62	1.36	0.68	0.43	0.29
No Checks	1.94				
Basic	4.07	2.08	1.06	0.57	0.34
ArrayPrefetch	2.17	1.12	0.57	0.31	**0.21**
DAG-Detection	4.05	2.04	1.05	0.56	0.37
ObjectCombining	4.07	2.06	1.06	0.56	0.34
ComputationMigration	4.12	2.30	2.28	1.34	1.12
+ ArrayPrefetching	2.61	1.95	1.28	1.32	1.10

Table 9.15: SOR: CRL, RMI and Jackal timings (seconds).

9.7.5 Successive Over-Relaxation (SOR)

Successive Over-Relaxation (SOR) is a well-known iterative method for solving discretized Laplace equations on a grid. We parallelized this program using one thread per machine; 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 2048×2048 (16 Mbyte) matrix. The execution times are given in Table 9.15. The Jackal compiler is able to optimize the SOR program, because it can

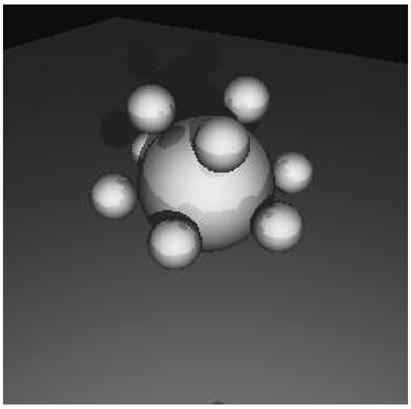


Figure 9.6: Ray-traced picture.

statically determine that an entire row from both neighboring partitions will be accessed. The compiler therefore inserts array-prefetching code to fetch an entire row (from both the left and right neighbor) before the innermost loop. Also, the compiler can determine that the inner loop executed by SOR is finite and small, so it removes all access checks from the inner loop by inserting array-prefetching code before the innermost loop. As can be seen from Table 9.15, these optimizations are highly effective for SOR (see versions with array prefetching enabled). On a single machine with array prefetching enabled, the Jackal program runs almost as fast as the *No Checks* version of SOR, so the access-check overhead has been almost eliminated.

DAG detection is not triggered because of the way the matrix is allocated: each thread creates the partition of the matrix it is going to operate upon. This creates aliasing between the rows of the matrix from the viewpoint of the compiler. This also confuses the object combining optimizer in the compiler and no object combining is triggered.

There is an opportunity to perform computation migration but enabling this optimization affects performance adversely. The sequential performance is also affected because of the costs of calling the spliced function indirectly and re-applying the arguments to the spliced function.

The Jackal version of SOR with array prefetching enabled slightly outperforms the RMI version, mostly because of the more efficient code that is generated for the main tight loop to perform the actual matrix operations SOR performs. The RMI version performance suffers from the overheads of the creation of the RMI handler thread and the object creations in the de-serialization process.

9.7.6 A Ray-tracer

We have implemented a parallel ray-tracer in Java. The ray-tracer application is different from the other applications examined in this thesis in that it does not employ a static working set but it requires the use of Jackal's global garbage collector.

The ray-tracer operates in two phases. In the first phase, each machine renders a

	#machines				
	1	2	4	8	16
run time (s)	243.1	190.9	148.2	115.0	104.3
trace time (s)	236.5	149.2	88.3	45.6	27.2
#local GCs	153	86	61	35	25
#global GCs	0	6	4	4	4
#GC messages	0	3910	9653	10814	10574

#local GCs and #GC messages are for processor 0 only

Table 9.16: Runtime statistics for the ray-tracer (seconds).

fraction of the screen bitmap; in the second phase, machine 0 assembles all bitmap fractions. The second phase is inherently sequential, so speedups can only be obtained for the ray-tracing part. The picture rendered is shown in Figure 9.6.

The ray-tracer operates on a two-dimensional array of *Color* objects whose values are computed using the ray-tracing technique. Ray-tracing works by “shooting” a ray of light from the observer’s eye through a pixel on the screen hitting a virtual set of objects. The color of the ray is initialized when it reaches a light source and the values are propagated and adjusted after retracting from any objects that have been “hit”. When retracting through the screen, the affected pixel is colored by setting its pixel to that of the retracting ray. The *Color* objects computed are also allocated at the computing machines.

Each *Color* object has a size of 84 bytes, including the DSM and object headers. After a thread has computed its piece of the screen bitmap, it flushes its bitmap to machine 0 (thus overwriting the pixels from a previously rendered picture). Since machine 0 receives bitmaps from all other machines, it will use many pages of memory to store the cached *Color* objects. This will trigger global GCs to remove dead *Color* objects from its address space. Occasionally, while tracing, another machine will request a global GC, because it has accumulated too many objects in its export table (see Chapter 5). The ray-tracer also creates many vectors and ray objects that almost immediately become local garbage.

For a 512 × 512 picture, the ray-tracer generates approximately 22 Mbyte of shared data. For the statistics, we generated 8 frames, for which the Jackal executable needs 243.1 seconds. Without access checks the ray-tracer requires 200.1 seconds. access-check overhead for the ray-tracer is thus 21.5%. For comparison, the bytecode generated by Sun’s JDK 1.3 on Linux, JIT enabled, renders the image in 134.58 seconds because of its very efficient memory management system. IBM’s JVM renders the image in 221.5 seconds. There is no equivalent CRL version because CRL does not contain any form of garbage collection of regions nor does it allow regions to be freed from the system.

Table 9.16 shows runtime statistics for the ray tracer, for various numbers of machines. Table 9.17 shows the same statistics but with object combining enabled. As more machines are added, each machine operates on a smaller chunk of the picture, so that the number of local garbage collections per machine decreases. In contrast, when machine 0 traverses the pixel array to generate the full picture, it will need more pages to cache the *Color* objects from other machines, and will therefore trigger global GCs. This behavior accounts for the global GCs on 4, 8, and 16 machines. The number of pages needed to

	#machines				
	1	2	4	8	16
runtime (s)	170.82	112.99	71.24	41.32	26.04
trace time (s)	169.88	107.91	65.87	34.74	18.92
# local GCs	502	289	296	225	106
# global GCs	0	2	2	2	2
# GC messages	0	44	75	111	165

#local GCs and #GC messages are for processor 0 only

Table 9.17: Runtime statistics for the ray-tracer, profile based object combining enabled (seconds).

Application	Basic/1	Best	Best/1	CRL/1
TSP	2.17	DAG-Detection	1.54	0.84
Asp	2.16	Comp.Migr+Arr.Pref.	0.92	0.76
Sor	2.10	ArrayPrefetch	1.12	0.92
Water	1.55	Array-of-Dag Detection (4)	1.04	0.94
BarnesHut	1.31	Obj. Comb./profile	1.19	1.28

Table 9.18: Jackal and CRL sequential overheads for each application, normalized to *No Checks* Jackal versions.

cache the *Color* objects increases only marginally with the number of machines, which explains why machine 0 does not trigger more global GCs on 16 machines than on 4 or 8 machines. The global GCs in the run on 2 machines are triggered by machine 1 because it has accumulated too many objects in its export table. When running on more machines, machine 0 will trigger global GCs before the other machines will, because it caches more objects.

The performance of the ray-tracer can be increased significantly by enabling object combining, specifically by allowing the use of a profile to enable the use of a slab allocator (see Section 8.3.4) for the allocation of the *Color* objects (that eventually constitute the pixmap). This allows many *Color* objects to share one DSM header and to be automatically aggregated when transferred over the network.

The sequential speedup is also to be attributed to the reduction of DSM headers which reduces the amount of time spent in the garbage collector (the number of local GCs increases but each GC is finished far sooner). The reduction in global GCs is also caused by the reduction in DSM headers, which causes the address space to be filled more slowly.

9.8 Summary

We have evaluated the performance of Jackal's optimizations for six applications (see Table 9.18 and Table 9.19). Each of these applications performed poorly without any optimization applied: either the sequential performance was poor because of access-check overheads or the parallel performance was poor because of the lack of aggregation. For both these problems multiple solutions have been proposed in this thesis: array prefetch-

Application	Basic/16	Best	Best/16	CRL/16
TSP	0.18	DAG-Detection	0.15	0.06
Asp	0.21	Comp.Migr+Arr.Pref.	0.09	0.05
Sor	0.18	ArrayPrefetch	0.11	0.07
Water	0.17	Array-of-Dag Detection (4)	0.10	0.10
BarnesHut	0.56	Obj. Comb./profile	0.41	0.28

Table 9.19: Jackal and CRL parallel speed increases for each application, normalized to *No Checks* Jackal versions.

ing and DAG detection for sequential performance improvement and computation migration, object combining and again DAG detection to enhance parallel performance. Each of these optimizations proved beneficial for at least one of the applications investigated. The access-check overhead without optimization ranges from a factor of 1.3 to 2.1. With optimization applied the access-check overhead becomes a factor of 1.0 to 1.5.

CRL is faster both in sequential and parallel performance. The difference with CRL is on average a factor 1.7 (compared to Jackal best). Part of this difference is due to performance differences of the base languages (C vs. Java). On average, the Pure C versions of the five applications are a factor of 1.2 faster than the Java versions.

For TSP the performance difference when compared to the CRL version is due to the access-check overhead. The CRL application programmer has been able to remove all access checks from the critical path. The Jackal compiler was not able to do the same.

For ASP the performance difference with the CRL version is due to the multithreading overhead imposed by the expensive threading package Jackal uses. Each message that is delivered causes about six thread switches, which results in poor performance. CRL does not support multithreading.

SOR suffers from control message overheads and the lower array throughput. The runtime system is unable to eliminate all of home-migration's associated control messages (see Table 9.9).

Water performs reasonably compared to CRL but performs poorly when compared to RMI. Both the CRL and Jackal versions operate on a too small granularity: one molecule at a time. The RMI version requests whole ranges of molecules at a time and optimizes communication by transmitting only the data that has actually been modified.

BarnesHut suffers from high synchronization overheads compared to CRL. The CRL version has been tuned to combine synchronization with communication, but unlike the Jackal version only the body under consideration is flushed. This is caused by the JMM which forces a thread that executes a synchronization statement to flush *all* of working memory while CRL allows the programmer control over which objects should be flushed from working memory and which should remain cached.

Programs exhibiting large arrays as their main data structure naturally benefit from array prefetching, see for example *SOR* and *ASP*. However, in some programs it is difficult to detect whether an array is indeed large and a large portion of it is accessed inside a loop. This problem is exhibited by for example *TSP*, where array prefetching is triggered but the added overheads of array prefetching completely remove any performance benefits. In the

CRL versions of *SOR* and *ASP*, the programmer manually performed array prefetching because CRL does not split arrays into multiple independently managed chunks.

Programs exhibiting static data-structures, such as multi-dimensional arrays or objects pointing to other objects, benefit greatly from DAG detection, as has been demonstrated by their use in *TSP* and *Water*. However, the problem with DAG detection is the lack of proper heuristics as to when, and when not to apply. This problem is most obvious when examining the performance of *ASP* with DAG detection enabled. The CRL versions, because they are programmed in *C*, manually inline data structures because this is a natural way to write *C* programs. However, this forces the programmer to decide object inlining depths aforehand; he may need to adjust these depths when the CRL program is optimized. Jackal allows these object inlining depths to be automatically adjusted through a command line parameter to the compiler without the need to modify a program's source codes.

When none of the other optimizations apply, our last resort is object combining. Object combining, although powerful, lacks the proper heuristics on when and when not to apply automatically. When possible, however, the object combining optimization can use a profile of object usage and creation statistics to guide Jackal's object combining decisions.

Object combining helps when large numbers of related objects are sent over the network but their relation is either too complex to be analyzed by techniques such as DAG detection or the overheads of managing multiple independent objects are too large. This situation occurs in for example *BarnesHut* where the aliasing relations are too complex to trigger DAG detection because of the use of aliasing and inheritance. The CRL version of *BarnesHut* indeed performs the same object-combining decisions, but these were performed by the programmer.

Chapter 10

Conclusions and Future Work

We have described Jackal, a system to run unmodified, multithreaded, Java programs on a cluster of workstations connected by a fast network. Jackal includes support for garbage collection, arbitrary numbers of threads per machine, and Java's object orientation and safety features such as automatic and transparent cast and array bounds checking.

To allow parallel execution on a cluster we simulate a large distributed shared memory. We rely on the compiler to add code to the program to detect accesses to objects or array segments which are not locally available or not currently up-to-date. This approach introduces two potential performance problems:

- the performance on a single processor can be poor because the overheads incurred by executing unnecessary access checks is prohibitive;
- the parallel performance can be hampered because Jackal communicates small pieces of data;
- the JMM requires that *all* data be flushed at synchronization points.

The goal of this thesis is to study compiler and runtime optimizations that address these problems.

As the performance evaluation has shown, several applications indeed suffer from inferior performance when no optimization is applied. The execution time of *SOR* and *TSP*, for example, nearly doubles when no optimization is applied (in both sequential and parallel cases). Array prefetching solves this problem for *SOR* and is straightforward to implement using standard induction variable analysis. However, it is not immediately clear when to apply this optimization, as performance is impaired when performing array prefetching for a loop over an array where the array is already locally available or the loop is small. This effect occurs when array prefetching is applied to *TSP*. One way to solve this problem is to do a profiling run first and gather usage statistics for each array and loop pair. The decision whether to apply array prefetching is then taken after examining the statistics. However this possibility is not explored in this thesis.

Some applications suffer from an excessive number of thread synchronization events which causes some objects to "bounce" around the network. A function shipping approach for these objects and code sites is more efficient than Jackal's default data shipping approach. One solution to this problem that we have examined in this thesis is to splice such synchronization blocks out to new functions and to execute those pieces of code at the machine where the lock object is located. This effectively amounts to performing computation migration for small pieces of code. Computation migration, however, is a more general technique that could even in its current form be applied more often, for example, to code that frequently accesses remote objects. This is possible future work.

Another optimization explored in this thesis is DAG detection. The goal of DAG detection is to remove access checks from leaves of the DAG and to transfer whole DAGs at a time over the network. For some applications this optimization is essential for achieving good performance, either sequential of parallel. For example, *Water* and *TSP* require DAG detection to reduce access-check overheads. However, not every application exhibits (static) DAG-like data structures in its heap, which restricts the applicability of the optimization. Also, applying DAG detection indiscriminately can worsen parallel performance by transferring too much data over the network. This happens when read-only data and modified data are aggregated. Our final fall-back is to allow the programmer to choose the number of objects (actually allocation sites) or DAG weight that should be part of a static DAG. Again, this is not a perfect solution because an application operating on multiple data structures might need different DAG weights for each data structure. A better solution to this problem would be to adjust the DAG weights by either executing a profiling run or to dynamically adjust the running code image. This is possible future work.

A final optimization we have considered is object combining. Object combining combines two objects if they are somehow related. Novel in this thesis is the flexibility of combinations that are allowed: virtually any two objects can be combined as long as a reference to one is stored in the other. Object combining has two effects: it reduces the time spent executing memory management code in the runtime system and it allows combined objects to be sent over the network, which potentially reduces the number of network roundtrip messages.

Object combining, however, introduces a new problem. Combining too many objects or the wrong objects will result in false sharing. In Jackal, falsely shared data is needlessly transferred over the network and needlessly diffed and patched at the home-node. If the compiler combines read-only objects with objects that are actively modified, the amount of data that could have remained cached at a given machine is reduced, increasing the amount of data that must be transferred over the network. The problem Jackal's compiler has to deal with is incomplete information:

- the compiler does not know if an object is shared between threads (escape analysis only makes a conservative guess);
- nor can the compiler always know the length of an array;
- nor can the compiler always estimate the number of objects allocated from a given allocation site, as these may depend on a command line parameter or an input file.

The solution chosen for Jackal is to perform a profiling run before production runs. This results in averaged communication statistics for all objects allocated at a given allocation site. This may not be sufficient information as objects allocated from two different allocation sites may have been sent to different destinations even if the number of object transfers is equal for both sites. Also, the program may consist of several "phases" where each phase introduces a new communication pattern that requires more or less object combining. The profile simply accumulates all sends and receives, disregarding the temporal effects and thus showing a false picture to the compiler.

Instead of employing profile information to guide object combining decisions, a better solution would be to undo object combining at runtime for those objects or allocation sites that have proven to be problematic. However, this possibility has not been explored in this thesis.

Overall, we can reach a number of conclusions about Jackal and Jackal's proposed optimizations.

- Each optimization significantly improves performance in at least some cases.
- Jackal attains 5-35% less speedup than CRL, a programmer-annotated DSM in C, mainly because:

    1. JMM's requires to flush *all* working memory and not just a minimal subset as CRL allows;

       x
    2. Jackal uses a heavy-weight threads package where CRL does not use or support threads. Often a thread-switch is required to switch to a message handler thread;
    3. Jackal's compiler is unable to always determine that an object shared between a number of threads will, at runtime, only be used by *one* thread at a time.

- To achieve the best performance, the proposed optimizations need good heuristics to decide when and when not to apply them.
- Jackal's overall strategy of employing compiler technology to enhance DSM performance instead of programmer aid is promising; however, more work on advanced alias analysis and object aggregation heuristics is needed to attain the performance of CRL.
- Systems like Jackal require very low-latency networks to attain good performance as the compiler cannot yet transform a program sufficiently to perform latency hiding techniques.
- The costs of executing Java synchronization statements remains high. In applications where synchronization is abundant, the incurred overheads impede speedup (Barnes-Hut). A different memory model where mutual exclusion statements are tied to a specific set of objects such as scope consistency would perform better.
- The latency of a single object transfer remains high because the high message transfer latency of the GM package (GM latency is 24 microseconds where other packages over the same network have achieved latencies of 9 microseconds).

Bibliography

[1] DAS project homepage: http://www.cs.vu.nl/das/.

[2] S.P. Amarasinghe, J.M. Anderson, M.S. Lam, and A.W. Lim. An Overview of a Compiler for Scalable Parallel Machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Portland, Oregon, USA, Aug 1993.

[3] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. A High Performance Cluster JVM Presenting A Pure Single System Image. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 168–177, San Francisco, California, USA, June 2000. ACM Press.

[4] H.E. Bal, R.A.F. Bhoedjang, R.F.H. Hofman, C. Jacobs, K.G. Langendoen, T. Rühl, and M.F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, February 1998.

[5] B. Blanchet. Escape Analysis For Object-Oriented Languages: Application To Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 20–34, Denver, Colorado, USA, November 1999. ACM Press.

[6] 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.

[7] H.-J. Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 197–206, Albuquerque, NM, USA, June 1993.

[8] B. Cheung, C. Wang, and K. Hwang. JUMP-DP: A Software DSM System with Low-Latency Communication Support. In *In the 2000 International Workshop on Cluster Computing - Technologies, Environments and Applications (CC-TEA'2000)*, Las Vegas, Nevada, USA, June 2000.

[9] T.M. Chilimbi, B. Davidson, and J.R. Larus. Cache-Conscious Structure Definition. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language*

Design and Implementation (PLDI), pages 13–24, Atlanta, Georgia, USA, 1999. ACM Press.

[10] T.M. Chilimbi, M.D. Hill, and J.R. Larus. Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Atlanta, Georgia, USA, 1999. ACM Press.

[11] J.D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff. Escape Analysis For Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, Denver, Colorado, USA, November 1999. ACM Press.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method Of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, USA, 1989. ACM Press.

[13] R. Cytron, J. Ferrante, B.K. Rosen, M. N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[14] J. Dolby and A. Chien. An Automatic Object Inlining Optimization And Its Evaluation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 345–357, Vancouver, British Columbia, Canada, 2000. ACM Press.

[15] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces, Principles, Patterns, and Practice*. Addison Wesley, 1999.

[16] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Quebec, Canada, 1998. ACM Press.

[17] E. Gagnon, L.J. Hendren, and G. Marceau. Efficient Inference of Static Types for Java Bytecode. In *Static Analysis Symposium*, pages 199–219, Santa Barbara, Calif., USA, 2000.

[18] D. Gay and A. Aiken. Language Support For Regions. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 70–80, Snowbird, Utah, USA, June 2001. ACM Press.

[19] S. Ghemawat, K.H. Randall, and D.J. Scales. Field Analysis: Getting Useful And Low-Cost Interprocedural Information. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 334–344, Vancouver, British Columbia, Canada, 2000. ACM Press.

[20] R. Ghiya and L.J. Hendren. Putting Pointer Analysis To Work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 121–133, San Diego, California, USA, January 1998. ACM Press.

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

[22] D. R. Hanson. Fast Allocation And Deallocation Of Memory Based On Object Lifetimes. *Software – Practice and Experience*, 20(1):5–12, 1990.

[23] W.C. Hsieh, M.F. Kaashoek, and W.E. Weihl. Dynamic Computation Migration in DSM Systems. In *Supercomputing '96*, Pitssburgh, PA, USA, November 1996.

[24] R.J.M. Hughes. A Distributed Garbage Collection Algorithm. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy, France, September 1985. Springer-Verlag.

[25] M.D. Hill I. Schoinas, B. Falsafi and D.A. Wood J.R. Larus. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 40–49, Paris, France, October 1998.

[26] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proceedings of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, Padua, Italy, 1996.

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

[28] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *ACM 1999 Java Grande Conference*, pages 15–24, San Francisco, CA, June 1999.

[29] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pages 213–226, Copper Mountain, Colorado, USA, December 1995. ACM Press.

[30] V. Karamcheti and A.A. Chien. Concert–Efficient Runtime Support For Concurrent Object-Oriented Programming Languages On Stock Hardware. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 598–607, Portland, Oregon, USA, 1993. ACM Press.

[31] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, pages 91–98, Hong Kong, 1996.

[32] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 Usenix Conference*, pages 115–131, San Francisco, CA, USA, January 1994.

[33] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[34] R. Kennedy, S. Chan, Shin-Ming Liu, R. Lo, P. Tu, and F. Chow. Partial Redundancy Elimination in SSA Form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):627–676, 1999.

[35] Robert Kennedy, Fred C. Chow, Peter Dahl, Shin-Ming Liu, Raymond Lo, and Mark Streich. Strength Reduction via SSAPRE. In *Computational Complexity*, pages 144–158, 1998.

[36] D. Kogan and A. Schuster. Collecting Garbage Pages with Reduced Memory and Communication Overhead. In *Proceedings 5th European Symposium on Algorithms*, pages 308–325, Graz, Austria, September 1997.

[37] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[38] P. Laud. Analysis for Object Inlining in Java. In *JOSES: Java Optimization Strategies for Embedded Systems*, Genoa, Italy, April 2001.

[39] P. Launay and J-L. Pazat. The Do! project: Distributed Programming Using Java. In *First UK Workshop Java for High Performance Network Compu ting*, Southampton, Sept. 1998.

[40] J. Maassen, R. Van Nieuwpoort, R. Veldema, H.E. Bal, T. Kielmann, C. Jacobs, and R.F.H. Hofman. Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, November 2001.

[41] M. W. Macbeth, K. A. McGuigan, and Philip J. Hatcher. Executing Java Threads in Parallel in a Distributed-Memory Environment. In *Proceedings of CASCON*, pages 40–54, Missisauga, ON, Canada, 1998.

[42] H. Massalin. Superoptimizer: A Look At The Smallest Program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 22-10, pages 122–127, New York, NY, 1987. ACM Press.

[43] D.C.J. Matthews and T. Le Sergent. LEMMA: A Distributed Shared Memory with Global and Local Garbage Collection. Technical Report ECS-LFCS-95-325, Dept. of Computer Science, University of Edinburgh, Edinburgh, UK, April 1995.

[44] SUN Microsystems. Java Object Serialization Specification, 1996.

[45] S.S. Muchnick. *Advanced Compiler Design And Implementation*. Morgan Kaufmann Publishers Inc. San Francisco, California, 1997.

[46] Y.G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27-7, pages 116–127, New York, NY, June 1992. ACM Press.

[47] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience (CPE)*, 9(11):1225–1242, November 1997.

[48] D. Plainfossé and M. Shapiro. A Survey of Distributed Garbage Collection Techniques. In *International Workshop on Memory Management*, pages 211–250, Kinross, Scotland, UK, September 1995.

[49] W. Pugh. Fixing The Java Memory Model. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 89–98, San Francisco, California, USA, June 1999. ACM Press.

[50] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000.

[51] K. van Reeuwijk, A.J.C. van Gemund, and H.J. Sips. Spar: a programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, August 1997.

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

[53] C. Ruggieri and T.P. Murtagh. Lifetime Analysis of Dynamically Allocated Objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, San Diego, Calif. USA, 1988.

[54] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the seventh international Conference on Architectural support for Programming Languages and Operating Systems*, pages 174–185, Cambridge, Massachusetts, USA, 1996. ACM Press.

[55] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Valle-Rai, P. Lam, E. Gagnon, and C. Godin. Practical Virtual Method Call Resolution For Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 264–280, Minneapolis, Minnesota, USA, 2000. ACM Press.

[56] K. Taura and A. Yonezawa. An Effective Garbage Collection Strategy for Parallel Programming Languages on Large Scale Distributed-Memory Machines. In *6th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 18–21, Las Vegas, NV, June 1997.

[57] D.N. Truong, F. Bodin, and A. Seznec. Improving The Cache Behaviour of Dynamically Allocated Data Structures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 322–329, October 1998.

[58] R. Veldema. Jade, A Recursive Ascend Parser Generator For Augmented Attribute Grammars (http://www2.informatik.uni-erlangen.de/ veldema/jade_docs.ps), June 1998.

[59] R. Veldema, R.A.F. Bhoedjang, and H.E. Bal. Distributed Shared Memory Management For Java. In *4th Annual Conference of the Advanced School for Computing and Imaging*, pages 256–264, Lommel, Belgium, June 2000.

[60] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. Runtime optimizations for a Java DSM implementation. In *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, pages 153–162, Palo Alto, Calif., USA, 2001. ACM Press.

[61] R. Veldema, C. Jacobs, R.F.H. Hofman, and H.E. Bal. Object Combining: A New Aggressive Optimization for Object Intensive Programs. In *Proceedings of the 2002 joint ACM-ISCOPE Conference on Java Grande*, pages 165–174, Seattle WA, USA, 2002. ACM Press.

[62] J. Whaley and M. Rinard. Compositional Pointer And Escape Analysis For Java Programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 187–206, Denver, Colorado, USA, November 1999. ACM Press.

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

[64] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a High-performance Java Dialect. In *ACM 1998 workshop on Java for High-performance network computing*, February 1998.

[65] W. Yu and A. Cox. Conservative Garbage Collection on Distributed Shared Memory Systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 402–410, Hong Kong, May 1996.

[66] 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 Symposium on OSDI*, pages 75–88, Seattle, WA, USA, October 1996.

Samenvatting

Voor het oplossen van werkelijk grote rekenproblemen en het adresseren van zeer grote hoeveelheden data is het gebruik van óf dure, grote computers óf meerdere, in parallel werkende, goedkopere computers vereist. Beide oplossingen hebben hun problemen. Bij het gebruik van één enkele snelle grote computer zijn de kosten vaak extreem hoog maar is het gebruik heel eenvoudig (vooral voor de programmeur). Bij het gebruik van een cluster van kleinere, in parallel werkende computers zijn de kosten lager, maar is de machine lastiger te programmeren doordat er expliciet berichten over het netwerk gestuurd moeten worden om de individuele computers te laten samenwerken.

Om het programmeren van clusters van computers te versimpelen zijn er twee paradigma's. Het eerste is om alle communicatie tussen de individuele computers zelf uit te programmeren en de netwerklaag deels te verstoppen in een functie-bibliotheek. De tweede aanpak is om de hele cluster als één enkele grote computer te beschouwen, door van het programmeren van losse machines te abstraheren. Dit abstraheren kan gebeuren door de introductie van een extra software-laag boven op de communicatie-laag of door de compiler van de applicatie code te laten genereren die geschikt is om op een cluster van losse computers te laten draaien.

In het systeem dat in dit proefschrift voorgesteld wordt, genaamd Jackal, is deze functionaliteit deels in een compiler en deels in een functie-bibliotheek geïmplementeerd. De compiler accepteert een programma dat geschreven is in Java en dat geschikt is om op één machine te draaien met een standaard compiler en met standaard Java bibliotheken. Verder nemen wij aan dat de programmeur zijn applicatie al parallel heeft gemaakt door meerdere threads te gebruiken. Nadat de Jackal-compiler de applicatie heeft gecontroleerd op fouten, genereert hij code die op meerdere individuele machines in parallel kan draaien.

Om in parallel te kunnen werken genereert de Jackal compiler extra code vlak vóór ieder gebruik van een geheugen-element, om te controleren dat het geheugen-element op de huidige machine aanwezig is (een toegankelijkheidstest). Als dit niet het geval is, wordt de meegeleverde functie-bibliotheek gebruikt om het geheugen-element op te halen bij de machine die het geheugen-element op dat moment in bezit heeft. In onze implementatie komt een geheugen-element overeen met óf een enkel Java-object óf een deel van een Java-array.

Bij een naïeve implementatie wordt nu bij ieder gebruik van een niet-aanwezig geheugen-element een bericht over het netwerk gestuurd, wat de efficiëntie niet ten goede komt. Om de efficiëntie te verbeteren, worden in dit proefschrift verscheidene optimalisaties voorgesteld: methoden om het aantal gegenereerde toegankelijkheidstesten te verminderen en methoden om geheugen-elementen te combineren. Door het aantal toegankelijkheidstesten te verminderen, zorgt de compiler er voor dat het programma sneller zal executeren op een enkele machine. Geheugen-elementen worden gecombineerd door te herkennen dat zij aan elkaar 'gelijmd' kunnen worden of deel uit maken van een boomstructuur. Door objecten te combineren gaan er voor een aanvraag voor één van de gecombineerde objecten de andere 'gratis' mee over het netwerk. Hierdoor zullen de gecombineerde objecten al aanwezig zijn voordat zij gebruikt worden. Dezelfde argumentatie is toepasbaar op de boomherkeningstechniek.

Een andere manier waarop het systeem in dit proefschrift de efficiëntie vergroot is door code zoveel mogelijk uit te voeren op de machine die de data bevat waar die code op werkt. Dit wordt bereikt door code waarin threads met elkaar synchroniseren uit te splitsen en te executeren daar waar het object waarop gesynchroniseerd wordt geplaatst is. Verder worden nieuwe threadobjecten daar geallokeerd waar de nieuwe thread ook zal executeren.

Curriculum Vitae

Personal information

Name: Ronald S. Veldema
Date of Birth: 13 July 1972
Nationality: Dutch
Email: veldema@cs.fau.de
Current address:

- Schleifweg 9,
- Uttenreuth-Erlangen, 91080
- Germany

Work experience

- 1997-1998: Teaching assistent for software engineering and assembler practical courses at the Vrije Universiteit, Amsterdam.
- 1998: Masters at the Vrije Universiteit, Amsterdam.
- 1998-2002: PhD-student in the parallel systems group, at the Vrije Universiteit, Amsterdam. Also coordinator for the Compiler Construction practical course.
- 2002-2004: Member of the research staff of the Department of Computer Science 2 of the University of Erlangen-Nurnberg, Germany.

Journal publications

1. J. Maassen, R. van Nieuwpoort, R. Veldema , H.E. Bal, T. Kielmann, C. Jacobs and R. Hofman. Efficient Java RMI for Parallel Programming. In *ACM. Transactions on Programming Languages and Systems*, pages 747 – 775, 2001.

2. R.Veldema, R.F.H. Hofman, R.A.F. Bhoedjang and H.E. Bal. Runtime Optimizations for a Java DSM Implementation. In *ACM: Concurrency: Practice and Experience*, pages 299 – 316, Vol. 15, 2003.

Conference publications

1. R.Veldema, C.Jacobs, R.F.H. Hofman, and H.E. Bal. Object Combining: A New Aggressive Optimization for Object Intensive Programs. In *Proceedings of the 2002 Joint JavaGrande/ISCOPE Conference on JavaGrande*, pages 165 – 174, Seatle WA, USA.

2. 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 *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 50 – 61, Showbird Utah, USA.

3. R. Veldema, R.F.H. Hofman, R.A.F. Bhoedjang and H.E. Bal. Runtime Optimizations for a Java DSM Implementation. In *Proceedings of the 2001 Joint JavaGrande/ISCOPE Conference on JavaGrande*, pages 153 – 162, Palo Alto, Calif., USA.

4. R. Veldema, T. Kielmann, and H.E. Bal. Java-specific Overheads: Java at the Speed of C? In *Workshop: Java in High Performance Computing*, at HPCN Europe 2001 Conference, Amsterdam, The Netherlands, June 25-27, 2001. Published as LNCS Vol. 2110, pages 685 – 692.

5. R.Veldema, R.A.F. Bhoedjang and H.E. Bal. Distributed Shared Memory Management for Java. In *ASCII'2000*, pages 256 – 264, Lommel, Belgium, June 2000.

6. R. van Nieuwpoort, J. Maassen, H.E. Bal, T. Kielmann, and R. Veldema. Wide-Area Parallel Computing in Java. In *Proceedings of the 1999 Joint JavaGrande/ISCOPE Conference on JavaGrande*, pages 8 – 14, Palo Alto, CA, June 1999.

7. J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173 – 182, May 1999, Atlanta, GA.