Improved DSM Efficiency, Flexibility, and Correctness

January 2010

Please cite as:
1 Introduction 5

2 A Proposal for OpenMP for Java 7
   2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
   2.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
   2.3 Differences to the C++ OpenMP Specification . . . . . . . . . . . . . . . 9
      2.3.1 Pragmas and Conditional Compilation . . . . . . . . . . . . . . . 9
      2.3.2 Extensions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   2.4 Specific Aspects of a Java Binding . . . . . . . . . . . . . . . . . . . . . . 12
      2.4.1 Java-OpenMP Memory Model . . . . . . . . . . . . . . . . . . . . 12
      2.4.2 Interaction between Parallel Regions and Java Threads . . . . 13
      2.4.3 Exception Management . . . . . . . . . . . . . . . . . . . . . . . . 14
   2.5 Runtime Environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
      2.5.1 Runtime Support Library . . . . . . . . . . . . . . . . . . . . . . . 14
      2.5.2 Object-pooling Support . . . . . . . . . . . . . . . . . . . . . . . . 15
      2.5.3 Environment Variables . . . . . . . . . . . . . . . . . . . . . . . . 15
   2.6 A Reference Implementation of Java-OpenMP in Jackal . . . . . . . . . . 15
      2.6.1 Performance evaluation of a Java-OpenMP Version of the Lattice-
          Boltzmann Method . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
   2.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3 Reparallelization and Migration 19
   3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
   3.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
   3.3 Reparallelization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
      3.3.1 Repartitioning of work-sharing constructs . . . . . . . . . . . . . . 22
      3.3.2 Loop schedule types . . . . . . . . . . . . . . . . . . . . . . . . . . 24
      3.3.3 Reductions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
      3.3.4 Context for added worker threads . . . . . . . . . . . . . . . . . . 25
      3.3.5 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   3.4 Migration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
      3.4.1 Thread Checkpointing . . . . . . . . . . . . . . . . . . . . . . . . . . 27
      3.4.2 Multi-process checkpointing . . . . . . . . . . . . . . . . . . . . . . 28
      3.4.3 Interaction with reparallelization . . . . . . . . . . . . . . . . . . . 29
   3.5 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
      3.5.1 Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
      3.5.2 Overheads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
      3.5.3 Speedup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.5.4 Migration .................................................. 32
3.6 Conclusion .................................................. 32

4 Heterogeneous Thread Migration ........................................... 35
   4.1 Introduction ................................................. 36
      4.1.1 Issues of heterogeneous thread migration .......... 37
      4.1.2 Issues of migration heuristics .................. 38
      4.1.3 Contributions ......................................... 39
   4.2 Jackal ....................................................... 39
   4.3 Mechanics .................................................. 39
      4.3.1 Example UDS .......................................... 41
      4.3.2 How to use UDS strings ................................ 42
      4.3.3 Record and Restore of thread stacks .......... 42
      4.3.4 Expert level aspects .................................. 43
   4.4 Heuristics ................................................... 46
   4.5 Related Work ................................................ 47
   4.6 Performance ................................................ 48
   4.7 Conclusions and Future Work ......................... 50

5 A Framework for Application Migration ................................... 53
   5.1 Introduction ................................................. 55
   5.2 Design Objectives .......................................... 56
   5.3 OGRE: Open Grid Research Environment .................. 56
      5.3.1 Architecture ........................................... 56
      5.3.2 Application Life Cycle ................................ 57
      5.3.3 Migration Strategy ..................................... 58
      5.3.4 Grid-wide File System ............................... 59
      5.3.5 Application Adapter .................................. 59
   5.4 Performance ................................................... 60
      5.4.1 GridFS .................................................. 60
      5.4.2 Migration ................................................. 61
   5.5 Related Work ................................................ 62
   5.6 Conclusions ................................................... 63

6 Tapir's Multi-core Support ............................................. 65
   6.1 Introduction ................................................ 66
   6.2 Language features .......................................... 67
      6.2.1 Functional language core ............................... 67
      6.2.2 Mutable Arguments are created by Flow expressions 69
      6.2.3 Reduction types .......................................... 71
      6.2.4 Support for OO-style polymorphism ............. 71
      6.2.5 Support for OO-style code reuse .................. 72
      6.2.6 Multi-dimensional arrays with named dimensions 72
   6.3 Implementation .............................................. 73
      6.3.1 Cuda .................................................... 73
      6.3.2 Threaded execution ..................................... 74
      6.3.3 Array Implementer Classes .......................... 75
   6.4 Performance ................................................... 75
      6.4.1 Synthetic micro-benchmarks ....................... 76
      6.4.2 Matrix Multiplication ................................ 76
6.4.3  LBM  ......................................................... 77
6.4.4  Computational network  .................................. 77
6.5  Related work ...................................................... 78
6.6  Conclusions ....................................................... 78

7  RDMA based DSM Protocols  ..................................... 79
7.1  Introduction ....................................................... 80
7.2  Related work ...................................................... 81
7.3  DSM protocol template ........................................... 82
7.4  Object requests by means of RDMA  ................................ 83
7.5  Processing the list(s) of cached objects  .......................... 85
7.5.1  Invalidation protocol handlers ................................ 85
7.5.2  RDMA-based update protocol handlers  ....................... 85
7.6  Performance ......................................................... 89
7.7  Conclusions ......................................................... 92

8  Tapir’s Model-checking Support  .................................. 93
8.1  Introduction ......................................................... 94
8.2  Self-consistency ..................................................... 96
8.3  Aspect oriented testing ........................................... 97
8.4  Application specific code transformations to reduce the state space  99
8.4.1  Transformation rules ............................................. 99
8.4.2  Compile-time predicates ........................................ 100
8.4.3  Propagation rules ................................................ 100
8.5  Application specific black boxing to reduce the state space  ......... 101
8.5.1  Locks, condition variables, and compile-time asserts  .......... 102
8.5.2  Discussion ......................................................... 103
8.6  Performance ......................................................... 103
8.7  Related Work ....................................................... 104
8.8  Conclusion ......................................................... 105

9  Huge Address Space VM on a Cluster  ............................ 107
9.1  Introduction ......................................................... 108
9.2  LVM Implementation .............................................. 109
9.2.1  Implementing the Address Space  ............................. 109
9.2.2  DSM support .................................................... 112
9.2.3  Object Allocation Strategies ................................... 113
9.2.4  Reducing Thread Migrations ................................... 114
9.2.5  Distributed Garbage Collection ................................ 115
9.3  Performance ......................................................... 116
9.3.1  Micro Benchmarks .............................................. 116
9.3.2  Application Benchmarks ........................................ 118
9.4  Related Work ....................................................... 119

10  A memory conserving DSM Protocol  ............................ 123
10.1  Introduction ....................................................... 124
10.2  Related Work ..................................................... 124
10.3  LVM .............................................................. 125
10.4  Replication Protocol .............................................. 126
10.4.1  Replication Protocol Implementation  ......................... 127
Chapter 1

Introduction

This Habilitation consists of a collection of related published papers by and with the author. Each of the following chapters is one paper included effectively verbatim with only the layouts unified. The various bibliographies are combined into a single bibliography to avoid text duplication and running page numbers are added across all papers. Each paper is preceded with the relevant publishing information for that paper.

The common theme of the included papers is how to create Distributed Shared Memory (DSM) systems that are as flexible as possible and still have some idea of correctness. Fortunately, many flexibility enhancements/optimizations are transparent to other optimizations/enhancements so that testing can be performed on a per-enhancement basis.

The first few papers center on making DSM systems more flexible and efficient by allowing running DSM programs to move around a Grid (a cluster of clusters). Because changing the number of threads in a running program is hard to adjust automatically, we require the help of the programmer. We do this by requiring the programmer to use JaMP, our adaptation of OpenMP for Java (Chapter 2). A programmer then only needs to program (1) in Java and (2) use OpenMP and can then deploy his program on the Grid. All other details are handled by the DSM layer and the cluster abstraction layers. The reparallelization itself is described in 3.

Because the Grid contains machines of different architectures, we created a new checkpointing mechanism that can cope with different word-sizes, stack-layouts etc. This mechanism can also be used to efficiently move threads around a cluster (thread-migration). This is explained in Chapter 4.

To ensure location independence in the Grid, we created a new Grid-middleware (OGRE) that provides a grid-wide file-system and performance monitoring facilities. OGRE is also responsible for the heuristics used to decide which cluster to move a running DSM application to (Chapter 5).

The above makes for very flexible DSM systems, however, at the cost of great system complexity. The question then is, can we ensure that the protocols used internally to provide the DSM’s shared memory illusion are still correct? This is the topic of the last batch of papers. For example, in the Jackal DSM system [121], the runtime system (excluding garbage collectors etc) of the DSM layer consists of 50,330 lines of C code (including comments, etc). Because C code is hard to analyze automatically (due to function pointers, pointer arithmetic, etc), we create a new general purpose language to program in.

This new programming language (Tapir) is a general purpose language suited for programming arbitrary programs in but contains features that make automatic program
CHAPTER 1. INTRODUCTION

analysis easier. Tapir takes features from Java, Haskell, C, and C++ and tries to leave out any features that may slow down program execution and program analysis. For example, we leave out standard object orientation, pointer arithmetic and function pointers but keep features such as bounds checking and classes and supplant removed features with alternatives.

One could argue that leaving out all these features may impede performance. We counter that by creating a Tapir compiler backend that can execute loops on graphics cards and can do auto-parallelization of functional code. The base Tapir language and its performance on a number of benchmarks is described in Chapter 6. We can thus be assured that a DSM protocol written in Tapir will be efficient and potentially more efficient than a hand-written one due to Tapir’s automatic parallelization.

Because it is hard to see if the language is complete enough to model RDMA based communications, we extended Jackal’s DSM protocol with RDMA based protocols. The resulting protocols are described in Chapter 7. Measurements show that object-fetch performance can be increased significantly using RDMA.

However, creating a DSM protocol that is bug-free is far from trivial, especially if said protocol intermixes RDMA based communications and normal message sends and allows multiple threads per machine. Automatically testing the correctness of Tapir programs (wherein DSM protocols are then to be written in) is currently via model-checking. Model-checking of concurrent programs is implemented by trying all interleavings of executions of threads and processes. Because this is an exponential problem (known as the state explosion problem), this requires a lot of memory to keep all partially executed states in memory to allow testing for duplicate states.

The first step we did was to create a model-checker for Tapir programs. This model-checker is written in Java and is partially generated from a given Tapir program by the Tapir compiler. The next problem we then face is that a single JVM will not suffice as it will quickly run out of memory and then rely on the underlying operating system’s swapping mechanisms. To reduce the state space and aid in defining tests, the Tapir language contains a number of features. These features are described in Chapter 8.

For this reason, we create a new distributed VM for Java, called LVM (Large VM). This VM is essentially a DSM for Java again but then written such that it is memory efficient by managing the memory of all machines in the cluster and implementing its own swapping mechanisms to circumvent the slow operating system’s swapping mechanisms. Unlike normal DSMs, it does not cache remove memory upon detecting non-local memory access, but rather moves the executing thread to the remote memory and does so very efficiently. Again this saves us the memory required to cache remote memory ranges. This base LVM implementation is described in Chapter 9.

Because the base DSM protocol used in LVM may perform badly on non-optimized programs, we extended the DSM protocol with a limited caching protocol. It selectively caches objects and arrays and only up-to a given memory limit. Unfortunately machine-local caching of objects and thread-migration to access remote memory for non-cacheable data-structures conflict requiring the cache of machine-local object to logically migrate upon thread-migrations which can be expensive. We explore a solution in Chapter 10.

Using LVM and a model-checker generated from a Tapir program, larger programs can now be model-checked, including simple DSM protocols written in Tapir.
Chapter 2

A Proposal for OpenMP for Java

Authors:
Michael Klemm, Ronald Veldema, Matthias Bezold, Michael Philippsen
University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3 • 91058 Erlangen • Germany
klemm@cs.fau.de, veldema@cs.fau.de, mail@msbezold.de, philippsen@cs.fau.de

Published in:
Proceedings of the OpenMP Shared Memory Parallel Programming (International Workshops IWOMP 2005 and IWOMP 2006)
Conference: International Workshop on OpenMP (IWOMP) ’06
Reims, France
June 12 – 15, 2006
Pages 409 – 421.
ISBN 3-540-68554-5
Publisher: Springer, Berlin, Germany.

Currently, there is no OpenMP definition for Java as there is for C/C++ and Fortran. However, we need Java to implement our DSM because it offers more implementation flexibility (no pointer arithmetic, function pointers) and we need OpenMP because Java supplies only unstructured parallelism (parallelism not bound to lexical scopes). We need this structuring later on to allow re-structuring of parallelism.
CHAPTER 2. A PROPOSAL FOR OPENMP FOR JAVA

Abstract

The current OpenMP 2.5 specification does not include a binding for Java. However, Java is a wide-spread programming language that is even used for HPC programming. We propose an adaptation of OpenMP to Java by retrofitting the basic OpenMP directives to Java and further propose some new concepts to make OpenMP fit into Java’s language philosophy.

We discuss how Java’s memory model matches OpenMP’s memory model and how the OpenMP bindings for Java and C++ differ. We also suggest how to achieve flexibility of an OpenMP implementation by allowing both Java threads (java.lang.Thread) and Java tasks (java.util.concurrent.FutureTask) as an underlying means of parallelization.

Support for object-orientation is added to allow OpenMP to better suit the Java programming model. For example, we suggest a parallel for-each loop over Java collections, OO-based reductions, and object-cloning semantics to adapt data-sharing clauses to Java. Also, we suggest a minimal runtime library to allow object-pooling to circumvent any implicit synchronization involved in object allocations.

Finally, we present some performance numbers for a reference implementation in a research compiler.

2.1 Introduction

Java becomes more and more pervasive in the programming landscape with numerous high-performance (and free) implementations (Sun’s and IBM’s JVMs, Jackal [121], etc.). For many applications Java’s performance is on-par with other programming languages (including C++ and Fortran) [11, 63]. Hence, Java becomes suitable for high-performance computing as well. Some advantages of Java over, for example, C/C++ and Fortran are its higher productivity, its safety features, and its large standard library. Finally, because of its high level of pervasiveness, many prospective HPC programmers are already experienced in Java but have no thorough knowledge of C/C++ or Fortran.

Java currently has two ways to allow parallel execution of programs. First, the Remote Method Invocation (RMI) facility allows distributed execution of a program that spans several nodes. Second, shared-memory parallelism is achieved by means of the Java Threading API. We argue that using Java’s threading model directly results in two unforeseen problems. First, it is a cumbersome and error-prone task to transform a sequential program into a parallel one by manually creating threads and implementing explicit data exchange by hand. Second, compiler analysis is made exponentially harder as the parallelism is hidden from it. For example, static data-race detection tools (for Java) such as ESC/Java [52] currently have to resort to complex model checking techniques to search for all possible interleavings of object usages (and thread creations). An OpenMP-style programming allows for easier static data-race detection.

With Java 1.5, the java.util.concurrent package provides more support for multi-threaded programming, i.e. it contains a large set of thread-aware collections and threading support mechanisms. For example, it contains code for forms of light-weight threads (tasks) as well as scalable thread-safe blocking queues and concurrent hashtables. The collections from java.util.concurrent are universally useful and applicable to both standard Java and the proposal of this paper, Java-OpenMP. However, the new task subsystem enlarges the landscape of expressible parallelism. With Java-OpenMP it is

---

1Java performance often depends on the programming style used.
2.2. RELATED WORK

easier to choose, even automatically, from the available types of parallelisms.

Finally, because Java has no support for fast thread-local storage, OpenMP’s data-sharing clauses (shared, private, etc.) fulfill a need in the Java environment. Java only allows one to simulate true thread-local storage by putting variables into the thread class, a reference to which can be acquired by calling the currentThread() method at any time.

Combining the above observations we propose to adapt the C++ OpenMP specification to fit Java and to retrofit any Java features onto it. We present the basic set of OpenMP directives and their suggested semantics under the proposed Java-OpenMP model. We further propose a series of extensions to the basic model so that OpenMP is a better match to Java’s object-oriented programming model.

2.2 Related Work

Various commercial and production quality compilers for C/C++ and Fortran, e.g. the Intel compiler suite [65] or the compilers of the Portland Group [66], can compile OpenMP programs to native code.

With OdinMP/CCp [21] for C/C++ and Omni OpenMP [104] as well as OpenUH [83] for both C/C++ and Fortran there also are open-source implementations of the OpenMP specification. These are source-to-source compilers that transform OpenMP programs to equivalent programs that make use of a native threading API. Source-to-source compilers rely on another compiler to generate the executable code. Only for the Itanium architecture, OpenUH uses the Open64 [64] compiler back-end to directly emit a native executable.

At present, we are not aware of any standardized Java binding of the OpenMP specification. An earlier paper on the JOMP project [23] dealt with a subset of OpenMP for Java. The JOMP compiler follows the source-to-source approach and transforms a JOMP program into an equivalent Java program, which uses the Java Threading API for parallelism. In JOMP (as in our proposal), special Java line comments are treated as OpenMP directives. The proposal we present here follows the OpenMP specification as closely and as completely as possible and suggests new features that allow a tight integration of OpenMP into the Java language.

2.3 Differences to the C++ OpenMP Specification

The semantics of most of the OpenMP features can be applied from the C/C++ OpenMP standard without any changes. This includes: //omp for, //omp section, //omp sections, //omp single, //omp master, //omp barrier, //omp flush, and others. Whereas we discuss only the necessary differences between Java-OpenMP and C++-OpenMP, we reason where we feel that Java’s design philosophy is better served by adapting the C++-OpenMP definitions.

2.3.1 Pragmas and Conditional Compilation

Although pragmas and conditional compilation are essential elements of the OpenMP standard, Java’s language specification does neither provide the pragma concept nor conditional compilation. Although one might use a C-style pre-processor on Java source code, such scenarios are discouraged as the mapping between the unprocessed code and the pre-processed code is potentially lost.
package omp;
interface CustomCloner<T> {
    T clone(T other);
}

Figure 2.1: Interface for creating custom copies.

Object a = ...;
int b = ...;
CustomCloner cloner = ...;
//#omp ... firstprivate(a:cloner, b)

Figure 2.2: Example for custom object copying.

With Java 1.5 the concept of annotations was added as an alternative. Annotations are type-safe meta-statements that the Java compiler, the JVM, and other tools are able to recognize. For example, annotations can be used to describe RMI stubs and skeletons so that they can be automatically created. Unfortunately, annotations cannot be used for Java-OpenMP for several reasons. First, they cannot be used at the level of Java statements or blocks, but only at the level of declarations (e.g. at class level, method level, etc.). Second, an annotation must always have a well-defined annotation type that in turn must be defined in some Java package visible to the Java compiler. Thus, a non-OpenMP compiler would have to provide dummy annotations of the same annotation types as an OpenMP-compliant compiler does. This is clearly not acceptable.

The only reasonable way to implement pragmas in an OpenMP binding for Java is not to modify the language itself but instead to add a special kind of Java line comment for OpenMP directives:

// #omp <directive-name> <clauses> newline

A non-OpenMP compiler treats such a directive as a regular Java comment and ignores it. The OpenMP-compliant compiler recognizes the line comment as an OpenMP directive. This approach has also been used by the JOMP compiler [23].

Similarly, we propose an adoption of the conditional compilation sentinels that are introduced by the OpenMP specification for Fortran. A Java line comment starting with //# is recognized as a conditional compilation sentinel. Again, the non-OpenMP compiler safely ignores the rest of the line while the Java-OpenMP compiler parses and compiles the statement as desired.

2.3.2 Extensions

This section proposes how OpenMP’s constructs can be applied or extended to an object-oriented context.

Data-sharing Attributes

The argument passing semantics used in the C++ binding for the data-sharing clauses are inappropriate for a Java binding. In the C++ binding the behavior depends on the data type. For example, if a shared x is a pointer to an object, the pointer is manipulated. But if x is of an object type, the copy constructor of the object is invoked. Hence, the C++ semantics of a firstprivate pointer would imply that only the reference is privatized whereas the object is still shared among the individual threads. A firstprivate object would result in a private copy.

Since Java does not have the concept of a copy constructor as C++ does and since it only has references to objects a programmer cannot choose whether an object is to be

\[\text{A Java reference is similar to a C++ pointer but pointer arithmetic is not allowed.}\]
2.3. DIFFERENCES TO THE C++ OPENMP SPECIFICATION

```java
LinkedLIst c = new LinkedLIst();
c.add("this");
c.add("is");
c.add("a");
c.add("demo");

///omp parallel iterator
for (String s : c)
    System.out.println("s");
```

Figure 2.3: Parallel iteration over a Java collection.

```java
package omp;
interface Reducer<T> {
    T reduce(T a, T b);
}
```

Figure 2.4: Interface for object-oriented reductions.

copied upon privatization or not.

The semantics of OpenMP’s data-sharing clauses have to be adapted to overcome
this limitation of Java. We propose the following solution: in case of a *shared* object
reference, the reference is copied, whereas *firstprivate* and *lastprivate* always create
copies of the object by means of Java’s `Object.clone()` method.

Moreover, an additional parameter may be added to the *firstprivate* and *lastprivate*
clauses to be able to specify a custom behavior for copying objects. That way, one
can for example specify a deep clone instead of a shallow one or use objects of classes
which do not implement the `Cloneable` interface. We introduce a special interface
(`CustomCloner`, see Fig. 2.1) which offers a method `clone` to clone an object in a
customized manner. Objects of this type can be placed right after each variable in the
*firstprivate* and *lastprivate* clauses, separated by a colon (see Fig. 2.2).

**Parallelization of Iterations over Collections**

Most Java programs make heavy use of Java’s collection API and for-each loops. For
example, a molecular dynamics application written in Java will probably be programmed
with collections of molecule objects rather than flat arrays of doubles. Without any ex-
tensions of OpenMP such Java programs cannot be parallelized by means of OpenMP’s
work-sharing clauses.

We therefore propose the constructs `#omp iterator` and `#omp parallel iterator`
(similar to `#omp for` and `#omp parallel for`) to allow iterations over container objects to
be parallelized (see Fig. 2.3). When these constructs are used, the iterator which is purely
sequential in regular Java is replaced by a parallel execution of the loop. Conceptually,
during compilation, the code of Fig. 2.3 is transformed by first calling `c.toArray()`, which
flattens the collection into an array. The loop itself is then rewritten into a traditional
for loop with an ordinary loop counter instead of the iterator. The resulting loop is then
parallelized by the `for` work-sharing construct. The same clauses as for the `for` construct
are allowed. Like ‘for’, ‘iterator’ has an additional convenience form called ‘parallel
iterator’, which is then transformed into a parallel region containing an ‘iterator’ region.

In the proposed form, the `iterator` construct is a specialization of the more generic
taskq construct that the Intel OpenMP compiler already provides [3]. With taskq the
programmer introduces a program region that encloses a set of task environments. Upon
entrance into the taskq region, the encountering thread appends each task region to a
work queue while it executes the enclosing structured block. All other worker threads
then dequeue tasks one by one and process the code contained in the task region. If the
enclosing taskq contains a loop, the loop is executed sequentially by the encountering
class BigIntegerReducer implements omp.Reducer<BigInteger> {
    public BigInteger reduce(BigInteger a, BigInteger b) {
        return a.add(b);
    }
}

BigInteger value = ...;
BigIntegerReducer reducer = new BigIntegerReducer();
// #omp parallel for reduction(reducer: value)
|
|...
|
|
Figure 2.5: Example of object-oriented reductions.

Because C and C++ lack a standardized iterator interface, a C/C++ binding has to provide such a generic construct to allow to parallelize iterator loops over collections. For example, in C++ iterators are mostly expressed by classes that overload the ‘++’ and ‘==’ operators. In contrast, Java provides for-each loops and the standardized Iterator interface. Hence, the proposed iterator construct offers sufficient expressiveness and is a straightforward extension of concepts known to Java programmers.

Object-Oriented Reductions
Currently, OpenMP only defines reductions for primitive data types and for certain operators. However, in Java’s object-oriented design philosophy most data is expressed as objects for which no reduction is allowed. Current practice forces the programmer to explicitly implement reduction algorithms.

We therefore propose an interface (see Fig. 2.4) that lets the programmer use objects in a reduction clause. The interface Reducer offers a method reduce, which reduces two objects. That way, the class definition of the objects, which should be reduced, does not need to be changed or enhanced and existing classes can be used by creating a new Reducer implementation. Different kinds of reductions can be implemented for every class by creating different classes implementing the Reducer interface. Fig. 2.5 shows an example of an object-oriented reduction, which defines a custom reduction for the BigInteger class from the java.math package.

2.4 Specific Aspects of a Java Binding

This section discusses the most important aspects of a Java-OpenMP binding. A potential OpenMP Java binding needs to respect the semantics and restrictions that are imposed by the Java Language Specification [57].

2.4.1 Java-OpenMP Memory Model

Java’s memory model [86] and OpenMP’s memory model [93] [59] are conceptually very close. Both Java’s and OpenMP’s memory models propose a relaxed-consistency model that supports the idea of a global memory where individual threads can temporarily fetch data from and then place it into a thread-local cache. At specific points in a thread’s life-time, the thread flushes its cache by copying modified data back into the global memory.
2.4. SPECIFIC ASPECTS OF A JAVA BINDING

OpenMP specifies that this ‘flush’ operation occurs when a flush or barrier construct is reached. Roughly, Java’s memory model specifies that the same occurs when a synchronization statement is entered or left. It is therefore only natural to reuse Java’s flushing behavior to implement the Java-OpenMP’s flush construct. For a naive implementation it would be sufficient to use an empty synchronized block—such as synchronized(this) {}—for any flush operation.

2.4.2 Interaction between Parallel Regions and Java Threads

Whereas C and C++ do not offer language constructs for multi-threading, Java’s language specification provides explicit threading support. Thus, a Java-OpenMP specification needs to take the coexistence of its parallel regions and the standard system thread package into account. It needs to state what Java-thread functionality is available to the programmer and what is disallowed. Limitations to the programmer need to be independent of the way parallel regions are mapped by the compiler. For example, it is possible to map a Java-OpenMP parallel region either to a Java-thread or to a ‘task’ from the java.util.concurrent package (from here on called Java-task). Java-threads are fully preemptive threads which are scheduled by the JVM while Java-tasks can be used as user-level cooperative threads. Java-tasks are then multiplexed upon a number of Java threads. The Java-OpenMP specification must therefore define sufficient (but not too severe) restrictions on the usage of Java’s threading features so that no harmful interactions with the transformation of parallel regions may result. The following list of restrictions serves this purpose. Please note that these restrictions also hold when java.util.concurrent is used for manual parallelization of a Java program. Thus, the restrictions are no artificial limitations that stem solely from the OpenMP Java binding. However, for Java-OpenMP the restrictions only apply within parallel regions. Outside, i.e. in regular Java code, all of Java’s features may of course still be used as usual.

- The programmer cannot use Thread.currentThread() to differentiate between regular Java threads and the threads that execute a parallel region. For a Java light-weight task Thread.currentThread() will report incorrect values as the light-weight tasks are multiplexed over a set of regular Java threads. Thread.currentThread() is mostly used to implement some form of thread-local storage. This is no longer necessary when OpenMP’s data-sharing primitives are available.

- The programmer may not use synchronized() for synchronization in parallel regions. Instead the various OpenMP thread synchronization constructs must be used. The reason for this is again the flexibility of the Java-OpenMP mapping to different threading models. If a parallel region is executed by tasks that are multiplexed onto several Java threads, synchronized() will not behave as desired.

- The programmer may not use Object.wait() because it may cause deadlocks of the threads and tasks used internally to implement parallel regions. For example, consider the case where two Java-tasks are multiplexed upon one Java-thread. A wait() inside one Java-task will block the entire Java-thread and therefore also the co-scheduled other Java-task(s).

- However, Object.notify() may be used as it is guaranteed to not block. This allows a parallel region to notify threads spawned by non-OpenMP parts of the application. For example, in an application where data is visualized by means of the Java Swing toolkit, even from within a Java-OpenMP parallel section, the application may need to notify the GUI threads that new data is ready for visualization.
2.4.3 Exception Management

The OpenMP specification disallows to prematurely leave a parallel region. This includes jumping out of a region using `break` statements or the like. The same applies to exceptions thrown inside a parallel region. For the C++ specification exception handling is not an issue because exceptions rarely occur in most C++ programs.

In contrast, Java makes heavy use of exceptions. Exceptions are not only visible at the programming level, but are also used for handling system events and errors (e.g. `NullPointerException`, `ArrayIndexOutOfBoundsException`, etc.). These system-level events may be thrown by many of the byte-code instructions of a JVM. Hence, every OpenMP-conforming implementation for Java needs to take special care when dealing with exceptions that are thrown inside parallel regions.

Unfortunately, it is not obvious how to appropriately react to an exceptional event. First, it is not an option to terminate the exception-throwing thread as it will no longer reach subsequent barriers, thus resulting in a deadlock of other threads in the team. Second, it is not permitted to prematurely proceed to the next barrier as the reason of the exception might still be affecting the program, again resulting in undefined behavior.

The desired behavior is that other threads of a team should abort as soon as possible if one thread has encountered an exception. For performance reasons, it is not acceptable that threads continuously actively poll for exceptions that might have occurred on other threads. For this purpose we propose the following semantics:

- If one thread throws an exception, it sets a cancellation flag for all other threads, registers its exception at the thread team, and then proceeds to terminate.
- All (other) threads check for the occurrence of an exception at cancellation points. Cancellation points are the start and the end of `#omp barrier`, `#omp critical`, `#omp parallel`, and work-sharing clauses.
- When a thread reaches a cancellation point and finds that another thread has requested cancellation, the thread itself is interrupted by an exception. This ensures that the thread’s stack frames are unwound and execution of the thread is terminated.
- The exception is re-thrown by the master thread after all threads have terminated execution of the parallel region. If more than one exception was thrown, the master thread randomly selects an exception and re-throws it.

Similar semantics have also been proposed for the JCilk [38] parallel programming language. Although JCilk does not define cancellation points, it allows to terminate asynchronous computations. Such computations are aborted as soon as possible whenever a exception occurs therein. If multiple exceptions are thrown in asynchronous computations, then JCilk randomly selects one to be handled by the corresponding `catch` block.

2.5 Runtime Environment

2.5.1 Runtime Support Library

The runtime library functions as proposed by the OpenMP specification can directly be adopted to Java-OpenMP. Java, however, does not support globally scoped functions as C++ does. Hence, the runtime library is to be implemented as a set of static methods.
2.6. A REFERENCE IMPLEMENTATION OF JAVA-OPENMP IN JACKAL

```java
package omp;
public class Omp {
    // runtime support functions
    public static int getThreadNum() { ... }
    public static int getNumThreads() { ... }
    // object pooling functions
    public static Object objectPoolGet(Class cls) { ... }
    public static Object objectPoolGet(Class cls, int elts) { ... }
    public static void objectPoolPut(Object obj) { ... }
}
```

Figure 2.6: Java runtime support functions and additional functions for object pooling.

in a class called `omp.Omp` (see Fig. 2.6). With Java 1.5, the qualified class name can be omitted as the Java-OpenMP compiler should automatically add a static import statement during compilation.

2.5.2 Object-pooling Support

Because of Java’s highly object-oriented programming style, most Java programs allocate lots of objects. To create arrays and objects, the `new` operator accesses the global heap. Many JVM implementations are synchronizing this access to guarantee that only one thread can perform a `new` operation or can garbage collect at the same time. Hence, Java programs that allocate lots of objects are less parallel than desired. An obvious solution to this JVM limitation is the usage of a per-thread object-pool.

This proposal does not restrict OpenMP’s implementation to Java threads but also, for example, allows Java tasks for implementing parallel regions (see Section 2.4.2). However, to create an efficient thread-pool, and to use the correct and most efficient form of thread-local data, the thread-pool implementation needs to know if a given Java-OpenMP implementation is based on Java-threads or Java-tasks. Hence, Java-OpenMP must provide an API (see Fig. 2.6) that supplies the programmer with an efficient object-pool abstraction.

The `objectPoolPut` method allows a programmer to store an object into the pool for later retrieval. If `objectPoolGet` is invoked and no object of the specified type is available, a new object is allocated from the global heap. The `objectPoolGet` method with two parameters is used to create array objects of a specific size.

2.5.3 Environment Variables

OpenMP 2.5 specifies a set of shell-level environment variables that can be used to control program behavior at runtime. We propose that these variables should be made available to an application by means of Java’s properties mechanism since this is the preferred way of passing environment settings to a Java application.

2.6 A Reference Implementation of Java-OpenMP in Jackal

Jackal [121] is a high-performance implementation of Java that provides a Distributed Shared Memory (DSM) both on Shared-Memory Processor (SMP) machines and clusters with fast interconnects.
CHAPTER 2. A PROPOSAL FOR OPENMP FOR JAVA

Figure 2.7: Speed-up and parallel efficiency of the Lattice-Boltzmann Method on upto 8 CPUs of the AMD cluster using the DSM of Jackal.

Our Java-OpenMP implementation in Jackal is in part compiler-based and in part Java-based. The compiler detects the directives and transforms them into an internal representation. This representation is carried right into the backend for optimization purposes so that Java-OpenMP-level analyses and optimizations can be performed there. The reference implementation is thus not implemented as a pre-processor but rather as part of the compiler. A compiler-based implementation has a number of advantages over a pre-processor. First, the code generated by a pre-processor needs to obey the rules of the target language which restricts the possible transformations that can be applied. Second, code reuse is increased as other compiler front-ends can share the same compiler intermediate code level infrastructure.

The Java-part of the reference implementation contains the runtime support functions specified by Java-OpenMP. It also contains additional support for data-sharing, management of the thread teams, and exception handling.

Our current Java-OpenMP infrastructure implements the basic OpenMP directives as discussed in Section 2.3 except for the potential extensions to the data-sharing and reduction clauses as outlined in Section 2.3.2.

2.6.1 Performance evaluation of a Java-OpenMP Version of the Lattice-Boltzmann Method

The Lattice-Boltzmann Method (LBM) [127] is used to simulate fluids using cellular automata. Space and time are discretized and normalized. In our case, LBM operates on a 2D domain divided into cells. Each cell holds a finite number of states called distribution functions. In one time step the whole set of states is updated synchronously by deterministic, uniform update rules. The evolution of the state of a given cell depends only on its neighboring cells.

The kernel was parallelized in a straightforward manner by placing Java OpenMP comments in the Java source code. The domain is decomposed along the y-axis, that is, the outermost loop is distributed over the worker threads. A similar scheme is used for the MPI parallelized version in [97]. Fig. 2.7 shows the speed-up and the parallel efficiency achieved by both the Java-OpenMP parallelized LBM and the MPI parallelized LBM.

For 8 nodes, Java-OpenMP achieves a speed-up of about 6.1. This is about 83 % of the speed-up achieved by the manually optimized LBM kernel that is written in C using
MPI for communication. The lower scalability is due to the DSM environment that strongly depends on the communication layer (in our case Ethernet). Jackal currently does not include message aggregation. Instead it transfers each cell individually whereas the MPI version transfers whole partitions, i.e. each process requests all the cells from neighboring processes with one MPI request.

2.7 Summary

In this paper, we have sketched a number of extensions and adaptations to the OpenMP C++ specification that are necessary to adapt it to Java’s object-oriented programming model. A syntax for OpenMP directives was proposed that allows a non-OpenMP compiler to safely ignore OpenMP directives. We have shown differences of Java and C++ that require changes to the OpenMP specification. For example, OpenMP’s data-sharing clauses need to be adapted to Java’s notion of object references. Parallel iteration over the items of a Java collection requires a different syntax and semantics. We discussed the Java and OpenMP memory model and described how OpenMP parallel regions can interact with regular Java threads. We further proposed semantics for managing exceptions in the context of parallel regions. Runtime support is provided by means of a class that not only contains the functions that are specified by the OpenMP standard, but in addition offers object pooling. A reference implementation of the Java-OpenMP support in the Jackal DSM system demonstrates Java-OpenMP’s viability.

Future Work

First, the performance evaluation of the Java-OpenMP prototype should be based upon standardized and well-known benchmarks. A Java-OpenMP version of the NAS Parallel Benchmarks (NPB) [13, 54] is in the works. The benchmark suite has to be backported from multi-threaded Java to sequential Java. Then, the OpenMP pragmas have to be inserted to re-parallelize the individual NPB benchmarks. Afterwards, the benchmarks suite is to be evaluated with large-scale clusters and SMPs.

Second, as soon as the upcoming OpenMP 3.0 draft is committed as a standard, the current prototype implementation has to be extended to follow the OpenMP 3.0 standard. If the upcoming standard also contains specifications of exception handling, user-level threads, and parallel execution of iterator loops, the new semantics have to be retrofitted onto the Java programming language as well.

Finally, we plan to port the Java-OpenMP binding into the Jikes RVM [9]. With the port to the Jikes RVM, we will show that a Java-OpenMP binding is not only feasible for our DSM environment but also usable on an SMP architecture.
Most cluster admins cap the maximum run time of a program on a cluster to, say one day. For many applications this is far from enough. We make DSM systems more flexible as described in this paper by allowing a program to move to another cluster to allow longer program runs. However, we must allow the case where the destination cluster is smaller/larger than the originating cluster. This paper investigates restructuring the parallelism in a program to allow this.
CHAPTER 3. REPARALLELIZATION AND MIGRATION

Abstract

Typical computational grid users target only a single cluster and have to estimate the runtime of their jobs. Job schedulers prefer short-running jobs to maintain a high system utilization. If the user underestimates the runtime, premature termination causes computation loss; overestimation is penalized by long queue times. As a solution, we present an automatic reparallelization and migration of OpenMP applications. A reparallelization is dynamically computed for an OpenMP work distribution when the number of CPUs changes. The application can be migrated between clusters when an allocated time slice is exceeded. Migration is based on a coordinated, heterogeneous checkpointing algorithm. Both reparallelization and migration enable the user to freely use computing time at more than a single point of the grid. Our demo applications successfully adapt to the changed CPU setting and smoothly migrate between, for example, clusters in Erlangen, Germany, and Amsterdam, the Netherlands, that use different processors. Benchmarks show that reparallelization and migration impose average overheads of about 4% and 2%.

3.1 Introduction

While offering novel computing opportunities, the boundaries between individual clusters of a computational grid are still visible to users. In addition to heterogeneity as a problem, the user is faced with a cluster’s job scheduling mechanism that assigns computing resources to jobs. Usually, the scheduler prefers short-running over long-running jobs and it prefers jobs that only need a small number of CPUs over more demanding ones. Short jobs with a only a few CPUs increase the cluster’s utilization, while long-running jobs or jobs that require many CPUs often cause unproductive reservation holes [131]. To be fair to waiting users the job manager terminates jobs that exceed their claimed resource limit. A terminated application loses all computed work. However, it is difficult to provide an exact estimation of a job’s runtime, as often runtime depends on the input and is influenced by unpredictable environmental issues (e.g., the load of the network, which in turn depends on a cluster’s overall load). Generally, two “solutions” exist for a user. First, a user can request a too long time slice and accept the penalty of more waiting time until the job runs. As an educated guess, a user might double the estimated time to avoid losing results upon termination of the program. Second, the program is rewritten into a number of smaller phases, which can then run within more predictable time boundaries.

Reparallelization and migration are not only a third solution to this problem but in addition they blur the boundaries between clusters. The user can start the application with a certain number of CPUs and a small time estimate at any cluster of the grid. If the application is about to exceed the time slice or if a more powerful cluster becomes available, the application checkpoints and migrates to another cluster, transparently adapting to the potentially different architecture and to a changed degree of parallelism.

Our solution consists of two parts: (1) OpenMP [92] programs can transparently alter the number of threads during the execution of a parallel region, and (2) checkpoint-based migration between clusters is supported which enables an application to halt on one cluster and to resume on another one. Both the origin and the target can be of different architectures. Reparallelization is crucial for migration, since the number of available CPUs is likely to change. And even without migration, reparallelization allows the next local resource reservation to request fewer or more CPUs depending
3.2. RELATED WORK

on the overall system load and queuing times. Our prototype is implemented on top of the Software Distributed Shared Memory (S-DSM) Jackal [120], a shared memory emulation for Java on clusters. Jackal’s compiler supports JaMP [74], an OpenMP 2.5 port to Java.

The paper is organized as follows. Section 3.2 covers related work. Section 3.3 describes the OpenMP reparallelization. Section 3.4 discusses the migration and the distributed checkpointing algorithm. Section 3.5 presents the performance of both OpenMP reparallelization and migration.

3.2 Related Work

To our knowledge, reparallelization and migration techniques for OpenMP programs are not available. Related work can roughly be divided into three categories: (1) reparallelization of OpenMP programs, (2) migration of processes and MPI programs, and (3) checkpointing. (2) and (3) are related as migration uses checkpointing.

Although the OpenMP specification allows to alter the number of threads per parallel region [92], in existing implementations except ours the number is fixed for the duration of the region. Adaptation of the thread count by the runtime system is restricted to code areas outside of parallel regions. Only the extension of OpenMP for irregular data structures proposed in [114] offers deferred cancellation. While new threads may not be created, worker threads can be scheduled to exit at certain cancellation points.

MOSIX [14], Sprite [41], and others, can migrate a process from one node to another. In contrast to our solution, they neither offer capabilities to change the degree of parallelism nor can they migrate on heterogeneous clusters. Our approach does not leave a process stub back at the old node to access immobile resources such as open files and network connections. Finally, we avoid kernel modifications, which we find unacceptable for general-purpose clusters.

Cactus [8] and DGET [46] can adapt to changes in the computing environment and acquire or release nodes while an application is running. However, both enforce own programming models in which the application has to extend a framework with callbacks to its application-specific functionality. Furthermore, our approach allows generic checkpointing without manual registration of data.

There is work that focuses on the migration of MPI programs from one cluster to another. In GrADS [119] the application registers its to-be-checkpointed data at a user-level checkpointing library. Similar to our approach, the application is migrated by checkpointing and restoring; the number of processors can be changed upon a restart. In contrast to our work, GrADS is limited to iterative MPI programs that are explicitly designed by the programmer for reparallelization. P-GRADE [76] checkpoints and migrates MPI/PVM applications. However, the application’s degree of parallelism is fixed after a migration. The mobile MPI programs of [50] and AMPI [67] do not perform true reparallelization. Instead, the application is over-decomposed for a high number of virtual processors that are in turn mapped to actual MPI processes by multiplexing them in MPI function calls. In contrast to our approach, a major limitation is the maximum degree of parallelism. Being unrelated to the application’s general scalability, an application cannot use more nodes than the number of virtual processors it started with.

Checkpointing [72] forms the basis of most migration approaches. Checkpointing libraries such as libckpt [95], or kernel modules such as BLCR [42] that dump the address space of a process to disk cannot be used in heterogeneous environments.
CHAPTER 3. REPARALLELIZATION AND MIGRATION

Porch [12] follows a compiler-based approach to provide heterogeneous checkpointing, but is limited to single processes. Multi-process checkpoints may be created by [103, 4, 89], and many more. LAM/MPI [103] offers multi-process checkpointing with BLCR and a coordinated (stop-the-world) algorithm. In [4], the application’s control flow is analyzed to find locations of checkpointing initiations that achieve a consistent global state. In [89], synchronized clocks are employed to maintain a consistent state for a checkpoint. Our checkpointing approach uses a coordinated algorithm for simplicity, as the overhead of writing the heap and the thread stacks to disk completely hides the coordination overhead.

3.3 Reparallelization

While prior work creates a large number of virtual processors and maps the virtual to the physical processors (called over-decomposition), our approach modifies the actual parallelization and data partitioning at the application level. The advantage is that this does not limit the maximum degree of parallelism as done by over-decomposition.

In our approach, the number of worker threads can be changed at certain points in the program, called adjustment points. They can be inserted either manually by means of the adjust directive (see Fig. 3.1) or automatically by the compiler (future work). At adjustment points, new threads can enter a parallel region or existing ones can be terminated.

Our reparallelization covers all OpenMP constructs. Below we first discuss reparallelization of work-sharing constructs. Then we examine the differences between the loop schedule types. Finally, we study adjustment issues of reductions and parallel regions.

3.3.1 Repartitioning of work-sharing constructs

OpenMP programs often process a data structure in parallel by means of the for work-sharing construct that assigns different parts of an iteration space to the available worker threads. Hence, the reparallelization of the for directive is crucial for the reparallelization of OpenMP codes.

According to [92], size and shape of the iteration space of an OpenMP for construct are known before the loop starts. The iteration space can be divided into chunks of a fixed size that are then processed by the worker threads.

A dynamic reparallelization must take into account what fraction of the iteration space is already computed, and what remains to be done by the changed number of threads, see Fig. 3.2. To be flexible, adjustment points may be placed at arbitrary code locations. Hence, at some point in time, some chunks are still unprocessed, others are partially processed (i.e. some of their iterations are done), and some chunks are completed. In the partially processed chunks some iterations are completed, while other iterations still need to be done. There might even be single unfinished iterations.
3.3. REPARALLELIZATION

If a worker thread is removed at an adjustment point in the body of a work-sharing construct, another thread has to take over the remaining work. Hence, a description of a partial chunk has to be created and stored in a set of still uncompleted chunks. In addition to an iteration space description (loop counter value, the lower and upper bounds, and the step width or—in more complex situations—a bit vector) the partial chunk description must store the number of the adjustment point \((i)\) that caused its creation, and the values of all live variables for subsequent use by another worker. Standard compiler analysis is used to identify this set of live variables. To copy the values, their type must be known to the compiler. Hence, our approach relies on type-safe environments.

When new threads are added to the work force, they can grab any unprocessed chunk and start executing the region’s code for that chunk. (We skip the question where the unprocessed chunks come from for now.) If no unprocessed chunks are left, new threads have to take on partial chunks. But instead of starting to execute the region’s code from its beginning, it is necessary to jump to the position of the adjustment in the code. Hence, we need a jump table to branch to the appropriate adjustment point (START\(_i\)).

With partial chunks and the jump table explained, we are ready to discuss the compiler’s code template for the \(i\)-th adjustment point (see Fig. 3.3). When an adjustment is requested, a worker first checks if it is selected for termination. If so, it stores a partial chunk description and terminates. The master thread (ID 0) adjusts JaMP’s internal data structures if the number of workers has changed. When a thread takes on a partial chunk for adjustment point \(i\), it jumps to the START\(_i\) label, loads the iteration space information and the live variables from the partial chunk description, and starts execution.

At the end of a work-sharing construct there must always be a barrier synchronization. If the parallel region has an adjustment point, the barrier code is more complex, since
partial chunks might remain to be processed by the existing threads. The skeleton code given in Fig. 3.4 shows that a work-sharing construct is completed when (1) all chunks have been processed, (2) no partial chunks are left, and (3) all worker threads have arrived at the loop barrier construct.

3.3.2 Loop schedule types

OpenMP defines different schedule types for the assignment of loop chunks to worker threads. While the compiler uses the common code transformation template discussed in Section 3.3.1 for all types, there are schedule specific issues.

If no loop schedule type is specified, the iteration space is divided such that every worker thread receives exactly one chunk. If an adjustment adds a new thread, at least one new chunk is needed as well. Hence, the whole iteration space has to be redistributed. This requires a bit vector in which a bit is set for every finished iteration. At the adjustment point, a new set of chunks can then be created by first computing the unprocessed iterations, dividing them by the new number of worker threads, and then assigning the same number of unprocessed iterations to each of the threads.

Since in a dynamic loop schedule threads request small chunks from a global work pile anyway, new worker threads can participate in the computation without further effort.

In a static loop schedule, all chunks are assigned to the worker threads at the beginning of the loop in a round-robin fashion. Therefore, this assignment has to be updated when the number of worker threads changes. Since a new assignment can only be computed if it is known which chunks have already been processed, every worker thread memorizes the list of completed chunks. Reassignment is done in two steps. First, all chunks are assigned as usual to the worker threads in a round-robin fashion. Since completed chunks are marked, they can be skipped later on. Step 1 creates an unbalanced load since some worker threads receive more unprocessed chunks than others. Step 2 relocates unprocessed chunks from over-loaded threads to under-supplied ones to achieve a better load-balancing. We have adapted an algorithm from [5] for this purpose.

An example of static reassignment is given in Fig. 3.5. The iteration space is distributed over two threads. Thread 0 has not yet completed any chunks, whereas thread 1 has computed four chunks (marked “X”). After doubling the number of threads, step 1 redistributes the chunks to four threads in a round-robin fashion. While threads 0 and 2 each receive three unprocessed chunks, the other two only receive one uncomputed chunk. To improve load-balance, step 2 moves two chunks to lighter-loaded threads.
3.3. REPARALLELIZATION

We now discuss the reparallelization of OpenMP reductions that combine the partial results accumulated by all threads into a single result at the end of a parallel region.

In work-sharing constructs, every single iteration contributes to the global result. In case of reparallelization, only termination of threads needs special treatment. There are two cases. First, a thread that is terminated at the beginning of a work-sharing construct still has to contribute its already accumulated partial result to the reduction. Second, a thread that is terminated inside the loop body stores its partial results in the partial chunk descriptor. When another thread takes on this partial chunk, it can no longer simply copy all the live variables from the partial chunk descriptor. Instead of overwriting it, the reduction value has to be merged with the thread’s partial result.

3.3.4 Context for added worker threads

When a worker thread takes on a partial chunk, all live variables are contained therein. On the other hand, when new worker threads are added to a work-sharing construct, they start execution at the beginning of the construct. If there are live variables that have been defined outside of the work-sharing construct, they have to be initialized with valid values for the new worker threads. For that purpose, the set of live variables is stored by the master thread directly before the work-sharing construct and a copy of it is loaded by the added worker threads.

In Fig. 3.6, a variable \( c \) of a class called Collector is created inside of the parallel region by every worker thread and used in some computation. Moreover, there is a variable \( z \) of type int. Both \( c \) and \( z \) are live variables from outside of the work-sharing construct. When a worker thread is added at the adjustment point, it has to receive valid values for \( c \) and \( z \). Therefore, the set of live variables is initialized from the live variables of the master thread.
void dependent(double[] array) {
  /*omp parallel*/
  int id = JampRuntime.getThreadNum();
  int cnt = JampRuntime.getNumThreads();
  int sz = array.length / cnt;
  int fr = sz * id;
  for (int i = fr; i < fr + size; i++) {
    // computation using array[i]
  }
}

/*omp parallel for*/
for (int i = 0; i < array.length; i++) {
  // computation using array[i]
}

Figure 3.7: Dependency to the thread ID.

void independent(double[] array) {
  /*omp parallel for*/
  for (int i = 0; i < array.length; i++) {
    // computation using array[i]
  }
}

Figure 3.8: Corrected example of Fig. 3.7.

The values of variables of primitive data types (such as \(z\)) are copied. The \(\text{clone}()\) method is used for objects of classes that implement the \(\text{Cloneable}\) interface. For other classes and if standard cloning does not produce a valid copy, special cloning facilities can be provided by the user.

3.3.5 Limitations

Our approach has the following three limitations:

a) Reparallelization can only be performed if the complete information about the parallelization and the partitioning of the work-sharing constructs is available. In Fig. 3.7 parallelization is done by OpenMP, but work distribution is programmed explicitly. The example code introduces an indirect data dependence from the thread ID to \(\text{array}[i]\). Due to that data dependence a reparallelization causes undefined behavior, as the thread with a given ID might have been removed. Hence, we disallow the use of \(\text{getThreadNum}()\) and \(\text{getNumThreads}()\) and the compiler issues warnings. Fig. 3.8 is the correctly parallelized version of Fig. 3.7.

b) If worker threads are added to or removed from a parallel region, the number of times every statement is executed can change. This might affect program semantics. Assume a thread is removed at the barrier in Fig. 3.9 Then fewer “B”s than “A”s are printed. If such a behavior is undesired, the programmer can disallow reparallelization by adding the \(\text{adjust(none)}\) clause to the \(\text{parallel}\) directive.

c) As mentioned above, the live variables of the master thread are copied to a newly created thread. However, this might not be the desired application’s semantics. For example, when a variable is used to store the current thread ID, the new thread would receive the ID of the master thread. The programmer has to be aware of this issue and deal with it accordingly. However, since in most OpenMP programs the use of a thread ID indicates a weak application design, we consider this to be an acceptable restriction of the programming model.
3.4 Migration

Checkpointing the DSM space of a Jackal application forms the basis of our migration approach. Checkpointing allows to save the computational state of an application [72]. The state can then be transferred to another cluster, on which the application is resumed. We have presented a compiler-based approach for migrating threads in heterogeneous clusters in [122]. It provides a means to checkpoint the computational state of a single thread, to move the state to another machine of a potentially different architecture, and to continue the computation there. For migration of OpenMP applications, this paper adds a coordinated checkpointing algorithm for multi-process applications. We first sketch how a checkpoint is created for a single thread. We then shortly describe the extensions made to support checkpointing of multi-process applications.

3.4.1 Thread Checkpointing

A generic stackframe format to store the current stack of a thread is the basis of platform-independence in [122]. For each call-site of a function, the compiler creates so-called checkpointers and uncheckpointers. Checkpointers map the stackframe of the function at that call-site to a generic, machine-independent format. In turn, uncheckpointers load a function’s stackframe from the generic format.

The computational state at a given call-site can be described by the live variables at that location. For each of the live variables, the compiler creates a unique Usage Descriptor String (UDS) that platform-independently describes the variable. The generic stackframe format consists of a set of tuples (UDS, value). As described in [122], the creation of the UDS is mainly based on the following assumption: the value of a variable \( A \) at a given point in a program is determined by the preceding computation \( H \) that affected \( A \). This computation has to be the same on all architectures. Otherwise, the program would compute different results on different architectures. Hence, the central idea is to encode \( H \) in the UDS. For brevity, only the rules for constructing a UDS for \( H \) are listed in Fig. 3.10. The details can be found in [122].
CHAPTER 3. REPARALLELIZATION AND MIGRATION

```java
1     int someFunction (int c) {
2     B0   int a = 0;
3     B0   int b = 1000;
4     B0   if (c != 0)
5     B1     a = a + 1;
6     B2     // live variables: {a, b}
7     B2     createCheckpoint();
8     B2     return a + b;
9   }
```

Figure 3.11: Java checkpointing example.

As an example of how to construct the UDS, let us consider a Java function (Fig. 3.11) with three basic blocks (B0 through B2). At the call-site of `createCheckpoint()` in B2, two variables `a` and `b` are live. According to the rules R1 and R7 of Fig. 3.10 the compiler creates the UDS “C:1000@B:0” for `b`, which reads as “variable `b` is initialized with value 1000 in basic block B0”. A more complex UDS is created for `a`. The compiler searches backwards from the call-site in B2 to find all reaching definitions for `a`. For the assignment in B1 it creates the partial UDS “+ a C:1@B1” (R6 & R7). It then continues to search for the contained `a`. This delivers “C:0@B0” (R1 & R7). Hence, the final UDS for `a` at the call-site is “+ C:0@B0 C:1@B1”, a platform-independent description of the computation of the value of `a` in line 8.

The algorithm above forms the basis for our checkpointing approach. A thread is checkpointed by sequentially calling the checkpointers for each function on the call stack. In addition, the reachable objects in memory are traversed and written to disk. In contrast to standard Java serialization [112], this is done asynchronously. Chunks of data are handed off to a service thread for compression and for writing the compressed data to disk with bulk transfers. Hence, the checkpointer does not need to wait for the disk to catch up. As soon as the checkpointer has finished, the application can continue while the checkpoint data is still being written to disk in the background.

### 3.4.2 Multi-process checkpointing

A Jackal application consists of a set of processes that together form the application. With JaMP, each process receives one OpenMP thread. However, Jackal processes contain not only application-specific worker threads but also execute several service threads such as the garbage collector and the finalizer thread that both are needed by the Java runtime. To checkpoint a multi-process application, we have implemented a coordinated checkpointing algorithm. It (1) stops all local threads, (2) ensures that no messages are on the network, and (3) checkpoints all the threads of a process.

A single-process application is checkpointed by saving the state of all threads and the heap to disk. This is accomplished by collecting all threads in a special barrier, effectively stopping all threads. Whenever, a thread reaches a synchronization point (e.g. `Object.wait()`, `Thread.sleep()`) it checks whether a checkpoint is requested. If so, it triggers the thread checkpointing algorithm. Otherwise, it proceeds with the regular synchronization code.

Already blocked threads that wait for a notification or a timeout to occur need a special treatment. To inform them about a new checkpoint request, we wake them up by means of a `CheckpointException`. The threads then checkpoint and re-enter their waiting state. Hence, from an application point of view, they seem to never leave their blocking state.

Multi-process checkpointing is achieved by means of coordinated checkpointing [72]. To request a checkpoint, the master node sends a broadcast to all other
3.5. **PERFORMANCE**

Table 3.1: Runtimes, overheads, and checkpoint sizes (in MB) for the benchmark suite.

<table>
<thead>
<tr>
<th>Thr.</th>
<th>LBM Time Adjust-</th>
<th>Check-</th>
<th>CP Overhead</th>
<th>SOR Time Adjust-</th>
<th>Check-</th>
<th>CP Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(sec)</td>
<td>ment point size</td>
<td></td>
<td></td>
<td>(sec)</td>
<td>ment point size</td>
</tr>
<tr>
<td>1</td>
<td>924</td>
<td>-2.5%</td>
<td>0.3%</td>
<td>117</td>
<td>430</td>
<td>0.9%</td>
</tr>
<tr>
<td>2</td>
<td>467</td>
<td>-2.1%</td>
<td>0.2%</td>
<td>122</td>
<td>221</td>
<td>2.9%</td>
</tr>
<tr>
<td>4</td>
<td>242</td>
<td>0.4%</td>
<td>0.3%</td>
<td>128</td>
<td>118</td>
<td>12.9%</td>
</tr>
<tr>
<td>8</td>
<td>127</td>
<td>2.6%</td>
<td>1.1%</td>
<td>136</td>
<td>65</td>
<td>15.9%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thr.</th>
<th>Crypt Time Adjust-</th>
<th>Check-</th>
<th>CP Overhead</th>
<th>Raytracer Time Adjust-</th>
<th>Check-</th>
<th>CP Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(sec)</td>
<td>ment point size</td>
<td></td>
<td></td>
<td>(sec)</td>
<td>ment point size</td>
</tr>
<tr>
<td>1</td>
<td>91</td>
<td>9.7%</td>
<td>3.1%</td>
<td>80</td>
<td>473</td>
<td>3.6%</td>
</tr>
<tr>
<td>2</td>
<td>52</td>
<td>9.9%</td>
<td>2.6%</td>
<td>80</td>
<td>234</td>
<td>2.3%</td>
</tr>
<tr>
<td>4</td>
<td>29</td>
<td>5.1%</td>
<td>5.5%</td>
<td>80</td>
<td>117</td>
<td>3.1%</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>3.5%</td>
<td>8.8%</td>
<td>80</td>
<td>60</td>
<td>3.0%</td>
</tr>
</tbody>
</table>

nodes. Due to the FIFO property of Jackal’s communication layer, this broadcast message pushes all data messages through the network to their respective receivers. As soon as another node receives a checkpoint request message, it informs its threads about the checkpoint request. All threads continue until they reach either a synchronization point or a call of `createCheckpoint()`. After all local threads are blocked, the nodes send a broadcast message to clear the network of application traffic. Then, all local threads start to write the checkpoint. This implements a race-free stop-the-world approach, since all threads block and the network is clear of messages. Hence, no rollback operations are necessary upon restarting from a checkpoint.

### 3.4.3 Interaction with reparallelization

When the application resumes from the checkpoint, the number of nodes that execute the program can change. If the new number is lower, the runtime system merges the images of the removed nodes into the resuming ones. If the number is increased, new nodes start with empty images.

The reparallelization runtime directly interacts with the checkpointing system by being notified of changes whenever an application is resumed from a checkpoint. The runtime then starts the reparallelization of the currently active parallel region and adapts the thread count such that each node receives one OpenMP thread. Data distribution is handled automatically by the Jackal DSM runtime system.

### 3.5 Performance

To show the feasibility of our reparallelization and migration approach, we have undertaken a set of experiments. The experiments were performed on a cluster of quad (2x dual core) AMD Opteron 2.0 GHz 64 bit nodes with 4 GB of main memory and Gigabit Ethernet. The cluster is located at the University of Erlangen, Germany.
CHAPTER 3. REPARALLELIZATION AND MIGRATION

Figure 3.12: Runtime per LBM time step and speed-up graph of LBM.

effects of network traffic, we only used one CPU per node. The results presented are
the average over 5 runs of each benchmark. Overheads are computed as the relative
increase of runtime of each benchmark without adjustment points and/or checkpointing.

3.5.1 Benchmarks

For our evaluation we implemented a compute intensive Lattice-Boltzmann Method
(LBM) [127] benchmark. In addition, we use JOMP’s Java-OpenMP port of the JGF
benchmarks [107, 23]. We skip section 1 of the benchmark suite since it solely contains
microbenchmarks for individual OpenMP directives, such as creation of parallel regions.
From section 2 and 3, we study SOR, Crypt, and Raytracer. (Euler uses the unsupported
OpenMP construct ordered. Sparse introduces data dependencies to the thread ID, which
we disallow. LU Fact and Monte Carlo are not suited to be executed on a DSM system
as the JGF versions contain large sequential fractions and/or suffer from false-sharing.)

LBM simulates fluids with cellular automata. Space and time are discretized and
normalized. In our case, LBM operates on a 3D domain divided into 120×120×120
cells that hold a finite number of states. In one time step the whole set of states is updated
synchronously by deterministic, uniform update rules. The kernel is parallelized in a
straightforward way; the time-stepping loop is parallelized with parallel and the loop
over the x-axis of the grid is parallelized with the for directive (default scheduling). We
have also parallelized the data allocation using parallel for such that the nodes that work
on a partition of the grid also perform the allocation. This is a well-known optimization
for OpenMP programs on NUMA architectures. The benchmark computes 50 time steps
over the 3D grid.

SOR solves a discrete Laplace equation with simple over-relaxation (200 iterations)
in a red-black style on an 10,000×10,000 grid. The outer loop is parallelized with the
3.5. PERFORMANCE

Figure 3.13: Runtime per SOR red-black iteration and speed-up graph of SOR.

Parallel directive while the inner loop over the grid is parallelized with the for directive and default scheduling. The data allocation was parallelized with parallel for. Crypt performs IDEA encryption and decryption of 140 MB of data and strongly depends on bit and byte operations. The main encryption/decryption loop is parallelized with a parallel for with default scheduling. The Raytracer renders a scene of 64 spheres in a picture with a resolution of 800×800. The main loop of the benchmark is parallelized with the parallel for directive and dynamic scheduling with chunk size 10. Hence, each thread renders a partition of the picture. Raytracer works on a read-only data set (the spheres) and represents the picture as an 1D array.

3.5.2 Overheads

Table 3.1 shows the runtimes of the individual benchmarks. Overall, the average overhead for supporting dynamic adjustment of the thread count is approximately 4%, which can be considered acceptable. The overhead is determined by the amount of work of the parallel region, as the code transformation adds a constant overhead. Checkpointing imposes a runtime overhead of roughly 2% on average when creating one checkpoint during the execution of the benchmark. The overhead is influenced by the data set of the application and is almost unrelated to the disk transfer rate. For realistic applications and realistic data sizes, the overhead is negligible (below 1%).

Inserting adjustment points into the SOR kernel decreases performance by up to 15.9% (see Table 3.1). This large overhead is caused by adding a constant overhead per adjustment point to a very low runtime per iteration. Thus, the relative overhead per iteration becomes significant. For LBM a negative overhead of roughly 2% can be observed due to processor caching effects.
3.5.3 Speedup

For the speedup measurements, we have set up Jackal such that half of the processes received OpenMP threads while the other half is idling. At benchmark half-time, the thread count is doubled. The additional threads are spawned on the idle nodes. This scenario is reversed for the removal of workers.

Fig. 3.12 shows the runtime per LBM time step over time. At time step 25, the number of threads is doubled. As can be seen on the left, the runtime roughly decreases by a factor of 1.8. A slow-down of about 2 occurs when the number of threads is halved (middle). This closely matches the speed-up behavior of LBM (on the right). The peak runtime after the adjustment in iteration 25 is caused by the DSM protocol that needs to redistribute the data after the reparallelization, i.e., that moves the data accessed (roughly 3.6 LBM cells) by the threads to their respective execution nodes. Note that such delays are caused by any NUMA system and the height strongly depends on the latencies of the NUMA implementation. In addition, a NUMA implementation that allows migration of data is desirable to avoid performance penalties after reparallelization.

A similar result is achieved for SOR. As Fig. 3.13 shows, the runtime of a single SOR red-black iteration is decreased by a factor of 1.8 when the number of threads is doubled at iteration 100. The runtime peak at iteration 26 is again caused by the DSM runtime that needs to redistribute data (10,000 arrays or 10,000 messages over the network). Crypt and Raytracer also show the desired speed-up behavior (not depicted for brevity), when the number of threads changes. When doubling the thread count, both applications achieve a speed-up of about 1.9, while they yield a slowdown of 1.9 when the thread count is halved. The runtime peak for Crypt and Raytracer is lower, as the data that needs to be redistributed is smaller.

3.5.4 Migration

To demonstrate our approach, we migrated LBM from the cluster at Erlangen, Germany, to a cluster at the Vrije Universiteit in Amsterdam, the Netherlands, and back. The cluster in Amsterdam uses dual Intel Pentium 3 CPUs with a 1 GHz processor clock, 1 GB of memory, Gigabit Ethernet, and Myrinet for each node. LBM is migrated two times: (1) at the 16th time step from Erlangen to Amsterdam, and (2) at the 33rd time step from Amsterdam back to Erlangen.

Fig. 3.14 shows the times for one LBM time step. After the 16th iteration we have manually aborted and migrated the application from Erlangen to Amsterdam. The number of CPUs was hereby increased by a factor of four. The performance roughly doubles as the target CPUs are slower than the originating CPUs. In time step 33, LBM is moved back to Erlangen. This time, the number of CPUs was halved. Please note, that the time to transfer the checkpoint image and to wait for the cluster reservation are not included. The time to transfer the checkpoint of LBM between the clusters roughly was 50 sec. The total queue time in the cluster queues was about 5 minutes.

3.6 Conclusion

We have presented a novel approach to reparallelize and to migrate OpenMP applications between clusters of different size and architecture. This helps to make the boundaries of individual clusters in a computational grid less visible. A user can start an application at an arbitrary cluster in the grid. When the time slice is about to be exceeded, a checkpoint
3.6. CONCLUSION

Figure 3.14: Migration of LBM.

can be created. The application can either migrate to another cluster or restart on the current system with a new reservation. Reparallelization automatically adapts to the new number of available processors. Reparallelization is restricted to (1) well-formed OpenMP programs, and (2) type-safe programming languages.

Benchmarking shows that the reparallelization imposes little overhead and scales as expected. When the number of threads is changed, the new parallelization achieves speed-ups that are comparable to the regular speed-up behavior of the application with that number of processors. The overhead of inserting extra code for adjustment points is almost negligible compared to the overall runtime. The same holds for the overhead of checkpointing the application state.
Chapter 4

Near Overhead-free Heterogeneous Thread Migration

Authors:
Ronald Veldema, Michael Philippsen
University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3 • 91058 Erlangen • Germany
veldema@cs.fau.de, philippsen@cs.fau.de

Published in:
Proceedings of the 2005 IEEE International Conference on Cluster Computing
Conference: Cluster ’05
Boston, Massachusetts, USA
September 26 - 30, 2005
Pages 1 – 10
Publisher: IEEE Computer Society, Los Alamitos, CA, USA.

The last chapter’s paper depended on a checkpointing package to checkpoint and restore program state between different architectures. Previous checkpointing packages used source code rewriting to enable heterogeneity which causes overhead during normal computation. The approach described here allows heterogeneity and incurs no run time overhead when not checkpointing. Additionally, using thread-migration, a DSM’s efficiency can be increased due to better load-balancing.
Abstract

Thread migration moves a single call-stack to another machine to improve either load balancing or locality. Current approaches for checkpointing and thread migration are either not heterogeneous or they introduce large runtime overhead. In general, previous approaches add overhead by instrumenting each function in a program. The instrumentation costs are then even incurred when no thread migration is performed. In this respect our system is near-overhead free: nearly no overhead is caused if no migration is performed. Our implementation instead generates meta-functions for each location in the code where a function is called. These functions portably save and rebuild activation records to and from a machine-independent format. Each variable of an activation record is described in terms of its usages in a machine-independent ‘Usage Descriptor String’ to enable heterogeneous, near overhead free thread migration with as few as possible changes to a compiler. Our resulting thread migration solution is, for example, able to move a thread between an x86 machine (few registers, 32 bits) and an Itanium machine (many registers, 64 bits).

Furthermore, we (optionally) move the decision on when and where to migrate to the application programmer instead of implementing a fixed ‘fits-all’ heuristics as in previous approaches.

4.1 Introduction

Thread migration is the process of moving a thread from one machine to another to, for example, improve load balancing, fix some resource requirement, or to reduce the amount of network communication by moving a thread closer to its required data.

Thread migration is required for applications whose load balance changes over time, or for non-dedicated clusters where at anytime someone could login to or logout of the constituent machines. When someone logs out, that machine becomes available for parallel computations and a thread could be moved to that machine, the reverse case when someone logs in.

Many thread migration packages (including the system presented here) give a programmer or compiler access to an API to perform thread migration. In general, the programmer or compiler inserts calls to a ‘migrate_this_thread()’ method in the code. This method causes all relevant information about the current thread — such as the thread’s stack and active registers — to be sent over the network to another machine where it is restarted. To the program, it then seems as if control just returned from the ‘migrate_this_thread()’ function as if nothing has happened. Note that an invocation of ‘migrate_this_thread()’ is nothing more but a ‘hint’ that can be ignored if no migration is required.

To apply thread migration in a grid environment there are two general problems to be solved. For one, since the machines in the grid are inherently heterogeneous the mechanics of how to migrate are challenging. Secondly, the problem of optimally scheduling threads to machines to optimize both for load balance and minimal communication is an NP-complete problem. Therefore heuristics are required to approximate the optimal solution, i.e. to decide when and where to migrate to. Also note, that no single heuristic will be optimal for every application since applications will have different communicative and computational behaviors.
4.1. INTRODUCTION

4.1.1 Issues of heterogeneous thread migration

There are numerous problems that a thread migration package has to solve to provide fully heterogeneous thread migration, i.e. to allow a thread to move, for example, from an x86 machine to a PowerPC machine.

Firstly, a translation step is required to actively translate values from one machine to another to handle endian-ness issues, different pointer sizes, and different bit encodings of floating point and integer numbers. Likewise, the layout of objects varies between architectures. Because of these differences, a thread migration system cannot send the bit-representation of entire activation records or objects from one machine to another.

```java
void foo(int a, int b) {
    for (int i=0; i<10; i++)
        zoo();
    < live variables : a, b, i >
}
(a) Java
```

```x86 pseudocode
void foo(int *(framepointer+8), int *(framepointer+12)) {
    for (register edi=0; edi<10; edi++)
        zoo();
    < live variables : register edi,
        *(framepointer + 8),
        *(framepointer + 12) >
}
(b) x86 pseudocode
```

```IA64 pseudocode
void foo(register in0, register in1) {
    for (register loc0=0; loc0<10; loc0++)
        zoo();
    < live variables : registers in0, in1, loc0 >
}
(c) IA64 pseudocode
```

Figure 4.1: Translation of an activation record (pseudocode).

Secondly, different architectures have different numbers and types of registers. See, for example, the fragment of Java code in Figure 4.1(a). On an x86 processor, the variables in the example could be allocated as shown in Figure 4.1(b). On an IA64 processor, the variables could be allocated as shown in Figure 4.1(c).

What is required is a mechanism to re-associate the registers and stackframe addresses with the common variable names. In the example, to migrate `foo()` from an IA64 to an x86 machine, we would need to re-associate register ’loc0’ with loop variable ’i’, and at the x86 machine map ’i’ to machine register edi.

Unfortunately, source code variable names cannot be used directly to re-associate the variables with their physical locations as naming information is lost in the compiler with increasing levels of optimization. For example, in general, we cannot directly re-associate register ’edi’ with source code variable ’i’ as that information is simply no longer available. While compilers that support both simultaneous debugging and optimization in such cases only provide approximations to the debugger, this will not suffice for our purposes.

Similarly, the ordering of live variables inside a compiler can vary between architectures as optimizations and architectural constraints may have reordered them differently due to machine-dependent optimizations and settings (for example caused by register promotion or early instruction scheduling). Even early in the compiler, this prohibits
the use of variable identifiers in the compiler’s intermediate representation.

Finally, the thread migration system should impose little overhead during normal execution (when no thread migration occurs). Ideally, the thread migration package should be so efficient that the added overhead is almost unnoticeable in application performance. The only noticeable effect should be an occasional drop in I/O-performance.

To allow heterogeneous thread migration to occur, there are two basic approaches:

1 to instrument the application’s source code to record the addresses and values of each variable and to restore each variable’s value in case the current invocation was performed because of restoration purposes. The foo in our example would be transformed to foo’. Added code in foo’, will upon calling zoo, save the values of the parameters and the loop index in a separate runtime stack. Upon thread-stack-restore, the reverse operation is applied. This however, means that a little bit of extra code at the start of ’foo’ is added to see if restoration is required and if so to do the restoration of the loop variable. As foo potentially contains more than one call statement, a test is required to determine after which control should be resumed at.

2 to reverse engineer the value of each source code level value by recording the transformations applied during optimization and code generation. This approach comes in two varieties:

   2a play-backward: record the changes applied when undoing the code generation and optimization processes back from machine code towards source code form.

   2b play-forward: record the changes applied when applying the code generation and optimization transformations from source code to the final machine code.

Both approaches are problematic. Instrumenting the source code as in solution 1, causes performance problems as the added code is executed even if no migration is performed. The cost of the instrumentation can easily double execution time. Solution 2 causes no runtime overhead but instead causes much complexity inside the compiler. A compiler will become increasingly complex with each added optimization pass as it needs to, besides performing and analyzing its own transformations and analyseses, record how to undo or redo its effects on the variables effected by it. Also, for machines where for code generation itself is non-trivial, the undo or redo mechanisms are possibly also often non-trivial as well.

This paper uses neither approach: the code to record the values of variables is located outside the function to-be checkpointed. Also, we do not use source code level variables for reassociation, but rather compiler intermediate variables that exist at some stage in the compiler.

4.1.2 Issues of migration heuristics

The second problem of using thread migration is due to the heuristics needed to decide when to migrate a thread and where to migrate it to. Many thread-migration systems implement a single heuristic in their runtime system that covers a specific range of problems. Our solution is to allow the programmer of an application to determine the heuristics (when and where-to migrate) of thread migration. We implement this in the form of meta-threads. A programmer inherits from the base meta-thread class and overwrites the methods that determine when and where to migrate to. A meta-thread can then be associated with a real-thread to determine its migration behavior. The exact syntax and semantics of meta-threads are discussed in section 4.4.
4.2. JACKAL

4.1.3 Contributions

Our new thread migration algorithm solves most of the problems mentioned above: a function’s activation record is written to a machine-independent format using a novel way to describe the individual variables. Furthermore, we have a mechanism to ‘undo’ and ‘redo’ machine specific optimizations. Together, this allows a thread-stack to be restored on a different architecture than where the snapshot was taken. Finally, the compiler emits specialized meta-functions for each call-site in a program to save and restore an activation record while imposing no overhead during normal execution. The contributions in this paper are thus:

a) a mechanism for implementing overhead free save-and-restore of a thread call-stack by generating meta-functions per call-site;
b) a meta-thread concept that allows the application programmer to formulate a migration policy.

4.2 Jackal

A short introduction to Jackal [121]: Jackal is a multi-threaded DSM system for Java that implements an object based DSM system. Multiple threads are allowed to share a single process space on a single machine to enable both the efficient use of SMP nodes in a cluster and to allow the overlapping of the communication of one thread with the computation of another. Jackal is an object based DSM which means that objects instead of pages are sent over the network. Read/write access detection is implemented in software as our compiler automatically adds code to explicitly test before each object access if an object is locally available (an access check). We use a static, ahead-of-time highly optimizing compiler, which means that Java source code (not bytecode) is translated to a static, native executable. Our compiler implements aggressive optimizations to reduce the number of inserted access checks to reduce overhead.

One of the advantages of implementing our new thread-migration algorithms in Jackal, is that it already has support for location independent file descriptors and heterogeneous object management (Jackal supports heterogeneous clusters) so that file descriptors and pointers can already be freely exchanged between different types of machines.

4.3 Mechanics

In this section, we describe the mechanics of our thread migration support: how can we move a thread from one machine to a machine of a different architecture. Our algorithm enables heterogeneous thread migration with few restrictions and nearly no runtime overhead while attempting to minimize the number of changes needed in the compiler. To be precise:

- The number of compiler passes that need to be aware of thread migration is kept small. High level optimizations such as loop transformations and method inlining are unchanged. Only the lower level code generation passes need to be changed to record at each call site, what its effects are to the live variables at that call site.
- We restrict ourselves to type-safe languages such as Java, C#, etc. (which is already a prerequisite for our DSM system, Jackal).
In our system, the compiler analyzes the given source code and automatically inserts
the calls to `migrate_this_thread()` using the algorithm proposed in [82], although
adapted slightly to reduce the number of inserted ‘sparse potential’ migration points
(see [82] for details). Note that, because the compiler inserts these calls, these calls
can potentially occur anywhere making saving of the complete call-stack an absolute
necessity. If our algorithm inserts too many migration points such that introduced
overheads become noticeable, the programmer has the option to place them manually
instead.

The key idea of the implementation of the mechanism hinges on five assumption-
s/observations:

1. Assume a function F that calls another function G at call instruction C.
2. At C assume that a number of variables \(X_1, X_2, X_3, \ldots, X_N\) are live. Let the total
   number of live variables at each call site be fixed.
3. Observation: when compiling F using a machine with P registers, P of the N live
   variables will be in registers and \((N - P)\) will be allocated on the call-stack. When
   compiling F using a machine with M registers, M live variables will be in registers
   and \((N - M)\) will be allocated in memory on the call-stack. This holds as long as
   N does not change for each compilation. We can assure this by using the same
   compilation flags for each target architecture. The result is that we can always
   create a mapping function Z that maps the stackframe at C from one machine
to a ‘universal stackframe format’ and from that to the stackframe of the other
   machine.
4. Let us look at a single live variable \(X_a\) at C on one machine. That variable is
   computed using some computation H. At the other machine \(X_a\) has to be computed
   using computation H as well as otherwise \(X_a\) would contain a different value
   on both machines. Different \(X_a\)’s are not possible as the same source code is
   compiled on both machines, therefore H is the same on both machines as the
   semantics of both compiled programs must be the same. The only difference may
   be in how and where H is computed.
5. Withholding low level transformations such as peepholing which would, for
   example, transform a constant multiply to a constant shift operation, H will use
   the same operations on both machines.

Taking the above assumptions/observations into account, we can conclude that if
we were to convert H to a small separate function \(H’\), \(H’\) would uniquely and machine
independently describe \(X_a\). The reason to look at \(H’\) instead of H is that instructions not
related to H are excluded as they do not influence the value of \(X_a\).

Our algorithm, takes \(H’\) and converts it to a string by effectively transforming \(H’\)
to a tree and performing a prefix traversal.

This effectively creates a hash function Z for each live variable: \(Z(X_a) =\)
\(\text{string}(H’(X_a))\). One aspect of this hash is that the resulting strings are machine inde-
pendent: if \(Z(X_a)\) evaluates to string Q on one architecture, then it must evaluate to the
same string on another architecture if the assumptions made above hold for the compiler
used.

Practically, our algorithm for creating the flattened \(H’\) string is as follows:
1. \(A = \text{ constant}\),
   generate: \(\text{string}(A) = \text{“C:<constant>”}\)
4.3. MECHANICS

2. “A = B”,
   generate: “string(A) = string(B)”
3. “A = call foo”,
   generate: string(A) = “call:<index of call in all calls inside the containing
   function>”
4. “A = param(X)”,
   generate: String(A) = “P:<index of param(X) in function’s parameter list>”
5. Upon “A = object_access(expression, field)”, generate: string(A) = “access:field”,
   and continue with the variables in ’expression’.
6. “A = B op C”,
   generate: string(A) = “<op>” + string(B) + string(C).
7. whenever rule 1 to 6 is triggered, generate: string(A) = string(A) + “@B:<basic-
   block-number>”. In this way, control dependencies are encoded in the UDS strings.

From now on we will refer to the string of H’ as the Usage Descriptor String (UDS) for
a live variable at a call site. Such a UDS describes machine independently how that
variable is computed. If two variables have the same UDS then they are computed in
the same manner and will necessarily contain the same value. For example, if in the
same basic block variables ‘x’ and ‘y’ are computed in the same manner (have the same
defining expressions), they will end up having the same values at runtime and will have
the same UDS.

<table>
<thead>
<tr>
<th>basic block</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo(int a, Object b, int c) {</td>
</tr>
<tr>
<td>a = a + 2;</td>
</tr>
<tr>
<td>int y = 0;</td>
</tr>
<tr>
<td>do {</td>
</tr>
<tr>
<td>int x = 0;</td>
</tr>
<tr>
<td>do {</td>
</tr>
<tr>
<td>zoo(a, b, c);</td>
</tr>
<tr>
<td>// live variables = {a,b,c,x,y}</td>
</tr>
<tr>
<td>} while (x++&lt;10);</td>
</tr>
<tr>
<td>} while (y++&lt;10);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Figure 4.2: (Running) example 1.

4.3.1 Example UDS

As an example, let us construct the UDSs for the example of Figure 4.2 At ’zoo in foo’
we have the live variables (’a’, ’b’, ’c’, ’x’, ’y’). The analysis pass takes this set and
performs a traversal over the control flow graph of the function.

During the traversal, the initialization of ’x’ is encountered and the UDS is updated
by appending the string ’C:0’ (rule 1). That happens in basic block 1 thus the string
’@B:1’ is appended to the UDS (rule 7). Likewise, ’y’ is given the string ’C:0@B:0’,
etc.

The tuples built for the call to zoo in function foo (’zoo in foo’) are thus:
• (x, “C:0@B:1”, int_type)
• (y, “C:0@B:0”, int_type)
• (a, “+P:1,C:2@B:0”, int_type)
• (b, “P:2@B:0”, object pointer_type)
• (c, “P:3@B:0”, int_type)

This example also demonstrates the need for encoding the basic block information in the UDSs to encode flow-control dependencies as otherwise the UDSs of ’x’ and ’y’ would end up with the same UDS ’C:0’. During restoration we could then no longer distinguish the two variables and, hence, could not restore the activation record.

4.3.2 How to use UDS strings

Again, consider the example from Figure 4.2. Here variable ’x’ may be located in register ’edi’ on an x86 machine and in register ’loc0’ on an IA64 machine. Both ’loc0’ and ’edi’ should have the same UDS string as they represent the same variable. After all high level optimization have run, we therefore record the transformations made to the live variables.

Therefore, during code generation, whenever an operation renames ’x’, ’y’, ’a’, ’b’, or ’c’, from a variable to a call stack location or register, or renames the location that it was already in, it is also renamed in the corresponding tuple. Register allocation will, for example, when spilling ’a’ to memory rename ’a’ in the tuple to the expression in memory where ’a’ is located. Fortunately, only the compiler passes that occur after the descriptor strings have been created need to rewrite these expressions. The machine independent, high level optimization passes (such as loop transformations and method inlining) are completely thread migration/checkpointing agnostic.

4.3.3 Record and Restore of thread stacks

Thus far we described how to machine-independently save and restore the variables of a single stackframe but not how to save/restore complete call-stacks. That is the topic of this section. Globally speaking, at runtime, checkpointing unwinds the call-stack from the migrate_this_thread activation record upward towards the thread’s run method while call-stack restoration operates in the reverse direction.

The unique idea of our approach is that for each call-site a checkpoint or restoration function is generated. We will name these functions checkpoint(C in F) and restore(C in F), for call-site C in function F.

<table>
<thead>
<tr>
<th>Type</th>
<th>field</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *</td>
<td>PC</td>
<td>address of call-site C in function F</td>
</tr>
<tr>
<td>char *</td>
<td>NAME</td>
<td>string “C in F”</td>
</tr>
<tr>
<td>int</td>
<td>INDEX</td>
<td>C’s index in the list of call-sites in F</td>
</tr>
<tr>
<td>void(*)()</td>
<td>CHKPT</td>
<td>address of checkpoint(C in F)</td>
</tr>
<tr>
<td>void(*)()</td>
<td>RESTORE</td>
<td>address of restore(C in F)</td>
</tr>
</tbody>
</table>

Our checkpointing algorithm operates in two passes at compile time. After all the machine-independent optimizations, we create a UDS tuple (from Section 4.3) for each live variable at each call-site. After code generation we generate two extra functions: checkpoint(C in F) that checkpoints function F at call-site C and restore(C in F) that performs the reverse operation. The original call-sites and functions are unaffected. At program startup, all the checkpoint and restore functions are registered with the runtime
4.3. MECHANICS

system in a “checkpoint_meta_table” as a series of tuples (see Table 4.1).

During checkpointing, when unwinding an activation record, we search for a tuple in the “checkpoint_meta_table” where the activation record’s pushed program counter matches the ‘PC’ entry in the tuple. The CHKPT field of the tuple provides the address of the checkpoint(C in F) method that is then invoked. Checkpoint(C in F) then outputs for each live variable V a tuple \{UDS(V), value(V), type(V)\}. As checkpoint(C in F) and restore(C in F) are machine-dependently generated for that specific call-site, they know where each variable is physically located. Figure 4.3 displays the checkpointing algorithm in pseudocode.

X = stackframe at top of call stack;
while X not at bottom of call stack
    do// unwind:
        PC = program counter at (X);
        search for tuple T matching (PC, *, *, *, *);
        in checkpoint_meta_table
        call write_string(T.NAME, T.INDEX);
        call T.CHKPT(X’s framepointer, T.NAME, T.INDEX);
        X = parent stackframe(X);
    done
write_string(“terminator”);

Figure 4.3: Checkpointing pseudocode.

The restoration process for a single activation record starts by reading a single call-site descriptor string “C in F” and searches for a matching tuple in the “checkpoint_meta_table” whose restore(C in F) is then invoked. Restore(C in F)—see Figure 4.3—starts by reading the activation record’s list of tuples \{UDS(L), value(L), type(L)\}. For each live variable expected to restore, it searches for \texttt{value(V)} by supplying the \texttt{UDS(V)}. Restore(C in F) then converts and assigns the value to the right location in either memory or register.

Transfer of control after a stackframe has been restored is implemented by performing an inter-function jump statement that jumps directly to the position in the function that was checkpointed.

Our thread migration/checkpointing code is ‘overhead free’ as the code of the function for which checkpoint/restore functions are generated is unchanged except for the addition of a label so that the restore function can transfer control to the original function.

4.3.4 Expert level aspects

There are a number of problems in performing thread migration that are not immediately obvious and only interesting for the expert so we include them here for completeness.

One such an aspect is the handling of call-saved registers. To handle call-saved registers, in addition to the frame-pointer, we pass a pointer to a structure that holds the values of each call-saved register (not shown in example for brevity) to the checkpointer. This structure is updated while unwinding the stack by the runtime system. The compiler generates a few bits in each “checkpoint_meta_table” entry to tell which call-saved registers are saved and where they are saved inside a given activation record.
// Generated for each function-call-site pair.
// A reference to this function is added in each
tuple in the checkpoint_meta_table

void Restore_C_in_F() {
    if (at end of checkpoint_file)
        return;
    // read the tuples for my activation record:
    L = read_activation_record_tuple_list();
    // restore next activation record:
    (F', C') = read string
    if (F', C') != "terminator" {
        search for tuple T
            matching (*, F', C', *, *)
        in checkpoint_meta_table
        call T.RESTORE();
    }
    // code here to restore my activation record using L
    // by writing to the parameters/local variables
    // in this activation record (that of ‘Restore_C_in_F’).
    jump to instruction after C in F
}

// called at checkpoint-restore startup:
void StartRestore() {
    (F,C) = read string
    search for tuple T
        matching (*, F, C, *, *)
    in checkpoint_meta_table
    call T.RESTORE();
}

Figure 4.4: Pseudocode for restoring checkpoints.

Dealing with Machine-Dependent Optimizations

For efficient native code for a given machine, machine-dependent optimizations need to be performed. This, however, makes heterogeneous checkpointing difficult. Our system has two complementary answers to this problem.

1. Usage of UDSs allows inter-block scheduling.
2. Support for ‘Undo’ and ‘Redo’ of machining specific optimizations by transforming the variables inside the generated checkpointing code.

As stated above, the usage of descriptor strings to identify variables allows us to move instructions to different basic blocks or within a basic block as long as the assumptions put forth in section 4.3 are upheld, we allow machine-dependent scheduling, delay slot filling, CSE, etc., without any further changes to the checkpointing code.

A more difficult problem with machine dependent optimization occurs if variables change their meaning or are eliminated after the UDSs have been generated. To support this, the compiler generates ‘undo’ and ‘redo’ annotations in the intermediate representation whenever such a change is made after the UDSs have been generated.

For example, assume that after the UDSs have been generated, a machine-dependent strength reduction pass is run. An ‘undo’ and ‘redo’ annotation is then generated that notes for which variable this happened and which transformations to apply to undo or

---

1To further clarify: machine-independent optimizations in our compiler are: object combining, method inlining, loop splitting/merging/unrolling, CSE, GCSE, object escape analysis, strength reduction, register promotion, etc. Machine-dependent optimizations are: live-range splitting, register allocation, instruction scheduling/selection, delay slot filling, predication, load/store elimination, etc., mixed with a combination of general optimizations, i.e., CSE, strength reduction, etc., repeated
redo the transformation. See, for example, the Java code in Figure 4.5.

```java
// original code:
void foo() {
    for (int L=0; L<10; L++) {
        A = L * M + K;
        print(A);
    }
}

// optimized code:
void foo() {
    A = K;
    for (; A < end; ) {
        A = A + M;
        print(A);
        // undo-annotation: L=(A-K)/M, triggered by 'A'
    }
}
```

Figure 4.5: Example 2, Strength Reduction.

The first thing to notice is that variable ‘L’ has been lost. To be able to save the activation record at the call to ‘print’ using the optimized version, and to restore at another machine using the unoptimized version we need to undo the transformation to recover ‘L’. The compiler therefore inserts an ‘undo’ annotation into the intermediate code of foo after the call to ‘print’. The contents of the ‘undo’ annotation are: when checkpointing variable ‘A’ apply the transformation (A-K)/M (the inverse of A=(L*M)+K) and output the result with the descriptor string of ‘L’. When generating the checkpoint function for ‘print in foo’, the compiler looks if there are any undo annotations for that call-site and invokes the runtime system to write the tuple \{UDS(L), ((A-K)/M, int)\}.

**Maintaining Consistency**

Whenever a thread migrates to another machine, the consistency information for each object that the thread has already accessed, must be shipped as well. As our consistency protocol is modeled after Java’s memory model [120], each usage of an object (detected by the aforementioned access checks) causes a reference to that object to be added to that thread’s working memory set. At a lock/unlock, that set is flushed: after examining the sharing state of the object the according protocol action (home-migration, read-only replication, a send of a patch/diff to the object’s home-node, etc.) is performed.

Migration a thread without migrating its working memory is flawed since changes already made to the data by the thread before migration won’t be visible at the target machine. The naive solution to ‘flush’ the thread’s working set before performing the thread migration does not work when the compiler’s access check elimination is used. Although conceptually there is an access check in front of every single object access, access checks in front of objects that are known to be already available can be removed for efficiency.

For example, see the code in Figure 4.6. Ignoring statement 3, the optimizer will remove the access check in statement 4 as the object is obviously already available due to the access check in statement 1. Consider what happens if statement 4 is removed. After thread migration, the object will not be faulted in at the new machine causing a potentially old object to be used, which may produce incorrect results.
access_check write: o;
o.field = 5;
perform-thread-migration;
access_check write: o;
o.field = 6;
flush;

Figure 4.6: Example 3, Access check optimization problem.

Since the perform-thread-migration statement will only sporadically migrate a thread, it is advantageous to allow the compiler to optimize for this (normal) case. As shown above, in case of thread migration, the current state of 'o' needs to be brought into the memory of the target machine. The solution we have taken is to perform a 'flush' at the source machine, to append all the working memory references to the thread’s migration message, and to perform all access checks again at the receiver side to fault everything, object-for-object in at the target machine.

4.4 Heuristics

Any thread migration implementation needs two heuristics: one to determine when to perform thread migration and another to determine where to migrate to.

Our solution is to optionally let the application determine both when and where to migrate a thread to. To each thread we associate an instance of class MetaThread that implements two methods: 'shouldMigrate()' and 'getMigrationTarget()'. Method 'shouldMigrate()' returns a migration urgency. An urgency of '0' means that no migration is needed. The higher the urgency the more need is indicated. These methods are invoked in response to a thread invoking migrate_this_thread(). If 'shouldMigrate()' indicates a need for thread migration, 'getMigrationTarget()' returns the machine number of where to migrate to.

It is the responsibility of the meta-thread implementation to ensure that not all threads make the same decision to move to some empty machine at the same time.

class FastMigrateMetaThread extends MetaThread {
    /** should we migrate now to another machine ?
     * returns urgency wherewith ('0' means no migration)
     */
    public int shouldMigrate() {
        return 1;
    }
    /** where should we migrate to */
    public int getMigrationTarget() {
        int me = RuntimeSystem.machineNr();
        int num = RuntimeSystem.nrMachines();
        int to = (me + 1) % num;
        return to;
    }
}

Figure 4.7: Example MetaThread.

A simple meta-thread that performs a round-robin style thread migration over a number of machines is shown in Figure 4.7. The shouldMigrate() method is called every N milliseconds (1000 by default), by the local machine’s MetaThreadManager
4.5. RELATED WORK

Related work can be roughly divided in two classes: those that use a compiler and those that implement their checkpointing/thread-migration algorithm inside a library/operating system.

A popular approach to checkpointing is to supply a library that saves and restores an entire process (see www.checkpointing.org). One such library is libchkpt [95] for user level checkpointing. Note that pure OS or library approaches do not support heterogeneity due to loss of information about types.

There are several systems that use a compiler to either optimize checkpointing performance or to enable heterogeneity.

Jessica2 was extended with thread migration in [130]. Here, a Java-JIT compiler was extended to record which code transformations were made during compilation. Checkpointing, uses a form of ‘play backward’ code generation: code transformations are replayed in reverse order starting at the generated machine code including renaming of registers etc. Machine registers are thus slowly transformed back to bytecode level (stack) variables which are then used for thread migration. However, Jessica2 does not support heterogeneity (x86 only).

The problem with this approach is that all compiler transformations need to record their influences on the local variables involved. Each optimization pass should recorded its effects to allow playback, while in our approach, all high-level optimizations are left untouched. Also, the decision of when to migrate is a fixed heuristic instead of our more flexible meta-thread approach.

Bouchenak et al. [17] created a system for JavaThread serialization based on de-compilation for Java-JITs. They checkpoint the Java operand stack in the interpreter because at that stage all of the thread’s state information is available. JIT generated code is decompiled to the same format that the interpreter would use to ensure that the checkpointer only has to deal with the Java operand stack. However, with increasing levels of optimization the process of decompilation can be expected to become increasingly difficult.

Dome [10] and Porch [12] create preprocessors for C++ (Dome) and C (Porch) programs to allow heterogeneous checkpointing. The preprocessor instruments each function of a program. Tests are added at each procedure to tests whether to perform restoration or not. Each variable modification is shadowed to a separate stack for checkpointing purposes. However, the added code causes substantial overhead during normal program execution.

Tui [108] is a system to provide checkpointing using the ACK [115] compiler kit. Each compiler pass is modified to maintain source code information. ACK does not perform much optimization (only peephole and instruction selection) which makes it easier to maintain source code information. If more optimization would be added to their system, the simple mapping of an activation record they use will no longer work.

PREACHES [110] offers heterogeneous checkpointing by creating checkpoints suitable for each architecture that the user might wish to restore on. Periodically, the state of a “master” process is sent to a number of “slave” processes for translation to
CHAPTER 4. HETEROGENEOUS THREAD MIGRATION

Each slave’s native data format. The problem is that always a slave of each different architecture needs to be available.

Plank et al. [96] use a compiler to exclude certain memory ranges from the checkpoint to reduce the amount of memory to be checkpointed with the aid of programmer supplied annotations. They, however, do not supply either heterogeneity or overhead-free checkpointing.

4.6 Performance

In this section we examine the performance of thread migration using our DSM system on simple micro-benchmarks and applications. In all the test programs below, we force the need for thread migration by always starting all threads at the same machine, e.g., a main loop spawns all threads.

All modifications made to the (standard) Java programs are the insertion of two lines: one to specify which meta-thread to use (at the start of the thread’s ‘run()’ method), and another (optional) line to specify where to ideally perform thread migration.

We use two x86 machines, each a dual processor machine running at 2.4 Ghz (32 bit, 6 integer and 8 FP registers), four AMD64 machines, running at 2 Ghz (64 bit, 16 integer and 16 FP registers), and finally, we use two IA64 machines (Itanium 2) running at 1 Ghz (64 bit, 128 integer and 128 FP registers). All machines are connected by Gbit-Ethernet.

Ping-Pong. If, at any time a load imbalance occurs it is imperative that a thread can be moved away at high speed.

We will first examine a micro-benchmark with a program using a single thread that migrates as often as possible between two machines. One machine is fixed at x86 while varying the architecture of the partner. Note that, fixing one partner bares no influence on the further experiments as we use a universal wire format. For this micro-benchmark, we have used the meta-thread shown in Figure 4.7.

The call to ‘migrate_this_thread()’ has been inserted manually at the top of a recursive ladder, the depth of which can be varied with a command-line parameter. In this manner, we can control the number of stackframes saved inside a single migration message. With a depth of ‘1’ only the stackframe of the Thread.run() method is saved and the overhead is minimal. The migration costs when using different stack-sizes and combinations of machines of different architectures is shown in Table 4.2. This table shows the costs of a single checkpoint (averaged over five runs).

Each activation record contains an ‘int’, a pointer to a PingPong thread-object, and some identification information to determine the checkpointer/renoster for that activation record. The timings in the table do not include the network wire time.

Traveling Sales Person (TSP) solves the problem for a 17-city input set. First, machine zero creates a distance table (holding the distances between each city) and a queue of partial path ‘jobs’. Next, each worker thread steals partial paths from that
4.6. PERFORMANCE

Table 4.2: Ping-pong migration costs

<table>
<thead>
<tr>
<th>#Activation records</th>
<th>Save (milliseconds) x86</th>
<th>Restore (milliseconds) x86</th>
<th>Message size (bytes) x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>1</td>
<td>0.29 2.54</td>
<td>911</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.31 2.55</td>
<td>3.131</td>
</tr>
<tr>
<td>10000</td>
<td>36.03 32.16</td>
<td>741.874</td>
<td></td>
</tr>
<tr>
<td>AMD64</td>
<td>AMD64 AMD64 AMD64 AMD64</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.16 2.01</td>
<td>911</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.23 2.17</td>
<td>3.131</td>
</tr>
<tr>
<td>10000</td>
<td>15.61 24.82</td>
<td>741.874</td>
<td></td>
</tr>
<tr>
<td>IA64</td>
<td>IA64 IA64 IA64 IA64</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.18 4.68</td>
<td>911</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.36 5.44</td>
<td>3.131</td>
</tr>
<tr>
<td>10000</td>
<td>54.77 260.13</td>
<td>741.874</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: TSP migration costs

<table>
<thead>
<tr>
<th>Application</th>
<th>Runtime (seconds)</th>
<th>Message size (KBytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSP / 1 machines / no migration</td>
<td>175.714</td>
<td>0</td>
</tr>
<tr>
<td>TSP / 1 machines / migration</td>
<td>175.933</td>
<td>0</td>
</tr>
<tr>
<td>TSP / 8 machines / migration</td>
<td>15.932</td>
<td>25.4</td>
</tr>
</tbody>
</table>

All workers are initially created at machine zero. Thread migration is advised by a generic meta-thread that attempts to use as many available processors as possible. On a cluster with $N$ machines, this causes exactly $N - 1$ thread migrations.

Table 4.3 shows the relevant statistics for TSP. Note that the runtime column is for end-to-end wall time for the application. The difference between the no-migration and migration rows using a single processor, is that with the no-migration rows no ‘migrate_this_thread()’ calls are inserted and the meta-thread-manager thread is disabled. However, as the meta-threads code is very simple (and executed in a different thread), there is little impact on performance.

Each migration causes 25 KBytes of data transfers. The migration data constitutes of only a few activation records (2 per migration), almost all of the message size (24 KBytes) consists of consistency information (see Section 4.3.4). This is caused by the thread-restart process which immediately tries to get read and write access to all the objects that it had read and write access to before performing the thread migration. The amount of data is low as the thread migration happens soon after program startup and the threads haven’t had much opportunity to cache a lot of objects from other machines. The migration latency for a TSP thread averages to 0.21 seconds which includes the mapping of all the data that the thread used at the source machine.

Successive Over-Relaxation (SOR) is a well-known iterative method for solving discretized Laplace equations on a grid (from the JavaGrande benchmark suite [107]). In SOR, 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 $2048 \times 2048$ (32 Mbyte) matrix. Table 4.4 shows the relevant statistics for SOR. Again, the runtime column is for end-to-end wall time for the application.
Table 4.4: SOR migration costs

<table>
<thead>
<tr>
<th>Application</th>
<th>Runtime (seconds)</th>
<th>Message size (KBytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR / 1 machines / no migration</td>
<td>35.43</td>
<td>0</td>
</tr>
<tr>
<td>SOR / 1 machines / migration</td>
<td>35.63</td>
<td>0</td>
</tr>
<tr>
<td>SOR / 4 machines / migration</td>
<td>20.11</td>
<td>64.4</td>
</tr>
</tbody>
</table>

Table 4.5: ASP migration costs

<table>
<thead>
<tr>
<th>Application</th>
<th>Runtime (seconds)</th>
<th>Message size (KBytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP / 1 machines / no migration</td>
<td>30.23</td>
<td>0</td>
</tr>
<tr>
<td>ASP / 1 machines / migration</td>
<td>30.33</td>
<td>0</td>
</tr>
<tr>
<td>ASP / 4 machines / migration</td>
<td>17.14</td>
<td>38.2</td>
</tr>
</tbody>
</table>

Each migration causes about 64 KBytes of data transfers. The migration data constitutes only a few activation records (2 per migration). Almost all of the message size (59 KBytes) is due to consistency information (see Section 4.3.4).

Like TSP, a simple meta-thread suffices to distribute threads over CPUs. The migration latency for a SOR thread averages to 0.13 seconds which includes the mapping of all the data that the thread used at the source machine.

All-pairs Shortest Path (ASP) 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 read row \( k \) of the matrix and use it to update their own rows. Table 4.5 shows the relevant statistics for ASP. Again, the runtime column is for end-to-end wall time for the application. Like in SOR, nearly all data inside a migration message is consistency information.

Like SOR and TSP, a simple meta-thread suffices to distribute threads over CPUs. The migration latency for an ASP thread averages to 0.19 seconds again including the mapping of all the data that the thread used at the source machine.

4.7 Conclusions and Future Work

We have described a system that strictly separates the mechanics of thread migration from its heuristics. The mechanics is based on a unique way to encode variable usage causing nearly no runtime-overhead in the case that no thread migration is performed. In our system, per-thread and per application migration policies can be selected. To reduce the amount of work needed to make a program aware of thread migration, a number of simple thread migration heuristics are pre-programmed. Combined, our benchmarks show that our system is fast enough to recover from a worst case thread distribution scenario.

The system as presented here allows a lot of flexibility in choosing thread migration heuristics. One question that this opens-up is if it is possible to create an advisory program that, when presented with a communication/computation/resource profile of a run of a program, can suggest which of our preprogrammed meta-thread classes to use for best performance.

One problem of the mechanics as presented here is that the size of the program grows with the number of call-sites in the program. The growth is tempered with compiler
analysis to skip parts of the call-graph not containing a migration point as migration can never occur from inside those parts of the callgraph. This can potentially be improved by generating tables instead of methods for each call-site.
Chapter 5

An Automatic Cost-based Framework for Seamless Application Migration in Grid Environments

Authors:
Michael Klemm, Ronald Veldema, and Michael Philippsen
University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3 • 91058 Erlangen • Germany
veldema@cs.fau.de, philippsen@cs.fau.de

Published in:
Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems
Conference: PDCS ’08
Orlando, Florida, USA
November 16 – 18, 2008
Pages 219 – 224
ISBN 978-0-88986-773-4
Publisher: ACTA Press, Anaheim, CA, USA
Editor: Teofilo F. Gonzalez

Chapter 3 described the mechanism that allow migration of a DSM program running on one cluster to another cluster, it did not implement the heuristics of when to move and where to move to. For the paper of Chapter 3, the copying of the checkpoints between clusters was manually performed and the program restarted with a new number of machines using command line flags. In this chapter’s paper we introduce a new grid computing infrastructure that performs all these tasks automatically. It also provides a
grid-wide file system and standard I/O is transparently managed.
5.1. INTRODUCTION

Abstract

Grid computing promises to seamlessly deliver computing power to computational scientists. However, scientists are faced with the problem of finding computing resources for their application and they have to cope with the systems’ batch operation mode that causes premature termination of applications if they exceed the reserved resource limits.

In this paper, we propose a middleware layer (OGRE) for a seamless Grid environment. OGRE automates the process of resource discovery and performs resource selection. Resource selection uses a cost-based migration strategy that allocates free computing resources on accessible clusters, i.e., OGRE automatically checkpoints, transfers the snapshot from one cluster to the next, and restarts the application there. A Grid-wide file system handles data accesses of migrated applications. OGRE can easily be adapted to new requirements due to its component-based architecture.

5.1 Introduction

Seamless and transparent Grid computing is still far away. Scientists still have to deal with the batch operation of their accessible clusters. They must submit jobs to cluster queues and request a fixed number of machines for a fixed amount of time. Jobs are prematurely terminated when they run longer than reserved or when nodes go down because of failures, software upgrades, etc. Therefore, scientists add periodic checkpointing [101] so that jobs can be restarted from a snapshot taken just before a job’s unplanned termination. However, sophisticated checkpointing complicates the code and distracts from doing science. Moreover, snapshots can often only be restarted on the particular cluster and node count they have been taken on. Hence, today’s Grid computing landscape is far from seamless.

This work proposes a novel solution for seamless Grid access. Our Open Grid Research Environment (OGRE) automatically discovers free resources in the Grid, selects resources for the application, and creates an execution plan for it. Guided by a cost-based model, OGRE can transparently migrate the checkpointed computational state of applications among the available remote Grid resources and thus finds a beneficial way of execution.

For transparent application execution in a Grid environment, OGRE performs five tasks. (1) Resource discovery queries accessible systems and determines free resources on these systems. (2) Resource allocation selects a system and allocates resources by submitting a job to the cluster management system. OGRE provides batch system communicators that interface with the cluster management systems for status queries and job submission when necessary. (3) Checkpointing stores the computational state of the application to disk for transfer. OGRE does not directly support checkpointing, but re-uses existing checkpointing solutions, e.g., [75], [12], or [103]. (4) Migration moves the snapshot to the target system. OGRE handles migration by means of a component for both automatic transfer of snapshots and a Grid-wide File System (GridFS) that spans several computing sites. (5) Restart waits until the reservation on the target system becomes active and resumes the application. OGRE also provides daemons that monitor the application’s execution life cycle.

It is too work-intensive to perform the above tasks manually. This paper shows how OGRE automates these tasks and hides many of the technical issues.
CHAPTER 5. A FRAMEWORK FOR APPLICATION MIGRATION

5.2 Design Objectives

The following objectives govern the design of OGRE.

The entire framework must reside in the user-space, since most clusters disallow to run or install system software, libraries, or modified OS kernels. Hence, OGRE must be self-contained in a single binary that the user starts on a cluster. The user-land approach also solves the authentication and authorization problem between accessible clusters, as OGRE runs under the user’s cluster-local account. Thus, there is no need to establish a global trust relation that is needed by Globus [53], for example.

For portability, OGRE is written in Java. OGRE-managed applications, however, can be written in any language. The application is linked with a small library that communicates with the OGRE server running on a cluster’s front-end. To allow for the Grid’s dynamic nature, the OGRE servers communicate using an OGRE-managed P2P network. There is thus no single server and thus no single point of failure that could cause all OGRE jobs to die.

Many clusters in the Grid are not directly connected to the Internet. For example, clusters can be behind firewalls, they use private IP address ranges, or are accessible only through NAT addressing. Fortunately, we can leverage SmartSockets [85] to penetrate firewalls through SSH connects. Problems with private IP addresses and NAT are solved by so-called hubs on the cluster front-end machines. Other solutions, e.g. Juxta-CAT [99], can easily be added to OGRE due to its component-based architecture.

5.3 OGRE: Open Grid Research Environment

This section presents the architecture and the high-level functions of the OGRE framework (see Fig. 5.1).

5.3.1 Architecture

The application adapter is a thin layer needed by OGRE-enabled applications. The framework runs in a daemon on one of the cluster’s front-end nodes. The application adapter establishes TCP/IP socket connections to the application communicator. Messages transferred are either migration-related messages or GridFS-related messages (for protocol details see Sect. 5.3.5).

The P2P communicator handles inter-cluster communication by means of the SmartSockets library [85].

The batch system communicator hides the interfaces of various cluster schedulers behind a common interface. It reads a cluster’s schedule (e.g. parses logs or executes status commands) and reports the reserved resources to the filter and scheduler component. In turn, the batch system communicator receives submissions from the migrator and accesses the cluster’s scheduler to launch an application.

The filter and scheduler receives a cluster’s schedule and analyzes it for free resources (so-called reservation holes). After the scheduler has created a list of reservation holes, it passes the list to the filter sub-component, removing reservation holes not suited for migration (e.g. with a too short time limit). After postprocessing, the cluster’s reservation holes are sent to all active OGRE instances.

When a remote OGRE instance receives the reservation holes (so-called bids), it passes them to the auctioneer to compute an application’s migration plan that describes when and where the application is to be migrated to. Before the migration plan is
computed, the bids are filtered according to application constraints, e. g. a minimal number of required nodes. An auction algorithm (see Sect. 5.3.3) evaluates the remaining bids, selects the subset of reservation holes (from all bids) that constitute the best migration plan, and forwards it to the migrator.

The migrator broadcasts the migration plan to all OGRE instances, which then request the appropriate reservations on their local clusters. Since reservation holes might have been consumed in the meantime so that the migration plan cannot be implemented as planned, OGRE must dynamically deal with early and late reservations.

5.3.2 Application Life Cycle

OGRE fully controls the application’s life cycle, i.e., the transitions between four different states.

An application starts in the submitted state on an arbitrary OGRE instance. That OGRE instance requests bids from all clusters and creates a first migration plan for the application (see Sect. 5.3.3). When the application is launched on the first cluster in the migration plan, its state is changed to running which is broadcast to all clusters.

Detecting that a job is about to exceed the reserved time, the local OGRE daemon asks the application to checkpoint and terminate. OGRE broadcasts the application’s new checkpointed state and waits for the next reservation in the migration plan (or creates a new migration plan if the current plan has been completely consumed). The cycle of transitions between running and checkpointed continues until the application has finished execution.

By explicitly managing application states, we can catch a number of error conditions. For example, if a reservation becomes available early while the application is already in state running, we can cancel the reservation.
5.3.3 Migration Strategy

The migration strategy must find a good tradeoff between the number of migrations and the computing power delivered to the application. To do so, it needs to assess the performance of the individual clusters, to quantify the computing power of a single cluster reservation, and to estimate the benefit of migrations. A graph-based algorithm can then compute a migration plan.

Assessing the Performance of Clusters

Because of the complexity of performance estimation \[6\] \[7\] the accessible clusters. Automatic performance estimation of applications on a given cluster is future work.

OGRE requires a per cluster profile containing the number of nodes in the cluster, processors per node, and a CPU speed indicator. In case of inhomogeneous systems, the user can specify a profile for each sub cluster. The profile can also hold the maximum permissible walltime limits per job for daytime and nighttime.

Quantifying the Performance of Reservations

Based on the simple performance characterization of a cluster, OGRE models the performance of a cluster reservation: The peak computing power of a cluster reservation \(r\) is a 3D cuboid. The length of the cuboid is determined by the walltime limit \(t_r\) of the reservation. The depth corresponds to the number \(p_r\) of Processing Elements (PE); the speed \(s_r\) of a PE is used as the cuboid’s height. The computational power \(P(r)\) of a reservation \(r\) is then determined as the volume of the cuboid:

\[
P(r) = s_r \cdot p_r \cdot t_r
\]

Estimated Benefit Function

While executing the application on a reservation yields the benefit given in formula \(5.1\), checkpointing, and migration cause penalties that reduce the benefit.

If the application migrates locally from one reservation \(r_1\) to another reservation \(r_2\) on the same cluster, we can express the benefit \(B_{\text{local}}\) as:

\[
B_{\text{local}}(r_1, r_2) = P(r_2) - (C_{\text{cp}}(r_1) + C_{\text{rs}}(r_2))
\]

where \(C_{\text{cp}}(r)\) is the cost of checkpointing with respect to a reservation \(r\) and \(C_{\text{rs}}(r)\) expresses the cost of resuming the application from the checkpoint. There is no cost of data shipping, since the application stays on the same cluster.

As the application does not make any progress during checkpointing, \(p_r\) PEs of speed \(s_r\) do not execute the application for the time \(t_{r,\text{cp}}\) that is needed to create the checkpoint. The cost function for checkpointing and resuming the application evaluates to:

\[
C_{\text{cp}}(r) = s_r \cdot p_r \cdot t_{r,\text{cp}}
\]

\[
C_{\text{rs}}(r) = s_r \cdot p_r \cdot t_{r,\text{rs}}
\]

For a remote migration between reservations on different clusters, an additional cost \(C_{\text{mg}}(r)\) reduces the benefit if the target reservation and the source reservation are
active while the checkpoint data is in transit. Again, we can express the cost as a loss of computing power:

\[ C_{mg}(r) = s_r \cdot p_r \cdot t_{r,mg} \] (5.5)

where \( t_{r,mg} \) is the time needed to completely ship the checkpoint data while the target reservation is already active. Hence, the benefit of a remote migration between two reservations \( r_1 \) and \( r_2 \) on different clusters is:

\[ B_{remote}(r_1, r_2) = B_{local}(r_1, r_2) - C_{mg}(r_2) \] (5.6)

### Computing the Migration Plan

From all incoming reservation holes, the auctioneer selects the ones that deliver the highest amount of computing power (according to our quantification) and cause a minimal number of migrations. Computing the migration plan is performed in two steps:

1. constructing the so-called migration graph
2. finding a migration path in the migration graph. The optimal migration path is the longest path (in terms of the benefit function) in the graph.

Each reservation \( r_1 \) is a node that is connected to another reservation \( r_2 \), if \( r_1 \) ends before \( r_2 \) starts. The weight of an edge between \( r_1 \) and \( r_2 \) is the result of the benefit function (see Sect. 5.3.3). For two reservations on the same system, it is \( B_{local}(r_1, r_2) \); otherwise it is \( B_{remote}(r_1, r_2) \). It is easy to create the migration plan by finding the longest path through the graph. The auctioneer then broadcasts the migration plan to all migrators in the framework. This ensures that each migrator knows about all reservations that an application might take during its life time.

### 5.3.4 Grid-wide File System

An application that is migrated to a new cluster may have accessed data on the storage subsystem of the previous cluster. This data must of course be accessible at the target system. The idea to transfer the accessed files to the target system before restarting the application is not viable, as only the OS can identify all open files. Our solution is to use a distributed file system that provides a means to transparently access data located on another cluster.

OGRE provides its own implementation of GridFS for three reasons. First, most distributed file systems require kernel modifications, which we consider unacceptable (see Sect. 5.2). Second, solutions for distributed file systems only provide access to remote data, but do not support global file handles that are still valid after a migration took place. Third, some solutions rely on Globus (see Sect. 5.5), which requires administrative rights for installation.

GridFS is implemented as a component of the OGRE daemon that runs on a front-end node of a cluster. It provides its own directory and data management and implements both a directory server for a global file name space and a data server that accesses the local disks. For fault tolerance and high availability, GridFS replicates data.

In addition, GridFS transparently redirects console I/O (i.e., stdin, stdout, and stderr) to a central console, on a machine designated by the user.

### 5.3.5 Application Adapter

The application adapter of OGRE that has to be included into a migratable application communicates with the application communicator through socket connections. For
any client programming language, there needs to be an adapter in that language. It implements the OGRE protocol to interact with the OGRE framework.

There are three migration-related messages. When a CheckpointSignal is received, the application writes a checkpoint and responds with a CheckpointPrepared message that contains the names of the checkpoint files. The application then waits for a CheckpointCopied message that indicates that OGRE has successfully retrieved the checkpoint data from the local disk and that the application can now safely terminate itself, effectively releasing its current cluster reservation.

There are ten GridFS-related messages: Open, Read, Write, Close, MkDir, ChDir, Rm, OpenDir, CloseDir, and ReadDir. The messages emulate the notion of the Unix system calls and expect the same arguments. For applications that do not need access to GridFS, it is sufficient to use an adapter that only implements the migration-related messages.

5.4 Performance

OGRE’s performance depends on two key performance measures: (a) the performance of GridFS (Grid-wide data access), and (b) the migration strategy that ships applications between clusters of the Grid.

We perform the measurements on four machines: (1) Woody, a cluster with quad-core nodes (2×2 cores, 3.0 GHz, 8 GB main memory, and InfiniBand 4x, located in Erlangen/Germany), (2) DAS3/VU and (3) DAS3/Leiden, two clusters of quad-core Opteron nodes (2×2 cores, 2.4 GHz, 4 GB main memory, and Myrinet 10G) that are located in Amsterdam/The Netherlands and in Leiden/The Netherlands, and (4) Altix, a shared-memory SGI Altix 330 (1.5 GHz Itanium CPUs and 32 GB main memory, Erlangen/Germany).

5.4.1 GridFS

For a distributed file system such as GridFS, two key measures are important: network latency and bandwidth. Latency for name look-ups and metadata updates and bandwidth for large files access.

Table 5.1 shows the single-machine performance of GridFS and the performance of GridFS that spans the Woody and DAS3/VU clusters. No other distributed file system can generally span several Grid sites hidden by firewalls or private address (see Sect. 5.5). In our Grid setup, only Secure Copy (SCP) through Secure Shell (SSH) connections is allowed between the systems.

To measure the latency of GridFS accesses, we use a micro-benchmark that creates many empty files (using Open and Close messages). Table 5.1 shows that latency is only 1.7 times slower when crossing the Internet (148 msec vs. 86 msec). As the ping latency between Woody and DAS3/VU is 12.2 msec, creating a GridFS file is roughly
5.4. PERFORMANCE

Figure 5.2: Runtime of a single iteration of the single-threaded LBM. The job is migrated from cluster to cluster.

ten times slower than a single IP packet, since a file creation involves eight messages (requests and acknowledgments that are forwarded over the P2P network).

Table 5.1 also compares GridFS and SCP when transferring large files. SCP is about two times faster than GridFS, as GridFS uses SmartSockets for transfers, which involves some message exchanges between the application and the GridFS servers. In the 2-machine case, GridFS is only 1.6 times slower, as SCP has to copy the test file two times: (1) from Woody to a public host and (2) from the public machine to the DAS3/VU cluster, as Woody and DAS3/VU are not directly accessible from each other. This indirect transfer is of course transparent to the GridFS user. We trade these performance losses against the higher transparency that GridFS provides for OGRE applications.

5.4.2 Migration

To demonstrate the feasibility of our migration framework, we migrate a compute-intensive Lattice-Boltzmann Method (LBM) application that simulates fluids [127]. The benchmark works on a 3D grid (200 × 200 × 200 cells of nineteen double values). We implemented two versions: a single-threaded version of LBM with a pure-Java binding of OGRE, and a version that runs on top of an OGRE-enabled version of Jackal [121], a distributed shared memory system for Java on clusters, and that is parallelized with OpenMP/Java [74]. To ensure that OGRE frequently migrates between the systems, we added an application constraint to only accepts bids from clusters that have not run the application the last two times.

Fig. 5.2 shows the runtime of the single-threaded version, when OGRE migrates it from system to system. Execution starts at the Woody cluster and proceeds to the Altix in Erlangen. The snapshot size for this migration is roughly 14 MB, as most LBM cells are zero, which enables a high compression ratio. As DAS3/Leiden was filled with jobs, it did not contribute bids that could have been incorporated into the migration plan. DAS3/VU, however, was idle at the time of the measurement. Hence, OGRE reported free bids and the auctioneer computed a migration plan that causes LBM to migrate to DAS3/VU.

The per-iteration runtime of the multi-threaded LBM is depicted in Fig. 5.3. This time the OGRE daemons on DAS3/VU, DAS3/Leiden, and Woody reported free re-
Figure 5.3: Runtime of a single iteration of the OpenMP-parallel LBM. The job is migrated from cluster to cluster.

sources for the application. An application constraint states that at least eight nodes are needed by the application. The application state of the multi-threaded LBM consumes 340 MB because of the DSM state that has to be stored to the snapshot as well as the LBM grid cells.

The runtime peaks are caused by the time needed to checkpoint the application state, transfer it to the target system, and resume the application. The peak includes an average of 16 minutes waiting time in the queues of the target cluster; the time needed for checkpointing, data transfer, and migration is almost negligible.

5.5 Related Work

Related work can roughly be divided into three categories: middleware layers for Grid applications, packages for application migration, and distributed file systems.

Globus [53], UNICORE [68], and Cactus [8] are packages for Grid-enabled applications. Globus is a set of libraries and tools (e.g. for job submission, data exchange, authentication). However, cluster specifics are not hidden from the user. Moreover, Globus requires administrative rights (e.g. to open firewalls) for installation and a global trust relation between accessible systems must be maintained. OGRE transparently utilizes clusters and does not require administrative rights or trust relations. UNICORE is a cluster environment similar to Globus and abstracts from cluster-specific aspects. The user graphically describes job dependencies and UNICORE submits the jobs and transfers the data automatically. UNICORE cannot migrate applications. In addition, GridFS transparently handles file access. Cactus often runs on top of Globus and provides a framework for parallel applications. Similar to our system, Cactus applications can migrate between systems, but suffer from the aforementioned problems of Globus. Moreover, it lacks a migration-aware global file system that OGRE provides.

MOSIX [14], Sprite [41], P-GRADE [76], AMPI [57], [50], and [74] migrate applications between accessible systems. MOSIX and Sprite are limited to single-process applications and only ship processes between homogeneous systems, i.e., systems of the same architecture and operating system. OGRE targets parallel applications that migrate in heterogeneous Grid environments. To solve the problem of open files, MOSIX and Sprite maintain a process stub at the previous system that handles data accesses. In
contrast, GridFS file handles are migration-aware. P-GRADE, AMPI, [50], and [74] support parallel applications and heterogeneous clusters. While P-GRADE enforces its own programming model for migratable applications, AMPI and [50] focus on MPI programs. OGRE is more generic, as it is fully agnostic of the application’s programming model. In [74], OpenMP/Java applications are migrated between clusters; although snapshots are created automatically, the user discovers free resources, manually moves the snapshot to the target, and resubmits the application to the cluster. OGRE fully handles these tasks transparently and fully automates application migration.

OGSA-GFS [32], GFS [56], GPFS [105], Lustre [113], and Gfarm [116] are distributed file systems. In contrast to our GridFS, none of the file systems is migration-aware; we solve this problem with globally unique file descriptors that remain valid after migrations. Moreover, GridFS is a user-space implementation, effectively alleviating the need to modify the cluster’s OS kernels. Finally, GridFS crosses cluster boundaries without the need to directly expose a cluster to the Internet or to soften firewall rules.

5.6 Conclusions

We have presented the OGRE framework that aims to create a seamless Grid computing environment. Using a cost-based migration strategy, OGRE transparently handles resource discovery and resource selection. A user only submits an application to one of the OGRE daemons and the framework automatically migrates it to free computing resources until the application finishes its computation. A Grid-wide file system provides a global name space for files, transparent data access, and migration-aware file descriptors for open files. Our measurements show that OGRE successfully migrates a compute-intensive application between several clusters and completely hides the cluster’s boundaries from the user, effectively creating a seamless computing environment.
Chapter 6

Safe and Familiar Multi-core Programming by means of a Hybrid Functional and Imperative Language

Authors:
Ronald Veldema, Michael Philippsen
University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3 • 91058 Erlangen • Germany
veldema@cs.fau.de, philippsen@cs.fau.de

Published in:
Proceedings of the 22th International Workshop on Languages and Compilers for Parallel Computing
Conference: LCPC ’09
Newark, Delaware, USA
October 8 – 10, 2009
Pages 1 – 15
Publisher: Springer, Lecture Notes in Computer Science

Because C is hard to analyze for either correctness tests or for automatic parallelization, we created a new general purpose language that allows both. The new language is unique in that it cleanly separates imperative and functional code where the functional code allows modifications of heap structures in an observable side-effect free fashion. This allows all non-dependent code to be parallelized (all for statements are automatically parallel)
CHAPTER 6. TAPIR’S MULTI-CORE SUPPORT

Abstract

Current (heterogeneous) multi-core environments such as gpGPU architectures are hard to program with normal imperative and object-oriented (OO) languages. There are two basic problems to tackle: (1) it is too easy to program race conditions and dead-locks with the available synchronization primitives, and (2) these environments do not support (or support inefficiently) the instructions required for efficient execution of OO programs, e.g., because function pointers and pointer arithmetic are lacking.

We address both problems with a new language that comprises both Functional Programming (FP) and OO programming. We solve problem (1) by auto-parallelization in the functional core where all loops and non-dependent calls can be executed in parallel. FP is to be used to write computational intensive code with safe concurrent memory access.

An alternative object model that does neither use pointer arithmetic nor function pointers but smart pointers/proxies (to implement polymorphism) as well as mixins and templates (to implement OO like code reuse) solves problem (2).

To cleanly integrate the two language cores, we propose a new integration model that even grants some restricted ways to access state from within FP mode.

With the new language and prototype compiler we can transparently parallelize code to target both Cuda and multi-core machines (without annotations from the programmer) and obtain good speedups.

6.1 Introduction

There is currently a trend in high-performance computing towards asymmetric multi-processors and application accelerators. However, these are all programmed using Fortran, C, or C-like languages for the most part.

At the low-end, we have the cheap gaming console market using the Cell processor [44]. Here a main-processor is surrounded by a number of vector co-processors, each with some local memory. The Cell is usually programmed in C. In gpGPU computing, GPUs are used to do the work. Here, a GPU has some local memory and a large number of computing elements on board. Larger setups can even combine multiple GPUs on a machine. Frameworks such as OpenCL [73], Cuda [90, 28], and Brook+ [22] allow these cards to be programmed with C dialects.

One could ask why, for example, C/C++/Fortran combined with OpenMP is not sufficient for programming GPUs and Cells. Certainly, this makes programming easier but does not guard against data-races and is only well-suited for shared memory systems. Message passing style programming (MPI) can also be used for programming Cells, but it loses performance in shared memory systems, cannot be used to program GPUs, and is difficult to program with. Functional programming can be mapped to both shared memory and distributed systems but not all programming problems map naturally to its programming style mainly because most FP implementations do not allow any state to be manipulated in functions. To generalize the FP programming model, we require some state to be manipulated. A good language should only allow this in a safe manner.

OO programming greatly simplifies program design. Unfortunately, direct translation of OO code to either Cuda or Brook+ is hard as many required features are not

---

1 A data-race is a situation where one or more threads are writing a variable concurrently while one or more threads are reading this variable.
supported (function pointers, pointer arithmetic, type-casting, etc.). Also, the decision on what can and cannot be run in parallel in an OO code is hard if these language features are used. Worse yet, polymorphism is usually implemented by means of a table of function pointers. Thus, even if the programmer does not use function pointers directly, the implementation uses them indirectly. Data access is hard to get right in OO languages in concurrent context due to the possibility of creating data-races to shared data.

We have designed a new language ‘HpcTapir’ that combines most of the advantages and avoids most of the problems of the approaches mentioned above:

- an OO-like programming model (for ease of programming) but without regular inheritance to avoid hidden function pointers;
- safe data access to make data-races hard to write;
- language level support of auto-parallelization so that the system can run any compute intensive loop in parallel without code or data annotations.
- fast, safe, flexible (multi-dimensional) array access (as much data will still be stored in arrays).

The main advantage of FP programming over OO programming is that any method can be run in parallel to another method (as long as no output-input relation exists between them). The same holds for ‘for’ loops, as each iteration can be executed in parallel without requiring annotations (such as in OpenMP). Because of this, we add an FP core to our language. In pure functional programming, this safety is achieved by not allowing a function to manage state at all (outside of function arguments). Since this is unfamiliar to most programmers, HpcTapir relaxes this restriction and allows state to be manipulated but in a strictly managed way that retains functional transparency. Also, the functional code stays close to the more familiar OO programming style. The OO core is meant for generic programming that does not fit into the FP philosophy. This separation allows the FP code to be strictly functional (meaning that FP methods can have no observable side-effects) while things such as I/O can be delegated to the OO code. Also, because data is strictly separated into read-only and write-only parts (more later), data can be partitioned and replicated without harm.

Many (good) optimizations require a closed-world assumption and for HpcTapir we therefore enforce a closed-world. This means that the entire program (and its libraries) are analyzed, optimized, and compiled in one step. In HPC, dynamic class loading is not an issue.

6.2 Language features

6.2.1 Functional language core

As argued above, functional languages allow all loop iterations to be run in parallel as ‘functional’ means (side-)effect free. Race conditions are thus impossible to write in functional code as no thread can influence another thread. In functional context we allow two kinds of references to objects: mutable and immutable ones (default). An immutable object cannot be changed. A mutable reference in HpcTapir is a tuple consisting of a reference to a read-only copy and a reference to a write-only copy, i.e., there are two objects per mutable reference. One cannot read from the write-only copy.
CHAPTER 6. TAPIR’S MULTI-CORE SUPPORT

As threads can no longer read what another is concurrently writing (it would read from the read-only copy) no data-races can occur in functional code and we achieve functional transparency.

A class becomes functional when marked with the **functional** keyword. A class without the new keyword remains plain OO. Array types are implicitly functional. Because the functional core alone does not create a general programming model, Hpc-Tapir contains both a traditional OO core and the new functional core. In non-functional classes, all data is accessible for both read and write access. In FP-code only functional objects and arrays can be changed that are reachable over references marked **mutable**.

Transitions between FP and OO mode are via method calls. We can therefore distinguish between four cases of method calls:

1. a call from an OO method to another OO method,
2. a call from an FP method to another FP method,
3. a call from an OO method to an FP method,
4. a call from an FP method to an OO method.

Cases (1) and (2) are handled just as in any programming language. Case (3) requires us to (deeply) clone any (functional and non-functional) arguments to ensure that arguments in the FP method are truly read-only and cannot be changed by any OO method that might be running concurrently to the FP method. If an argument of an FP call is a non-functional object, a copy is made and also assigned back afterwards. The last case (4) calls an OO method from an FP context that is potentially already running in parallel. Because the OO code there has no knowledge of parallelization at the FP level, some form of concurrency control (in the current implementation via locks) is generated to sequentialize the call to the OO method. To ensure data safety, no direct field access is allowed from functional to mutable classes. Method calls must be used instead to allow easy automatic lock insertion. Consider the following example:

```java
class Arg {
    int field;
    void inc(int v) { field += v; }
}

functional class FCaller {
    void foo(Arg c, int[S] arr) {
        c.inc(arr[0]);
    }
}

class Init {
    void zoo() {
        Arg a = new Arg();
        int[S] arr = new int[16];
        FCaller c = new FCaller();
        c.foo(a, arr);
    }
}
```

Here the non-functional method `Init.zoo()` calls the functional method `foo()` from `FCaller` that in turn calls the non-functional method `inc()` from class `Arg`. Conceptually, this code is translated to:

```java
class Arg {
    int field;
    void inc(int v) { field += v; }
}

functional class FCaller {
    void foo(Callee c, int[S] arr) {
        // FP to OO (due c.inc):
        // must insert lock()
        system.lock(c);
        c.inc(arr[0]);
        // likewise inserted system.unlock(c);
    }
}
```

Consider the following example:

```java
class Arg {
    int field;
    void inc(int v) { field += v; }
}

functional class FCaller {
    void foo(Callee c, int[S] arr) {
        // FP to OO (due c.inc):
        // must insert lock()
        system.lock(c);
        c.inc(arr[0]);
        // likewise inserted system.unlock(c);
    }
}
```

```java
class Init {
    void zoo() {
        Arg a = new Arg();
        int[S] arr = new int[16];
        FCaller c = new FCaller();
        c.foo(a, arr);
    }
    // OO cloned: re-assign.
    a = clone_a;
    // No reassign of 'arr',
    // since it is functional
}
```
6.2. LANGUAGE FEATURES

Naively, the non-functional Arg does not need to be cloned as it is protected by a lock in FCaller.foo(). However, there could be other user-created threads that modify the object without being aware of any auto-parallelization and hence omit the locking. In general, by creating a private clone before entering functional code, object synchronization is not needed in non-functional code elsewhere. If Arg.inc() were to call a functional method again, the lock would be released and re-acquired around that call to ensure that no dead-locks occur.

6.2.2 Mutable Arguments are created by Flow expressions

As described above, arrays and other functional objects are by default read-only in FP methods. To allow state to be manipulated (the key to performance in all processors), a data declaration must be marked mutable which then names both a read-only copy and a write-only copy of that data. Any read of such a variable/object-field/array-index will read from the read-only copy and all writes will go to the write-only copy.

Like object variables, local variables are read-only. HpcTapir enforces single assignment semantics (by requiring local variable declarations to have an initialization expression and parameters to be read-only), see the following example:

```java
functional class LocalDemo {
    void foo() {
        int tmp = 123; // OK
        tmp = 5;  // illegal
    }
}
```

The semantics of using both read and write copies of data become clearer when using objects or arrays. The original code below on the left makes an assignment to d[0] and then prints a value afterwards. Unlike intuition, this will not print ‘1234’, but the ‘old’ value of the array element. This becomes clearer as we inspect the code transformation that is internally performed as listed on the right: the read-only copy is consulted for the print.

```java
// original source code:
functional class Demo {
    void foo(mutable int[LEN] d) {
        d[0] = 1234;
        debug_print( d[0] );
    }
}
```

```java
// internal representation:
functional class Demo {
    void foo(int[LEN] readonly_d, int[LEN] writeonly_d) {
        writeonly_d[0] = 1234;
        debug_print( readonly_d[0] );
    }
}
```

The only way to create a mutable reference is to pass a reference to a read-only object to the ‘flow’ expression →. Flow expressions are only allowed as arguments to FP methods. The left operand to → is the ‘input’ and the right operand is the ‘output’ of the flow. The output of the flow is the write-only copy of the input after the callee has returned. For example, see the right-hand side of Fig 6.1.

Here, the OO-method User.zoo() allocates an array and calls the FP-method Demo.foo. Because an FP method is called from an OO context we create a read-only copy of ’d’. Due to the → operator, a write-only copy of ’d’ is created as well. Inside Demo.foo() both the read-only and write-only copies are available as the mutable reference. After returning from Demo.foo(), new_d is the name under which the previous write-only copy is read-write accessible (as it is back in OO context). If the → operator is used in FP context, the result (RHS) would be read-only of course. These
class User {
    void zoo() {
        int [LEN] d = new int[16];
        d[0] = -1;
        // foo is functional!
        Demo.foo(d -> new_d);
    }
}

// Internal representation
class User {
    void zoo() {
        int [LEN] d = new int[16];
        d[0] = -1;
        // expanded ‘->’ creates a tuple
        int[LEN] readonly_d = clone(d);
        int[LEN] writeonly_d = clone(
            readonly_d);
        Demo.foo(readonly_d, writeonly_d);
        new_d = writeonly_d;
    }
}

Figure 6.1: Example of a Flow expression

functional class MyThiz {
    int val;
    // read only 'this' coming in,
    // return mutable copy of 'this'
    mutable MyThiz goo() {
        val = 4;
        // no return statement needed
        // mutable 'this' reference
        // coming in
        void bar(mutable this) {
            val++;
        }
    }
}

functional class ThizUser {
    void go() {
        MyThiz x = new MyThiz();
        // call on readonly 'x'
        mutable MyThiz y = x.goo();
        // call on mutable 'y'
        y.bar();
    }
}

Figure 6.2: Mutable this and its use.

parameter mechanics are a restricted way in which FP code can touch state.

At the end of User.zoo(), d[0] is still -1 and new_d[0] will be 1234. Because this semantic is counter-intuitive at first, the compiler will give an error on reading from a mutable. This behavior can be explicitly overridden by adding an ‘!’ after the access, e.g., debug_print(d[0]!).

A (syntactical) problem now occurs if a functional method wants to modify this. Then either (1) the this reference coming in must already be a mutable reference or (2) a write-only copy of this must be allocated beforehand (and returned).

Syntactically, we differentiate between these two cases by giving (1) an explicit mutable this parameter annotation and giving (2) a mutable method modifier. In (2), the function also cannot return a value explicitly (as it is done implicitly). A method in such a context would have two return values: the changed this and the actual return value.

For example, see Fig 6.2. MyThiz.goo changes this (a write to this.val) and is therefore marked mutable with an explicit return type of MyThiz. The write-only copy of this is retrieved in the caller using a plain assignment (on the right-hand side). In MyThiz.bar(), the this reference must be mutable already before calling.

Because the write-only copy cannot be examined (because it is write-only), data-races cannot occur as they specifically require concurrent read and write accesses. Concurrent writes to the same variable are allowed (writing to the same write-only variable), but they indicate a program bug and are not guaranteed to be atomic. Also, concurrent reads access the same read-only copy of a data structure. Deadlocks cannot occur since auto-parallelized functional code does not use locks. The only locks used are those to protect non-FP objects that are used from within an FP context. But these
locks are under system control (and not programmer control) and are made deadlock
free by HpcTapir.

6.2.3 Reduction types

As mentioned above, each iteration in a ‘for’ loop can be executed in parallel in
functional context as no iteration can see any changes made by another iteration (and
local variables are single assignment). Unfortunately, this allows no way to communicate
between loop iterations although this would certainly simplify programming. The
alternative would be to rely on recursion to simulate this (the standard FP solution),
which would be inefficient. For this reason HpcTapir offers reduction types as they
are portable and can be implemented efficiently on both Cuda and regular multi-core
architectures. The order in which reductions are applied is undefined to allow some
cores to apply reduction faster than others.

A reduction type is allocated before a loop and initialized with some value and an
operator type, currently +, −, /, *, max, and min (no user defined reductions
yet). Inside the loop, values are ‘sent’ to the reduction variable using the ⇐ operator.
At the end of the loop, the result of the reduction can be extracted by coercion to its
base type. Reduction variables can only be declared as local variables and form the
only exception to the rule that local variables are single assignment (due to their special
semantics). Reduction variables are write-many-read-once variables, and cannot be read
from (compiler enforced) after that single read.

In the following example, a new reduction variable \( m \) is declared and initialized with
zero. Then, in the (potentially parallel) loop, we iterate over all array elements and send
each element to the + operator (with the old value of \( m \)) and assign the result back to \( m \)
each time. The system guarantees that the assignment to \( m \) is concurrently safe. At the
end of the loop, we cast the reduction type to a plain integer and return it.

```java
functional class Reducer {
    int get_max(mutable int[LEN] arr) {
        int<=> m = 0;
        for (int i<LEN> = 0 to LEN) {
            int v = arr[i];
            m <= v;
        }
        return int(m);
    }
}
```

6.2.4 Support for OO-style polymorphism

As argued above, regular OO inheritance is not optimal for performance and is not
well-suited for some parallel hardware. Therefore, HpcTapir does not sport inheritance.
Instead, we offer proxy classes that can be used to express polymorphism but without
the disadvantages of regular inheritance.

Proxy objects are interfaces to other objects. A proxy dynamically tests what object
it is proxying for and dispatches to its methods. It is important that proxies can be
implemented without function pointers. Hence, there is no need for clever compiler
analysis to allow method inlining (which can trigger other optimizations in turn).

In Fig. 6.3 there is a proxy class \( A \) and two classes \( B \) and \( C \) that can be proxied by
\( A \). The method \( \text{foo} \) in \( A \) is abstract; \( B \) and \( C \) provide concrete implementations. Later,
the two proxy objects \( a1 \) and \( a2 \) are created. If \( \text{foo} \) is called on them, the call is
forwarded to the proxied \( \text{foo} \) of the proxied object \( a \) and \( b \), respectively.
CHAPTER 6. TAPIR’S MULTI-CORE SUPPORT

// HpcTapir source code:
proxy class A {
    void foo(); // abstract
}

class B proxy<A> {
    void foo(); // concrete
}

class C proxy<A> {
    void foo(); // concrete
}

B b = new B();
C c = new C();
A a1 = new A(b); // proxy to b
A a2 = new A(c); // proxy to c

// calls b.foo()
// calls c.foo()
a1.foo();
a2.foo();

Figure 6.3: A proxy class and proxied classes (left) and its semantics (right).

mixin class A {
    // Needs to be implemented by a class that mixes this in.
    mixin void foo(); // abstract
    void zoo() { foo(); }
}

class B mixin<A> {
    void foo() {}
}

Figure 6.4: A mixin class and mixee.

The call is implemented using a switch statement over the type-ID of the proxied class. Polymorphism is reclaimed as a proxy can 'point' to differently typed objects at run-time.

It is possible to use a fixed switch statement here because HpcTapir explicitly uses a closed-world compilation model which allows us to scan the entire program for proxied classes and generate such a fixed switch statement. With regular inheritance this is not possible for polymorphic calls in general (even with a closed world assumption).

Note that both implementations of foo (i.e., all proxied objects) can be inlined into the switch because the concrete implementations are not hidden behind function pointers.

6.2.5 Support for OO-style code reuse

The other purpose of inheritance is code reuse. Code reuse in HpcTapir is via templates and mixins. Templates are parametric classes as known from, for example, C++. A list < int > declaration, for example, instantiates a new type with a new unique name with its generic type variable replaced by 'int'.

A mixin is an incomplete class that requires some extra methods to become a full, compilable class. For example, see Fig. 6.4. Here, the methods of A are added (verbatim) to B. As mixin method A.foo is abstract, mixees must implement it.

6.2.6 Multi-dimensional arrays with named dimensions

Other sources of overhead in modern managed languages such as Java and C# are array bounds checks and the lack of multi-dimensional arrays. Array bounds checks are very convenient if only they were virtually free at run-time. HpcTapir therefore uses index types to be able to perform most bounds checks at compile time. In HpcTapir, to access dimension X of an array[., X, .], an index type tagged with X must be
6.3. IMPLEMENTATION

A cast of an 'int' to an index type 'X' requires a range cast. Type equivalence of multi-dimensional arrays is then via structural equivalence.

For example, the following code:

```java
int[P, Q] arr = new int[1024, 64];
int z = ...;
int x<P> = range(P, z);
int y<Q> = range(Q, 123);
arr[x, y] = 1234;
```

declares a 2D array with new variables P and Q as its dimensions. To access array `arr[z, 123]`, the 'int' variable 'z' and the value '123' must first be cast to the range of P or Q, resp.

6.3 Implementation

In functional classes, conceptually all non-dependent statements and all for-loop iterations could be run in parallel. Of course, the question is which part of the available parallelism is beneficial to actually exploit to gain performance, given that parallelization setup and tear-down overheads exist. In the current prototype, a heuristics decides if a loop warrants parallel execution. The programmer can override the (conservative) heuristics using the `par` keyword: `par for` and `par { .. }` will force parallel execution.

The heuristics works bottom-up and assigns a benefit value to all expressions, for example, a multiply gets a 3, a division gets a 4, addition gets a 1, etc. The benefit value of a block of statements is the sum of its constituents. An 'if' statement returns the maximum benefit value of both branches. A 'for' statement multiplies the benefit value of its body by some constant.

A 'for' statement is executed in parallel if a benefit threshold is reached. The threshold is different for the parallelization method(s) requested. Currently, the programmer can request two parallelization methods from our compiler: threaded execution or execution using Cuda for graphics platforms.

6.3.1 Cuda

With Cuda a function can be invoked a number of times in parallel on a GPU. To parallelize a loop, we must therefore move the loop’s body to a new function and, in the old loop’s place, insert code to request the Cuda runtime to call this function once per iteration in parallel. The arguments (and objects reachable from them) must be copied explicitly to the GPU beforehand and need to be copied back afterwards.

Execution of the parallel calls is by execution of a number of threads in parallel. The threads are organized in groups. Thread groups execute in SPMD fashion and can share a little bit of shared memory among them for fast intra-group communication. We use this small shared memory to implement fast reductions (see Sec. 6.2.3).

If Cuda code generation is enabled and if the loop-parallelization heuristics suggest parallelization, we attempt Cuda code generation. Unfortunately Cuda hardware is not (yet) always IEEE compliant. Also deep call stacks are not possible (so deep recursion is not possible). The HpcTapir compiler therefore examines loops to disable Cuda code generation for Cuda-unfriendly loops. If all goes well, we splice the loop body out to a new function and internally mark it as 'Cuda'. For example, the left functional loop is translated to the (pseudo) Cuda code on the right:
6.3.2 Threaded execution

Besides generating Cuda, HpcTapir can also target the main processor’s cores as long as the heuristics enable loop parallelization. If parallelization is warranted, we perform a blocking transformation and tag the outer loop with "parfor" so that we know which loop to run in parallel afterwards:

```java
int foo() {
    int m = 0;
    parfor (int i<LEN>=0 to LEN by 16) {
        int v = arr[i];
        m <= v;
    }
    return int(m);
}
```

In a second pass, we lift the new inner loop into a new method of a new class that is proxyable by a Runnable proxy. Any work stealing algorithm can then be applied to support various heterogeneous multi-core/multi-machine architectures. From the example above we effectively generate:
// generated:
int foo() {
    int<max> m = 0;
    ParLoop p;
    for (int i<LEN>=0 to LEN by 16) {
        p = new ParLoop(i, m, arr);
        Runnable r = new Runnable(p);
        system.invoke_parallel(r);
    }
    system.wait_invoke_parallel();
    m = p.m;
    return int(m);
}

At program start a number of work-stealing threads are started to steal jobs from the system. Each of those will invoke the run method of ParLoop in parallel.

6.3.3 Array Implementer Classes

There are multiple ways in which a (multi-dimensional) array can be implemented. For example, a multi-dimensional array can be implemented sparsely, or using arrays-of-arrays, or flattened (using arithmetics to map all the indexes to a single index). Besides differencing layouts, array memory management can be varied as well. For example, the array could be allocated in the gpGPU’s memory or in main-memory, or (parts of) the array could be cached on the graphics card side.

To allow the programmer to influence array memory management, we propose that array types can be annotated with an implementation type. For example:

```plaintext
float [N,M] arr<Cuda2DF> = new float[16,32]<Cuda2DF>;
arr[0,0] = 1.23f;
```

This means that all manipulations of the array type are via the Cuda2DF class. This class has get/set methods to access elements of the array. Internally in the compiler, the above is rewritten to:

```plaintext
Cuda2DF arr = new Cuda2DF(16,32);
arr.set(0, 0, 1.23f);
```

Because data access by the main CPU is necessarily different from accesses by the GPU, we need to generate different codes for the getters and setters based on where they are used.

For example, the above Cuda2DF will initially allocate the array on the GPU. The main-CPU versions of the get/set methods maintain a cache of array elements to avoid copying the entire array to main memory. The frontend versions of the get/set methods have direct access to the backing array and are inlined for extra performance.

Using array-implementer annotations is different from C++ style operator overloading of [] in clearer syntax although the same effect is achieved. Note that the annotation is part of the type so that different annotations make type assignment incompatible.

6.4 Performance

We test performance on a number of synthetic micro-benchmarks and a few application benchmarks. The machine used for testing is a Quad Core Intel (2.5 Mhz, x86-64, 4 GByte memory, 12 MByte cache, g++ (GCC) 4.4.0) equipped with a Quadro FX 3700M graphics card with Cuda 2.2.

For actual code generation, the current prototype generates C++ code for final compilation to machine code. This allows quick prototyping of new language features and the use of standard messaging libraries, computational libraries, etc., but performance
relies on the C++ compiler. Note that the current implementation does not yet allow both Cuda and the main cores to process the same loop at the same time.

### 6.4.1 Synthetic micro-benchmarks

To illustrate the performance benefits of proxies versus inheritance, we call a proxied method \(2 \times 10^9\) times in a loop and compare that to doing the same in C++ using inheritance (where the method is then marked ‘virtual’). The results are illustrative: 4362 ms for C++ versus 834 ms for the HpcTapir version (same C++ compiler used as back-end). The performance increase is solely because of the increased ability to inline.

To illustrate the gains of using multi-dimensional arrays (including our range tests in the type system, see Sec. 6.2.6) we compare against Java 1.6 which uses dynamic bounds checking and arrays-of-arrays. In this micro-benchmark we increment all elements of a 3D array of integers a number of times. For this benchmark HpcTapir requires 451 ms while it takes Java-1.6 739 ms to finish.

### 6.4.2 Matrix Multiplication

Our matrix multiplication is a very simple standard implementation:

```cpp
functional class multiplier {
    void par_mul2(float[N,N] a, float[N,N] b, mutable float[N,N] c) {
        for (int x = 0 to N) {
            for (int y = 0 to N) {
                float<+> sum = 0.0f;
                for (int k<N> = 0 to N) {
                    float f = b[y, k],
                    float g = a[k, x];
                    sum += (f * g);
                }
                c[y, x] = float(sum);
            }
        }
    }
}
```

In plain-OO context, we allocate and initialize the matrix and then pass control to a functional class to perform the actual computation. The functional method takes two input matrices and produces a result matrix. The innermost loop uses a reduction variable. We must use a reduction variable here because a normal ‘float’ variable to sum the product would be a write-once variable that cannot carry loop-dependencies.

In the first step, the HpcTapir compiler flattens the outer two loops to a single loop:

```cpp
parfor (int i = 0 to sqSize) {
    int x<N> = range(N, i % N);
    int y<N> = range(N, i / N);
    ...
}
```

This single outer loop then becomes the target for Cuda code generation or threaded code generation (depending on which parallelization method is compile-time enabled).

<table>
<thead>
<tr>
<th>Program Version</th>
<th>Matrix Multiplication (seconds)</th>
<th>LBM (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential-Tapir</td>
<td>26.4</td>
<td>Sequential-Tapir</td>
</tr>
<tr>
<td>Threaded-Tapir (4 cores)</td>
<td>15.0</td>
<td>Threaded-Tapir (4 cores)</td>
</tr>
<tr>
<td>Cuda-Tapir</td>
<td>5.1</td>
<td>Cuda-Tapir</td>
</tr>
<tr>
<td>Cuda-native</td>
<td>0.9</td>
<td>Cuda-Tapir, w/ArrayClass</td>
</tr>
</tbody>
</table>
For the measurements, we use 200×200 matrices. As Table 6.1 shows, the threaded version gains some speedup on 4 cores. The bottlenecks are the overheads of starting and stopping threads and of managing the shared job-queue. The Cuda-Tapir version gains an impressive speedup over the plain threaded version.

A carefully hand-optimized native implementation of matrix multiplication in Cuda, is faster because it uses the fast on-chip memory to cache matrix elements and uses a blocking implementation. Although it is known to be important [106], HpcTapir does not yet exploit fast on-chip memories. A back-end Cuda optimizer is needed to reach hand-optimized performance.

### 6.4.3 LBM

LBM [34] is a 3 dimensional implementation of the Lattice Boltzmann Method for doing fluid-flow simulations. Internally the application uses a functional class for doing the actual computation. The functional code is called once per iteration. A flow-expression is used in the main iteration to create a mutable reference to the output data of that iteration. LBM uses two data structures, a read-only array of booleans that tells if position (x, y, z) contains an obstacle and a mutable array of cells. We use 64 x 64 x 128 cells (with each cell containing a 19 element array) with 16 iterations. Because every iteration transitions from mutable to functional context, the data needs to be cloned. This causes severe copying overheads.

As Table 6.2 shows, the threaded LBM gains some speedup (approx. 1.8) on 4 cores. Without the array package, Cuda-Tapir sees some slow-down because almost all time is now spent in copying the data to and from the GPU’s memory. Only with the array classes do we gain good performance since most data transfers can be avoided as data remains on the GPU at all times.

### 6.4.4 Computational network

A computational network is a directed graph. Each node computes a function over its inputs and stores the result as its output. In this benchmark there are four types of nodes. The nodes’ functions are very simple: sum, xor, bitwise and, and a multiplication of all input bits. Each node has two input edges from randomly selected other nodes. Each type of node is proxied by a proxy class to allow polymorphic programming; we can use a single array of references to proxies to the various types. Note that a normal OO implementation would fail to compile on Cuda (no function pointers).

In this benchmark we examine how well Tapir style objects are managed. This benchmark creates $2^{20}$ graph nodes where each graph node consists of four objects: a proxy object to a node object (four classes), a node-state object (mutable), and an array of references to other nodes.

<table>
<thead>
<tr>
<th>program version</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential-Tapir</td>
<td>2.7</td>
</tr>
<tr>
<td>Threaded-Tapir (4 cores)</td>
<td>1.8</td>
</tr>
<tr>
<td>Cuda-Tapir</td>
<td>3.6</td>
</tr>
</tbody>
</table>

Even for this more complex polymorphic code, HpcTapir can generate parallel code. The threaded version gains a speedup of 1.5 (limited because of the size of the CPU caches). The Cuda version is slowest because each graph node executes a different function, which causes too much divergence for Cuda’s SPMD processors. Unfortunately, our prototype’s simple heuristics do not exclude Cuda code generation...
here. Cuda execution also requires a copy step of the (many) objects from main memory to Cuda and back which is not incorporated into our Cuda-enabling heuristics. The threaded variant wins as the copy step is avoided and ‘normal’ processor cores can execute the code.

6.5 Related work

While many functional languages, e.g., Haskell [111], strive for functional purity (no dynamic state and no side-effects at all), we argue that the use of mutable state is acceptable, as long as it is managed in a type-safe manner. While effects can be encapsulated into Monads, it is not as efficient as the mutables introduced here. Instead of embedding a functional language into an imperative one, one could also embed an imperative sub-language into a functional one, as done in data-parallel Haskell [79]. The latter only allows simple (data parallel) expressions to be run on the GPU. With HpcTapir one can put arbitrary code on the GPU. Mutable data-structures can also be added to a functional language as for example in Objective Caml (http://caml.inria.fr/) or Id [15]. However, adding read+write variables comes at a cost of a loss of parallelizability and/or analyzability. Tapir seperates read and write variables to mitigate this.

In imperative languages one way of adding parallelization is via annotations, for example, in OpenMP [93, 23, 74]. Here a for-loop is prefixed with an annotation that states the degree of parallelism to use and how to manage the data touched in the loop. However, OpenMP does not guarantee data safety (absence of data-races) where our functional code does so.

Other approaches to new languages for HPC computing focus on programmability (not performance) to quickly get some code running and/or to make very readable high-level code. Chapel [26] has an SPMD programming model (using tasks and parallel-for) but without data safety guarantees. X10 [27] is a Partitioned Global Address Space (PGAs) language based on a subset of Java. In X10, one creates tasks or parallel loops that can access shared data without restrictions and thus can create data-races. Distribution annotations are used to distribute an array over multiple machines. HpcTapir’s array-annotations are more low-level which should give the programmer more influence over data management.

6.6 Conclusions

We demonstrate that a tight integration of FP and OO with well-defined boundaries allows easy parallelization (no data-races, dead-locks, or live-locks). Adding safe data manipulation enables familiar OO-style programming in FP code with good performance (speedup of up-to 15 with Cuda over 4 threads). To target simple, heterogeneous multi-cores such as Cuda, plain inheritance is not well suited. An alternative object model such as the proxies suggested here is required that can deliver good performance and reasonable programmability.
Chapter 7

Evaluation of RDMA opportunities in an Object-Oriented DSM

Authors:
Ronald Veldema, Michael Philippsen
University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3 • 91058 Erlangen • Germany
veldema@cs.fau.de, philippsen@cs.fau.de

Published in:
Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing
   Conference: LCPC ’07
   Urbana-Champaign, Illinois, USA
   October 11–13, 2008
Bd. LNCS 5234
Pages 217–231
Publisher: Springer, Berlin
Editors: Vikram Adve and María Jesús Garzarán and Paul Petersen.

In this paper we increased the efficiency of the Jackal DSM protocol by trying a number of RDMA based DSM protocols. Mainly for object fetch we were able to get good performance increases. We did this paper to see if the Tapir language needed to be extended to allow inter-process RDMA based communication support besides its current asynchronous RPC support.
CHAPTER 7. RDMA BASED DSM PROTOCOLS

Abstract

Remote Direct Memory Access (RDMA) is a technology to update a remote machine’s
memory without intervention at the receiver side. We evaluate where RDMA can be
usefully applied and where it is a loss in Object-Oriented DSM systems. RDMA is
difficult to use in modern OO-DSMs due to their support for large address spaces,
advanced protocols, and heterogeneity. First, a communication pattern that is based on
objects reduces the applicability of bulk RDMA. Second, large address spaces (meaning
far larger than that of a single machine) and large numbers of machines require an
address space translation scheme to map an object at different addresses on different
machines. Finally, RDMA usage is hard since without polling (which would require
source code modifications), incoming RDMA messages are hard to notice on time.

Our results show that even with RDMA, update protocols are slower than invali-
dation protocols. But RDMA can be successfully applied to fetching of objects in an
invalidation protocol and improves performance by 20.6%.

7.1 Introduction

A Software Distributed Shared Memory (S-DSM) system allows for easy distributed
programming by making a cluster seem like a single, big computer. The current
proliference of Java programmers increases the importance of Java DSMs.

Recent cluster interconnects can directly and efficiently read/write another machine’s
memory by means of explicitly programmed Remote Direct Memory Access (RDMA).
Note that an RDMA operation is performed without cooperation from the receiving
machine, except for the initial setup of RDMA-able memory spaces. With Infiniband,
RDMA can be up to 6-10 times faster than send/receive based primitives. For example,
using 3Ghz CPUs, a 1 byte RDMA costs about 2 µs whereas a normal message send
(including protocol processing to deliver the packet to the application layer) takes 17
µs. It therefore seems promising to employ RDMA in DSM systems to implement
memory consistency protocols. However, due to a number of restrictions and the lack of
message receipt notification, protocols can become more complex, so that performance
is reduced.

Java DSMs must implement the Java memory model using some memory consistency
protocol. There are two basic memory consistency protocols: invalidation protocols and
update protocols. In an invalidation protocol, a machine asks a ‘data-owning’ machine
(called the home-node), to send over the requested data (fetch), and caches it locally
until it is invalidated (and sent back, i.e., flushed). In an update protocol, a machine
broadcasts its changes to all (accessing) machines so that a ‘write’ to some data causes
communication while a read of some data will not cause communication. To reduce
communication load, changes can be aggregated for delayed bulk broadcasts till a
later synchronization action. Update protocols can be implemented solely by means of
RDMA, i.e., without any explicit messaging.

Below, this paper investigates the RDMA opportunities in an invalidation protocol
and in two update protocol alternatives.

Our prototype implementation uses Jackal [120, 121], a Java based S-DSM system,
because of its support for pluggable DSM protocols and its simple mark-and-sweep
Garbage Collector (GC) that makes RDMA-based DSMs easier to implement (as objects
do not move during a program’s runtime). Jackal compiles a Java program directly to
an optimized native executable. We currently have code generators for x86, AMD64,
7.2. RELATED WORK

PowerPC, and IA64. Any multi-threaded Java program can as such run without change on a cluster of workstations. Where some DSM systems transfer MMU pages over the network (fixed 1 to 4 Kbyte chunks of memory), Jackal’s granularity is a region: a single Java object or a 64 KByte chunk of an array. This automatic array chunking reduces the possibility of false sharing.

Finally, Jackal is very flexible. It supports a very large virtual address space where each machine adds its memory to the global pool without limits on the number of participating machines. This large address space is one of the main obstacles for RDMA use. To build this large address space, it is necessary to translate object references between machines. This address translation scheme works as follows. At object allocation, each object is assigned a cluster-wide unique Global Object Reference (GOR). When sending an object reference to another machine, that objects’s GOR is sent instead, combined with a type-descriptor structure of the object pointed to. When the target machine receives the GOR, it consults a hash table to see where the object’s local copy is allocated. If no local copy has been allocated yet at that machine, the type-descriptor is used to locate the local machine’s meta-class instance for that object. That meta-class contains enough information (type, size in memory, etc.) to allocate the local copy and store the new pointer in the table. Afterwards the local copy’s address is used.

This scheme therefore has the following characteristics: copies of objects at different machines have different addresses, each reference sent over the network needs to be translated, and finally, copies are allocated lazily so that we cannot be sure that for each object a local copy always exists before sending references to other machines. These features make an RDMA based implementation harder.

7.2 Related work

Both invalidation and update protocols exist in many variations and can operate under different memory models. For an (older) overview of DSM systems see [98].

To our knowledge, only few attempts have been made to use RDMA in a DSM protocol. In each of [91, 45, 69] a page oriented Home-based Lazy Release Consistency protocol (HLRC) is optimized with RDMA. In [91] multiple page diffs (the changes made by the local processor) are sent to their home via RDMA. In [45] diffs are applied by RDMA to the home-node copy. Our system, however, is not page based but object based. Also, we examine the opportunities offered by RDMA for both invalidation and update protocols. In [69] it is investigated how to allow multiple threads per process but by using VIA style network interconnects. Our system can use multiple threads per process as well, but uses different protocols to achieve this.

Other related works compare invalidation and update protocols (without RDMA usage) and program transformations to best utilize them. While the authors of [47] show some performance gains for update protocols, we avoid their extensive manual source-code transformations.

Munin [25] allows the programmer to choose from a time-out update protocol and a number of lazy release protocols. Whereas Jackal allows the protocol to be specified per object, in Munin this can be done per (global) variable. Munin’s global address space is restricted. Both [47] and [25] do not use RDMA based protocol implementations and hence might benefit from the RDMA versions presented here.
### 7.3 DSM protocol template

Java’s memory model prescribes that at the entry and at the exit of a synchronized block, any changes to memory caches made by a thread must be ‘published’ so that other threads can pick up those changes. Besides this, synchronized blocks guarantee mutual exclusion of threads. We implement the mutual exclusion part by sending a message to the owner of the object of the synchronized block and then waiting for an acknowledgement message. The owner will only send this acknowledgement if access to the synchronized block is granted. While an object has a lock associated with it, it is not eligible for home-migration (under invalidation protocols) to avoid having to migrate locking state across machines when migrating an object. Object.wait(), timedwait(), notify(), and notifyAll() are implemented similarly.

Let us discuss the DSM protocol template first that works for both update and invalidation protocols. For concrete invalidation and update protocols different implementations of the three operations: `start_read(Region r)`, `start_write(Region r)`, and `process_cached_regions()` need to be selected. In our implementation, these functions test some flag bits in a region and then pass control to the appropriate protocol handler for that region so that protocols (invalidation or update) can be specified per region.

Our compiler inserts conditional calls to `start_read` and `start_write` into the code. We call these ‘access checks’. For example, the write access to the ‘p’ field at line 4 of the source code on the left hand side of Fig. 7.1 causes the access check in lines 11–12 to be generated. A region is already locally available if the corresponding bit in the thread’s `write_bitmap` is set. Optimization passes in the compiler remove as many superfluous access checks as possible, see [121]. If the region is not yet locally available, the `start_write` and `start_read` functions cause the DSM protocol to fetch a copy of the object. After mapping the copy locally, a reference to the mapped region is added to the thread’s `cached list`. On this list update and invalidation managed regions coexist. Whereas an invalidation protocol fetches a fresh copy of a region after every single thread synchronization statement (as the use of ‘synchronized’ causes data invalidations), the update protocol fetches it only once whereafter it remains mapped. At each `lock/unlock`, the `cached list` is traversed and each region is flushed (invalidation protocol) or broadcast (update protocol). In case of an invalidation protocol, a diff of a modified region is sent to the object’s home-node. If the region was not modified, only a notification is sent that there is now one user less. These messages are used to implement lazy flushing, home-migration, home-only states, etc. Jackal’s invalidation protocol is a multiple-writer protocol with home-migration similar to [48]. It uses lazy-flushing for
cluster-wide read-only regions. In case of an update protocol, a diff is used to update all other existing copies.

As said before, Jackal uses an address translation scheme. Whenever a reference to an object is to be sent over the network, a GOR and some type-information is sent instead which the receiver maps to the local copy causing copies of objects to have different local addresses on different machines. In our prototype, the programmer can select an update protocol per object/array via a simple Java API. This sets the object’s ‘update protocol flag’ and performs an all-to-all communication to exchange the local object addresses. This communication is required for RDMA protocols, as each machine needs to know the remote addresses of all the cached copies of an object. Each of `start_read/write` and `process_cached_regions()` test the object’s ‘update protocol flag’ to determine the correct protocol handler. This all-to-all communication to exchange the addresses of local copies is needed only once at program start and is therefore not an issue for program performance. The following two sections study which parts of the general DSM protocol can/cannot benefit from RDMA.

### 7.4 Object requests by means of RDMA

Whenever an object is not locally available, the access check invokes either `start_read` or `start_write`, depending on the type of accesses that follow. The object is then requested. At receipt of the object by the requestor it is locally mapped by setting the thread’s accessibility bits and by adding it to the cached-list of the requesting thread. This section shows that some parts of the object request protocol can be done by RDMA. Others must rely on regular send-receive pairs.

The fetch request itself cannot be sent via RDMA because of the following reasons. First, with RDMA, the home-node would have to periodically poll memory to determine message arrival. As the home-node will have other Java threads running, the program would need to be instrumented with polling statements. Due to Jackal’s goal of running unchanged multi-threaded Java programs (not necessarily in SPMD style), we cannot require the Java programmers to insert polling statements in their codes. We therefore would have to resort to automatic insertion of polling statements. This causes performance problems as the frequency of polling is either too high or too low, both of which would adversely effect performance. Second, an RDMA-ed fetch request would need to correctly update any protocol state maintained at the home-node. This would involve allocating and freeing data structures from memory, updating the accessibility state vectors, potentially sending invalidation messages, etc. These operations are too complex to manage by using only simple RDMA transfers.

We therefore need to apply a normal send-receive protocol for sending fetch-request messages. Message receipt at the home-node causes a special communication thread (an ‘upcall thread’) to wake up from a blocking-receive. It handles the message and sends the object back to the requestor. In contrast, object receipt by the requestor can be handled by RDMA as the receiver can actively wait for the message to arrive, since the message will (definitely) take only a short amount of time to arrive.

Fig. 7.2 shows the pseudocode for issuing fetch requests. First, the requestor figures out how large the combined received object and its protocol data will be. If this fits in a pre-allocated and pre-registered RDMA-able memory region (`jackal_rdma_alloc`), the fetch request is sent with the address of the local RDMA buffer. Otherwise the object will be sent by the home-node’s communication thread as a normal message. An interesting insight is that the home-node cannot directly place the object’s data into
void fetch_request(javaObject x) {
    // determine the size of the receive area to allocate:
    int reply_msg_size = 1 + x.size() + dsm_protocol_overhead_reply(x);
    int home = x.home_node();
    // allocate a RDMA receive area from the device’s buffer pool
    jackal_rdma_t *rmda_descr = jackal_rdma_alloc(home, reply_msg_size);
    if (rmda_descr) {
        // a suitable RDMA receive area was found,
        byte *end_msg_byte = rdma_descr->memory[reply_msg_size];
        *end_msg_byte = 0;
        send_fetch_request_with_rdma_reply(home, rdma_descr);
        // wait for the RNIC to copy the data in place
        while (*end_msg_byte == 0) {}  
        process_object_reply_message(rdma_descr->local_memory);
        jackal_rdma_free(rdma_descr);
    } else {
        ack_t ack;  // create a condition variable
        send_fetch_request_with_normal_reply_msg(home, &ack);
        // wait for then signal from by process_object_reply_message()
        thread_condition_wait(&ack);
    }
}

Figure 7.2: An efficient way to use RDMA for fast object fetching.

the requestor’s memory because in general the requestor needs to execute additional protocol code upon message receipt. For example, in a situation where there are other threads that are concurrently executing at the requestor and that already had write access to the requested object, tests need to prevent their changes from being overwritten. A naive RDMA-write initiated by the home-node cannot detect such concurrent writes as it would require it to examine the requestor’s states, the (current) requestor’s copy and its twin.

We therefore RDMA the complete reply message in the format of the normally sent protocol message. In other words, process_object_reply_message is always invoked for a fetch-reply message, regardless of whether the message is received normally or via RDMA. The overhead of active polling for the (RDMA-ed) message receipt is acceptable even if concurrent threads at the requestor may be slowed down that way. Note that the active memory polling for message receipt in the RDMA cannot be circumvented, nor can the CPU be freed in the meantime. We can’t free the CPU using thread-yield or sleep statements as either causes slow operating system calls or takes longer than a message latency. It can be argued that polling memory in a tight loop could saturate the memory bus. Fortunately however, the reads from memory in the polling loop run out of the processor’s cache which is updated by a processor’s internal consistency protocols on modern CPUs.

To summarize, for fetching objects, we must send the request as a normal message while the reply message can be sent by RDMA. The reply message, cannot write to the object in place in order to allow multi-threaded execution. Note that our RDMA-rpc implementation is similar to what certain MPI implementations do internally for managing acks. See for example [84].

We will now examine the opportunities of RDMA use (regardless of a performance gain/loss) when we need to update copies of objects on other machines.
7.5 Processing the list(s) of cached objects

Each thread maintains four lists of cached objects, one for machine local read-only regions where some other machine(s) are modifying it (local read-only), one for cluster-wide read-only regions (lazy flushing), one for objects that are used by only one machine (home-only), and one for locally modified objects. At each entry and exit of a synchronized block, all regions on these lists must be examined and processed (except those on the lazy flush list). Depending on a region’s flag, the region is managed by the invalidation protocol handler or by an update protocol handler. This section discusses where RDMA can be used in those protocol handlers.

7.5.1 Invalidation protocol handlers

For each region on the list of locally modified regions we create a diff. These diffs are then streamed to the home-node in 4 KByte packets. By streaming the diffs (instead of buffering them to send them all at once) the home-node can already process incoming diffs while the invalidator still continues to create them. Likewise, for each region on the thread’s read-only list, one-user-less messages are streamed to the home-node. The region is removed from either list as soon as the messages have been sent. The home-only/lazy-flush lists are left alone. Upon receiving a diff or a one-user-less message, a state-machine quickly performs any necessary state changes to implement home migration, invalidation, or read-only replication.

Due the same reasons that prevented RDMA from being applicable to object requests (polling requirements, too complex for RDMA management, etc.), diff messages and one-user-less messages in the invalidation protocol cannot be transferred by means of RDMA either. Hence, invalidation protocols cannot exploit RDMA capabilities when propagating changes.

7.5.2 RDMA-based update protocol handlers

We have developed two alternative update protocols that solely use RDMA. The first one updates remote objects/arrays in place. Diffs between modified regions and their twins are created and applied (by RDMA) to the remote copies at all other machines. This is a true zero-copy protocol performing RDMA from one Java heap to another. The second update protocol stores the above diffs in a large intermediate array first, one per target machine. These arrays are then broadcast via RDMA to all other machines for local processing at their earliest convenience.

For simplicity of our prototype implementation, our current update/RDMA protocols do not allow objects containing reference fields. For such objects the invalidation protocol handler must be used. To illustrate the problem, consider the example in Fig. 7.2. Here we have two machines, 0 and 1. Machine 0 initially holds objects A and B, where A is marked for update-protocol management. The addresses of the copies of A are therefore known at all machines. However the addresses of the copies of B at the other machines are unknown.

Now a thread at machine 0 writes a reference to B in the R field of A. This eventually causes a broadcast of A to machine 1. However, to ensure correctness, the R field of A at machine 1 should be translated to the local copy of B (and potentially allocating the copy of B if it did not already exist). A simple RDMA of A however would not do this, and write a copy of A with an illegal R field. Any circumvention of this problem would no longer make the protocol zero-copy.
void update(Region r) {
    diff_t d = changes to r in respect to twin(r); apply 'd' to twin(r);
    for all machines p:
        int64_t remote_address = r->region_hash[p], remote_twin = r->twin_hash[p];
        RDMA 'd' to 'p' at remote_address and remote_twin;
}

Figure 7.4: Update protocol handler, alternative 1: update in place.

To allow references in update-protocol managed objects the best solution would be to create a copy of the object in RDMA-able memory, replace references to their remotely valid equivalents, and RDMA the copy one-after-the-other to each machine. This is problematic as we need to know the remote addresses of any referred to objects, not only for update-protocol managed objects. This would cause memory shortage problems for maintaining the translation tables and additionally, large processing overheads as each machine would need to translate each reference for each machine to broadcast to.

Because of the difficulties outlined above, we support update protocols only for objects containing no reference fields at all (and default to an invalidation protocol for these).

**Alternative 1, updating objects in place.**

To allow broadcasts of local modifications to objects to their copies on remote nodes by means of in-place RDMA, the entire Java object heap must be mapped and registered with the RDMA-device. Fortunately, this is not a problem with modern Infiniband hardware. Modifications to a region are found by comparison against its twin, which is a copy of the region from since it was last processed. Conceptually, for each region in the list of modified regions we invoke the update method shown in Fig. 7.4.

Note that we must update both the remote object and its remote twin because of the following scenario. Assume two machines that write to an object with two fields, X and Y. One machine exclusively modifies X the other Y. Machine 0 writes to X. It then broadcasts the change to machine 1 and updates its own twin. If the other twin on machine 1 would be left untouched, the next synchronized statement (by machine 1) would cause a diff to be created for field X, causing X to be broadcast back to machine 0 overwriting any changes to X at machine 1 made in between the two broadcasts. Our solution is to update both the remote region and the remote twin.
7.5. PROCESSING THE LIST(S) OF CACHED OBJECTS

This protocol is a zero-copy protocol, since we RDMA the changed fields from one object directly over the corresponding fields in the object copies at the other machines.

To increase performance, we deal with ranges of fields or array elements that have changed at a time instead of processing single fields at a time. This intra-object coalescing of fields and array-element indexes allows to perform some bulk-RDMA. The same coalescing of changed fields inside single objects is used in Alternative 2 of the update protocol below as well.

To illustrate what exactly can and what cannot be shipped by bulk-RDMA, consider Fig. 7.5. Here, two objects, P and Q, have been allocated consecutively in memory. Each object is prefixed with an object header that contains information for the garbage collector, DSM, and the object’s meta-information (pointer to the method table for the object, etc.). After the object-header reside the object’s fields. The fields marked with an ‘X’ have been changed since the last lock/unlock (read: Java’s synchronized).

During diff-creation, we first process object P, and create one RDMA-range for the combined fields S and T, and another RDMA-range for its field W. For object Q, we create RDMA-ranges for its field S and another for the combination of its fields U and V. To apply the diffs, we therefore have to perform four single RDMA-puts. We cannot merge the RDMA-ranges of P’s W and Q’s S due to the intervening object header (as the header is not read-only and furthermore, machine-local).

To summarize, this update protocol can be implemented with RDMA only. The upside of this update-protocol version is thus that there is virtually no overhead per object: (ranges of) changes in objects are copied from one machine directly to the copy on another machine. The downside is that there is no inter-object bulk-communication; only some intra-object bulk-communication is possible.

Alternative 2, RDMA-broadcast of diff-arrays.

A potential problem with alternative 1 of the RDMA-update protocol is that there is little potential for bulk communication (it only supports some intra-object bulk communication). The second update protocol rectifies this by supporting inter-object
CHAPTER 7. RDMA BASED DSM PROTOCOLS

```c
void process_arrived_diff_arrays() {
    for each machine m and m != myself:
        patch m.diff_array in locally;
        RDMA m.diff_array.seq_num to 'm' at m.diff_array.remote_ack_seq_num
}
void find_locally_created_changes() {
    for all update_region r on thread->cached_modified_list:
        for each machine m:
            append diff(r) to m.diff_array,
            m.diff_array.seq_number = m.current_bcast_seq_num;
}
void broadcast_new_diff_array() {
    for each machine m:
        while m.current_bcast_seq_num != m.diff_array.acked_seq_num:
            // wait RDMA-ed ack
            m.current_bcast_seq_num++;
        RDMA m.diff_array to 'm' at 'm.remote_diff_array';
}
```

Figure 7.6: Update protocol handler, alternative 2: the three steps to process diff-arrays

Bulk communication. It builds arrays of diffs and broadcasts these arrays in one go by means of RDMA, to all machines. Each machine then needs to periodically poll local memory to see if a new set of diffs has arrived and then process them. If polling is performed often enough, and if the received diff-arrays are easy to process, overheads are low. In our implementation, diff-array elements consist of a target address, a length in bytes, and the changed bytes. The methods needed to process the list of modified regions are given in Fig. 7.6.

For the example of Fig. 7.5, we would create one diff-array containing the diff-ranges {P.S-P.T}, {P.W}, {Q.S}, and {Q.U-Q.V}. These diff-ranges are then copied into RDMA-able memory and copied by one single RDMA to all other machines.

The diff-array protocol therefore globally performs the following three steps. We first check for diff-arrays that have arrived from other machines. Any incoming diffs are applied to the local regions and their twins. Secondly, we create local diff-arrays, one for each target machine (for regions that are not already handled by the invalidation protocol). Finally, the diff-arrays are broadcast by RDMA. Note that a shipment of a diff-array must have been acknowledged before shipping the next diff-array.

To understand the need for an ack-protocol, think of two machines 0 and 1, and two objects, A and B. Machine 0 first updates A and broadcasts its modifications by placing them into the diff-array in machine 1. Let us assume that directly afterwards, machine 0 were to update B and broadcast the changes to B with a new diff-array. If machine 1 had not yet acked the processing of the first diff-array, the second broadcast would overwrite the first diff-array. The first diff and the update to A would be lost. We therefore need an acknowledgment scheme.

We have implemented the acknowledgments with single-word RDMA-writes for efficiency. As soon as a machine has processed a diff-array it performs a single-word RDMA to the diff-array originator. The acknowledgement protocol itself has thus a very low overhead.

To summarize, this second update protocol can also be implemented with RDMA only. The upside of this update-protocol version is that we allow both intra- and inter-object bulk-communication to occur by sending arrays of diffs at a time. The downside of this protocol is that the receiver needs to periodically test if a diff has to be processed.
7.6. PERFORMANCE

Table 7.1: Micro-benchmark results

<table>
<thead>
<tr>
<th></th>
<th>1 machine</th>
<th>2 machines</th>
<th>8 machines</th>
</tr>
</thead>
<tbody>
<tr>
<td># Locks/second</td>
<td>568181</td>
<td>566051</td>
<td>1768</td>
</tr>
<tr>
<td># Barriers/second</td>
<td>—</td>
<td>15015</td>
<td>9174</td>
</tr>
<tr>
<td>Object-request latency (no RDMA)</td>
<td>—</td>
<td>49.9 µs</td>
<td>49.9 µs</td>
</tr>
<tr>
<td>Object-request latency (RDMA)</td>
<td>—</td>
<td>24.6 µs</td>
<td>24.6 µs</td>
</tr>
<tr>
<td>RDMA-invalidation-flush-bandwidth</td>
<td>1.0 GByte/s</td>
<td>64.0 MByte/s</td>
<td>13.3 MByte/s</td>
</tr>
<tr>
<td>RDMA-in-place-update bandwidth</td>
<td>1.2 GByte/s</td>
<td>91.0 MByte/s</td>
<td>9.6 MByte/s</td>
</tr>
<tr>
<td>RDMA-diff-array-update bandwidth</td>
<td>1.2 GByte/s</td>
<td>84.0 MByte/s</td>
<td>7.7 MByte/s</td>
</tr>
</tbody>
</table>

If the receiver does not react quickly enough, the sender will need to wait a long time for diff-processing acknowledgements.

7.6 Performance

Two aspects are important for DSM performance, the latency of fetching objects and the available bandwidth for flushing or broadcasting modifications. This section analyzes performance with some micro-benchmarks and two applications. We use Water and LU from the SPLASH benchmark suite [128]. Both are irregular and challenging and thus stress the DSM protocols. Regular applications or applications with little communication are not well suited for showing protocol performance as differences are rarely visible. Also, Water and LU form the extremes of a spectrum: whereas Water uses many small objects, LU uses only one single array with larger contiguous modifications.

Our measurements were performed on a cluster of dual Xeon 3.20 GHz "Nocona" machines (800 MHz bus, 666 MHz front-side bus) with 2 GByte RAM each. The cluster uses an Infiniband interconnect (10 GBit/s). We use our own low-latency communication package that maps directly to the Infiniband driver's libraries. Our communication package maps RDMA-puts directly on top of the Infiniband verbs layer to get the best possible performance out of Infiniband. Note that normal message sends also use the Infiniband verbs layer directly so that both normal message sends and RDMA sends are fully optimized.

Micro-benchmarks. We first evaluate Jackal’s performance for some simple primitive operations so that we can eliminate these as the sources of overhead in later benchmarks. The relevant data is given in the upper part of Table 7.1 (to simplify presentation, we only present numbers for 1, 2, and 8 machines (read: ‘1, 2 and many’)).

To perform a synchronized block in a loop using one or two machines costs about the same (568181 vs. 566051 locks per second). This is due to the low contention ratio for the lock. The communication costs are very low given that only very small messages are needed and most of the time these messages are handled entirely by the communication thread (the thread that handles all incoming normal messages; RDMA messages of course bypass this thread). On eight machines, the machine that holds the lock object becomes overloaded with request and release messages. Hence, performance drops. Lock contention does not yet seem to be a problem however.

For the barrier micro-benchmark we use a Java object that consists of two fields, a barrier-entry limit and a barrier-entry counter. The main work is performed in a synchronized method that increments the counter. If the counter reaches its limit, the methods calls Object.notifyAll(), otherwise Object.wait(). Each entry/exit of the synchronized method and the execution of the wait method causes an invalidation of the
class MolData {
    double [[[]data] = new double[NUM_DIMENSIONS][NUM_ATOMS];
    ...
}

class MoleculeEnsemble {
    MolData[][] f = new MolData[MAX_ORDERS] [getNumMolecules()];
    ...
}

Figure 7.7: Water’s main datastructures.

cached objects and sends the barrier object to its home.

The lock, unlock, Object.notify(), and Object.wait() are implemented as normal
messages sent to the home-node of the barrier object where they attempt to locally lock
the object. If the lock succeeds or the wait finishes, acknowledgement messages are sent
back to the remote machine to allow it to continue. In total, including flush messages and
synchronization messages, we achieve a barrier time of $109 \mu s$ per barrier on 8 machines
($1.19_{174}$). While the absolute number may seem high, given the much lower latencies
for, say, an MPI barrier on Infiniband, Java’s semantics add significant overhead to a
barrier. For example, to handle multi-threading, Java requires an implementation to flush
working memory, to send synchronization related messages, to handle Object.wait() and
Object.notify(), and finally to wake up threads from thread-pools to handle protocol
messages. With all this overhead, $109 \mu s$ is quite good.

Object request speed is measured by traversing a linked list. Each access to the ‘next’
field in a linked node causes that node to be fetched. For a list containing $N$ elements,
we thus get $2N$ messages (one request message, one reply message with the node’s
data). We then take the average time required for a single list node fetch. Enabling
RDMA for object requests almost halves the latency for fetching a node (even though
only one side of the round trip can be optimized via RDMA). The numbers include the
time needed for state updates, address translations, and for allocating a local copy for
every list node. Regardless of whether 2 or 8 machines are used, the times are the same
due to the low protocol processing overheads.

User level bandwidth is measured as follows. All machines (1 thread per machine)
concurrently execute a loop and change each $N$-th element of a 32KByte array. The
modifications are propagated via a single, empty synchronized block. This inner loop
is performed 20,000 times to give us an indication of the application-level bandwidth
available ($32K * 20,000 / \# \text{seconds used}$). The same benchmark is used for the
invalidation protocol and the two update protocols. When only 1 or 2 machines are
used, both update protocols outperform the invalidation protocol. With larger numbers
of machines, the invalidation protocol wins due to the high overheads in both update
protocols. Note that for all bandwidth measurements, RDMA is also used for region
requests (repeatedly for the invalidation protocol, once for the update protocols).

Water performs an (N-square) N-body simulation of water molecules coded as in
Fig. 7.7. We simulate only 1728 molecules to stress protocols. The innermost array
elements (the NUM_ATOMS dimension of MolData), contain the actual molecule data.
The other data here is read-only and is cluster-wide read-only replicated by the protocols.
Note that NUM_ATOMS equals ‘3’ here (for two hydrogen atoms and one oxygen
atom). This stresses protocols as modified data is encapsulated in many of these small
Table 7.2: Application results in walltime (seconds)

<table>
<thead>
<tr>
<th>Application</th>
<th>1 machine</th>
<th>2 machines</th>
<th>8 machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Water, no RDMA</td>
<td>56.4</td>
<td>41.2</td>
<td>16.5</td>
</tr>
<tr>
<td>Water, RDMA-request invalidation</td>
<td>56.4</td>
<td>40.2</td>
<td>13.1</td>
</tr>
<tr>
<td>Water, RDMA-in-place-update</td>
<td>56.9</td>
<td>41.1</td>
<td>28.8</td>
</tr>
<tr>
<td>Water, RDMA-diff-array-update</td>
<td>56.9</td>
<td>36.6</td>
<td>20.0</td>
</tr>
<tr>
<td>LU, no RDMA</td>
<td>47.3</td>
<td>30.1</td>
<td>19.6</td>
</tr>
<tr>
<td>LU, RDMA-request invalidation</td>
<td>47.3</td>
<td>32.9</td>
<td>18.9</td>
</tr>
<tr>
<td>LU, RDMA-in-place-update</td>
<td>47.0</td>
<td>37.5</td>
<td>26.1</td>
</tr>
<tr>
<td>LU, RDMA-diff-array-update</td>
<td>47.0</td>
<td>41.1</td>
<td>23.2</td>
</tr>
</tbody>
</table>

arrays.

An invalidation protocol with switched on RDMA-request, improves performance by 20.6% on 8 machines (13.1 seconds with RDMA-request versus 16.5 seconds with regular messaging, see Table 7.2).

Regardless of the number of machines used, both update protocols are slower as we pay two penalties. First, although changes are broadcast to every machine (eliminating the need to explicitly fetch them), the changes are not used by every processor. The exact set of consumers is hard to detect by the DSM protocol without changing Water’s source code. In contrast, the invalidation protocol pulls the changes to only those machines that require them.

The second penalty is due to Java’s lack of true multi-dimensional arrays (Java provides only arrays of references to sub-arrays). Since the data of all the innermost 1D arrays are not contiguous in memory, the first RDMA update protocol (updates in place) needs a large number of RDMA transfers. On the other hand, the diff-array RDMA update protocol version has a lot of administrative overhead for each diff-array. Even the higher efficiency of the RDMA hardware (compared to normal send/receive) cannot overcome this. The slight advantage of diff-array RDMA-update protocol on only two machines is quickly lost with increasing numbers of machines.

LU factorizes a dense matrix, encoded as a single flattened array of doubles. Due to the blocking technique used, every machine accesses only a few linear segments of array elements. Hence the number of region fetches needed is less than in Water. Use of RDMA improves the performance of the invalidation protocol only slightly by 3.6% (19.6 versus 18.9 seconds).

The update protocols are always a loss for LU. Unlike Water, LU’s threads write larger consecutive chunks in a single linearized matrix. Because of array chunking, fewer updates and hence fewer RDMA broadcasts are needed. A few hundreds of 2 KByte array segments are broadcast instead of tens of thousands of 24 byte broadcasts as in Water. LU also suffers from the effect that broadcast data is often not (immediately) used by the receiving CPUs. Hence the invalidation protocol (with RDMA fetch) is faster.

Finally, there are many array sections on the list of modified regions that are actually unmodified since the last list processing. This happens because in update protocols, modified regions are never removed from the modified-regions list, causing many empty diffs. However, since we still need to test every single array element for potential modifications at each synchronization, the processing requirements of the update protocols increase. In Water, this effect does not occur since each processor writes the same water molecules each time. Of course, the invalidation protocol does not suffer from this
effect as its regions are removed from the modified-regions list at invalidation time (but each access afterward triggers an access check to add it again to one of the flush-lists).

7.7 Conclusions

In our system, invalidation and update protocol managed regions can coexist. We found that RDMA can be successfully applied to invalidation protocols and have designed two update protocols that solely use RDMA. We have seen performance improvements of up to 20.6% using RDMA for object-fetching. Without source code changes (for example, those suggested by [47]), even when modern RDMA hardware is used throughout, update protocols are still slower than invalidation protocols. This is because of three main reasons. First, when adding address translation to allow large address spaces, the cost of protocol processing grows large in update-protocols as they need, per-machine processing.

Second, message aggregation is hard to do with current Infiniband RDMA implementations as they currently lack remote scatter. Ideally, we would like to present the RDMA hardware with two lists of I/O vectors. One I/O vector for where to copy the data from at the local machine, and another I/O vector for where to copy the data to at the target machine. The current Infiniband VERBS allows only very limited use of I/O vectors.

Finally, another RDMA-feature currently missing is signalled-IO. Signalled RDMA would cause an interrupt at the receiver once data has been copied. This would not only allow to free the CPU when waiting for message replies but it would also allow us to immediately reply to unexpected incoming messages (instead of periodically tested for them).
Chapter 8

Tapir: Language Support to Reduce the State Space in Model-Checking

Authors:
Ronald Veldema, Michael Philippsen
University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3 • 91058 Erlangen • Germany
veldema@cs.fau.de, philippsen@cs.fau.de

Published in:
Arbeitstagung Programmiersprachen
Conference: ATPS ’09
Lübeck, Germany
Oktober 1
LNI Proceedings, abstract page 364; 2860â–74, Vol: P-154
Lecture Notes in Informatics (LNI) der Gesellschaft für Informatik e.V. (GI).
Publisher: Bonner Köllen Verlag (2009)
Editors: Stefan Fischer, Erik Machle, and Rüdiger Reischuk.

Where in the paper of Chapter [5] we discussed the basic language features for parallelization and general programming, in this Chapter’s paper we discuss the language features that allow easier model checking be reducing the number of states that need to be explored by the model-checker. Where other languages that explicitly support model checking are too abstract to generate code for, Tapir explicitly allows for good quality code generation.
CHAPTER 8. TAPIR’S MODEL-CHECKING SUPPORT

Abstract

Model-checking is a way of testing the correctness of concurrent programs. To do so, a model of the program is proven to match properties and constraints specified by the programmer. The model itself is created by disregarding irrelevant program details.

The biggest problem in model-checking is the number of program states that need to be tested. Tapir, a simple but familiar object-oriented language and accompanying tool chain, addresses this problem in two ways. First, the programmer can provide application specific program transformations that reduce the state space. Second, for selected classes and methods, the programmer can provide an alternative implementation: a slim black box version for use in model-checking that abstracts away many details of the full fledged implementation.

Tapir’s aspect-oriented test case generation combined with black-boxing allows model-checking of low-level library code.

8.1 Introduction

System software is critical software that includes such things as operating systems, networking libraries, and middle-ware. In general system software provides some service to application code, and is hence essential for system stability. There is therefore a need for ‘extreme testing’ of system code which we implement by means of model-checking. Because system software is often used by multiple concurrent threads or processes, the software must be made safe with respect to concurrency. Model-checking provides this concurrent testing naturally.

Model-checking a large library or complex service provider is hard due to two reasons. First, model-checking requires complete, self-contained programs which by definition libraries or service providers are not. To model-check these, a service user (i.e., a test case) must be supplied as well. For exhaustiveness, however, many service users are required, better still, in combination with each other to increase test coverage. Second, model-checking itself is time-intensive as it performs a complete state enumeration over a huge state space. This requires a large memory to hold all computed states (to be able to detect cyclic program states) and a lot of time to perform the state space enumeration itself.

Tapir has solutions to both of the above problems. First, Tapir supports Aspect Oriented Programming (AOP) to weave a service or library together with one or more service users creating many different complete (test-)programs. Second, Tapir allows the programmer to aid in state space reduction by both providing program transformations that reduce the model-checker’s state space and by providing an additional slimmed version of a class that is used during model-checking.

The Tapir language is a subset of both Java and C++ with their worst and unneeded features removed and some new features added. As such, Tapir’s language should be familiar to many. Tapir adds aspect orientation (covered in Sec. 8.3) for comfortable test case generation. Compile-time predicates, program transformations and analysis rules are added to enable state space reductions (see Sec. 8.4). The base Tapir language features:

- classes and templates, but no inheritance,
- no type casts, pointer arithmetic, nor function pointers,
- no exception handling support,
- no static/global variables,
8.1. INTRODUCTION

Figure 8.1: Tapir tool chain.

• some functional programming support.

All these restrictions allow for easier program analysis and thus indirectly make state-space reducing optimizations easier: the first two items provide strong static typing and eliminate some types of errors. Removed exception handling simplifies control-flow. Removed support for global variables and added support for functional programming eliminate sources of non-local reasoning which eases program understanding.

Parallelization and distributed programming are explicit. In Tapir, a class can be marked with process or thread. A process class will have its run method invoked at virtual machine start-up. When a process is running it can create instances of thread classes and start them by invoking their start methods. Process classes communicate among each other via Remote Procedure Calls (RPC) in active-message fashion [123]. This level of communication abstraction allows systems programs to be written without knowledge of communication media. Communication between threads is via process memory.

The Tapir tool chain (see Fig. 8.1) cannot only generate a C/C++ implementation from the Tapir code. It can also generate program specific model-checkers (one generated per test case). When generating the (C++) implementation, the weaver removes all statements from the Tapir program that deal with AOP. What remains is a true library. That code is passed to the Tapir compiler which generates a C++ library from it. For model-checking, the aspect weaver generates all possible permutations of weavings to create the test case programs. Each of these new Tapir programs (now without the AOP statements) is passed to the Tapir compiler to generate a program (test case) specific model-checker. Each of these model-checker programs is generated in Java, in the same way that the Spin [60] model-checker generates its C code for the Promela language. Such a Java program is then fed to a JVM for execution. Because of the model-checker’s large memory requirements, a specialized JVM such as LVM (Large Virtual machine) [123] is preferable. LVM’s language extensions for memory optimizations and support for distributed execution on a cluster meet the model-checker’s large memory requirements.

The main contributions made in this paper are: (1) a technique to allow the programmer to specify, at language level, multiple implementations of a class, method, or statement (a version used during model-checking and a version used during code gener-
8.2 Self-consistency

Typical model-checking approaches express properties external to the code or use a formulaic approach. We take a code-centric approach.

We call an application self-consistent if all incorporated checks succeed for all possible runs. In Java/C/C++, such checks are often implemented by assert statements: `assert(X)` where X evaluates to a boolean condition. If X evaluates to false the program aborts, if true, nothing happens. These assert statements are then placed all over the program to test the self-consistency of its data structures.

For many applications, it is also important to know that eventually an event will occur, that a given program state will be reached, or that after some event some other condition should hold. Although many model-checking environments provide specialized formalisms to express such properties, they can also be explicitly programmed by means of asserts with the help of counters. E.g., to test that after event X, eventually Y should hold, one could write:

```java
boolean X_happened = false;
boolean Y_happened = false;
int Y_wait_counter = 0;

if (X)
    X_happened = true;

if (X_happened)
    assert {
        Y_wait_counter++;
        if (Y)
            Y_wait_counter = 0;
        assert(Y_wait_counter < MAX_Y_WAIT_COUNTER);
    }
```

Linear Temporal Logic (LTL) [43] is a formalism that is meant to describe such event sequences (until/after X, Y should hold, etc.). CTL (Computation Tree Logic) [43] extends LTL by adding that a condition holds on all possible paths following a program state. By using model-checking we can guarantee CTL properties as it tests all possible states.

Some problem domains have explicit time requirements as well. Again, instead of a special formalism to express real-time properties, we support a minimalist approach by providing a `system.currentClockTick` method so that programs can check their to-be-checked properties with asserts as well. When model-checking, the currentClockTick intrinsic returns the number of executed instructions of the non-deterministic machine. When used for model-checking, it will return the number of instructions executed by the simulated machine. When used for code-generation, it works as expected. Using the currentClockTick() method inside assert statements allows us to test real-time properties of programs. For example:

```java
long start = system.currentClockTick();
work();
long end = system.currentClockTick();
assert((end-start) < N);
```

when executed under the model-checker, the aspect will test if `work()` will always execute in less than N instructions.
Note that it is the responsibility of the programmer to write sufficient asserts into his code to allow the model-checker to find all errors. While for simple tests automatic assert generation seems feasible, tests of program semantics is definitely the programmer’s job.

Unfortunately, the above solutions have the potential to increase the state space as each timer and counter variable becomes part of the state of a process. That can be partially fixed by means of the techniques discussed below.

8.3 Aspect oriented testing

As Tapir is meant for programming system software, a Tapir program will in general not be complete in the sense that it can link to an executable. What is needed is some start-up environment that invokes the functions of the service provider to allow its functionality to be tested. Moreover, because a library can often be used in different ways (by calling its methods in different orders, for example), many different individual programs need to be generated where each generated program uses the library in a different way.

To create a test, a set of classes and methods is woven into the library under test. We call such testing code (that consists of ordinary Tapir classes and methods) a ‘scenario’. Thus, the Tapir weaver works as follows; for a given joint-point, a number of scenarios may apply. For example, for a join-point X(), multiple scenario’s may contain a matching X() method. For each permutation of scenarios that apply to a given join-point, the weaver generates an output file in which each join point is replaced by a call to the method with that name from inside the scenario. When there are multiple scenarios in the permutation, the methods from that scenario are called in the order of the scenario permutation.

```java
class FileStream join_point<FS_Mixin> {
    boolean is_open;
    StorageDevice device;
    Lock l;
    int open(string name) {
        is_open = true;
        join_point<test_open(ret, name)>;
        return ret;
    }
    byte read_byte(int fd) {
        Block b = device.get_block();
        join_point<test_read(ret)>;
        return ret;
    }
    void close(int fd) {
        is_open = false;
        join_point<test_close(fd)>;
    }
}
```

Figure 8.2: A file stream.

To illustrate this, let us examine the simple Tapir program from Fig. 8.2. It depicts a simple file I/O class. Here, for example, join_point<test_open()> can be replaced by a corresponding call to method test_open from one or more test scenarios.
scenario S1 {
    void test_open(int ret, string name) {...}
    void test_read(byte ret) {assert(is_open);...}
    void test_close(int fd) {...}
}
class FS_Mixin {
    void main() {
        int fd = open("S1");
        byte val = read_byte(fd);
        assert(val == 42);
        close(fd);
    }
}

scenario S2 {
    void test_open(int ret, string name) {...}
    void test_read(byte ret) {...}
    void test_close(int fd) {...}
}
class FS_Mixin {
    void main() {...}
}

Figure 8.3: Two test scenarios for Fig. 8.2.

From the two test scenarios from Fig. 8.3 four new Tapir programs (four permutations) are created by the weaver. One that looks only at scenario S1 which causes the join-point statements to be replaced by calls to S1’s test_open, test_read, etc. The second generated Tapir program is similar and considers only scenario S2. The other two permutations replace each join-point with two calls, one per scenario. For example, join_point<test_open()> will be replaced by (S1’s test_open(); S2’s test_open(); ) in the third permutation. In the last permutation, scenario S2 is considered first followed by scenario S1. This causes a replacement snippet: (S2’s test_open(); S1’s test_open(); ).

Whenever a join-point causes a call to a scenario’s method, that scenario’s method is also added to the class being woven into. This allows a weaved-in method to use the members of the class where the join-point was performed. For example, this allows the use of is_open in the assert statement in the test_read method of scenario S1.

The clause join_point<FS_Mixin> in the first line of Fig. 8.3 causes the methods and fields of class FS_Mixin from a test scenario to be added to the class FileStream. For the first two permutations, the main method of either S1 or S2 is copied in. For the other two permutations, we cannot bring in two equally typed main methods of course. Instead, we concatenate the two bodies to create a single new main method.

As not all permutations of scenario weavings are valid, the programmer can restrict the set of generated permutations by means of additional syntax:

- excludes X says that this scenario excludes scenario ‘X’.
- repeat Y says that this scenario can be repeated up to Y times inside a permutation. By default a scenario is allowed to occur only once inside a permutation.
- exclusive says that this scenario cannot be combined with any other scenario.
- after X says that this scenario can only be put into a permutation if scenario X has already been added.
8.4 Application specific code transformations to reduce the state space

To support threaded execution in the model-checker, Tapir has a `context_switch` instruction. Execution of this instruction allows the model-checker to non-deterministically execute another (or the same) thread or process. This instruction spans the state space in the model-checker to create all possible interleavings of thread and process executions. Removing even one single context-switch thus has the potential to greatly reduce the state space.

While some context-switch instructions can potentially be removed by generic (optimization) rules, many can only be removed if application specific knowledge is applied. For example, many context-switches can be removed if the programmer would provide the compiler with knowledge that a piece of code only accesses thread-private data (and thus cannot be influenced by other threads). It is hard (and most times impossible) for compiler analysis alone to determine that data is actually thread-private.

This is why Tapir supports programmer supplied program transformation rules. These allow the programmer to code such programmer level information that can’t be extracted easily (or at all) by static compiler analysis alone. These transformation rules and propagation rules are coded by the programmer in his source code alongside his Tapir classes so both are inside the same file(s). The rules themselves are interpreted by the compiler so no compiler internals are exposed to the application programmer.

A program transformation rule needs to express two things: (1) when to allow the transform and (2) what to transform into what. The ‘when’ question is almost always non-local: information from somewhere allows a transformation elsewhere. To allow programmer supplied transformations, we must therefore (application specifically) propagate information using (predicate) propagation rules and provide a mechanism for describing program transformations. Both mechanisms are supplied in the language (as opposed to hard-coded inside the compiler).

The implications of bad (and good) applications of block-boxing are discussed in Sec 8.5.2.

8.4.1 Transformation rules

```java
int open(string name) {
    l.lock();
    work_open();
    is_open = true;
    l.unlock();
    return ret;
}
private void work_open() {
    context_switch;
}
```

Figure 8.4: A propagation rule example.

Consider the `open` method of the FileSystem example of Fig 8.4. Because `open` might be concurrently invoked, its state is protected by lock `l`. For some reason there is a `context_switch` statement in `work_open` which is called from `open`. However, because there is a lock/unlock pair surrounding it, the context-switch statement
might be superfluous when no other thread can interfere. This knowledge is application specific as, for example, locks are programmed using ordinary Tapir statements and so provide no language level 'hints' to atomicity. To be more precise, the lock class is a meta class (from Sec. 8.5).

To get rid of the context-switch in this example, the program can provide a transformation rule like:

```plaintext
transformation_rule {
  guard: LockedToken[x]
  pattern: { context_switch; }
  target: { }
}
```

The semantics is that when the pattern’s statement (the context-switch statement here) is found in the code while the guard holds, the pattern’s statement is replaced by the target’s (empty) statement. The guard is a compile-time predicate. How that works is described below.

### 8.4.2 Compile-time predicates

With the `predicate` statement, the programmer can tell the compiler that something holds for a program variable. Compile-time predicates have no code generated for them at all. For example, the `Lock.lock()` method can have:

```plaintext
predicate LockedToken[this]
```

at its end. The compiler propagates this predicate through the code and can then prove that `LockedToken[1]` holds in `FileSystem.open`. The propagation itself is facilitated using programmer supplied propagation rules, see below in Sec. 8.4.3.

Note that transformation rules and predicates encode application level knowledge. The compiler does not need to know that the `Lock.lock()` and `Lock.unlock()` methods are special in any way.

In addition to using these compile-time predicates as guards in program transformations, they can also be used in `mustHold` annotations on fields. For example:

```plaintext
class FileStream join_point <FS_Mixin> {
  bool is_open must_hold: LockedToken[1];
  ...
}
```

The compiler will flag an error if the predicate has not been propagated to any statement that uses `is_open`. Extended syntax allows to distinguish read and write accesses. The `mustHold` annotations over class fields are similar to type-states, see [40][88].

### 8.4.3 Propagation rules

Consider the following propagation rule that states that if the predicate `LockedToken[x]` holds for variable `x` on a control-flow path that leads to the assignment `y = x`, then the predicate `LockedToken[y]` holds afterwards:

```plaintext
propagation_rule {
  if_holds_before LockedToken[x]
  with y = x;
  then_holds_after LockedToken[y]
```
8.5. APPLICATION SPECIFIC BLACK BOXING TO REDUCE THE STATE SPACE

The statement for a propagation rule (here `'y = x'`) can include field accesses and calls to allow heap and inter-procedural predicate propagation. In case of multiple control-flow-path predecessors, "if_holds_before<0> P[x] and if_holds_before<1> P[x]" is used, where '0' and '1' denote predecessor paths.

Note that transformation guards must be compile-time entities and not run-time entities as they are used to enable compile-time transformations. The compiler must prove that a guard always holds. We do this by propagating information.

Instead of compile-time predicates, run-time predicates could be used also. However, due to the time-intensive nature of model-checking, making the model-checked code as small and fast as possible is imperative. Adding run-time tests is therefore counter productive.

8.5 Application specific black boxing to reduce the state space

```java
meta class StorageDevice {
    Block[] blocks;

    Block get_block(int id) {
        return blocks[id];
    }

    void put_block(int id, Block b) {
        blocks[id] = b;
    }
}
```

Figure 8.5: Meta class.

Abstract-data types hide implementation from specification. We re-introduce the concept here by allowing the programmer to provide two implementations of a class: a simplified implementation used during model-checking and a complex version used during code generation.

The specification of a type can be modified in three ways: by marking a field, a method, or a whole class with the `meta` keyword. In the FileStream class of Fig. 8.2, a StorageDevice class is used to perform actual device I/O. The full code of this class is rather lengthy and its implementation is transparent to the FileStream class under test. It would be both impractical and computationally intensive to use it in model-checking since the state space would explode. To alleviate this problem, a simplified version that (loosely) implements the same interface contract can be provided, see Fig. 8.5. In the example, when Tapir's compiler generates the library, calls to `get_block` and `put_block` are generated (whose implementation is to be externally supplied). When Tapir generates the model-checkers it uses the code of the meta class.

To summarize, when marking a field as `meta`, the field itself is used during model-checking, but in the generated implementation code, calls are emitted to getter and setter methods. When marking a method `meta`, the method's code is used during model-checking but when generating the implementation code, the code of the function needs to be supplied externally. Marking a class as `meta` causes all its methods and fields to be marked `meta`. 
8.5.1 Locks, condition variables, and compile-time asserts

Note that in Tapir, lock objects and condition variable objects, are themselves regular Tapir objects. To ensure atomic execution of statements, Tapir provides the `atomic` statement-block which is only allowed to occur in meta statements as it only has meaning for the model-checker:

```
atomic { statements }
```

The actual implementation of the class must ensure its own atomic execution if needed. Atomic blocks are guaranteed to be atomic inside the model-checker by not allowing either the programmer or the compiler to insert context-switch instructions in them.

To allow waiting for a specific condition to become true, Tapir supplies the `wait_until` statement:

```
wait_until <condition> statement
```

which blocks the current Tapir thread until the condition holds. Once the condition holds, the statement is executed. Each `wait_until` implicitly adds a context-switch statement so that if the condition does not hold, another thread can run. We can use atomic blocks and the `wait_until` statements to implement the lock object as follows:

```java
class Lock {
    meta private void blocking_lock() {
        wait_until UnlockedToken[this];
        Thread t = Thread.current();
        wait_until value == 0 atomic {
            owner = t;
            value = 1;
        }
        UnlockedToken[this] => LockedToken[this];
    }

    meta void unlock() {
        LockedToken[this];
        atomic {
            owner = null;
            value = 0;
        }
        LockedToken[this] => UnlockedToken[this];
    }
}
```

Let us look at the `blocking_lock` method. First, we assert that the compile-time predicate LockedToken holds over 'this'. Afterwards we store the reference to the current thread for later use. As soon as the value becomes zero, the current thread becomes the lock owner and the value is set to one to indicate that the object is currently locked.

If UnlockedToken[this] holds, the transition predicate changes the predicate to UnlockedToken[this]. We need an explicit transition statement here as simply asserting `predicate(LockedToken[this])` at the end of `blocking_lock` would cause to have both UnlockedToken[this] and LockedToken[this] valid on the same reference (this) at the end of the method. There would then be no way to disambiguate between a correct predicate use here and a detected problematic usage.

The unlock method expects the LockedToken property to be set on it and so has the predicate asserted. It then atomically resets the owner and the value before compile-time predicate translation.
8.6 PERFORMANCE

The code of the Condition variable is similar. A `wait_until` statement blocks on a thread’s `signal` field. A list of blocked threads is used for waking them up once signalled. Condition.Signal, takes one thread from the list, and sets its `signal` field.

Because locks and condition variables use `wait_until`, the more they are used, the larger gets the search space for the model-checker to explore.

Deadlock detection is implemented by the Lock and Condition objects themselves as they count the number of blocked threads and compare it against the number of running threads by an assert statement. Live-lock detection is currently not implemented as it is harder. For example, it could happen by message exchanges, back-off algorithms, etc.

8.5.2 Discussion

While black-boxing helps to reduce the state-space, it also allows for divergence of the model-checked system and the final system. If the black-box exactly implements the semantics of the actual implementation, then no problems will occur. If the black-box provides semantic changes in allowing less than the actual implementation, the system could show errors not present when the black-box (meta) types are replaced by their actual implementations. If the black-box allows more in some way (in inputs, outputs, etc.) compared to the actual implementations they replace, the final system will not show the errors during model-checking.

It is the responsibility of the programmer to provide accurate replacement meta types. This problem is endemic, however, to all systems doing black-box replacement. At least in the solution proposed here, the places where abstraction from ‘reality’ occurs are clearly marked. Also, this guarantees that the structure of the whole program is the same as in the actual implementation while this cannot be guaranteed if the specification or model are completely separate from the implementation.

8.6 Performance

To measure model-checking performance, we use one application generated by applying one test-scenario to our file system example. We first measure the effectiveness of the propagation rules and user-level optimization rules and next the effectiveness of using meta classes to remove code complexity.

The code of the benchmark is too big to include verbatim, so we sketch it here. The whole program starts by creating two process objects. Each process then creates a thread to run `main()`. Each `main()` then creates a `FS_tester` thread to run the FileSystem tests and waits for completion (using a condition variable object). Each `FS_tester` thread accesses the file system object to perform a series of writes and reads and uses (runtime) asserts to check self-consistency. Because there could be multiple `FS_tester` threads, the file system object is protected by a lock object. Outside of these threads, the Tapir runtime library creates two additional Tapir threads per Tapir process for providing services (a thread for managing incoming messages from other Tapir processes and another for waiting for global system termination). In total, there are thus two processes to model-check, each with four threads running and interacting and a number of objects allocated by them.

We then create two versions of the application. One version (A) uses the file system class described above and a companion Device meta class. A second version (B) of the application replaces the FileSystem class with a meta class (making the Device meta class redundant). Version (B)’s FileSystem methods have just enough implementation
Table 8.1: File System Benchmark

<table>
<thead>
<tr>
<th></th>
<th>Average Time</th>
<th>Heap Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Optimizations</td>
<td>342 s</td>
<td>11 GByte</td>
</tr>
<tr>
<td>Propagation</td>
<td>163 s</td>
<td>7 GByte</td>
</tr>
<tr>
<td>Propagation + Meta FileSystem</td>
<td>91 s</td>
<td>4 GByte</td>
</tr>
</tbody>
</table>

*Dual Opteron 246, 2 GHz, 12 GByte of memory.*

to satisfy their interfaces. Furthermore, the benchmarks include Tapir’s runtime so that measurements include the time needed to non-deterministically start-up the service threads and model their interactions (which already contain a number of locks, etc. internally).

We then run the model-checker twice on version (A): once with and once without propagation rules. For version (B) we time the model-checker with propagation rules enabled.

Table 8.1 gives performance results. Without optimizations (version (A), no propagation), memory usage and run time are high. With optimization (version A with propagation), two context switch instructions are removed (over a total of 12). However, one of those is in a loop and hence critical. This causes the big performance gains seen. Version (B) makes the code simpler (fewer instructions to simulate) and removes some object allocations (smaller process state).

Memory requirements are high because every new state discovered in the search space is placed into a hash-table. We use a 64 bit hash key computed over the simulated processes. The hash table ensures that no state that has already been visited is expanded again. Memory usage is kept low by maintaining diffs of the simulated processes’s memory whenever possible instead of creating a full-blown copy of the process each time it is stored into the model-checker’s hash table.

8.7 Related Work

Except for the language restrictions, the programmer is free to code anything in Tapir. This includes recursion which is not allowed by some other software model-checkers (SPIN/Promela [60], NuSMV [35]). Tapir also supports concurrency (processes and threads) which other software checkers cannot handle, e.g. Moped [2]. In general, model-checkers that use state-machines (like Petri-networks [29]), cannot handle recursion but can handle concurrency. Stack-machine based model-checkers cannot handle recursion but can handle concurrency efficiently. Approaches that use virtual (Von-Neumann) machines can handle both (such as we do here). They can handle both but require more details in their models to operate and so generate a larger search space compared to the restricted techniques.

Most model-checking approaches use separate implementation and model-checking languages. This can cause both implementations to diverge such that a bug-free, model-checked application can still show logic bugs in the implementation, see [94]. Tapir uses one single language with only abstracting away low-level code in meta classes. This approach should scale better than model-checking the complete application directly such as CMC [87] does.

The use of Aspect-orientation has been proposed before for various tasks from implementing the parallelization itself [109] to adding unit-tests [16]. But to our knowledge, AOP has not been used in a tool chain to generate permutations of test
programs.

Condate\cite{124}, extends GCC with external rules for semantic analysis. For example, one might write a rule that states that between a call to \texttt{malloc()} and a call to \texttt{free()} there should be a test against NULL. We achieve the same effect by a combination of predicates, propagation rules, and transformation rules at language level but for optimization purposes. Cobalt \cite{81} is similar to Condate, but uses its rules to generate a static optimizer in a compiler while in Tapir rules are taken from the program and interpreted in the compiler for extra flexibility.

Unlike our language-level approach, some approaches use compiler based static state space reduction, e.g. \cite{129} which does not allow application-level knowledge to reduce the search space. Tools such as F-Soft \cite{55}, Bandera \cite{36} and JavaPathFinder \cite{80} take a program and extract a specification for an existing program verifier by removing program details automatically. In contrast, Tapir leaves it to the programmer to decide which program details are superfluous or essential. Other techniques to reduce the search space, such as program slicing \cite{61}, and predicate reduction or predicate abstraction \cite{39} are orthogonal to Tapir’s approach and could be added to our tool chain.

\section{Conclusion}

Using AOP to generate test programs for libraries is a useful first step towards automatic library testing. What is further required are things such as coverage analysis (how much of the library is covered by a non-deterministic run of a library test), input value boundary tests, etc. The use of meta classes to separate code into model-checking specific and implementation specific parts is key to allow model-checking for some types of software, especially if it interacts with hardware. Even if such problems are absent, black boxing code inside simplified meta classes can severely increase model-checking performance (at the cost of false negatives/positives).

Even though programmer written optimization rules can be hard to write and get right, they can help performance enormously. Still, by providing them in libraries the programmer may even never see their use.
Chapter 9

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

Authors:
Ronald Veldema, Michael Philippsen
University of Erlangen-Nuremberg
Computer Science Department 2
Martensstr. 3 • 91058 Erlangen • Germany
veldema@cs.fau.de, philippsen@cs.fau.de

Published in:
Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing
Conference: LCPC ’07
Urbana-Champaign, Illinois, USA
October 11–13, 2008
Bd. LNCS 5234
Pages 187–201
Publisher: Springer, Berlin
Editors: Vikram Adve and María Jesús Garzarán and Paul Petersen.

Because model-checking is such a memory intensive application, we investigate a specialized DSM for efficiently managing all memory of all machines in a cluster. For example, when allocating new objects/arrays we use heuristics to allocate on a lightly loaded machine and allocate it close to objects that will be used together. Specialized DSMs for memory conserving purposes is a new research area.
Abstract

To solve problems that require far more memory than a single machine can supply, data can be swapped to disk in some manner, it can be compressed, and/or the memory of multiple parallel machines can be used to provide enough memory and storage space. Instead of implementing either functionality anew and specific for each application, or instead of relying on the operating system’s swapping algorithms (which are inflexible, not algorithm-aware, and often limited in their fixed storage capacity), our solution is a Large Virtual Machine (LVM) that transparently provides a large address space to applications and that is more flexible and efficient than operating system approaches.

LVM is a virtual machine for Java that is designed to support large address spaces for billions of objects. It swaps objects out to disk, compresses objects where needed, and uses multiple parallel machines in a Distributed Shared Memory (DSM) setting. The latter is the main focus of this paper. Allocation and collection performance is similar to well-known JVMs if no swapping is needed. With swapping and clustering, we are able to create a list containing $1.2 \times 10^8$ elements far faster than other JVMs. LVM’s swapping is up to 10 times faster than OS-level swapping. A swap-aware GC algorithm helps by a factor of 3.

9.1 Introduction

There are problems that require extremely large numbers of objects (hundreds of gigabytes to terabyte(s)) and that are as such not bound by processor speed, but rather by the amount of available memory. Examples are simulations with large numbers of ‘units’, e.g., either molecular or fluid particles [102]; combinatorial search problems, e.g., finding the most frequent sub-graph in a set of other graphs, which requires to store all the graphs already processed; model checkers, which run a program on top of a (simulated) non-deterministic Turing machine (NDTM) and for each non-deterministic choice, the NDTM creates a copy of the simulated machine to explore both choices to check that no illegal program states can occur. Memory requirements for all of the above range from multiples of hundreds of gigabytes to a terabyte and above.

Since it is too costly or impossible to plug in enough memory into a single machine, programmers squeeze their code and rely on the operating system’s swapping. Both of which is suboptimal. Reimplementing data structures and algorithms to reduce memory consumption takes time that is better spent implementing functionality and ensuring program correctness. Also, the operating system’s virtual address space implementation not only does not know what data is truly most recently used, but also the amount of virtual memory available (including swap space) is fixed and limited. Extending the amount of swap space is a tedious task for the system administrator and permanently reduces the amount of disk space available to the user. Finally, few operating systems compress swap space or exploit the aggregate memory and swap space available in a cluster.

Our LVM (Large address space Virtual Machine) swaps objects to disk in compressed format and provides a simple distributed shared address space to use all of a cluster’s memory and disk space. While this functionality could also be implemented by the programmer, a virtual machine solution provides a separation of concerns. The programmer can concentrate on the correctness and efficiency of the application code instead of optimizing the low-level address space consumption. Also, address space optimizations for one particular program are often useless for the next program whereas
9.2. LVM IMPLEMENTATION

Our compiler frontend \[121\] generates a register-based intermediate representation which is similar to LLVM \[77\]. This is fed into LVM that employs both an interpreter and a Just-In-Time compiler (JIT) to execute the code. The code is first interpreted, and if found important enough, it is compiled to native code. To ensure portability, our JIT is very simple: we compile a LVM-function first to C-code and from there to a shared library that is dynamically linked while the program is running.

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

In LVM we focus on memory-conserving compiler optimization. For example, LVM performs escape analysis \[33, 78, 126\] and allocates objects that do not escape the allocating function/thread on the stack instead of the garbage collected heap to reduce pressure on the garbage collector.

9.2.1 Implementing the Address Space

The main problem with implementing a huge, distributed address space is addressing objects flexibly and efficiently. Implementing an object reference as a direct memory pointer is inflexible because it does not allow objects to easily move in memory and because it provides little information for analysis. On the other hand, a reference should be small, since there usually are many references that need to be kept in memory and are manipulated often. For performance, they should not be larger than the operand width of machine instructions. For these reasons, we employ 64 bit references to encode an object’s location in memory in a cluster. Every access to an object thus first needs to decode the object reference to retrieve a local object pointer. While this translation costs at run time, it allows us to access an address space that spans multiple machines. This indirection scheme is fully Java compatible as references are transparent in Java.

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

An object reference is structured as shown in Fig. 9.1. The machine field encodes the
class Data {
    int value;
    Data() {
        value = 12345;
    }
    void foo() {}
}
public static void main(String args[]) {
    Data d = new Data();
    d.foo();
}

Figure 9.2: Java example and resulting LVM code.

number of the cluster node on which the object is allocated, the segment field indicates which segment on that machine it resides, and the object-id fields gives the offset (in multiples of 32 bytes) at which the object is found in the segment.

Because references are no direct memory addresses they need to be decoded. Fig. 9.2 shows a snippet of Java code and its corresponding LVM code where this is required. The latter is simplified, as we have switched off escape analysis, type inference, method inlining, and the removal of superfluous reference decoding.

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

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

Fig. 9.3 shows the pseudo code for refToObjectPtr. Because of its ubiquity, refToObjectPtr is also used as the GC barrier. Whenever the garbage collector needs to wait for all threads to stop, all threads are gathered in gc_barrier_enter();

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

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

Whenever the limit of the number of in-memory segments has been reached, the
javaObject *
refToObie ctPtr(object_reference_t ref) {
    if (gc_requested) gc_barrier_enter();
    if (ref == 0) throw_null_pointer_exception();
    test_dsm(ref);
    Segment*s = locate_segment(ref.seg_number)
    s->update_timestamp();
    javaObject *q = s->data + (32 *
    ref.get_seg_index());
    return q;
}

Segment seg_arr[MAX_SEGMENTS_PER_MACHINE];

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

Figure 9.3: Decoding a reference to an object.
oldest segment in memory is compressed\footnote{For compression we use the LZO library since it combines high compression speed with reasonable compression ratios\cite{1}.} and swapped out. This ensures that the operating system’s swapping mechanism is never triggered, as LVM will never use more memory than core memory. LVM speeds up swapping by delaying all swap-out operations until the next swap-in operation. Which segment is least recently used is determined by a logical clock that is set by \texttt{update_timestamp()} upon each access. \texttt{Update_timestamp()} increments a global variable and sets the segment’s timestamp to it. This only takes a few machine instructions. To further increase performance, LVM can be directed to swap-out a number of its oldest segments instead of just one segment when a memory shortage is encountered. The result is that most disk-IO can be performed in parallel. Note that after the operating system’s page level swapping loads a page, the OS does not track individual page hits. In contrast, LVM knows exactly which segments have been used last.

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

\section*{9.2.2 DSM support}

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

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

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

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

Thread migration traverses the stack using the \texttt{prev\_frame\_offset} links. For each activation, a translation table entry of the form $\{\text{prev\_function}\rightarrow\text{name}, \text{offset\_in\_stack}\}$
is added and sent with the stack to the receiver. The receiver uses the translation table to plug in new function addresses (as the receiver might have allocated functions at different addresses). Migration therefore takes a stack traversal at both sender and receiver with an additional hash look up per stack frame at the receiver to find function addresses for given function names.

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

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

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

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

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

### 9.2.3 Object Allocation Strategies

LVM implements Java’s automatic memory management. It tries to allocate objects in the following order: (1) try first to allocate the object in an in-core memory segment.
boolean equal_arrays(int[] a, int[] b) {
    for (int i=0; i<a.length; i++)
        if (a[i] != b[i]) return false;
    return true;
}

Figure 9.4: Comparing two arrays.

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

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

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

The syntax for our (optional) directive is:

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

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

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

### 9.2.4 Reducing Thread Migrations

There are a number of simple Java constructs that can potentially cause excessive thread migrations. See, for example, the code in Fig. 9.4. If the arrays ‘a’ and ‘b’ are allocated on two different cluster nodes, each array element comparison will cause two thread migrations (once to the machine holding ‘a’ and once for going back to access ‘b’).

For this reason, we provide a small class library containing elemental operations on arrays. To be exact, we provide methods for fast addition, subtraction, multiplication, and division of two arrays. In addition, Java’s class library already offers `java.util.Arrays.equals()` and `java.lang.System.arraycopy()` to compare and copy two arrays. LVM’s optimized methods test if both arrays are local, and if so, they do a local operation. If one of the two arrays is remote and the other one is local, the local array is sent to the remote cluster node which then executes the operation locally. This reduces the communication load to a total of two messages instead of 2·N messages for an
N element array. To reduce the load on the heap, LVM does not allocate the remote copy on the garbage-collected heap, but instead it is allocated on the system heap. This reduces the pressure on LVM’s garbage collector. Note, that because we allocate objects on the system’s heap we bypass LVM’s swapping mechanism as well. For this reason we reserve a bit of the system’s memory for this purpose in advance.

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

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

9.2.5 Distributed Garbage Collection

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

We preferred a distributed mark-and-sweep collector over a copying collector (generational or otherwise) since the latter waste half of the memory which is intolerable given our project’s goal of an efficient huge object space (in our benchmarks, intra-segment free-list fragmentation is no problem). Moreover, unlike some distributed garbage collector schemes, we do not separate into local and a global garbage collection phases, again due to memory concerns: to support machine-local GC’s, a machine must keep track of incoming references, which can grow to a large set. Also, the gains compared to only using a global GC are low [117]. Hence, LVM starts a garbage collection phase whenever a cluster node hits its local heap usage boundary. It then requests a GC thread to be started on every cluster node.

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

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

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

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

9.3 Performance

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

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

9.3.1 Micro Benchmarks

To measure the performance of object allocation and object access, we allocate (see Fig. 9.5) and traverse (see Fig. 9.6) linked lists of increasing lengths. To be exact, we start the program, create and traverse a list in a loop (10 iterations), and exit the program. After a list summation, each list becomes garbage. The VM is restarted for each new list length.

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

To show that LVM-directed swapping is much more efficient than OS-level swapping we disabled LVM’s swapping module (see LVM+OS-swap numbers) and instead relied

\footnote{Instead of using physical pointers that become invalid when a segment is swapped in at a different memory address, LVM implements the free list as offsets from the start of the segment to the next free space within the segment.}
9.3. PERFORMANCE

Figure 9.5: LinkedList Creation.

Figure 9.6: LinkedList Traversal.
on the OS’s virtual memory implementation. Note that in OS-LVM, the code still contains calls to `refToObjectPtr`. It is interesting to see that with OS-level swapping the system becomes very unresponsive as soon as the OS starts to compete for memory against the JVM or the LVM-OS version. This competition also impacts messaging speed as I/O buffers compete for memory as well. When LVM’s internal swapping is enabled, OS performance does not suffer because enough memory is always reserved for it.

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

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

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

### 9.3.2 Application Benchmarks

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

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

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

The memory requirements (see Table 9.1) are extreme due to the number of states that need to be explored and the corresponding hash tables for them. To reduce the number of thread migrations, JCheck heavily uses the optimized arraycopy, treeCopy, and treeEquals methods (see Section 9.2.4). Thread migration mostly happens whenever a thread attempts to publish a new state in hash tables belonging to other threads.
Lazy swap GC is a big gain for JCheck; GC is three times faster with it which shows most clearly on the 1 machine measurements (where run time is only 2.3 times faster as GC is only a portion of the run time). When using more than one machine, preformance is greatly influenced by the speed in which the thread’s hash tables are kept up-to-date to allow pruning of the search space. With eight machines (threads), this becomes hard. The increased heap usage with eight machines is caused by missing search-space pruning opportunities (and thereby lowering speedup). Swap compression allows a 1 MB segment to be compressed to a 65 KB file on average.

The Griso sub-graph locator finds occurrences of a graph P in another graph K. Since nodes and edges may be rotated, a complicated graph isomorphism test is needed. The algorithm first creates a set of permutations of P with the outgoing edges of each node permuted to create a set. This set is reduced by only allowing canonical forms of the graphs into the set (while also converting K to its canonical form). The set is then partitioned into N parts, so that each of N worker threads can locate embeddings of a permutation of P in K.

Memory consumption (see Table 9.2) is large since all canonical forms of the permutations of P need to be stored (again excluding a standard JVM). Fortunately, with increasing numbers of machines, the memory requirements per machine drop such that with the graph sizes that we have chosen, with 8 machines the graphs almost fit in the cluster’s memory (1864 MByte per node, with an LVM limit of 1700 MByte causes 164 MByte worth of graphs on average that need to be swapped). This results in the superlinear speedup seen when going from 4 to 8 machines. Unfortunately, Griso can take very little advantage of the class-libraries provided. This causes the high thread-migrations counts.

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

### 9.4 Related Work

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

**Operating systems.** Swapping and compression of swap space is a technique usually associated with operating system implementations and out-of-core applications.
In [37] Linux was adapted to compress MMU pages before swapping them out to disk. A small cache is used to store pages being compressed. In [30,100] Linux is adapted to divide memory into two parts. One holds compressed pages, the other uncompressed. Instead of swapping out an uncompressed page, the system first attempts to compress the page and to place it into the compressed memory area. That avoids many disk-IO operations. Our system is implemented on top of an OS. Hence, LVM is not restricted to the MMU’s 4K page sizes, it is portable, and allows for multiple techniques to reduce memory pressure (escape analysis, lazy GC swapping, etc.).

Operating systems can also increase a process’s available memory by remote swapping or remote paging. One approach is described in [51]. Here, a special I/O device ‘nswap’ is registered in the kernel. Related to this is [49], where the lowest-level page-manager in an operating system is made cluster-aware. Both approaches perform remote paging to allow idle nodes to cache pages of heavily loaded nodes to decrease reliance on slow disks for swapping. In contrast, LVM starts to remote-allocate objects once local memory is full and performs thread migration to access them afterwards.

**Distributed garbage collection.** A modified Linux notifies the Jikes RVM in [58] that a page is about to be swapped out. Whenever this happens, the GC creates a list of outgoing references from that page. Objects on that page are then part of the root-set for the GC’s marking phase. All references to objects on swapped-out pages are ignored in subsequent collections. In contrast to LVM, RVM is restricted to a single machine and to the size of the OS level virtual address space. Closest to our distributed garbage collector is the system of [117]. However, it targets ABCL/f instead of Java, uses a traditional data Fetching DSM system (with the associated memory overheads discussed above), and assumes that all data fits into core memory.

**Locality directives.** Related to our locality based directives is ccmalloc [31], an alternative to malloc that allows to allocate something close to some other ccmalloaced block of memory. However, the authors’ goal is cache optimization instead of swap optimization. Moreover, they target C instead of Java.

**Out-of-core & DSM.** There have been a number of Java-based DSM systems. An overview of DSM systems can be found in [98]. We will, however, concentrate on out-of-core in combination with DSM.

The interaction between, out-of-core applications, compilers, and execution on a DSM system is investigated in [19]. The authors perform source code analysis to add an inspector-executor style parallelization method. An inspector finds probable data usages and hands these over to the executor for the execution of the program. We perform no source code analysis to detect parallelism but rely on Java threads explicitly created by the programmer. Moreover, our different DSM style that relies on thread migration instead of data fetching is advantageous for memory-greedy applications. The compiler analysis for out-of-core applications in [20] inserts prefetch instructions to fetch array data from disk. The techniques described here are orthogonal to LVM: instead of inserting prefetch instructions, the LVM front-end compiler could try to call refToObjectPtr as early as possible.

LOTS [24] is closest to LVM. It is also a DSM that can swap out objects to disk. However, the mechanisms and techniques are quite different. LVM compresses data on disk while LOTS does not. Furthermore, LOTS can only use a third of the available memory/disk space for storing objects. LVM uses a virtual machine approach, while LOTS is provided as a C++ library. LOTS is an object-based DSM that migrates data and that therefore pays the memory penalty for storing proxy objects, diffs, and twins of pages. These overheads sum up significantly so that LOTS cannot support large address spaces, especially when large numbers of small objects are used. LVM uses thread
migration and has no per-object DSM overheads. Finally, LVM manages standard Java code and migrates threads through a cluster automatically. LOTS requires manually inserted acquire and release statements to control data consistency and to use the C++ library constructs provided.
Chapter 10

A DSM protocol aware of both thread migration and memory constraints

Authors:
Ronald Veldema\textsuperscript{1}, Bradford Larsen\textsuperscript{2}, Michael Philippsen\textsuperscript{1}

\textsuperscript{1} University of Erlangen-Nuremberg
Computer Science Department
Martensstr. 3, 91058 Erlangen, Germany
veldema@cs.fau.de, philippsen@cs.fau.de

\textsuperscript{2} University of New Hampshire
Computer Science Department
brad.larsen@gmail.com

Published in:
Proceedings of the 20th IASTED International Conference on. Parallel and Distributed Computing and Systems
Conference: PDCS ’08
Orlando, Florida, USA
16 – 18 November, 2008
Pages 291-295
ISBN 978-0-88986-773-4
Publisher: ACTA Press, Anaheim, CA, USA
Editor: Teofilo F. Gonzalez
(This paper received a best paper award)

The previous Chapter’s paper described the base DSM system for memory conservation. However, some level of object/array caching is needed for good performance which is contra to our goal of memory conservation. We therefore only selectively cache objects and only up-to a given memory limit. The main contribution in this paper is to combine object caching and thread-migration while maintaining cache coherence.
Abstract

A DSM protocol ensures that a thread can access data allocated on another machine using some consistency protocol. The consistency protocol can either replicate the data and unify replica changes periodically or the thread, upon remote access, can migrate to the machine that hosts the data and access the data there. There is a performance trade-off between these extremes. Data replication suffers from a high memory overhead as every replicated object or page consumes memory on each machine. On the other hand, it is as bad to migrate threads upon each remote access since repeated accesses to the same distributed data set will cause repeated network communication whereas replication will incur this only once (at the cost of increased administration overhead to manage the replicas).

We propose a hybrid protocol that uses selective replication with thread migration as its default. Even in the presence of extreme memory pressure and thread-migrations, our protocol reaches or exceeds the performance that can be achieved by means of manual replication and explicit changes of the application’s code.

10.1 Introduction

There are many problems that require a large memory, larger than a single machine’s core memory or even a whole cluster’s combined core memories. To name some examples: combinatorial search problems, problems that use large graphs, particle simulations with large numbers of particles, etc.

The DSM protocol presented here solely addresses these classes of problem sizes where swapping is always needed. It is implemented as an extension of LVM [123], a virtual machine for Java that adds a distributed shared memory. LVM supports these large problem sizes efficiently by implementing its own swapping of objects to disk instead of relying on the operating system. In a cluster context, each machine adds its memory and disk space to the available global memory. Thread migration is used to access remote objects. However, to avoid excessive thread migration selective object replication is needed. This paper presents such a protocol that limits object replication to curb both memory usage and the amount of thread migration required.

We use thread migration by default for two reasons. First, all DSM protocols that fetch data for their operation—whether using lazy-, release-, entry-, or scope-consistency protocols—require copies of data for calculating local changes. In effect this at least halves available application memory. Second, we cannot afford to fetch huge numbers of objects locally.

Contributions in this paper are: how to allow both replication and thread migration in the same protocol efficiently, how to limit replication to a fixed pool of memory, and a simple heuristics to decide which objects to replicate. Note that no other DSM protocol that we know of fully addresses the problem of maintaining memory consistency of shared data under thread migration.

10.2 Related Work

Most DSMs (including the ones mentioned here) assume that applications completely fit into memory and that sufficient memory is available to keep two or more copies of all data. Our protocol only replicates a small part of the data. No other system performs
10.3 LVM

The effectiveness of our previous LVM prototype is shown in [123]. That prototype relied solely on thread migration for remote object access. I.e., whenever a remote object is accessed, the remote reference is examined to see which machine owns the remote object and the thread is sent to that machine to access the object locally.

Each cluster node’s local address space is divided into 1 MB segments. In an object reference, we encode the machine number, the segment number, and the offset within the segment as shown in Fig. [10.1].

Because references are not direct memory addresses they must be decoded: Before each reference usage there is a call to refToObjectPtr that, given a reference, returns the memory address in the local machine. There is of course a compiler pass that eliminates

our lazy diff-pulling on thread migration or limits the amount of memory available for replication.

Some page-based DSM systems, e.g. the Coherent Virtual machine (CVM) [118] and Millipede [70], can improve performance by selectively applying thread migration. In general, if a given page is written often enough, thread migration is applied (where we do thread migration by default and replicate objects with a lazy self-consistency protocol). Unlike with LVM, with Treadmarks/CVM/Millipede, the available memory does not grow when machines are added.

CRL [71] is a DSM library (for C) that provides an API for wrapping regions of memory to shared-objects (start-read(X), start-write(X), end-read(X) or end-write(X)). Upon start-read/write(X), the region X is mapped locally. MCRL [62] extends CRL with thread migration. A start-write(X) now causes migration of the current activation record to the machine that hosts X. Under some heuristics, some reads cause computation migration as well. Overall, we differ from MCRL's protocols in various ways: we perform thread migration by default and only optionally replicate. Also, we use an update protocol rather than a caching protocol. Neither CRL nor MCRL are memory-aware and both are implemented as libraries where we use a VM approach that is transparent to the programmer.

Jessica2 [130] is a JVM that allows both data caching and thread migration. Data caching is used for remote object access. A migration policy allows the home of an object (i.e. where its meta-data is maintained) to change. The decision is based on a comparison of machine access ratios. Thread migration is used here to allow a programmer to implement application level load-balancing where we use thread migration to manage remote access and use data replication to reduce communication load. Also, Jessica2 is not memory-aware, whereas LVM is.

An overview of DSM systems can be found in [98]. We will focus on out-of-core combined with DSM.

LOTS [24] is closest to LVM. It is also a DSM that can swap out objects to disk. However, its mechanisms are very different. Furthermore, LOTS can only use a third of the available memory/disk space for storing objects (due to its traditional DSM that requires diffs/twins) so that no large numbers of objects can be used. Also, LOTS uses thread migration and has no per-object DSM overheads except for objects replicated with the memory bounded replication facility introduced in this paper, which is specifically designed to use only a fixed, small amount of memory. Finally, LOTS requires manually inserted acquire and release statements to control data consistency and to use the C++ library constructs provided. All of this is generated by the compiler in our approach.
superfluous repeated calls to refToObjectPtr within basic blocks.

```c
javaObject* refToObjectPtr( object_reference_t ref) {
    if (is_remote(ref)) {
        if (is_locally_replicated(ref)) ref = get_replica(ref);
        else migrate_thread(ref, get_machine(ref));
    }
    Segment*s = locate_segment(ref.seg_number);
    javaObject *q = s->data + (32 * get_seg_index(ref));
    return q; }
```

Figure 10.2: Decoding a reference to an object.

The cost for refToObjectPtr (Fig. 10.2) is non-trivial, but not extreme. The important part is the call to is_locally_replicated(ref) which ensures replica use. If no local replica is available, thread migration is performed. Except for this new line, the code is the same as that of [123] and needs not be understood further.

10.4 Replication Protocol

If an object causes excessive migration and the read-to-write-ratio is high, higher application performance and better load balance can be achieved by replication. Our replication protocol is correct with respect to the Java Memory Model (JMM) in that we guarantee JMM semantics for properly synchronized programs but not for incorrectly synchronized programs.

Our replication protocol is aware of memory constraints and is able to perform well with thread migration. The protocol limits the maximal amount of memory that is used for replicated objects, which are managed under an update protocol—changes to an object are synchronized with the copies of that object. Of course replication consumes memory for replicas and twins, that are necessary for later calculating what changes have been made. But by only allowing a limited amount of memory for replication, this concern is greatly alleviated.

To enable use of the replication protocol the forceReplicate library function must be invoked on an object. In response the protocol then replicates the object on all machines. This method can either be inserted into the program by the programmer, or it is automatically called by our replication heuristics (Section 10.4.3). If a machine does not have enough memory left for replication purposes, a machine can deny a replication request. The cluster node that owns the object keeps track of which other nodes have successfully created a replica of that object.

To implement replication, each LVM instance maintains a hash table of references to object/replica pairs (diffs are thus per object). As shown in Fig 10.2 every time a
remote reference is accessed, refToObjectPtr searches the hash for a local replica. If none is found migrate_thread is called.

For performance reasons and to allow the programmer a modicum of control over the replication protocol, replicability must be annotated to a class type explicitly:

```java
#pragma replicable

class Data {
    int value;
    Data() { RuntimeSystem.forceReplicate(this); }
}
```

We only generate the call to is_locally_replicated(ref) in refToObjectPtr for such annotated types. This increases performance of non-replicated object access as it avoids a useless hash lookup and results in smaller code. Because Java array types cannot be given names (cf. typedef in C), a use-case distinction of different arrays is hard. All array types are therefore implicitly replicable. The general rule for adding the replication annotation is to do so only for objects that are relatively seldom allocated.

To allow polymorphism for annotations, we use static type analysis in cooperation with two replication pragma usage rules. First, for any type X marked replicable, all sub-classes of X are automatically replicable. Second, only classes that directly inherit from java.lang.Object can be marked replicable.

### 10.4.1 Replication Protocol Implementation

The replication protocol must interact with three pre-existing subsystems: the thread migrator, the synchronization subsystem, and the garbage collector. We discuss these in turn.

Replication must interact with thread migration because to implement the semantics of the Java memory model, changes to objects need to be made available to the modifying thread after its own migration. Consider a scenario where a thread changes replicated object P on machine 0, then migrates to machine 1 where it accesses a object Q. If it then again uses P but this time on machine 1, the thread should see the change that it made earlier when it was still running on machine 0.

![Figure 10.3: Lazily pulling thread-local changes.](image)

To implement this, we could require that threads eagerly pull their changes to replicated objects upon migration. Since this would transmit the full pool of replicated objects on each thread migration, we instead pull these changes lazily. Each thread maintains a table of object reference/node number pairs, where the node number is that of the machine that holds the most recent copy of the object. When a thread accesses
an object that it has previously modified at a different machine, it pulls the diff from
the node with the most recent copy, applies the diff locally, and updates the table to
indicate that the current machine now has the most recent copy. This is illustrated in
Fig. 10.3. We see LVM running on two cluster nodes. Machine 0 has the original of
object A and machine 1 has a replica. Since thread 0 has modified A (1), the original
A is out-of-date. When thread 0 modifies the replica of A, it adds the entry \{Reference
of A, 1\} to its local changes table. Next, thread 0 migrates to machine 0 (2). There it
references A (3). Because there is a local version of A, migration is unnecessary. Instead,
the thread consults its migrated local changes table, finds the \{Reference of A, 1\} entry,
and pulls the changes from machine 1 (4). Once the changes have been applied to the
local copy, which in this case is A’s original version, thread 0 can finally access
A (5).

Replication also affects the synchronization protocol as the latter is tightly coupled
to Java’s memory model. Multiple copies of an object must be kept in sync. Our solution
is to let a thread publish its changes whenever it executes a monitor operation, i.e. upon
lock, unlock, wait, notify, notifyAll. Upon any of these operations, the executing thread
flushes its local changes table, causing all copies of that object to be synchronized. For
properly synchronized programs, these updates are atomic since shared variable access
requires lock/unlock for mutual exclusion from the programmer.

To save space we cannot go through the protocol line-by-line, instead we discuss it
globally: for each entry in the thread’s local changes table, the thread tells the machine
with the most recent version to send its changes to the object’s owner. The object’s
owner applies the diff to the original and forwards the diff to every other machine with
a replica of that object. Each replicator next applies the diffs to its local copies. Since at
this point, all copies of the object have an identical state, the object’s owner can tell the
initiating machine that synchronization is complete.

Finally, to free unused replicas and twins, the replication mechanism must also
interface with the garbage collector. The logic is straightforward: replicas can be
collected, whenever the ‘original’ is collected. Thus, whenever, the ‘original’ is marked,
we send messages to mark its replicas to also keep them alive. During the GC sweep
phase, unmarked replicas/twins/objects are disposed of.

10.4.2 Volatile Variables
If in Java a field is marked ‘volatile’, it is guaranteed that first, no other access is
reordered over the volatile access and that, second, the cache is flushed after the
manipulation of the volatile field. We implement the first requirement in the compiler
by tagging instructions in our intermediate language as ‘volatile’. This causes the
instruction to be treated the same as calls to lock and unlock. The second requirement
is ensured by letting the compiler generate a call to Thread::flush(). Hence, a volatile
access has the same (protocol-level) effect as lock or unlock.

10.4.3 Automatic Replication
Intuitively, replication of an object is a good idea if the number of reads is high and the
number of modifications is low. However, it can be hard for the programmer to correctly
detect objects with such a favorable read-write ratio and to insert forceReplication calls
to the appropriate objects. We therefore provide a simple heuristics that automatically
calls forceReplication once an object has caused a threshold number of thread migrations
of a suitable read/write ratio (each thread migration is caused by either reading or writing
a field). However, we still require potentially replicable objects to be marked replicable
so that for most objects, the costly replica hash-lookup can be avoided. Currently, we require a 4:1 read to write ratio with at least a hundred migrations caused by a read. This simple heuristic may fail in several cases. First, to keep memory consumption low, the hash table needs to be fixed to a small size. Otherwise, if every Java reference that causes replication would be recorded the hash would grow huge (especially given LVM’s goal of a large object-space). The small size causes us to overwrite hash entries in case of key collisions. A too high number of key collisions can effectively disable replication.

The second problem is caused by an increment of a wrong counter. Consider a read-caused migration followed by a write access. At the target machine, the thread immediately after arrival modifies another object that lives on the target machine. Since the read counter is incremented but the write counter is not, the replication engine might incorrectly assume that replication is a good idea. Here is a (simple) example: \( \text{tmp = obj1.field; tmp = tmp + 1; obj1.field = tmp; } \) When execution migrates to the machine that holds ‘obj1’, the counters will record this migration to be caused by a read. The counters will ignore the subsequent (local) write operation. There are many variations on this theme.

Finally, the heuristics do not monitor the frequency of synchronization actions nor their costs. If synchronization is needed frequently, fewer objects should be replicated. Also, larger objects should be replicated less aggressively for the same reason. Ideally, the observed frequency should be dynamically/adaptively applied to the read/write heuristic we already use. Such heuristics are however delicate.

The programmer can combat these effects in two ways. First, one can disable automatic replication if the heuristics turn out to be too simplistic, or second, one can experiment with different applications of the \textit{replicable} pragma.

\subsection{Benchmarks}

Our cluster nodes are quad-core Xeon “Woodcrest” processors running at 3.0 GHz with 4 MB Shared Level 2 Cache per dual core and 8 GB of RAM. Although the machines have 8 GB RAM, we restrict LVM to 1.7 GB to artificially increase memory pressure. The Infiniband interconnect can communicate with 10 GBit/s bandwidth per link and direction. LVM, internally uses MPI for communications. We use Intel MPI 3.1.038 over Infiniband for our measurements. Standard JVM measurements are performed on a standalone machine equipped with only 2 GB of physical RAM. This ensures that LVM and JDK run with the same memory pressures. We use Sun JDK 1.6 with standard options which is referred to as “JDK” below.

For each of our three applications we consider four versions. In \texttt{no-repl} we have no replication at all. In \texttt{manual-repl} we did not use replication annotations but instead for each object that needs replication, we allocate a per-thread copy manually by changing the application codes. Third, \texttt{force-repl} uses the no-repl code versions but adds \texttt{forceReplication} and the \texttt{replicable} pragma. Finally, \texttt{auto-repl} adds the \texttt{replicable} pragma only and lets the auto-replication heuristics select objects to replicate. All times reported are in seconds and are wall times, the mean numbers show the per-machine numbers. We do not compare against raw MPI implementations of these benchmarks for two reasons: the parallelism and programming models are different, and they would rely on the operating system’s swapping mechanism.

\textbf{Ocean}. Ocean is a Java port of the corresponding Splash2 code \cite{128}. Ocean studies large scale water movements in an ocean based on eddy and boundary water currents. It
**Table 10.1: Ocean results.**

<table>
<thead>
<tr>
<th>Mean Thread Migration Count</th>
<th>Mean Max Heap Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machines</td>
<td>1</td>
</tr>
<tr>
<td>manual-repl</td>
<td>n/a</td>
</tr>
<tr>
<td>force-repl</td>
<td>n/a</td>
</tr>
<tr>
<td>auto-repl</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Wall Time (seconds)

<table>
<thead>
<tr>
<th>Machines</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>manual-repl</td>
<td>3172</td>
<td>1390</td>
<td>688</td>
<td>345</td>
</tr>
<tr>
<td>force-repl</td>
<td>3336</td>
<td>1403</td>
<td>695</td>
<td>358</td>
</tr>
<tr>
<td>auto-repl</td>
<td>3423</td>
<td>1388</td>
<td>683</td>
<td>339</td>
</tr>
<tr>
<td>JDK</td>
<td>64041</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

is implemented using a red-black Gaus-Seidel multigrid equation solver [18].

Ocean’s main data-structure are a number of 4D arrays of doubles. Only the outer arrays are modified, and the inner dimensions are ideal for replication. This is especially suited as the outer dimension(s) will occupy most memory. The relevant statistics are shown in Table 10.1. Unfortunately, the no-repl version is so slow that it does not run under the time limits imposed by our cluster’s fair use policies.

For all versions, we can see that speedup is good. Ocean’s auto-repl version wins over the manual-repl version (for 2 or more machines) because it is hard to find and copy the correct data structures in the manual-repl version. In the force-repl version we replicated any replicable object. Over those objects, the auto-repl version replicated some objects that were hard to find manually. Both auto-repl and force-repl we gave 256 MB of replication memory, which is small relative to the heap size. The heap size for manual-repl is slightly larger as it replicates some objects unnecessarily (see two machine case).

**JCheck.** JCheck is a model checker for programs written in a Java dialect. Starting from some initial state, it tries all possible thread interleavings (the state-space) to find reachable error states. Each state consists of at least 14 objects including a large array.

Each machine stores a copy of the hash table of states already tried to avoid duplicate parallel searches. These duplicated hash tables cause (application level) loss of parallelism as each machine’s hash needs to be kept reasonably in sync with the other’s. In JCheck it pays to replicate the often-accessed, read-only control structures. Also, to reduce the number of thread migrations, all versions of JCheck heavily use the optimized arraycopy, treeCopy, and treeEquals methods (see [123]). The results are given in Table 10.2.

Where manual-repl takes under an hour (2297s) on one machine, JDK (using OS-swapping) takes over 10 hours (37515s) on that machine. For either VM, time is mostly dominated by swapping. The overhead of doing the replica-lookup per object access can be clearly seen in JCheck: manual-repl has no lookups where force-repl/auto-repl do. This causes the difference of 2297s vs. 2866s using only one machine. With 8 machines, search space pruning becomes hard as threads can’t synchronize their hash tables fast enough. This causes the slow down of manual-repl when going from 4 to 8 machines.

The force-repl and auto-repl versions win over manual-repl in JCheck because with manual-repl object equality of two replicated objects must be implemented by comparing object contents. Instead, with force/auto-repl a reference comparison of
10.5. BENCHMARKS

Table 10.2: JCheck Results.

<table>
<thead>
<tr>
<th>Machines</th>
<th>Mean Thread Migration Count</th>
<th>Mean Max Heap Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Machines 1 2 4 8</td>
<td>Machines 1 2 4 8</td>
</tr>
<tr>
<td></td>
<td>no-repl n/a 962109 631524 513232</td>
<td>no-repl 10442 4962 4895 2592</td>
</tr>
<tr>
<td></td>
<td>manual-repl n/a 44234 91266 38582</td>
<td>manual-repl 10455 7654 3963 2828</td>
</tr>
<tr>
<td></td>
<td>force-repl n/a 30865 40773 169388</td>
<td>force-repl 10443 5270 5932 3582</td>
</tr>
<tr>
<td></td>
<td>auto-repl n/a 32408 30869 66202</td>
<td>auto-repl 10442 4926 4895 2592</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wall Time (seconds)</td>
</tr>
<tr>
<td></td>
<td>no-repl 2498 2808 1934 1406</td>
<td>no-repl 2498 2808 1934 1406</td>
</tr>
<tr>
<td></td>
<td>manual-repl 2297 938 404 996</td>
<td>manual-repl 2297 938 404 996</td>
</tr>
<tr>
<td></td>
<td>force-repl 2866 1072 1787 370</td>
<td>force-repl 2866 1072 1787 370</td>
</tr>
<tr>
<td></td>
<td>auto-repl 2760 927 976 208</td>
<td>auto-repl 2760 927 976 208</td>
</tr>
<tr>
<td></td>
<td>JDK 37515 n/a n/a n/a</td>
<td>JDK 37515 n/a n/a n/a</td>
</tr>
</tbody>
</table>

Table 10.3: Griso Results.

<table>
<thead>
<tr>
<th>Machines</th>
<th>Mean Thread Migration Count (×1000)</th>
<th>Mean Max Heap Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Machines 1 2 4 8</td>
<td>Machines 1 2 4 8</td>
</tr>
<tr>
<td></td>
<td>no-repl n/a 34644 25242 12366</td>
<td>no-repl 7413 3715 1864 932</td>
</tr>
<tr>
<td></td>
<td>manual-repl n/a 83 294 506</td>
<td>manual-repl 7384 3700 1856 934</td>
</tr>
<tr>
<td></td>
<td>force-repl n/a 9 19 15</td>
<td>force-repl 7384 3700 1856 934</td>
</tr>
<tr>
<td></td>
<td>auto-repl n/a 96 6480 295</td>
<td>auto-repl 7384 3697 1854 930</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wall Time (seconds)</td>
</tr>
<tr>
<td></td>
<td>no-repl 10121 176400 54600 21180</td>
<td>no-repl 10121 176400 54600 21180</td>
</tr>
<tr>
<td></td>
<td>manual-repl 10121 2427 847 406</td>
<td>manual-repl 10121 2427 847 406</td>
</tr>
<tr>
<td></td>
<td>force-repl 9760 2963 745 764</td>
<td>force-repl 9760 2963 745 764</td>
</tr>
<tr>
<td></td>
<td>auto-repl 11637 4896 3608 797</td>
<td>auto-repl 11637 4896 3608 797</td>
</tr>
<tr>
<td></td>
<td>JDK 389629 n/a n/a n/a</td>
<td>JDK 389629 n/a n/a n/a</td>
</tr>
</tbody>
</table>

replicas suffices. The overall winner is auto-repl with 208s on 8 machines. Even though our replication heuristics is very simple, it is competitive compared to force-repl and manual-repl by being more aggressive in replication. Note that the force/auto-repl versions required far fewer code changes than the manual-repl version and that we only allow 1 MB of memory for replication in JCheck.

**Griso.** The Griso Subgraph Locator finds occurrences of a (sub) graph P in a (super) graph K. Due to potentially rotated nodes/edges this requires costly graph isomorphism tests. Memory consumption is large since all canonical forms of the permutations of P need to be stored. Fortunately, the memory consumption scales with the number of available machines. The results are shown in Table 10.3.

JDK performance (389629s) suffers from the semi-random memory accesses that do not affect LVM as its allocation regime automatically puts related nodes/edges on the same segment.

The results clearly show that some version of replication is absolutely necessary. Although our thread migration is fast, performing billions of migrations is deadly. We allow the protocol maximally 256 MB of memory.
Note that with few machines, the speed in which the auto-repl version "learns" which things to replicate (the nodes and edges of the super graph) is slow, which is seen in the high thread migration counts. Only when using 4 – 8 machines sufficient data is gathered for the heuristic to work. Heap usage for all protocol versions are about the same. Manual-repl wins with respect to speedup because it has no administrative data to maintain per replica.

10.6 Conclusions

We have described a DSM protocol that by default uses thread migration and applies selective object replication to remove excessive thread migrations. A surprising result is that with only little memory set aside for replication (1MB for JCheck, 256 MB for Ocean and Griso), the number of thread migrations already shrinks to reasonable levels. Both automatic and forced replication also nicely alleviate the programmer from the task of manual replica management. A simple heuristics to decide what to replicate by counting how many migrations were caused by read and write accesses has shown to be very effective. It is either more aggressive in how soon to replicate or it finds objects that benefit from replication and that the programmer has overlooked. By annotating which types are candidates for replication, replica statistics management is easier (and therefore the heuristics can be too). Overall, the heap sizes are about the same.
Bibliography


