IMPLEMENTATION AND OPTIMIZATION OF THREAD-LOCAL VARIABLES FOR A RACE-FREE JAVA DIALECT

by

Yi Zhang

School of Computer Science
McGill University, Montréal

December 2011

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Despite the popularity of Java, problems may arise from potential *data-race* conditions during execution of a Java program. *Data-races* are considered errors in concurrent programming languages and greatly complicate both programming and runtime optimization efforts. A *race-free* version of Java is therefore desirable as a way of avoiding this complexity and simplifying the programming model.

This thesis is part of work trying to build a *race-free* version of Java. It implements and optimizes *thread-local* accesses and comes up with a new semantics for this language. An important part of implementing a language without races is to distinguish thread-local data from shared data because these two groups of data need to be treated differently. This is complex in Java because in the current Java semantics all objects are allocated on a single heap and implicitly shared by multiple threads. Furthermore, while Java does provide a mechanism for *thread-local* storage, it is awkward to use and inefficient.

Many of the new concurrent programming languages, such as OpenMP, UPC, and D, use “sharing directives” to distinguish shared data from thread-local data, and have features that make heavy use of thread-local data. Our goal here is to apply some of these language ideas to a Java context in order to provide a simpler and less error-prone programming model. When porting such features as part of a language extension to Java, however, performance can suffer due to the simple, map-based implementation of Java’s built-in `ThreadLocal` class. We implement an optimized mechanism based on programmer annotations that can efficiently ensure class and instance variables are only accessed by their owner thread. Both class and instance variables inherit values from the parent thread through deep copying, allowing all the reachable objects of child threads to have local copies if syntactically specified. In particular, class variable access involves direct access to thread-local variables through a localized heap, which is faster and easier than the default map mechanism defined for `ThreadLocal` objects. Our design improves performance significantly over the traditional thread-local access method for class variables and provides a simplified and more appealing syntax for doing so. We further evaluate our approach by modifying non-trivial, existing benchmarks to make better use of thread-local features, illustrating feasibility and allowing us to measure the performance in realistic contexts. This work is intended to bring us closer to designs for a complete race-free version of Java, as well as show how improved support for use of thread-local data could be implemented in other languages.

Résumé

Malgré la popularité de JAVA, de potentiels accès concurrents aux données peuvent causer des problèmes à l'exécution d'un programme. Les accès concurrents aux données sont considérés comme des erreur par les langages de programmation et compliquent grandement le processus de programmation et d'optimisation. Une version de JAVA sans accès concurrents serait la bienvenue et simplifierait ce processus.

Cette thèse n'est qu'une partie d'une recherche plus importante visant à établir une version de JAVA *sans accès concurrents*. Elle implémente et optimise les accès en *thread local* et introduit une nouvelle sémantique pour ce langage. Une part importante de l'implémentation d'un langage *sans concurrence* est de distinguer les données locales de thread des données partagées car ces 2 types de données doivent être traitées différemment. Ceci est complexe en JAVA, car avec la sémantique actuelle, tous les objets sont alloués en un seul tas (heap) et implicitement partagés entre plusieurs threads. De plus, le mécanisme de stockage en *thread local* de Java est étrange et inefficace.

Plusieurs des nouveaux langages concurrents, comme OpenMP, UPC et D, utilisent des "directives de partage" pour distinguer les données partagées des données locales de thread, et ont des structures faisant un usage avancé des données locales de thread. Notre but ici est d'appliquer certaines idées de ces langages dans un contexte JAVA dans le but de fournir un modéle de programmation plus simple et plus fiable. Cependant, apporter ces fonctionnalités sous forme d'extension a JAVA peut en affecter les performance du fait de la structure de la classe `ThreadLocal` de JAVA. Nous implémentons donc un mécanisme qui garantit efficacement que seul le processus propriétaire accède aux classes et variables d'instances. Aussi bien les classes que les variables d'instances héritent des valeurs du processus parent par copie, ce qui permet aux objets de processus enfants d'avoir des copies locales si précisé dans la syntaxe. En particulier, l'accès à des variables de classe utilise un accès direct aux variables du processus local via un tas local, ce qui est plus rapide et facile que le mécanisme par défaut de mappage défini pour les objet `ThreadLocal`. Notre conception améliore le performance de faon significative comparé à la méthode d'accès au processus local traditionnelle pour les variables de classe et fournit une syntaxe simplifiée et plus attrayante. Nous évaluons ensuite notre approche en modifiant des outils de test (benchmarks) complexes existants pour faire un meilleur usage de leurs fonctionnalités en processus local, ceci illustrant la faisabilité et nous permettant de mesurer les performances dans un contexte réaliste. Ce travail a pour but de nous rapprocher de la conception d'une version JAVA *sans concurrence* aussi bien que de montrer comment un support amélioré des données en thread local pourrait être implémenté dans d'autres langages.

Acknowledgements

I would like to thank my supervisor, Clark Verbrugge, who taught me a lot of academic knowledge and was always patient to my questions. He did not only help me with my study but also encouraged me to practice presentation in front of a large group of people and be social during conferences.

I would also like to thank my lab mates in Sable Research Group. They come from different countries with different culture but they are all very nice and knowledgeable. In particular I would like to thank Nurudeen Lameed, from whom I learnt both about compilers and life.

I would also like to thank my friends, nearby and far away, and my family, who have been constantly supporting and encouraging me.

This work was supported, in part, by the IBM Toronto Lab and the National Science and Engineering Research Council of Canada (NSERC).

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Java is a general-purpose, object-oriented language with built-in features for concurrent programming. The Java programming language and the Java class libraries provide basic concurrency support while high-level concurrency APIs provide more convenient tools. This feature makes Java convenient for multithreaded programs where several threads cooperate to finish a task. The memory management scheme in current Java facilitates communications between threads because all the objects are shared among threads.

Unfortunately, although this is convenient, it makes it all too easy for a programmer to write programs that contains *data-races*. *Data-races* occur when two or more threads try to access the same variable, the accesses are not ordered (protected) by synchronization, and at least one of the operation is a write. Data-races are typically considered errors in concurrent programming, and greatly complicate optimization efforts and language semantics. A potential solution to this problem is to provide and enforce separation between shared and unshared, or *thread-local* data. The use of *thread-local* data in shared-memory concurrent programming is typically seen as a performance enhancement, but also has important semantic implications. If threads have their own, unique copies of data, unsynchronized intermediate and local computation is possible, and this has the further important benefit of reducing or avoiding complex, subtle data-race issues [MPA05, BA]. This is a strategy taken in many concurrent languages and libraries; newer concurrent languages such as UPC [UPC05], OpenMP [Ope08], and D [DM] allow for extensive thread-local allocation through declarations, with little or no special data access syntax. In other popular concurrent languages and environments, such as Java and PThreads, however, thread-local storage is implemented through existing language mechanisms, in the case of Java requiring object creation and special method calls to create and use the thread-local data. This results in awkward and non-trivial syntax for programmers, as well as relatively high runtime overhead in practice, making the whole mechanism much less appealing.

In this work we develop an optimized implementation mechanism for thread-local data in a Java dialect. This effort has two major goals. First, we demonstrate that significant performance improvements are possible for programs that actually use thread-local data by using a straightforward, specialized implementation design for thread-local data. Modern memory management and the JDK implementation already uses thread-specific heaps for data that can be determined to be thread-specific through analysis [MPJ11], and our work complements that effort by showing improvements are also possible for programmer-specified thread-local data. Our work also aims at the larger goal of facilitating and encouraging *data-race-free* (DRF) language design and adoption. The DRF property has importance with respect to both programmability and to optimization potential; many ambiguities and complexities found in the current Java Memory Model [MPA05] arise from the need to define semantics for programs that do not have this property, while the presence or lack of a DRF property has a large impact on compiler optimization of multithreaded programs [AH90, Š11, MSM$^+$11, Boe11]. Providing strong, language-level, syntactic guarantees for thread-local usage thus has great value in terms of language design, including incremental dialects of popular languages such as Java, but is naturally predicated upon efficient implementation designs.

To further both goals we base our investigation and design on a DRF variant of Java, in which thread-local access is the *default mode* for all fields and it is shared data that must be specifically identified through language syntax. This allows us to develop the basic optimization and to build a context for deeper investigation of DRF Java designs. We used a modified *Jikes* RVM 3.1.1 [jik], wherein we implemented a "thread-local by default" memory system as well as optimized code-generation for thread-local data access.

1.1 Contributions

This thesis makes the following specific contributions:

- We propose a new method to realize the concept of thread-local data in a Java Virtual Machine, as currently provided by `ThreadLocal` objects. Our design avoids direct object allocation and allows for transparent data access, significantly improving both usability and access costs.
- To demonstrate and experimentally evaluate and our design, we implement our method in *Jikes* RVM 3.1.1 [jik]. Thread-local isolation of both the static and heap variables is taken care of through a *deep-copy* mechanism during the start-up period of each thread. This initial design supports basic *copy-in* mechanisms as found in other concurrent languages, and is amenable to further optimization through lazy copy mechanisms [ZDG+10].
- We evaluate our design using both micro-benchmarks to measure thread-local behavior itself as well as larger realistic programs. For the latter, we transform existing Java benchmarks to make full use of thread-local data, and experimentally evaluate both memory costs and runtime performance. Our data shows that while there is a significant memory impact for a program which dynamically creates many, short-lived runtime threads, most thread-local versions offer performance comparable to non-thread-local implementations, with the added advantage of being trivial to verify as DRF.

1.2 Outline

This thesis contains 7 chapters (including this introductory chapter). *Chapter 2* provides background information for this thesis. This includes a description of the overall architecture of *Jikes* RVM and some of its important components. It also includes a discussion about the meaning of our work which is motivated by our focus on achieving a DRF property through ubiquitous use of thread-local data. This is followed in *Chapter 3* by discussion of the existing Java thread-local facility and the requirements of a Java dialect context. *Chapter 4* gives a full description of our implementation design, which is then experimentally evaluated in *Chapter 5*. We describe related work in *Chapter 6*. Finally, conclusions and future work are presented in *Chapter 7*.

Chapter 2
Background

This chapter provides background information that is helpful in understanding the remainder of this thesis. We start with a brief description of *Jikes* RVM, the open source Java Virtual Machine (JVM), in *Section 2.1*. We will give an overview of some parts of the *Jikes* RVM architecture and explain how these parts are related to our work. This thesis is part of a larger project aiming at providing a complete implementation of a JVM for a race-free Java dialect. We will introduce the larger picture in *Section 2.2*.

2.1 *Jikes* RVM Framework

Jikes RVM is a Java virtual machine developed for research purposes. The most significant feature of this open source project is that the core architecture is implemented in Java. This feature makes it easier to implement and explore new ideas about virtual machines within the *Jikes* RVM framework. The architecture can be mainly divided into three components: the JIT (Just-In-Time) compilers, the runtime service, and the memory management system [jik].

2.1.1 JIT Compiler

Java class files contain an intermediate language called bytecode which can be executed on any platform that has a JVM. JVMs either use interpreters to execute the code by mimicking the underlying virtual machine or use compilers to convert the bytecode to machine code to execute directly. In the case of *Jikes* RVM, the latter approach is employed and multiple JITs are used to compile the code: a baseline compiler for rapidly producing simple compiled code, and an adaptive optimizing compiler that produces better code at greater cost.

The baseline compiler in *Jikes* RVM follows a template-based approach and uses a switch statement to deal with different bytecode instructions, generating corresponding assembly code instructions and further machine code instructions for each case, as is shown in Figure 2.1.

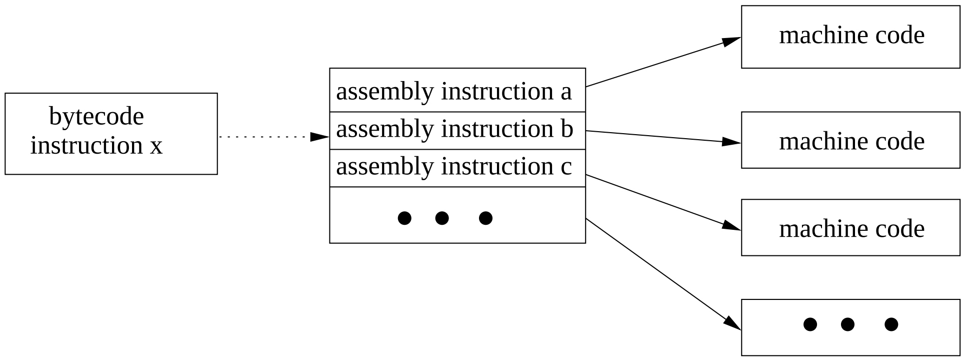


Figure 2.1 Code generation for a bytecode instruction. Here, bytecode “x” is compiled to three assembly instructions “a”, “b” and “c” which are then used to generate binary machine code instructions specific to the current architecture.

The baseline compiler does not perform any optimizations and the generated assembly codes behaves very much like the interpreter implementing a basic stack machine. An adaptive optimizing compiler can be used to further optimize the methods observed to be executed very frequently or which are deemed computationally intensive. In our implementation, we turn off the adaptive optimization compiler and perform implementations only within the baseline compiler. We choose the baseline compiler because it is easy and convenient to modify and to change the semantics of a bytecode, we just need to take control of the process of converting one bytecode instruction to several assembly code instructions, shown by the dotted arrow in Figure 2.1.

2.1.2 Runtime Service

The runtime service in *Jikes* RVM provides an environment for correct execution of Java application programs. We only introduce in detail the parts related to managing objects and threads which are most related to our work. Runtime service also provides the following functions:

- providing a special bootstrap mechanism to start the basic virtual machine
- triggering compilation and linking of methods
- managing generated code storage
- delivering hardware and software exceptions

Object Model

There are two categories of data-type in the Java language: primitive and reference data-type. Primitive variables like integer and double store the primitive value directly. A variable of reference data type stores the address of the object it points to. The object model defines how objects are represented in memory. An object is a piece of memory space which is separated into header and actual object fields as is shown in Figure 2.2. Each partition in the header holds specific information about the object. Furthermore, because each partition is at a fixed position within the object the virtual machine can easily get the needed information about the object by adding a certain amount of displacement to the object reference value. This makes accessing object fields and methods dispatching fast and convenient.

One special, important word in the header points to the Type Information Block (TIB) which applies to all the objects of the same type. The TIB has a *virtual method table* for invoking methods on an object and a pointer to another object representing its class. This will be introduced in the next section.

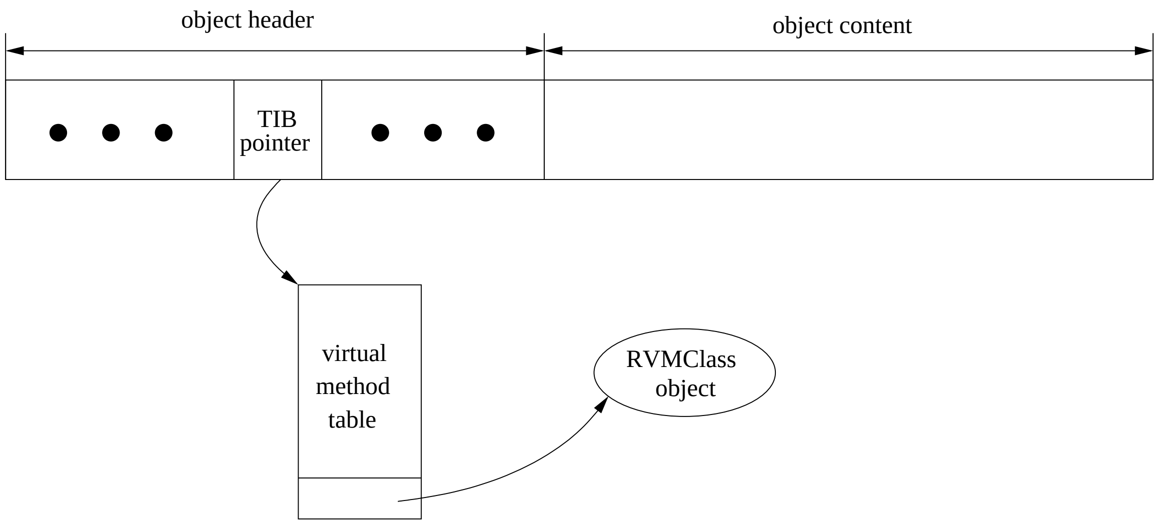


Figure 2.2 Object representation in memory

Class Representation

Classes in *Jikes* RVM are represented by an `RVMClass` object. Whenever a new class file is loaded, the class information is interpreted and stored in the corresponding fields of this object. Figure 2.3 shows the most important fields of the `RVMClass` class. The name of a field clearly tells what information is kept. For example, the `contructorMethods` field stores all the constructor methods, and the `modifiers` field uses a short variable to indicate the modifier information of the class (final, public/private, etc.). Similarly, each field of a class is represented by an object of `RVMField` class extending the `RVMMember` class giving information about the represented field. Figure 2.4 shows part of the `RVMField` class. Important to our work, the `modifiers` field can tell whether this field has the “volatile” modifier or not. Once a class file is loaded and resolved in *Jikes* RVM, its information is stored in form of an `RVMClass` object and information concerning each of its fields is stored in form of `RVMField` objects. Given any object, we can trace to its `RVMClass` object following arrows shown in Figure 2.2 making it quite convenient to retrieve information about itself and its fields.

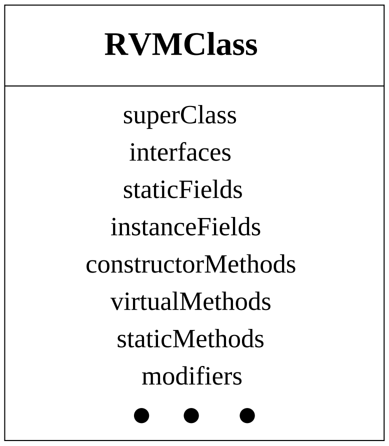


Figure 2.3 RVMClass

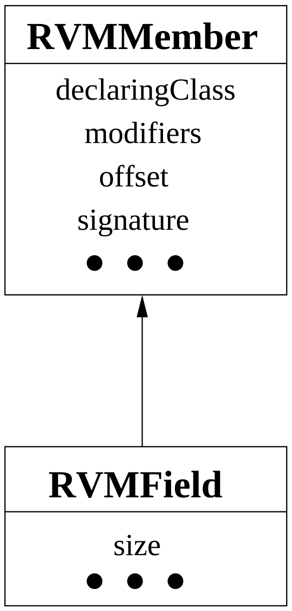


Figure 2.4 RVMMember and RVMField

Thread Management and Representation

All Java threads including application threads and garbage collector threads derive from `RVMThread`. Each `RVMThread` is bounded directly with a native thread which is scheduled directly by the underlying operating system. This greatly simplifies the task of Java threads scheduling since *Jikes* RVM does not take control of thread-processor mapping.

What is more of our interest is how Java threads are represented. An `RVMThread` object wraps exactly one Java thread. It has state fields to indicate the states of the Java thread, such as whether it is running or blocked. The `RVMThread` objects have their own `public final void start()` method which will put the Java thread into the queue of an available processor and call the `public void start()` method for the actual Java thread. Given this one-to-one relationship between the `RVMThread` object and the Java thread, we can easily add thread-specific space by modifying the associated `RVMThread` class.

2.1.3 Memory Management

The memory management system takes care of object allocation and garbage collection. The Memory Manager Toolkit (MMTK) in *Jikes* RVM supports a variety of types of memory management schemes. Different memory managers divide the virtual memory into spaces with different functionality and may choose different garbage collection schemes. In our implementation, we use the basic semi-space collector as one of the simplest collectors to modify and thus accommodate the way we generate duplicated data in supporting thread-local.

A semi-space collector is a copying collector that stops the program during garbage collection. It divides the space into two partitions. One is *from space* and the other is *to space*. When a program is executing, the objects are allocated into the *to space* until this space become full and the collection is triggered. Before the collection really starts, the *to space* becomes the *from space*, and vice versa. During the collection, the collector copies all the objects reachable by the program from the *from space* to the *to space*. As a result, the objects that are not reachable are left in the *from space* and get collected. This whole process is repeated until the garbage collection is finished and the program continues to execute. The semi-space collector is easy to understand but has the disadvantage of requiring very large contiguous space.

2.1.4 Linux/IA32 VM Conventions

Jikes RVM is fully supported on two main platforms, the AIX/PowerPC (PowerPC processors running AIX) and the Linux/IA32 (Intel 32-bit architecture running Linux) [ABC+02].

Our work is implemented on the Linux/IA32 platform. Linux/IA32 has its own stack, calling and register conventions. Our implementation should carefully comply with those conventions, especially the register convention since we change the generated assembly code manipulating registers. There are eight general purpose registers in the Linux/IA32 convention. For *Jikes* RVM, two of them are dedicated registers. Among them, ESI is the thread register pointing to the `RVMThread` object associated with the current thread which is very important to our implementation since we are dealing with thread specific spaces. The calling convention defines what should be done in the prologue and epilogue of a callee, meaning the way of passing parameters, passing return values and jumping back to the caller. How *Jikes* RVM passes parameters and the return value matters most to our project because our modifications to compiler generated code include inserting function calls. For this we use the EAX register for the parameter and return value. This register is considered a *caller save* register, meaning that it can be used for temporaries and the value is not preserved across calls.

2.2 Race-Free Language Design

Our work is intended to optimize thread-local variables in Java so that they are easier to program with and faster to access when compared with the current `ThreadLocal` class. The actual meaning of this optimization goes beyond improvements in execution speed, and was originally motivated by the need to offer a well-performing, but also simple and programmer-friendly data-visibility model for popular concurrent languages. A larger and more complete picture of the language design, motivation, and context is provided in previous work by the authors [VKZ11] and is summarized below.

2.2.1 Data-Race-Freedom and Optimization of Java Programs

While data-races have been long-recognized as errors or concerns in concurrent programming, current interest in ensuring data-race-free behaviour arises from recent work in developing memory model semantics for modern languages. Proving correctness and establishing the bounds of allowable compiler optimization is possible in the presence of data-race-free programs, but is much more complex if this property is absent. The Java memory model, for instance, provides attractive *sequential consistency* guarantees for data-race-free programs, but brings significant complexity in trying to bound the behaviour of racy programs [MPA05].

Part of the problem is in trying to draw a balance between programming simplicity, the ability to reason about program behaviour, and the need to give hardware and compilers sufficient latitude to perform traditional optimizations. Take the *sequential consistency* (SC) memory model as an example. It is one of the strictest memory models and forces all the instructions to be executed in a global order as they appeared in the actual program. It is quite simple for the programmers to program with this memory model because they can easily understand the program's behavior. However, SC also eliminates chances for compiler optimizations which often involve reordering instructions. This is an example of a memory model that is extremely easy to program with but imposes significant limits on optimizations. Ideally, a memory model should be easy to program with, while also allowing as much optimization as possible. The recently revised Java Memory Model provides a weaker consistency model that attempts to do just that. This model gives guarantees of sequential consistency for *correct* programs, ones without data-races, and a more complicated and weaker semantics for programs that do have data-races. Subtleties in the latter effort are in fact a continuing source of concern, and several researchers have pointed out that many very common optimizations, including simple reordering of statements, are technically prohibited in the current memory model for Java [Au07, ŠevčíkA08, CKS07]. A very important reason for this situation is that many optimizations making assumption about control flow cannot be applied to program blocks involving shared variables. The main difficulty comes from the fact that the shared variables are not clearly identified in current Java and therefore, a highly-conservative assumption needs to be applied to compiler optimization, in the worst case assuming that any code blocks may potentially contain shared variables. Because so many optimizations are thereby prohibited, researchers are trying to find a more reasonable boundary between the forbidden and the allowed optimizations and there has been no definite solution to this issue yet.

Abstractly, such complexities are reduced for DRF, or "correctly synchronized" programs, since they provide a well-defined input model [AH90]. Optimizing compilers can use synchronization points as the boundaries of optimizations and thus generate only sequential consistent executions. Therefore, a way to bypass the difficulties in validating optimization for racy programs is to somehow detect that a program is race-free and ensure that this property is preserved throughout optimization. Unfortunately, this is a very complex problem. Static approaches have been developed but suffer from accuracy concerns since the DRF property is essentially a runtime program behavior. A program as simple as the one shown in Figure 2.5 can cause challenges for static analysis. Dynamic approaches have also been defined, and can accurately report races happening during execution but add considerable runtime overhead and of course do not give static guarantees of race-freedom. We discuss this further in *Chapter 6*.

```
Thread 1                    Thread 2
while( condition1 )         y = x;
x = 1;
```

Figure 2.5 Example of a program that may be correctly synchronized depending on the runtime behavior. If `condition1` is always false at runtime, thread 1 will not write to x and therefore, will not conflict the read of x in thread 2. In this case the program is race-free. It may, however, not be possible to statically determine that `condition1` is always false.

2.2.2 A Race-Free Execution Model

In contrast to all the efforts that either try to prove the permissible optimizations in order to comply with the complex memory model or which aim at detecting race-conditions in an existing program, we would like to guarantee a data-race-free program from scratch. The ideal model uses a conceptually trivial mechanism to make it impossible for a program to contain data-races and thus gives an effective method for an optimizing compiler to clearly identify shared and thread-local data. There are two basic principles behind our changes to the Java language as listed below:

- All the static and heap variables are thread-local by default. Thus every thread has its own copy of these variables. Two threads accessing the “same” static fields through the same syntax are in fact modifying their own private copies without conflicts.

- Shared data is still supported. All shared fields are explicitly tagged with the keyword *volatile*, and accesses of different threads reach the same memory location. Such data make use of the current Java volatile syntax.

There are several good reasons for reusing the idea of volatile data. In the current Java semantics, accesses to volatile data are guaranteed to be transparent to all threads. This means if one thread modifies a volatile data, all threads can observe the change in the value right away. This is because a compiler must insert a memory barrier for each access to a volatile variable and this barrier synchronizes all cached copies of variables with main memory. As a result, volatile data will never cause race conditions. Furthermore, use of volatile data also has ordering implications. The order of accessing two volatile data must be the same for all the threads. This imposes synchronization boundaries for compiler optimization.

This execution model provides very useful guarantee. It limits the sharing variables between threads to volatile variables which are statically and clearly identified. This greatly simplifies programming effort, and also optimization design, which no longer requires advanced analysis or strong conservative assumptions to ensure data-race-freedom is preserved throughout optimization.

The starting point of implementing this execution model is to support default thread-local data in Java and make sure the more intensive usage of thread-local data will not compromise performance too much. This leads to the next chapter where we consider how the current `ThreadLocal` class works and present the basic idea for our approach to optimizing it.

Chapter 3
Semantics and Implementation Strategy

Basic and existing support for thread-local storage is provided in Java through a class called `ThreadLocal`. This class requires no special implementation, using a thread-specific map to associate `ThreadLocal` objects with specific values. In this way each thread can store and access unique data (logically) within the same `ThreadLocal` object by using the `ThreadLocal` object as a map key. Although this design sits nicely within the object-oriented paradigm of Java and avoids any special-casing, performance can be unsatisfactory. In this chapter, we will look at both the original design using `ThreadLocal` objects and our new design resulting in new semantics. *Section 3.1* gives a detailed exposition of the original design and our concerns with it. *Section 3.2* gives an overview of how thread-local data is used in the race-free Java variant we target.

3.1 Original Design

In a typical Java Virtual Machine all object data is allocated on a heap, a single area of shared memory. Naturally, and by design, all the variables are potentially accessible to all the threads. This includes instance data, provided a thread is able to reach the data through some chain of references, and also *static* class fields, which can be accessed without holding any reference to an object of the class. Either mechanism allows data created in one thread to reach another, "escaping" a single thread's context [WR99]. To store private, thread-specific data the program can allocate `ThreadLocal` objects, which as discussed above interact with a map-based mechanism in order to simultaneously "hold" distinct data for different threads. A single `ThreadLocal` object can be shared among threads, but when a thread reads or writes to it the thread accesses only its local version of the data stored therein. The map-based approach within `ThreadLocal` variables is quite flexible, allowing for thread-specific version of data for all threads, and cooperating with garbage collection in the sense that a thread's private data only becomes collectable once the owning thread dies.

The map-based approach requires that each thread is able to translate `ThreadLocal` objects into specific values. Each Java thread has a `threadLocals` field which has type `ThreadLocal.ThreadLocalMap`. This is a *weak reference* map associating `ThreadLocal` objects with a thread's values considered to be stored within. Basic read and write access to `ThreadLocal` data is provided by 2 (generic) methods, `<T> get()` and `void set(<T> object)`. Whenever a thread-local variable is accessed through one of these methods, the `threadLocals` field is read to get the map, and the `ThreadLocal` object is then used as a key to retrieve the appropriate value. An example of using thread-local data in Java is given in Program 1 below.

Program 1 A program using `ThreadLocal` class in Java

```
class A {
     public static ThreadLocal<B> localItem = new ThreadLocal(){
         protected synchronized Object initialValue() {
              return new B();  /* B’s constructor is called */
         }
     };
}

class C extends Thread(){
     public void run(){
         B localValue = A.localItem().get(); /* read from a
                                              ThreadLocal subclass */
         B newValue = new B();
         A.localItem.set( newValue ); /* write to a ThreadLocal
                                        subclass */
     }
}
```

Although this design is quite portable, involving only Java code, there are several drawbacks to using the `ThreadLocal` class for actual thread-local variables:

- The thread-local variables are not actually inherent to the Java language. As a result, the syntax to use them is complicated, and quite different from normal variable access. Reading and writing require method calls, and to set a default, initial value a programmer needs to override the <T> initialValue() method.

- The mechanism makes every access to thread-local data expensive. Each time a read or write is requested, the `threadLocals` map must be retrieved from the `Thread` object, and the current `ThreadLocal` object used as key to get the related value. This requires multiple levels of indirection.

- Initialization of thread local variables for child threads can only be done manually. A programmer may want a parent thread to define an initial value automatically when launching children threads, a common pattern, given that children threads often need an initial value to complete further tasks. To tackle this problem, a separate mechanism, using a subclass of `ThreadLocal` called `InheritableThreadLocal` is then used. This class allows for a method, `Object childValue(Object parentValue)` to be implemented to receive an initial value from parent thread. Program 2 shows the difference between the use of `ThreadLocal` class and `InheritableThreadLocal` class. In the `LocalExample` class, the `id` field is of `InheritableThreadLocal` class and the `color` field is of `ThreadLocal` class. The child thread inherits the value of the `id` field from the parent thread while the `color` field is not initialized.

  The main concern here is that while this does indeed enable children to receive parent data, it does not provide any default mechanism to ensure the child data is thread-specific to the child thread—-by default, parent threads pass an object reference, and are not required to (deep-)clone the object. This conflicts with the spirit of thread-local variables, which is to guarantee each thread has its isolated thread-specific datum within a given *ThreadLocal* object. The default mechanism allows both parent and child to have access to the same object, and is thus an easy source of potential race-conditions.

Program 2 A program using `InheritableThreadLocal` and `ThreadLocal` class.

```
class LocalExample extends Thread {
   private static InheritableThreadLocal id =
      new InheritableThreadLocal();
   private static ThreadLocal color = new ThreadLocal();
   static int count = 0;

   public void run() {
      if ( count == 0 ) {
         count++;
         id.set( new Integer( count ));
         color.set( "Green" );
         System.out.println( "Parent " + id.get() + " "
                                                   + color.get() );
         LocalExample child = new LocalExample();
         child.start();
      }
      else if ( count > 0 ) {
         System.out.println( "Child " + id.get() + " "
                                                   + color.get() );
      }
   }
}

public class Test  {
   public static void main(String args[]) {
      LocalExample a = new LocalExample(  );
      a.start();
   }
}

output:
Parent 1 Green
Child 1 null
```

3.2 Our Design

Our design is intended to optimize support for thread-local data within a new, race-free Java dialect. Here we summarize details important to understanding the nature of our thread-local implementation. Further details of the language and justification for it being race-free, are available in [VKZ11].

The principle idea behind our approach is to make thread-local the default option. A programmer wanting different threads to use the same object then needs to explicitly declare the data as shared, for which we repurpose the *volatile* keyword. Non-volatile data, be it static class variables or general instance variables are then guaranteed to be thread-local. For instance variables this is afforded by making distinct copies of any accessible data at thread initialization time. Access to thread-local, static class variables is optimized through a table look-up based mechanism.

This approach overcomes the concerns expressed above with respect to the current use of `ThreadLocal` objects in Java described above. That is:

- Thread-local data becomes inherent within the language semantics. All the variables are thread-local automatically, and both thread-local and shared data uses the same access syntax.
- Thread-local instance data is guaranteed to be thread-local by reachability properties within the heap, and so requires no extra access overhead. Static thread-local data is stored within a thread-specific look-up table, such that the same data is located at the same offset for each thread. Access thus only requires an additional indirection to reach the thread table, with offset values known at (JIT) compile-time. This saves a lot of time compared with searching a map.
- Children threads can automatically inherit object values from parent thread when started. Thread-local properties are easily guaranteed in a real sense by (deep-)copying the data, ensuring parent and child objects are indeed distinct.

Program 3 shows an example of programming in this paradigm. The static field `localItem` in class `A` is not specified by *volatile* and so a thread of class `C` is operating on a local copy of that field. In contrast, the thread is operating on a shared copy for field sharedItem because there is a *volatile* modifier. The syntax for accessing the two kinds of fields are the same and the programmer just needs to use the modifier to tell the compiler the difference.

Program 3 A program using thread-local static variables in a race-free design.

```
class A {
     public static B localItem;
     public static volatile D sharedItem;
}

class C extends Thread(){
     public void run(){
         B localValue = A.localItem;  /* read from thread local
                                         static variable */
         A.localItem = new B();       /* write to a thread local
                                         static variable */

         D sharedValue = A.sharedItem;   /* read from shared
                                             static variable */
         A.sharedItem = new D();        /* write to shared
                                             static variable */
     }
}
```

Chapter 4
Optimized Implementation

Our implementation work is performed in *Jikes* RVM, allowing us to take advantage of the full Java Virtual Machine context. In this chapter, We begin with a general description of how to ensure thread-local properties in *Section 4.1*. We argue for the validity of our theory by dividing the objects into 4 categories and discussing the thread-local property category by category. Next, we provide the implementations details for instance fields and static fields respectively in *Section 4.2* and *Section 4.3*. In *Section 4.2*, we exploit reachability for instance fields by tracing class hierarchy of thread objects and employing an *object cloner* for deep-copying. This is followed by a discussion in *Section 4.3* about static fields for which we not only use an *object cloner* for reachability but also maintain a separate, thread-specific *value heap* to speed up accesses.

4.1 Thread-Local Access

Our purpose is to make each variable thread-local by default unless the programmer uses the *volatile* modifier to explicitly specify that a variable is shared. Whenever a thread starts, the virtual machine thus goes through every reference field reachable by the instantiated thread to deep-copy all objects whose references are not specified as shared. Note that not only instance variables are copied but also class variables are copied and stored into thread-specific memory locations, as described in the *Section 4.3*. In order to do this fully and effectively, we need to identify all sources of reachable objects.

The process of finding reachable objects is not unlike the tracing process of mark-and-sweep garbage collection. We begin from a known root set, transitively following references. Fig. 4.1 shows this process for one child thread launched by its parent thread. The arrows show the relationship between objects. Object A and Object C represent objects referenced from static fields accessed by the parent thread. Object B and Object D stand for objects contained within the instance fields of the parent thread that are reachable through parameters passed to the child thread constructor, and stored within the instance fields of the child thread object.

In the basic Java implementation, after a thread starts we can define 4 categories of reachable objects, corresponding to the tracing roots. *Non-escaping local* and *Escaping local* contain objects created by the current thread. *Inherited* and *Published* contain objects created by other threads but the current thread can reach them through fields accesses. We will argue how our implementation can make sure that each of these categories are thread-local unless otherwise specified by the programmer. Examples of these categories are given in Program 4. Note that the categories are in view of a single current thread. They do not overlap in one thread but one object may belong to different categories with respect to different threads.

Non-escaping local These are local objects instantiated in the `public void run()` method which are neither passed to another thread nor published through static class fields. Those objects *never escape* from the belonged thread. The *object a* in Program 4 belongs to this category.

Escaping local These are local objects instantiated in the `public void run()` method which are either passed to another thread or published through static class fields. The *object b* in Program 4 belongs to this category from the view of the `Parent` thread. It *escapes* from the `Parent` thread to a `Child` thread.

Inherited These are objects whose references are passed as arguments of the thread constructor or instance fields of the `Thread` and/or `Runnable` object. The *object b* in Program 4 belongs to this category from the view of a `Child` thread. It is stored in an instance field of a `Child` thread. Therefore, this object is shared between a `Parent`

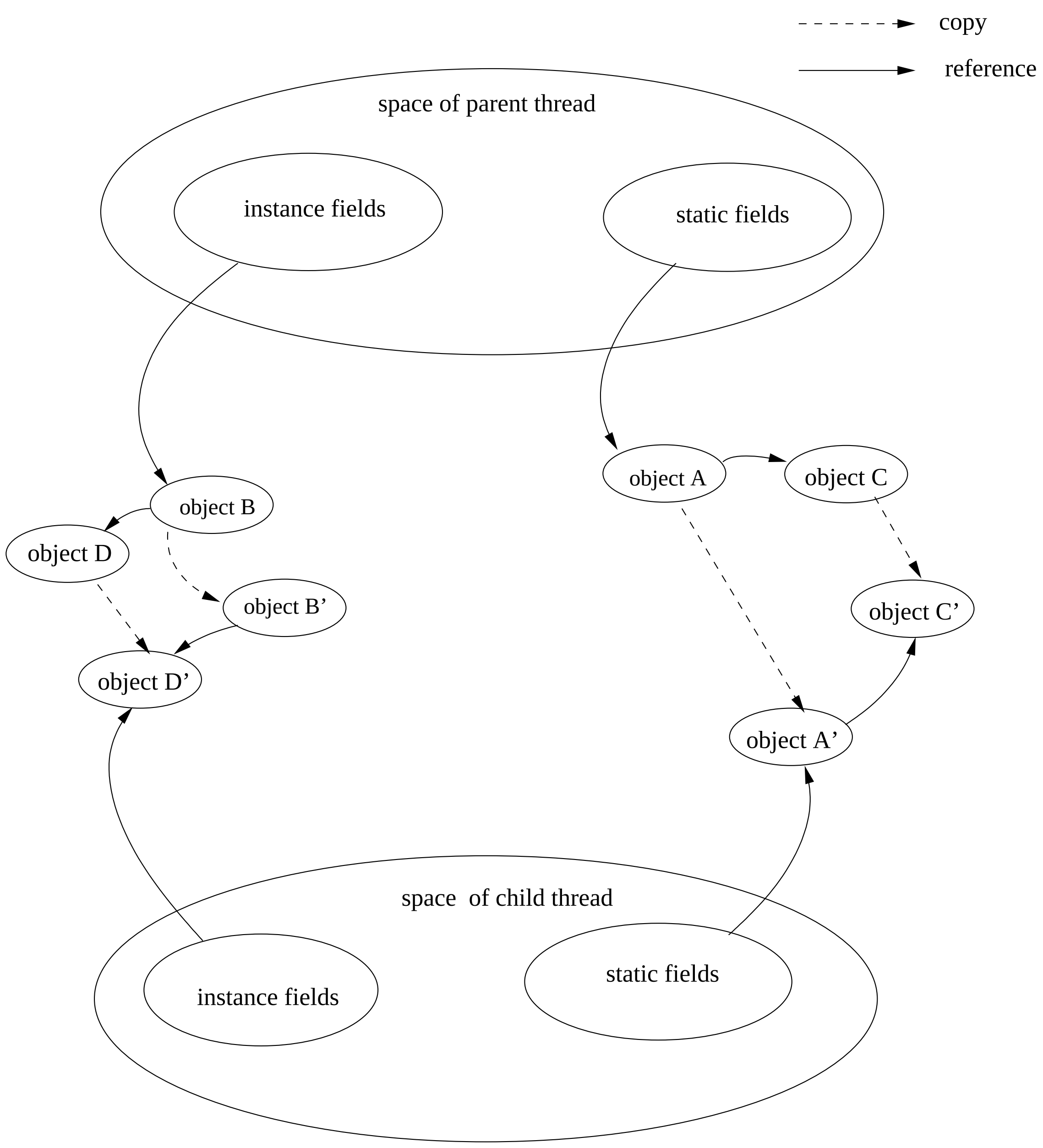


Figure 4.1 Deep-copying for thread initialization

thread and a `Child` thread. In another word, the `Child` thread *inherits* those objects from the `Parent` thread.

Published These are objects published through static fields of any class. The *object y* in

Program 4 in both `Parent` and `Child` thread belongs to this category. Both `Parent` and `Child` thread can access these objects. Actually, any other threads, if they exist, can also reach these objects.

For the first category, the objects are instantiated locally and only the current thread holds references to them. These objects can never escape the thread even in the base version of Java. Objects in *Escaping local* escape the thread in which they are created either by being stored within a thread instance variable or by being stored in a static field. The former become *Inherited* objects while the latter become *Published* objects. Objects in *Inherited* are shared between parent thread and child thread. Objects in *Published* are shared among all the threads. The main requirement is thus to create unique versions of *Inherited* and *Published* objects (and their reachable referents) for the current thread. During the initialization phase at the start of a thread, our modified virtual machine iterates through all the reference fields and checks for the *volatile* modifier. If not found, the object is deep-copied to ensure there are thread-specific versions.

4.2 Application Instance Fields

A limitation of our approach is that we only consider application fields defined by developers here. The fields defined in classes of the original Java API are not copied because we assume the classes in Java API will not cause race conditions. Furthermore, objects of most Java API classes do not refer to objects of the developer defined classes or access any fields of the latter classes because they are not aware of the existence of the latter classes. Here, we only consider 2 exceptions among the most commonly used Java API classes whose fields point to developer defined classes. They are `Thread` *Section 4.2.1* and containers *Section 4.2.2*. A safer method is to check all the used API classes to see if they have fields referring to developer defined classes, but this is left for future work.

For *Inherited* objects, we need to iterate through all the application instance fields of the current thread to copy the objects that are reachable. We trace the class hierarchy for `Thread` objects and its `runnable` field, using *object cloner* to copy both *scalar* type objects and array type objects.

Program 4 Four categories

```
class X {

      //static field that can be accessed by any threads
      public static Y y;
}

class Parent extends Thread(){
     public void run(){

            //local object that never escapes parent thread
            A a = new A();

            //object which is passed to children thread
            B b = new B();
            Child child = new Child(b);
            child.start();

            //accessing static field
            Y y = new Y();
            X.y = y;
     }
}

class Child extends Thread(){
     B b;
     Child( B b ){
            super();
            this.b = b;
     }

     public void run(){

            //accessing static field
            Y y = new Y();
            X.y = y;
     }
}
```

4.2.1 Class Hierarchy for Thread and Runnable

The application fields are defined in different levels of subclasses of the `Thread` class. Therefore, to find out all the objects reachable from instance fields we need to trace through the class hierarchy of the current `Thread` object. All subclasses that the current thread extends from are extracted and the objects referenced by those fields are checked and copied. Furthermore, there is a `runnable` field of type `Runnable` defined in the `Thread` class. A `Thread` object may reach a `Runnable` object with a lot of application fields. Similar to the application fields in `Thread` object, these fields are dispersed among different level of subclasses of `Runnable`. These instance fields may hold objects shared with other threads as well. Therefore, we need to use the same method to trace the class hierarchy of the `Runnable` object in order to find all the objects reachable from this point.

4.2.2 Object Cloner

Each thread is assigned an *object cloner* at the start. The object cloner is in charge of cloning every object reachable from its owner. For each object, the object cloner first does a shallow copy which only duplicates the primitive type variables and reference values. Then, a recursive approach is employed to guarantee the objects are fully copied; this is especially important for collection and array type objects. A map is used to prevent an object from being copied twice, and to properly reconstruct copied object references.

Scalar Type Object

A scalar type object is a single object that is not an array. Every object in *Jikes* RVM consists of 2 parts, the header and the fields value. The header part gives type and garbage collection information. Fig 4.2 shows the structure for a scalar type object. To copy a single object, the object cloner first allocates a space and initializes the header for the new object. Next, it does a memory copy from the second part of the original object to the new object. After this shallow copying process, the object cloner will iterate through each field of the current scalar type object. If the field is primitive type, we need to do nothing. If the field is an instance field, the `copyObject( Object fromObj, RVMType type )` method will be called recursively.

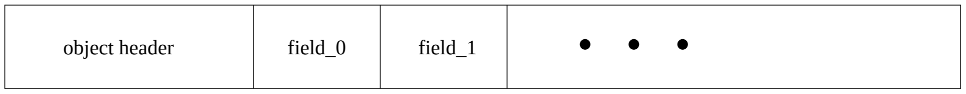


Figure 4.2 Structure of scalar type object

The Java API contains a large group of container classes which are used to hold multiple objects. Although containers are also scalar type objects, we need a special way of copying them for 2 reasons. First of all, containers are defined in Java API but have references to objects of the developer defined classes. We assume only the field referring to developer defined classes will cause race conditions. Therefore, we shallow copy rather than deep-copy other fields which are mostly primitive type data used to indicate information like size and capacity. Note that while it is possible that other fields in containers may need to be deep-copied (such as head-pointer for a linked list), we do not address that issue here and leave it for future work. Furthermore, in each container the field referring to objects of developer defined classes is an array type field. This array is used to buffer all the elements in a container. The object cloner extracts the array and use the copying scheme for array type objects, which we will describe below.

Array Type Object

Fig 4.3 shows the structure for an array type object. Similar to a scalar type object, the object cloner allocates the space and initializes the header for the new array type object. Instead of deep-copying each field as scalar type object, however, the object cloner deep-copies each element object for array type object.

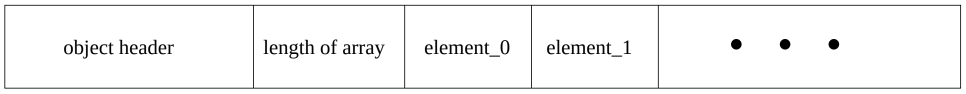


Figure 4.3 Structure of array type object

Map

There are several cases where a naive deep-copying scheme will cause problems. Program 5 will get stuck in an infinite loop because 2 objects reachable by instance fields of the thread are pointing to each other, as is shown in Figure 4.4. A naive deep-copying scheme will continue to deep-copy each field and result in an infinite number of copies of `C` objects and `B` objects. In Program 6, an instance field and a static field are referring to the same object as is shown in Figure 4.5. A basic deep-copying method will copy this object twice. The first time is when deep-copying the instance field `b` and the second time is when deep-copying the static field `C.b`. Because of those cases, we need to take an approach to make the object cloner know whether an object referred has already got a local copy for the current thread.

Program 5 Objects referencing to each other

```
class A extends Thread{
      B b;
      C c;
      run(){
        this.b = new B();
        this.c = new C();
        this.c.b = this.b;
        this.b.c = this.c;
      }
}

class C{
      B b;
}

class B{
      C c;
}
```

Similar to the case where algorithms traversing each node in a graph needs to record visited node, the deep-copying method need to find a way of keeping track of the copied object. The object cloner realizes this by recording the address of the new copy. A map

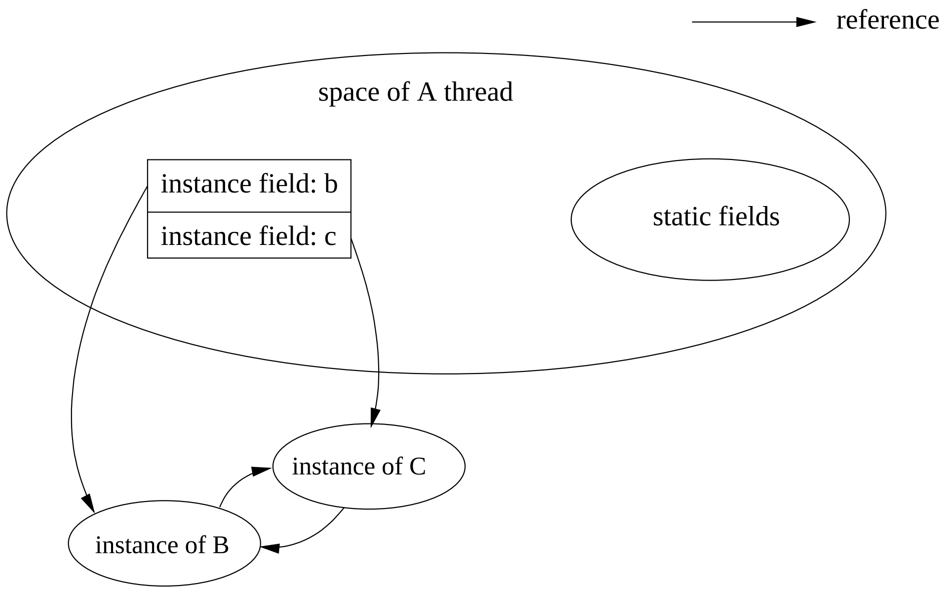


Figure 4.4 Objects referring to each other

Program 6 An instance field object referencing to static field object

```
class A extends Thread{
      B b;
      run(){
        this.b = C.b;
      }
}

class C{
      static B b = new B();
}
```

is used to keep <old address, new address> pairs. Before copying any object, the object cloner first searches the map with the old address as a key. If the object is already copied for the current thread, a new address value associated the copied object is returned. If there no entry for the old address in the map, the object cloner will continue to copy this object. The garbage collection is disabled during this process to make this approach work. Otherwise, the map will not correctly identify a copied object since it might get a new address by being moved elsewhere.

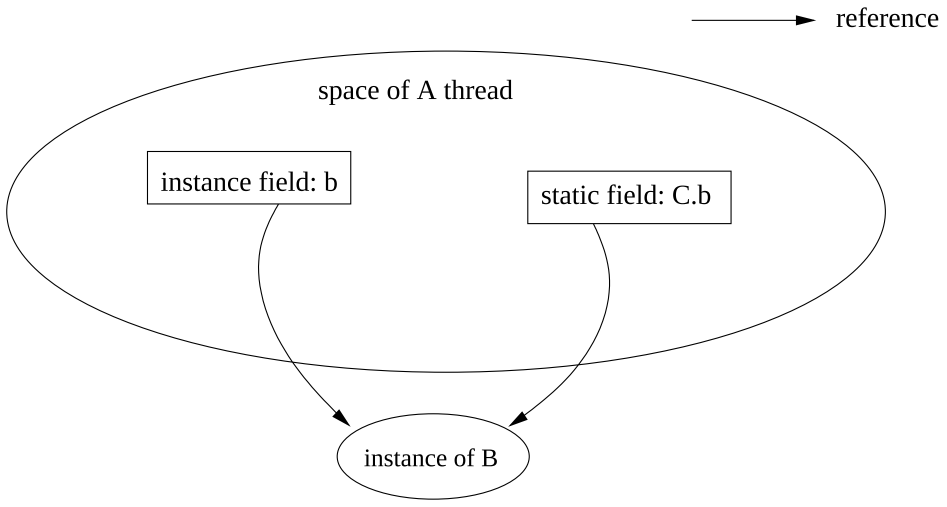


Figure 4.5 Fields referring to the same object

4.3 Localized Value Heap for Static Fields

For the last category, we leverage the existing mechanism of *Jikes* RVM for storing static fields. Within *Jikes* RVM, all static variables are held in a (global) table called JTOC (*Jikes* RVM Table of Content). We duplicate this table, attaching a local copy to the new thread, while also deep-copying all the references contained within. This process has the added advantage of resulting in much faster access than through an equivalent static `ThreadLocal` reference. Each static variable is already assigned an offset when loading and resolving the corresponding class. This offset indicates the relative positive with respect to the start of the table, and is unique for each static variable. Our implementation preserves this property for all the threads; for a thread to read a static non-volatile field of a class, it merely indexes its own version of the static variable table using the variable index already defined during class load-time. Note that this design does not entirely replace the original, global table of statics—there is still a global table holding the shared static variables that are specified by the programmer to be *volatile*. Figure 4.6 illustrates the idea in the case of Program 3. Thread 1 and thread 2 each have a local copy of static field `A.localItem` at the same offset in their table. If any of the threads would like to access `A.sharedItem` they look it up in the global table.

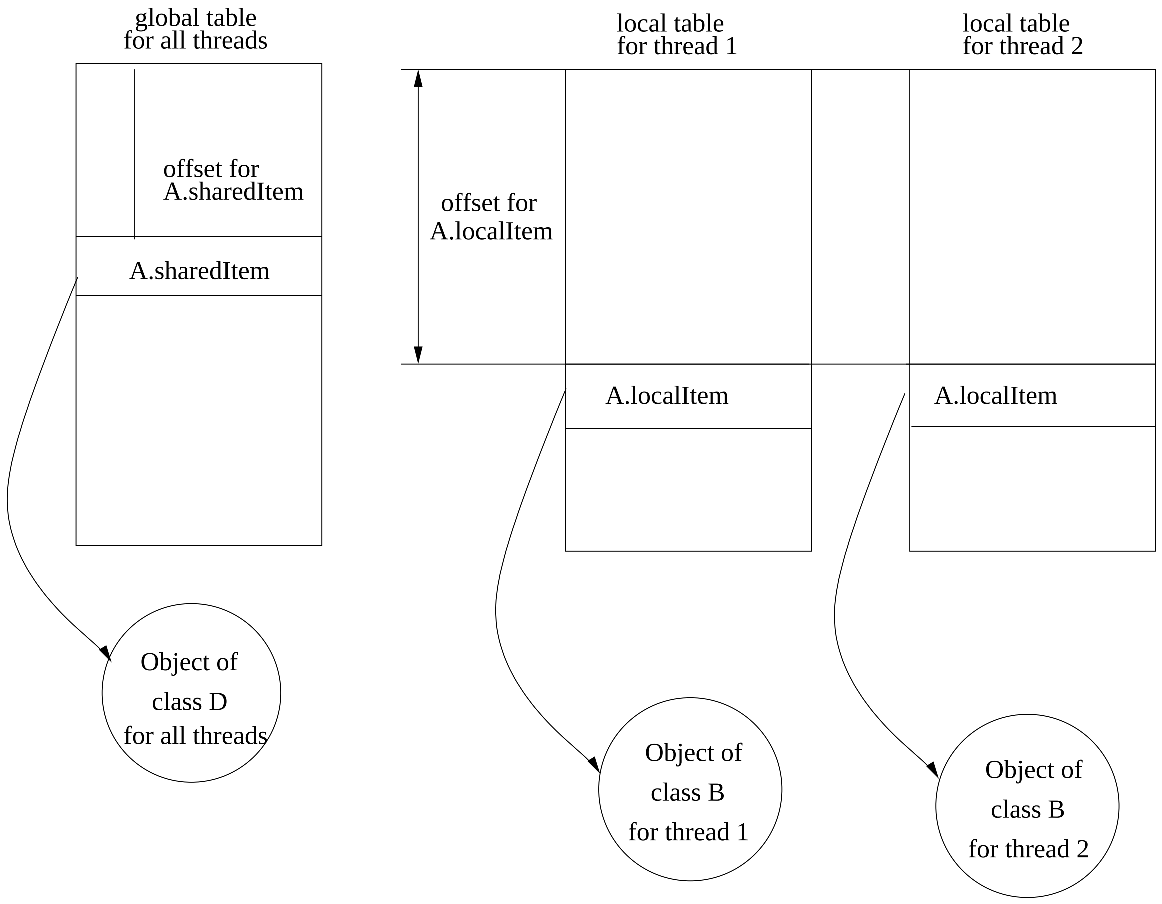


Figure 4.6 Local value heap(s) for static fields.

4.3.1 Localized Heap with JTOC

In *Jikes* RVM, the JTOC which is an array of integers, stores pointers to all static fields. Whenever a new class appears in the executed program, its corresponding class file is loaded. The VM will extract the class information concerning class name, fields, methods, supper classes and interfaces, storing them in *Jikes* RVM format like `RVMClass`, `RVMField`, `RVMMethod`. The class is resolved through this process and each field is assigned an unique offset stored in an associated `RVMField` object. The offset is measured from the start of the object if this field is an instance field. For static fields, the offset is measured with respect to the start of the JTOC.

In the original version of *Jikes* RVM, JTOC is a global table shared by all the threads and a dedicated machine register (the global JTOC register) holds the reference pointing to this table. We make a thread-local copy for JTOC at the start of each single thread. When a thread needs to read and write to a static field, instead of reading the global JTOC register, it reads the `slots` field which is added to the `RVMThread` class and adds the offset to this reference value to get the reference to its own version of the static thread-local fields.

4.3.2 Assembly Code Generation

To change the semantics of the language, we need to change the way the virtual machine behaves by modifying the machine code generated according to bytecode instructions. Two groups of bytecode instructions are involved in static field reads and writes. The first group involves operations for static fields in resolved classes whose class information is already stored in form of `RVMClass` and `RVMField` objects. The second group involves operations for static fields in unresolved classes whose class information is not available at this moment. For both groups, the first step is to tell if the related static field is a *volatile* field. This is a functionality that should be included in the JVM but has not yet been implemented in *Jikes* RVM. The second step is to branch to the right path with this information. The whole process needs a combination of static and runtime information.

Reads and Writes of Resolved Static Fields

For static fields in resolved classes, the virtual machine can get modifier information easily from the `RVMField` object and generate only one pass of the assembly code. This process is shown in Figure. 4.7. The generated assembly code instructions for reads and writes are shown in Figure. 4.8 and Figure. 4.9. After the first instruction, the EAX register[1] gets the base address of the thread-local table by reading the `slots` field of the `RVMThread`. This field is represented by the red rectangle (the one with slash lines) in Figure. 4.10. After the second assembly code instruction, the virtual machine gets the location of the slot associated with the static field by adding the field's offset to the base address stored in EAX. The field slots is shown as a green rectangle (the one with horizontal lines) in Fig. 4.10,

[1]a caller-saved register in IA32 systems used by compilers for storing temporary values. Caller-saved registers are registers whose value may be changed across a function call.

storing the address for the local object. For read operations, this address is pushed on the stack. For write operations, the value on top of the stack is popped and stored into this slot.

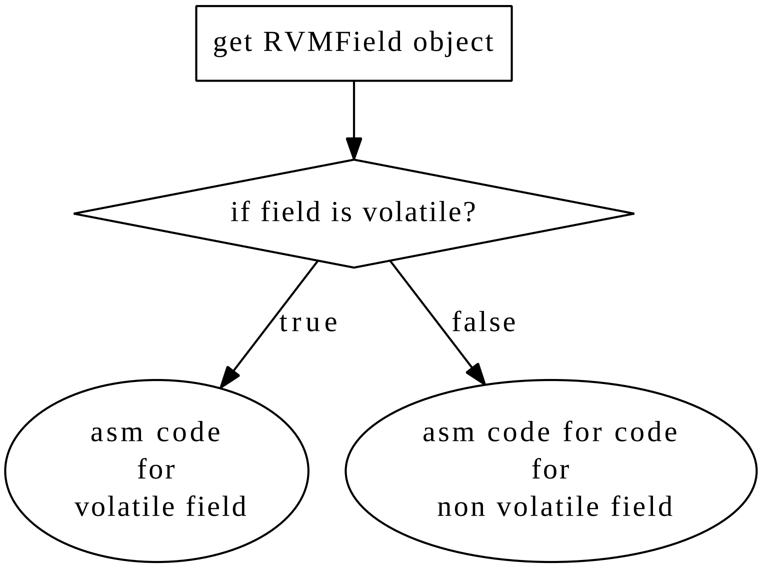
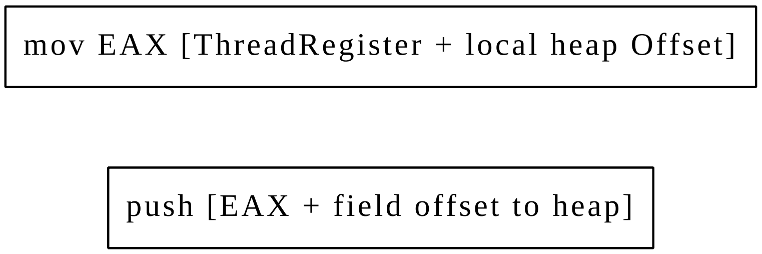

get RVMField object
if field is volatile?
true
false
asm code for volatile field
asm code for code for non volatile field

Figure 4.7 Virtual machine behavior for resolved get/putstatic bytecode



```
mov EAX [ThreadRegister + local heap Offset]

push [EAX + field offset to heap]
```

Figure 4.8 Assembly code for resolved getStatic bytecode

Reads and Writes of Unresolved Static Fields

For static fields in unresolved classes, there is no relevant information available in the virtual machine at compile time. Fortunately, the modifier information will be available at run-time when the assembly codes are executed. As is shown in Figure 4.11, the first thing the virtual machine does is to generate code for triggering the class loading and resolving process. We need to insert the assembly codes for retrieving field modifier information after

```
mov EAX [ThreadRegister + local heap Offset]
```

```
pop [EAX + field offset to heap]
```

Figure 4.9 Assembly code for resolved putStatic bytecode

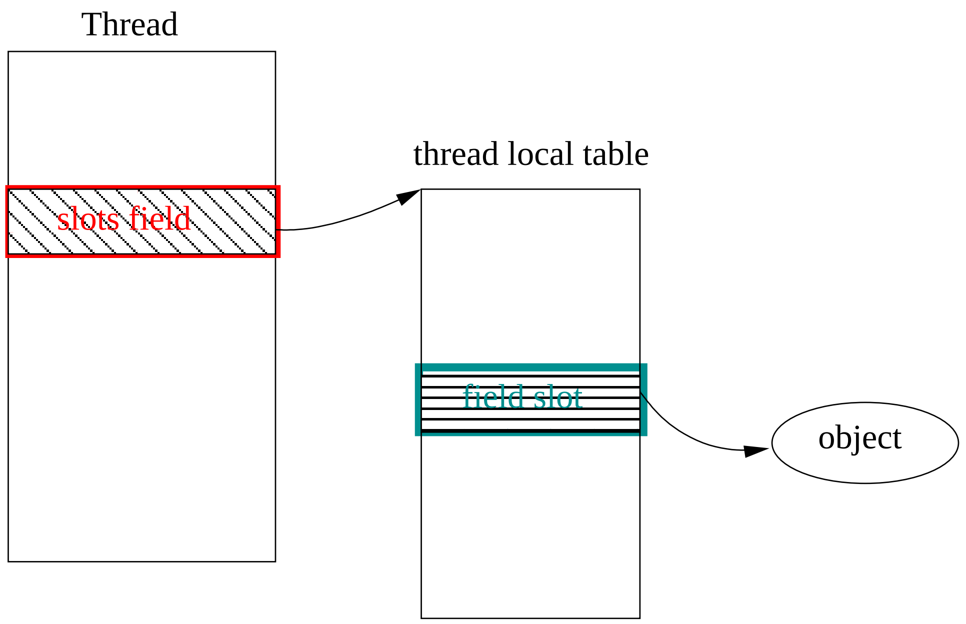


Figure 4.10 Assembly code illustration

this process. Because the modifier information is absent until run-time execution, assembly code for both branches should be generated.

Figure 4.12 shows this process. The logic is almost the same as that for resolved fields in Figure 4.7 but it is applied to generated assembly code rather than VM behavior at compile time. The way of extracting modifier information to judge whether a field is volatile is also different. Rather than getting this information directly from the `RVMField` object, we need to call a method `isVolatile` with a reference id as a parameter at run-time. Each field reference in *Jikes* RVM is in form a of `FieldReference` object associated with an unique id and a resolved field in form of `RVMField`. Therefore, inside the called method `int isVolatile(int memberId)`, we use the id to find the resolved field and check if this field is volatile or not.

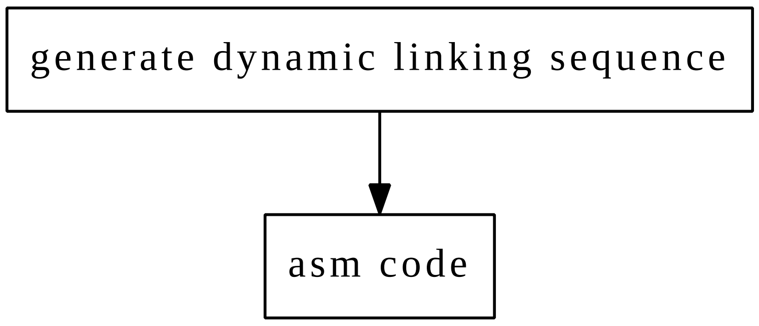


Figure 4.11 Virtual machine behavior for unresolved get/putstatic bytecode

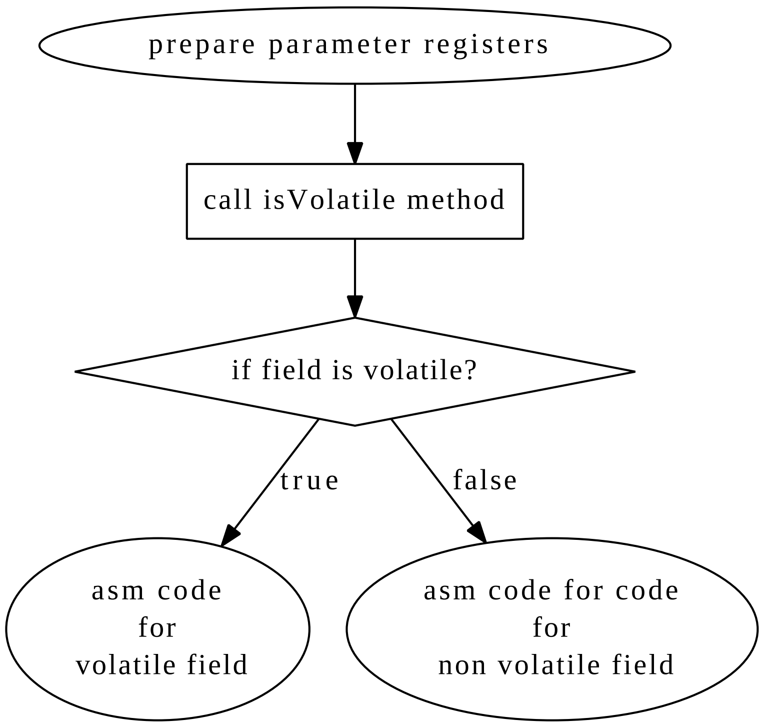

prepare parameter registers
call isVolatile method
if field is volatile?
true
false
asm code for volatile field
asm code for code for non volatile field

Figure 4.12 Assembly code for resolved getStatic bytecode

4.3.3 Garbage Collector

A basic semi-space copying garbage collector is chosen over other more sophisticated garbage collectors because its simple structure facilitates fast implementation. As a copying garbage collector, it needs to determine all the objects reachable and copy these objects to the other half of the space. The remaining objects in the first half of the space are discarded. The tracing phase starts from objects in root sets which are assumed to be reachable. One root set is the static root set which contains all the static fields. In the case of *Jikes* RVM, this takes the form of the JTOC which is duplicated to hold extra copies for each thread. The garbage collector is changed to be aware of these extra static root sets.

There are multiple garbage collector threads in *Jikes* RVM and it is necessary to fairly distribute among them the slots in the static roots of different application threads. This is more complicated than distributing slots from only one single static root. To begin with, the garbage collector needs to find out the quota, a heuristic approximation of tracing load which is the number of slots one garbage collector is assigned and it needs to be calculated like this:

$$quota = \frac{numStaticField * numOfThread}{numOfGCThread} \tag{4.1}$$

We use the number of static fields (numStaticField) times the number of application threads (numOfThread) to get the total number of slots. Next, we divide this total number of slots by the number of garbage collector threads (numOfGCThread) to get the quota. The garbage collector then finds the slot, uniquely identified by using both slot index (slotIndex) and thread index (threadIndex). It uses the following calculations:

$$preSlots = (GCOrdinal - 1) * quota \tag{4.2}$$

$$slotIndex = (preSlots \bmod numOfGC) + 1 \tag{4.3}$$

$$threadIndex = \frac{preSlots}{numOfGC} \tag{4.4}$$

Each garbage collector thread is assigned an ordinal number (GCOrdinal). Equation 4.2

gets the number of slots that have been processed by garbage collectors with ordinal numbers smaller than the current one. The result is used in 4.3 and 4.4 to identify the starting point. Then it continues to process the next slot following the flow shown in Figure 4.13. It checks if the slot contains a valid reference and traces the reachable objects from this reference. The garbage collector keeps going and stops at two cases. The first case, shown by the blue edge (the longer dotted edge) in Figure 4.13, is when it finishes its quota. The second case, shown by the green edge (the shorter dotted edge) in Figure 4.13, is when the garbage collector reaches the end of the entire root set which is the last slot of the last thread. During this process, each garbage collector thread keeps track of not only the current slot and but also the current static root being processed.

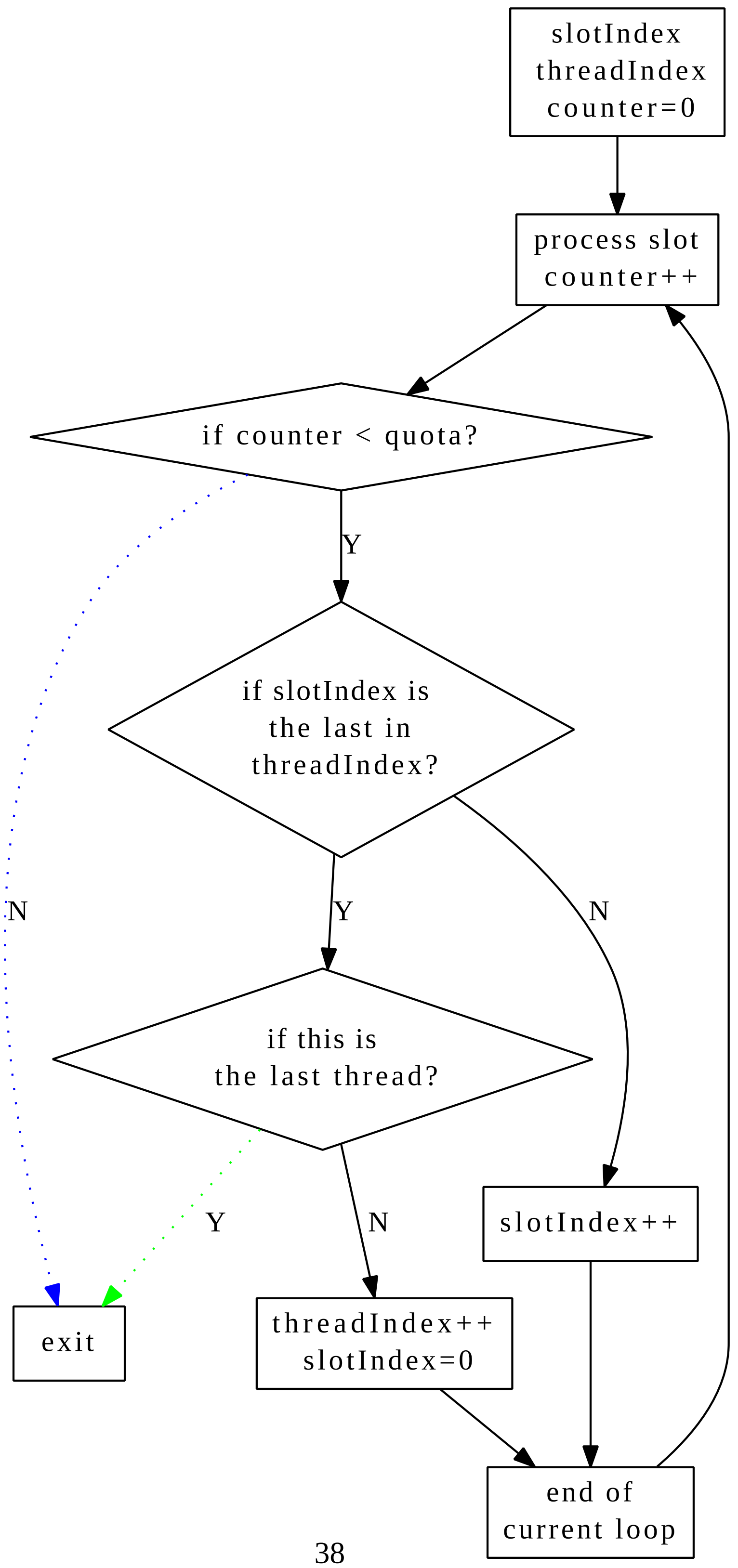


Figure 4.13 Assembly code for resolved getStatic bytecode

Chapter 5
Experiments

While our design reduces the normal overhead of `ThreadLocal` access, it also introduces significant overhead in the deep-copying process. In this chapter, we use both micro benchmarks and non-trivial benchmarks to experimentally verify and evaluate performance of our design. In *Section 5.1*, we use micro benchmarks to compare the performance of current `ThreadLocal` in Java API with static thread-local data in the new semantics. In *Section 5.1*, we use non-trivial benchmarks ported from existing sources to our semantics to show the programmability and functionality of the new semantic.

Programs written in the current Java language are run in *Jikes* RVM 3.1.1 and the modified versions are run in our modified virtual machine. There are two separate code generation paths: one in the baseline compiler and the other in the adaptive optimization compiler. In our prototype work we modified only the baseline compiler for simplicity, and so we turn off the adaptive optimization compilers on both virtual machines to be able to measure comparable results. For garbage collection we used the basic semi-space copying garbage collector in both JVMs, as the easiest to modify to understand the new root sets created by the duplicated static variable tables. Comparison with a fully optimized compiler and the advanced generational collector within *Jikes* RVM would be interesting, but is left for future work.

5.1 Micro Benchmarks

Micro test cases contain loops that repeatedly perform simple read and write operations on static, thread-local fields. The benchmarks written in the new dialect require no effort other than using the keyword *static*, while the benchmarks written in Java wrap variable data within a `ThreadLocal` type. Different groups of micro benchmarks emphasize the performance of different aspects of our design. In the Java Virtual Machine, a class needs to be *resolved* the first time it appears during the program execution. During a class resolution, the class file is loaded, static initializers are executed and symbolic references are replaced with direct references. After a class is resolved, information about this class is kept by the vitural machine. Therefore, retrieving information for a field in a unresolved class requires loading and resolving the class first while a field in a resolved class does not need extra effort. We need to consider the performance for accessing both these two kinds of fields. Micro tests in *Section 5.1.1* include only accesses to resolved static thread-local fields. Micro tests in *Section 5.1.2* focus on operations on unresolved static thread-local fields.

5.1.1 Accessing Resolved Thread-Local Fields

This group of test cases shows performance for resolved thread-local fields. All the current released versions of *Jikes* RVM do not distinguish volatile data from non-volatile data. This is a fast way of getting the entire JVM working but a full implementation of JVM should certainly be capable of telling the two different data groups. Our implementation of the new semantics adds this function to *Jikes* RVM and also brings in some overhead. As mentioned in *Section 4.3.2*, retrieving modifier information is very straightforward for resolved thread-local fields, adding little overhead in telling volatile data. Therefore, the overall performance would be dominated by the actual data accessing mechanism. For this reason, we use this group of benchmarks to fairly determine the baseline performance difference between our table-based approach and original, map-searching approach. We change the benchmarks with respect to two aspects, which are the number of fields (in *Section 5.1.1*) and the number of threads (in *Section 5.1.1*). The number of fields in each thread decides how many entries each map or table will have. It may potentially change the accessing time. The number of threads decide how many maps or tables the program will have. It may also affect data accessing time. Therefore, both of these two aspects are important.

Number of Fields Increasing versus Time and Memory

Figure 5.1 shows the case where there are 3 threads busy reading and writing all the static thread-local fields. We increase the number of static fields from 1 to 29. Unsurprisingly, the speed of our dialect is much better than that of the original Java using the current `ThreadLocal` class. We improve performance by a factor of two despite the fact that our system needs to determine whether a field is volatile or not while the original system does not do so. As the number of fields in each thread increases, the total execution time and the garbage collection time of benchmarks in both groups increases linearly. Note that the GC time increases at a slightly slower rate than the total time; our version adds only a small amount of extra data, mainly in terms of an extra table for each thread, and the amount of collectable data is minimal in both language variants.

The memory requirement remains at 10MB as the number of fields increases for the new semantics while the memory requirement remains at 6MB for original Java. For both the new semantics and original Java, more static fields does not require more table or map allocations and the memory space required is therefore unchanged.

Number of Threads versus Time and Memory

Table creation implies that for a given amount of static data, our overhead costs should increase in proportion to the number of threads. Figure 5.2 thus shows micro-tests reading and writing to 4 thread-local static fields, but varying the total number of threads created. As expected, the total execution time increases, although the result is not linear. Our optimized thread-local static variables easily outperforms the `ThreadLocal` class version when there are few threads, but as the number of threads increases there is also an increase in the slope of the curve, while the old version of Java maintains a linear relationship. The separate curve for GC time shows that this is primarily due to increased GC costs. The local heap for each thread is added to the static root set when collections begin, and thus

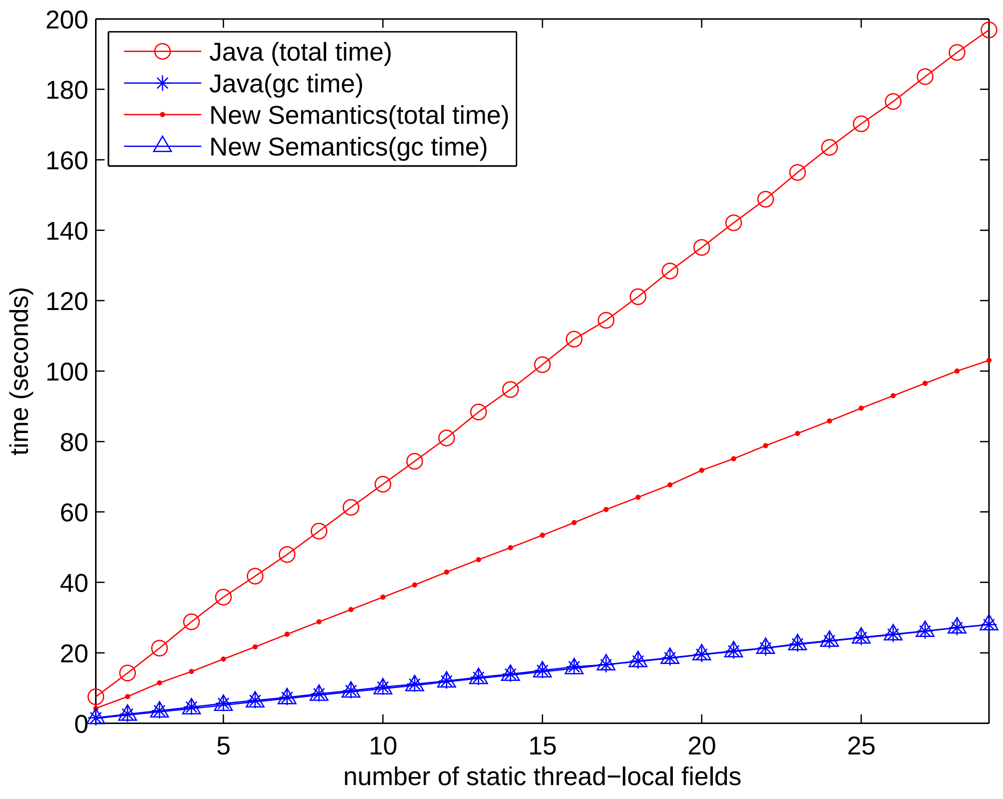


Figure 5.1 Time required for 6000000 reads and writes to each of 1 to 30 resolved static, thread-local fields. The time value is the average of 20 runs.

increases the time spent root scanning. This effect is verified in Figure 5.3, which shows execution time subtracting GC time. In this case growth is again reduced to linear. We expect improvements to the GC costs would be possible with a more sophisticated collector that understood and exploited the thread-specific separation our design creates.

Figure 5.4 shows the changes in required memory space as the number of threads increase. For both the original Java and the new semantics, the memory requirement increases in proportion to the number of threads. In both cases, there is a fixed amount of memory allocated to each thread as it starts. While the original Java associates a map to each thread the new semantics associates a table to each thread. The former approach is flexible in size and can be adjusted to the number of entries, while the latter has a fixed table size and has

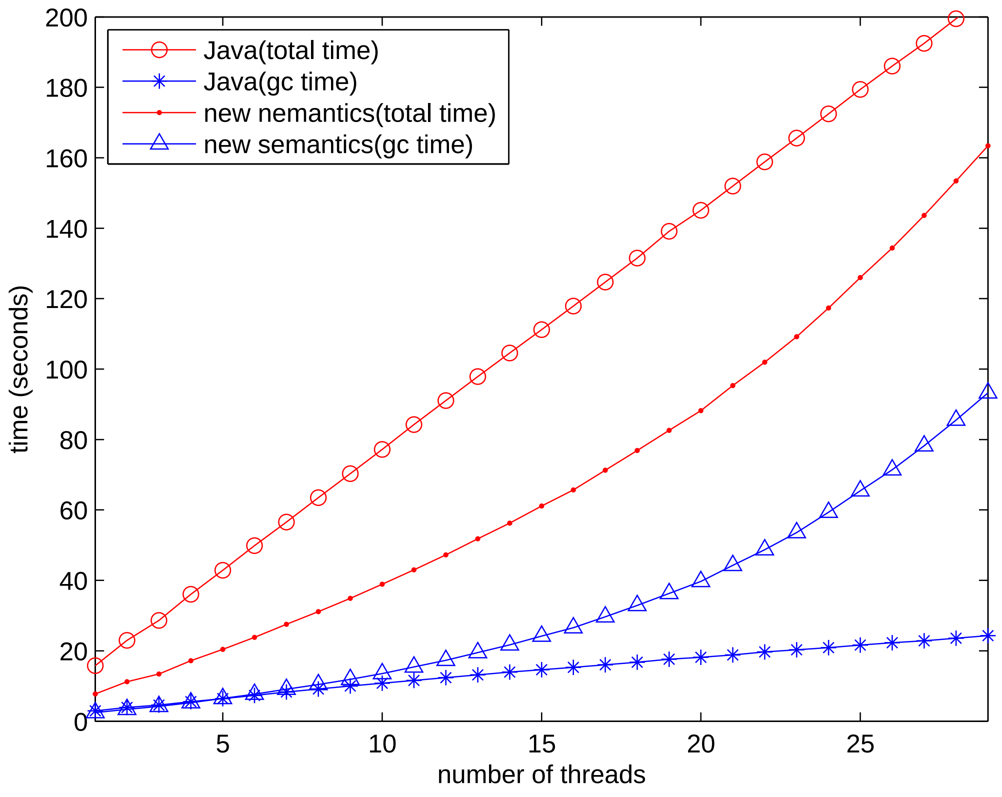


Figure 5.2 Number of threads versus time for 6000000 reads and writes to 4 resolved static, thread-local fields. The time value is the average of 20 runs.

a quite large initial size, accommodating 128K entries. This results in the different slope of the line. The memory requirement for new semantics can be decreased by using a table that can start small and be dynamically expanded.

5.1.2 Accessing Unresolved Thread-Local Fields

In this case, just to examine the specific cost of accessing unresolved fields we show one benchmark with three threads reading and writing to 10 unresolved static thread-local fields. As is mentioned in *Section 4.3.2*, retrieving modifier information for those fields takes more efforts and results in larger overhead. The extra overhead to distinguish volatile field from non volatile will compromise our performance improvements. Therefore, this

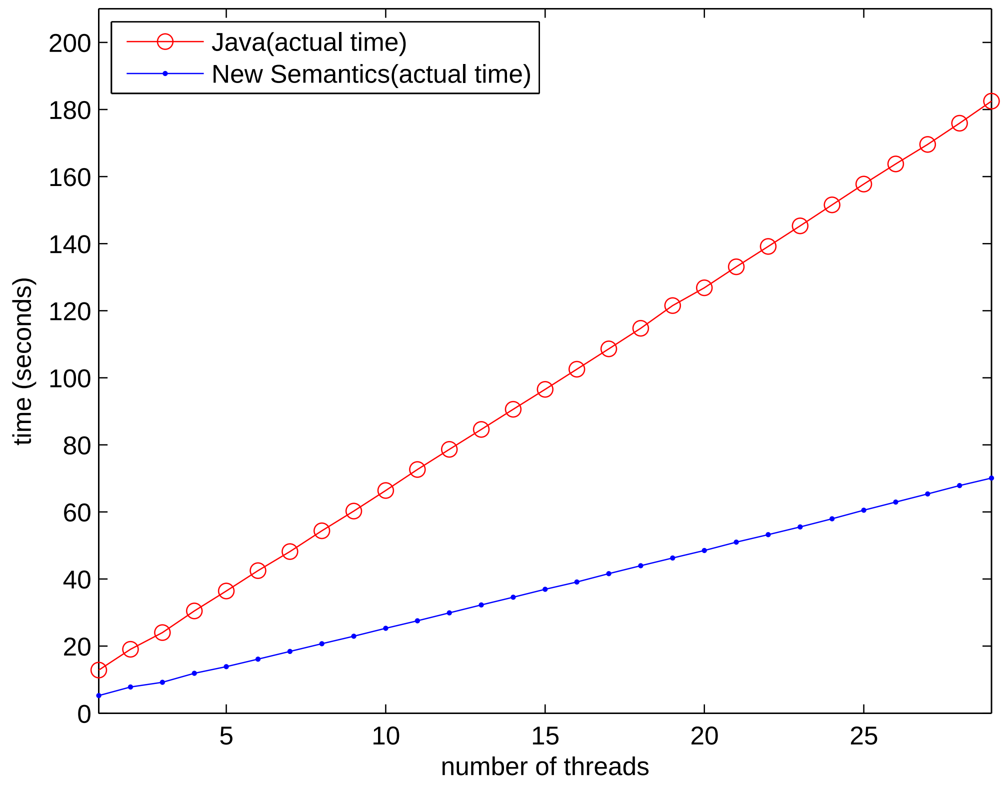


Figure 5.3 Number of threads versus time for 6000000 reads and writes to 4 thread-local static fields, deducting GC time. The time value is the average of 20 runs.

test is used to show part of the inevitable overhead introduced by implementing volatile data for a fully functional JVM. As is shown in Figure 5.5, both our new semantics and the original Java add some overhead compared to accessing resolved fields. This is because the virtual machine first needs to load and resolve the corresponding class to access unresolved data. The garbage collection time does not change for both cases. Another observation is that although the new semantics still outperforms the original Java the degradation of the former is significant larger than that of latter. This is caused by judging whether a static field is volatile or not. One way to reduce this overhead is to use a bit vector to indicate whether a static field is volatile or not rather than calling a function to read the modifier. Another way is to reduce *false unresolved access*, which means executing code for unre-

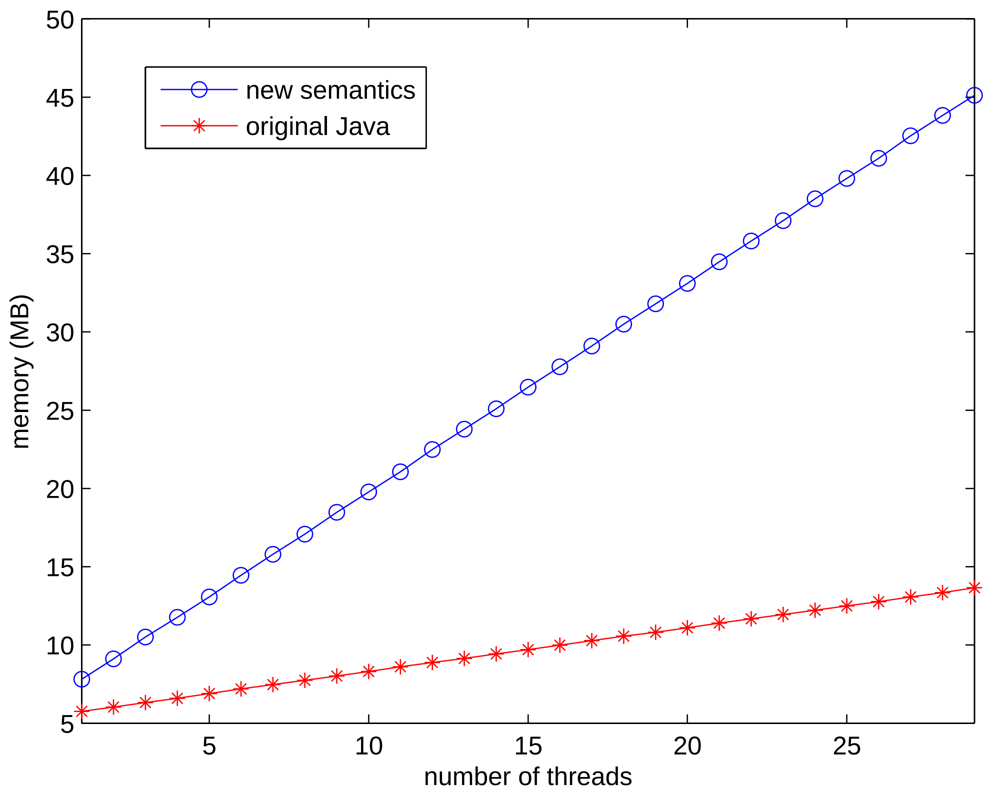


Figure 5.4 Number of threads versus memory for 6000000 reads and writes to 4 thread-local static fields.

solved fields despite actual accessing resolved fields. Our test case containing a huge loop is such case. The code generated for the loop is the same and the same code for accessing unresolved field will be executed for 6000000 times while actually only the first access is to an unresolved field. This can be improved by recompiling the code after the first access so that only the first access executes expensive code. This is a traditional method and although the baseline compiler of *Jikes* RVM does not do that, the adaptive optimization compiler in *Jikes* RVM can do such optimization.

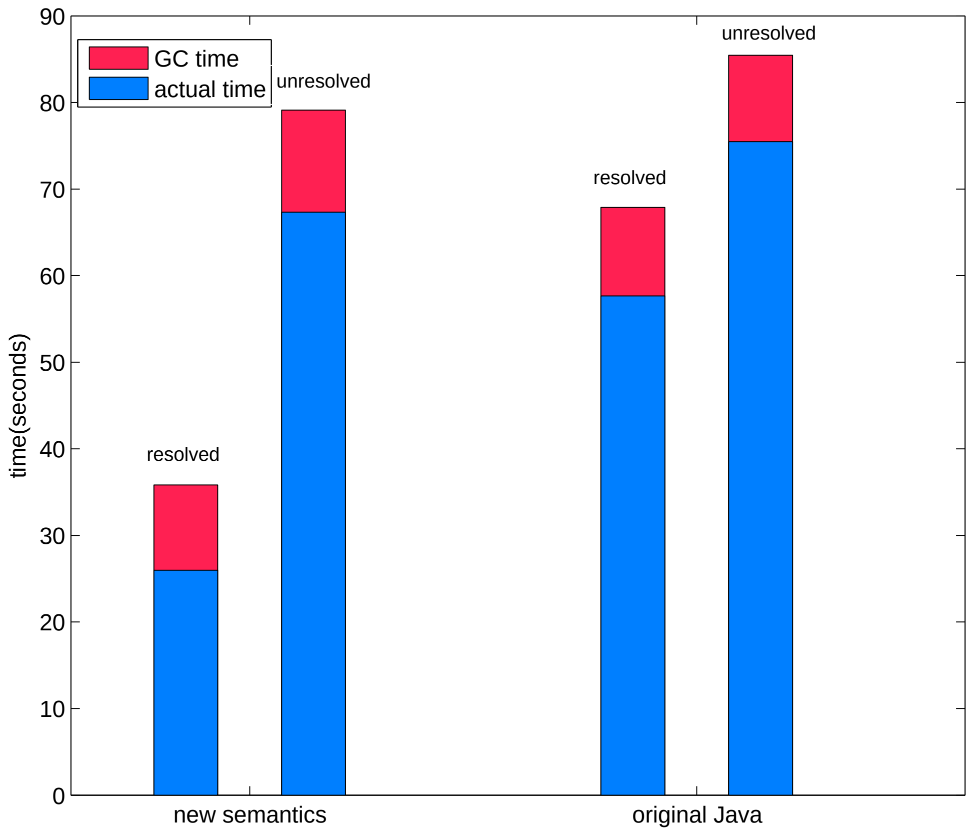


Figure 5.5 Comparison between time required for 6000000 reads and writes to both 10 unresolved and resolved static, thread-local fields. The time value is the average of 20 runs.

5.2 Non-trivial Benchmarks

Intensive thread-local access does not give a realistic picture of performance differences; different programs will in practice make greater or lesser use of shared versus thread-local data, and the relative impact will depend on how execution costs are distributed throughout a program's activities. We thus further investigate performance using a suite of larger, non-trivial test cases. These multithreaded benchmarks are selected from various sources. In all cases changes were required to make the programs comply with our new, thread-local/shared semantics. We manually identified the shared fields in different classes and tagged them with *volatile*. These changes respected the original program's use of shared and private data. Table 5.1 summarizes the benchmarks, giving each benchmark a short description. Table 5.2 includes dynamic data measuring the relative number of variable accesses (important to understanding how much impact changes to variable access methodology could possibly have), the total number of threads created, and a basic measurement of the overhead introduced by the per-thread table allocation we perform.

There are two groups of test cases for non-trivial benchmarks. *Section 5.2.1* compares the performance of the new semantics and the original benchmarks which do not make use of the `ThreadLocal` class. In *Section 5.2.2*, we modified the *series* benchmark to compare performance of the new semantics and use of the `ThreadLocal` class in a real case. The reason why we choose only the *series* benchmark is stated at the beginning of that section.

5.2.1 New Semantics versus Original Benchmarks

Figure 5.6 shows the performance in terms of execution time. Here we evaluate behavior of the modified benchmarks run under our semantics in relation to the execution time of the original, unmodified benchmark, which is using all shared data. We do not introduce new `ThreadLocal` objects to guarantee thread-local access for original Java benchmarks because this would require manually changing all the thread-local fields and also each access to those fields. To compare different benchmarks clearly, all the execution times are normalized to 1.0 by dividing the original execution time. Since garbage collection costs contribute a lot to the total time and our design has a clear impact on GC costs, we separate normal execution and GC time. In this experiment, we do not expect to see improvements, and are mainly trying to ensure that using thread-local by default does not introduce too much cost.

From this data, it can be seen that most benchmarks run on our modified virtual machine at about the same speed as on the original virtual machine. As expected, there is no performance improvements for this group of benchmarks as for micro benchmarks because the original Java benchmarks here do not use `ThreadLocal` objects to guarantee thread-local access as do the micro benchmarks. Nevertheless, we do anticipate performance improve-

Table 5.1 Benchmarks description and sources.

name	description	sources
P/C	A standard producer/consumer benchmark: 3 producers produce 12000 objects and 3 consumers consume them. They share the container holding the value and update the value.	Sun Java Tutorial [sun]
traffic	144 drivers and cars will pass through a rotary intersection. For each round, 1 car thread and 1 driver thread navigate together around a rotary. They share fields that give information about traffic condition, driver's decision and car's location as well as speed. They each have local copies of data indicating destinations, entrances, driver's driving style, road segments information.	Sable Research Group
roller	A variation on the classic "roller coaster" concurrency problem; 7 passenger threads compete for 7 seats in 1 roller coaster thread. They share fields that indicate ride number and whether certain seats are taken.	Sable Research Group
bank	8 threads transfer funds between a checking account and a savings account. 4 of them deposit and withdraw money from the checking account while the other 4 use the savings account.	Doug Lea [Lea99]
series	This benchmark computes the first N=100000 Fourier coefficients of the function $f(x) = (x+1)^x$. The computation of N coefficients are independent of each other and are distributed among 3 threads.	Java Grande Forum Benchmark Suite [gra]
mtrt	2 threads render a raytraced image. The threads share the canvas on which they need to cast the spot, but each thread does raytracing of its own section using independent calculations	SPECJVM98

Table 5.2 Benchmarks related data. The access columns shows static variable access density (average number of static variable accesses/millisecond), and the alloc time column gives the normalized table allocation time as a percentage of total execution time.

name	shared access	local access	number of total threads	number of active threads	alloc time
P/C	0	0	7	7	0.0%
traffic	455	91	289	3	12.0%
roller	414	53	9	9	0.0%
bank	0	0	9	9	0.0%
series	12	0	4	4	0.0%
mtrt	0	0	3	3	0.0%

ments would be possible after this design is ported from the baseline compiler implemented currently to the optimizing compiler, which under our design would allow more aggressive optimization to those benchmarks with multiple threads.

A clear exception which shows considerable performance deterioration in our design is the *traffic* benchmark. The fifth column in Table 5.2 indicates the crux of the matter is table allocation which contributes to 12% of execution time. There are 2 factors that result in this situation, both of which can be understood from the data shown in columns 3 and 4 of Table 5.1. First, we note that this program creates 144 threads for drivers and 144 threads for cars (and a main thread). This heavy rate of thread generation interacts poorly with our design choice to eagerly duplicate data during thread initialization. This accounts for the bulk of the introduced overhead. A second notable factor is seen in the field access density which is defined as number of static variable accesses per millisecond. Static variable accesses are modified to let the language support thread-local variables inherently, and while this is certainly faster than using `ThreadLocal` objects it does not out perform static variable accesses in the original Java. In the original version of *Jikes* RVM, accessing static fields is done by accessing all data in a single table and this makes the original Java more data-cache-friendly than the new semantics. This is particularly true for the *traffic* benchmark, given the large number of short-lived threads, each with their separate copies of static data in the new semantics.

Note that we do not separately measure or account for heap variable accesses; these access costs are the same as in original virtual machine. There may be a cache impact from this duplication as well, but the only extra overhead for instance data is due to deep-copying included in the alloc time at thread start.

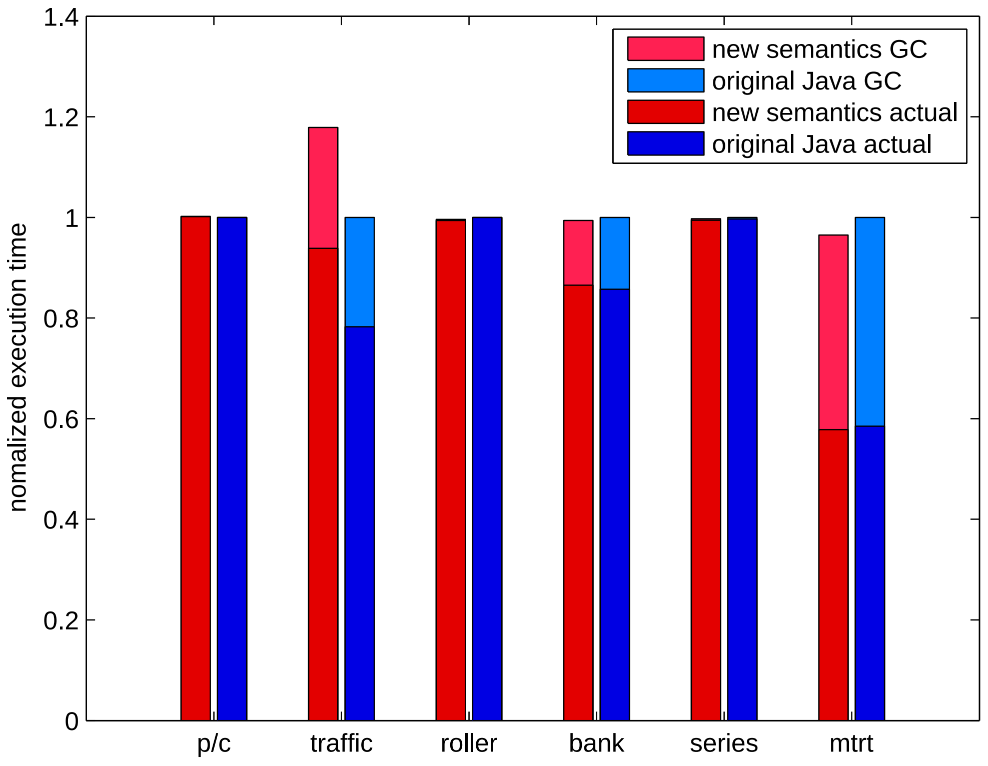


Figure 5.6 Normalized execution time of non-trivial benchmarks, comparing behavior of the modified benchmarks under our new dialect with their unmodified execution times under the current Java language. The red portion of each bar identifies the portion of time spent in garbage collection. The time value is the average of 50 runs.

Heap data is shown in Table 5.3. Duplicating data for each thread is expected to increase memory requirements, and this is apparent in the maximum memory requirements measured for our benchmarks. One thing worth noticing is that the number of active threads rather than number of total threads matters. This can be justified by the *traffic* benchmark. Despite the heavy cost due to table allocation inside the *traffic* benchmark, the actual memory requirement are larger but not significantly larger compared with the huge number of threads it launches. This is because although the *traffic* benchmark creates a lot of new threads the threads are running in turn and at each moment there are only 3 active threads. Interestingly, while there is an impact, the data does not show an increase proportional to number of active threads, as found in our micro benchmarks. Individual thread tables contribute to an increase in memory consumption, and overall increases are significant, but the relative amount of actual thread-local versus shared data is a more important factor. The *roller* benchmark has 9 active thread and quite a lot thread-local data. Therefore it shows a very high increase memory requirements. This is because it needs to create and hold onto local copies for all thread-local data for all the threads.

Table 5.3 Memory Usage

name	Java(MB)	new semantics(MB)
p/c	7	11
traffic	6	9
roller	8	17
bank	6	9
series	21	22
mtrt	13	16

From all the data-sets, it is clear that an important avenue for improvement in our scheme is to reduce the cost of the deep-copying performed during thread initialization. For class variables, a lazy, copy-on-write (or copy-on-access) scheme may be able to help the situation by delaying the deep-copying process until a thread's first access to this static field. Implementing this with an additional check before access would incur other overhead, but a coarser scheme using write-barriers as is common in some GC designs [BH04] could be quite efficient.

5.2.2 New Semantics versus Benchmarks with ThreadLocal

The non-trivial benchmarks targeting shared variables in multithreaded programs do not really make use of thread-local accesses very often. To investigate the effect of the new semantics on thread-local accesses we need to manually change the benchmarks so that they frequently use thread-local data in form of uses of the `ThreadLocal` class. We choose the *series* benchmark in which several threads are calculating Fourier coefficients and storing the result to a shared 2×1000000 double type matrix. The matrix is divided into different partitions and each thread is in charge of calculating one partition. The character of this benchmarks makes it easier to change shared data accesses to thread-local accesses through `ThreadLocal` class. On the one hand, calculation of each partition in the matrices is independent and the shared matrix can be easily replaced with several smaller thread-local matrices. On the other hand, because the benchmark contains only regular accesses to matrix elements the thread-local accesses can be conveniently identified and replaced with `get()` and `set( Object object )` methods. The modified Java program we get contains 2000000 thread-local accesses in total.

Table 5.4 New semantics versus original Java with `ThreadLocal`

name	total time (milliseconds)	GC time(milliseconds)	memory(MB)
new semantics	16610.9	475.05	44
original Java	16820.7	486.85	37

Table 5.4 shows the result of the changed *series* benchmark. The time value is measured by 50 runs. Disappointingly, there is little improvement in speed from the original Java to the new semantics. Despite the large number of accesses to `ThreadLocal` objects in this CPU intensive benchmark the expensive code generated by the baseline compiler for calculation of each datum dominates the execution of the program and dilutes the access time improvements. An advanced optimization compiler has a better chance of showing the reduced overhead that we expect from our design based on our micro benchmarks. The increase in memory has similar justification with previous tests. One interesting observation is that because we split the shared large matrices into several small matrices the memory requirements for both the new semantics and the original Java are increased by an amount around the size of the matrix. We suspect this is an artifact of the GC design of the semi-space collector in *Jikes* RVM. The unmodified version stores the entire array in a single *large-object-space*, a special area of memory used for large objects that are not collected frequently, while partitioned matrices with reduced size in the modified code are allocated into the semi-space, and thus require twice the memory.

Chapter 6
Related Work

6.1 Related Work

Our work is mainly related to three areas and in this chapter we will introduce other people's work in those areas. *Section 6.1.1* talks about the efforts in justifying compiler optimization under the Java Memory Model [MPA05] and trying to guarantee a race-free property for a given program. Our work is motivated by the difficulties in both these two efforts. *Section 6.1.2* introduces modern concurrent program languages using sharing directives to specify shared data. This is the source of our inspiration to provide a similar language feature for Java. Our implementation provides threads with private value heap and separate object copies. Other researchers also tried to explore thread-local space features in Java and we introduce their work in *Section 6.1.3*.

6.1.1 Data-Race-Freedom and Optimization

Efforts trying to bound the behavior of racy programs make the Java Memory Model very complicated and researchers have been focusing on finding out exactly what optimizations are allowed. Ševčík, for instance, applies semantic techniques on code transformation to validate the correctness of compiler optimizations [Š11]. He attempts to prove that combinations of the classes of transformations he defined can provide sequential consistency for DRF programs and also guarantee that no *out-of-thin-air* values for racy programs are generated (a central tenet of the JMM). *Out-of-thin-air* values are a consequence of simple attempts to define semantics for the racy programs in Java. If not carefully handled memory model semantics can be used to justify arbitrary data showing up in variables. This obviously unrealistic and undesirable behaviour is known as *out-of-thin-air* data. The JMM spends considerable efforts to avoid *out-of-thin-air* data.

Marino and his colleagues argue that relaxed memory models do not necessarily lose the property of preserving sequential-consistency (SC) [MSM+11]. They find that many optimizations are either already SC-preserving or can be modified to have this property, and so the subset of SC-preserving optimizations are sufficient for improving program performance. Nevertheless, some of their benchmarks still show a performance gap between compilers only allowing SC-preserving optimization and those that use a more complete optimization suite as well.

One way of avoiding the troubles in validating optimizations under different memory models is to determine whether the programs are DRF by detecting existing race conditions. No-overhead, static approaches are possible [BLR02, AFF06], but tend to suffer from excessive numbers of false positives, and therefore, state-of-the-art race detectors are principally dynamic ones. Unfortunately, while much more precise, these bring very non-trivial runtime overhead, with state-of-the-art dynamic race-detection currently resulting in approximately an 8-fold slowdown [FF09]. Faster and more accurate dynamic race detection requires cooperation from hardware. Marino *et al.*'s *DRFx* system, for example, uses hardware-driven detection, throwing a runtime exception when a violation of SC occurs [MSM+10]. Less specialized hardware is required by Greathouse *et al.*'s performance-counter-based, demand-driven method, which turns on race detection only when there is likelihood of data-sharing among different threads [GMF+11].

Our effort targets the memory model for the Java language as well but is different from all those above. We would like to make sure that no data races exist no matter how programmers program with the language. Many designs have in fact explored this general approach in the past. *Guava*, for example, a race-free version of Java, uses a type system to explicitly distinguish between objects that are shared and those that are not. Bacon *et al.* define three disjoint type-classes so the program can tell a compiler whether an object can be moved from one thread to another, can only stay in one thread, or if it can be shared [BST00]. Chandrasekhar and Martin classify objects from another angle, letting programmers apply different protection mechanisms to objects by specifying different types [BR01]. These designs do not focus on *thread-local* data itself, which both guarantees thread-specific data-values and offers a uniform syntax to access the "same" data in multiple threads. *Loci* is a type system for thread-local data in Java [WPM+09]. It does static checking based on the programmer's intention for thread-locality expressed through annotation and change thread-specific fields into `ThreadLocal` classes. The type system separates the heap into thread-specific parts and shared parts. There is an important difference between their work and ours other than exploring thread-specific instead of thread-local property. When they find a violation to the programmer's intention, the program does not compile. This makes the method essentially a static approach. In our case, the program always compiles even if the resulting program behavior is not quite what the programmer expects. Like our work, they also notice the merit of distinguishing thread-specific data explicitly to allow more code optimization in multithreaded programs, but the technique is more directly aimed at improving garbage collection.

6.1.2 Thread-local Data and Sharing Directives

Using sharing directives to explicitly specify objects that are shared is a common concept in parallel and multi-process programming. Our interest in better supporting thread-local data in Java is derived from the syntactically and programmatically simple design for shared data found in languages such as UPC [UPC05], Titanium [YSP+98], OpenMP [Ope08], as well as extensions to C found in the Microsoft compiler [micb] and others. In UPC, for instance, each thread has a private address space separate from a common, shared memory area. Objects are allocated by default in the local space, with the same object name used to reach different data in different threads. Shared data must be explicitly marked *shared* to be accessible by multiple threads. In Microsoft C, the extended storage-class[1] attribute *thread* is used with _declspec as an annotation to declare a thread local variable. When

[1] C has a concept of 'storage-class' defining the scope (visibility) and life time of variables and/or functions. Examples of other storage-class modifiers are *static* and *extern* keywords.

a new thread starts, it is allocated some thread specific memory in the process heap to hold these variables [mica]. Our approach here aims to support a similar concept ported to Java, repurposing the *volatile* Java keyword to differentiate shared from thread-local content. This has the advantage of offering a very familiar-looking environment to Java programmers, straightforward porting of existing Java programs, and close identification with ideas found in new languages such as UPC and OpenMP.

6.1.3 Thread-local Implementations in Java

While direct, optimized support for thread-local storage in Java is novel to the best of our knowledge, significant optimization work exists for exploiting thread-*specific* data properties. In Java, a basic memory management scheme allocates all the objects in a single heap, shared among all the threads. Synchronization constructs, as well as garbage collection algorithms thus have to assume any given data may in fact be shared, and take appropriate access precautions. If data can be determined through analysis to be only accessible by one thread, however, individual thread data can be allocated to distinct partitions of the heap [DGK$^+$02] [Ste00], or objects can allocated within the runtime call stack [MKB09]. Most works in this area, however, aim at reducing allocation, GC, and synchronization overhead [MPJ11], rather than focusing on supporting general race-free guarantees.

Chapter 7
Conclusions and Future Work

Data-race-free (DRF) languages are an important direction in concurrent language research, eliminating much of the complexity of the memory consistency concerns that affect existing concurrent languages. In this work we have described how support for a thread-local by default data-model could be added into a Java Virtual Machine, significantly improving the overhead otherwise incurred by using the existing `ThreadLocal` implementation design. The intention here is to encourage DRF approaches and demonstrate they can be competitive with current languages, and this is largely borne out by our experimental work with larger benchmarks. In most cases we are able to show that even when the program is modified to use a race-free semantics based on thread-local data, performance is not reduced over the original, naive and fully shared execution. Unfortunately, even programs that use `ThreadLocal` intensively do not always improve by using our technique instead. For the *series* benchmark, there is a little improvement over the original benchmarks with `ThreadLocal` objects. Since micro-benchmarks show the improvement can be large, we believe this to be an artefact of our initial prototype design—the baseline compiler is convenient to modify but also generates expensive code that can dilute the benefits. In the future, we want to port the mechanism currently applied to the baseline compiler to the more sophisticated optimizing compiler to generate better code.

Future work will focus on providing and supporting a full DRF language design for Java based on extensive use of thread-local data. This requires improving our implementation with respect to the programs that create great numbers of dynamic threads, as well as tailoring the garbage collection strategy. Our approach here shows feasibility and good performance are possible, but there are several places where our design introduces significant bottlenecks. Lazy copying or more advanced designs based on virtual memory would greatly mitigate our current thread creation overhead, as would precise forms of escape analysis [WR99] that could better identify data that is already thread-specific. Memory costs would also be reduced by making use of more complex memory management for static thread-locals, such as through an expanding table design. Finally, it seems possible to also apply our design outside of any novel DRF semantics, simply to speed-up existing uses of `ThreadLocal` in plain Java programs. Such work may encourage use of safer, thread-local program design even in existing Java semantics. This would, however, require additional program analysis to verify that use of a `ThreadLocal` wrapper object can be trivially replaced by direct (thread-local) value access.

Last but not least, a type system is needed to guarantee that only shared objects can be assigned to shared fields. Naively, each class requires two definitions: one for thread-local instances and the other for shared instances with fields tagged with *volatile*. If an object will stay with in one thread during its life time, it is declared as an instance of the thread-local version of its class. Otherwise, it is declared as an instance of the shared version of its class. When a thread-local object needs to be shared at a certain point, it should be copied to the public space with the shared version of its class. Providing a model for the transformation can be an interesting topic.

Bibliography

[ABC$^+$02] Bowen Alpern, Maria Butrico, Anthony Cocchi, Julian Dolby, Stephen Fink, David Grove, and Ton Ngo. Experiences porting the Jikes RVM to linux/IA32. In *2nd Java(TM) Virtual Machine Research and Technology Symposium (JVM'02*, pages 51–64. USENIX Association, 2002.

[AFF06] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 2–14, New York, NY, USA, 1990. ACM.

[Au07] David Aspinall and Jaroslav Ševčík. Java memory model examples: Good, bad and ugly. In *VAMP 2007: 1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs,* Sep 2007.

[BA] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78.

[BH04] Stephen M. Blackburn and Antony L. Hosking. Barriers: friend or foe? In *Proceedings of the 4th international symposium on Memory management*, ISMM '04, pages 143–151, New York, NY, USA, 2004. ACM.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230. ACM, 2002.

[Boe11] Hans-J. Boehm. Performance implications of fence-based memory models. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 13–19, New York, NY, USA, 2011. ACM.

[BR01] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 56–69, New York, NY, USA, 2001. ACM.

[BST00] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 382–400, New York, NY, USA, 2000. ACM.

[CKS07] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2007.

[DGK$^+$02] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In *Proceedings of the 3rd international symposium on Memory management*, ISMM '02, pages 76–87, New York, NY, USA, 2002. ACM.

[DM] Digital Mars. D Programming Language. http://dlang.org/index.html.

[FF09] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 121–133, New York, NY, USA, 2009. ACM.

[GMF+11] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Frank, Ramesh Peri, and Todd Austin. Demand-driven software race detection using hardware performance counters. In *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 165–176, New York, NY, USA, 2011. ACM.

[gra] Project: Java Grande. `http://www.epcc.ed.ac.uk/research/java-grande/`.

[jik] The Jikes RVM Project. `http://jikesrvm.org/`.

[Lea99] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[mica] An In-Depth Look into the Win32 Portable Executable File Format Part 2. `http://idletester.com/xcontract/PDF/Look%20into%20the%20Win32%20Portable.pdf`.

[micb] Thread Local Storage. `http://msdn.microsoft.com/en-us/library/4ax54352.aspx`.

[MKB09] Peter Molnar, Andreas Krall, and Florian Brandner. Stack allocation of objects in the cacao virtual machine. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 153–161, New York, NY, USA, 2009. ACM.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Princi-*

ples of programming languages, POPL ’05, pages 378–391, New York, NY, USA, 2005. ACM.

[MPJ11] Simon Marlow and Simon Peyton Jones. Multicore garbage collection with local heaps. In *Proceedings of the international symposium on Memory management*, ISMM ’11, pages 21–32, New York, NY, USA, 2011. ACM.

[MSM+10] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. DRFx: a simple and efficient memory model for concurrent programming languages. In *PLDI ’10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 351–362, New York, NY, USA, 2010. ACM.

[MSM+11] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. A case for an SC-preserving compiler. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI ’11, pages 199–210, New York, NY, USA, 2011. ACM.

[Ope08] OpenMP Architecture Review Board. OpenMP application program interface. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008. Version 3.0.

[ŠevčíkA08] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. `http://homepages.inf.ed.ac.uk/s0566973/jmmtrans.pdf`, 2008.

[Ste00] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the 2nd international symposium on Memory management*, ISMM ’00, pages 18–24, New York, NY, USA, 2000. ACM.

[sun] Sun Java Tutorial. http://java.sun.com/docs/books/tutorial/index.html.

[UPC05] UPC Consortium. UPC language specifications v1.2. `http://www.gwu.edu/~upc/publications/LBNL-59208.pdf`, May 2005.

[VKZ11] Clark Verbrugge, Allan Kielstra, and Yi Zhang. There is nothing wrong with out-of-thin-air: compiler optimization and memory models. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 1–6, New York, NY, USA, 2011. ACM.

[Š11] Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 306–316, New York, NY, USA, 2011. ACM.

[WPM+09] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for Java. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 445–469, Berlin, Heidelberg, 2009. Springer-Verlag.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 187–206, New York, NY, USA, 1999. ACM.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, September 1998. Special Issue: Java for High-performance Network Computing.

[ZDG+10] Chao Zhang, Chen Ding, Xiaoming Gu, Kirk Kelsey, Tongxin Bai, and Xiaobing Feng. Continuous speculative program parallelization in software. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 335–336, New York, NY, USA, 2010. ACM.