UPPSALA
UNIVERSITET

IT 13 013
Examensarbete 45 hp
Februari 2013

Evaluating the Accuracy of Annotations in the Loci 3.0 Pluggable Type Checker

Nosheen Zaza

Institutionen för informationsteknologi
Department of Information Technology

To my beloved mom

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Evaluating the Accuracy of Annotations in the Loci 3.0 Pluggable Type Checker

Nosheen Zaza

This thesis work investigates the accuracy of Loci, a static type checker in expressing thread-locality at compile time. To do this, we need to capture both thread-locality at runtime, and thread-locality expressed statically using Loci. We present the framework we built to find these measurements, describe the process of adding Loci annotations to two multi-threaded benchmarks, and measure the accuracy of expressed thread-locality using the developed framework. We found that Loci annotations could express a thread-locality rate of 99.9% in terms of object count for a small benchmark of 1436 LOC, using a total of 15 annotations, and a rate of 83.5% for a part of Xalan from the DaCapo benchmark suite, composed of more than 85,000 LOC, using 451 annotations. These results show that Loci can be used to capture a high-rate of thread-locality with a small annotation overhead.

Handledare: Tobias Wrigstad
Ämnesgranskare: Konstantinos Sagonas
Examinator: Ivan Christoff
IT 13 013
Tryckt av: Reprocentralen ITC

Acknowledgements

I would like to thank my thesis supervisor, Tobias Wrigstad, for his continuous help and valuable feedback. Many thanks to my dear parents and sisters, for always being there to support me in every way. Special thanks to my dear husband, Amanj Sherwany, who happens to be the developer of Loci 2.0. Your help and support made my work a lot easier. I would also like to take this chance and thank you for making my life better everyday. Finally, I thank my little feline friend, Victoria for making the dull days more bearable, and forbidding me from getting glued to my computer chair.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Thread-locality is an important concept in the context of multi-threaded computing. Thread-locality of a datum means that it is used by one thread only. Computations performed on thread-local data take place inside a single thread, which permits giving guarantees, and performing optimizations only possible in sequential code to such computations. Detecting thread-local data simplifies reasoning about concurrent programs.

To express thread-locality in Java, Wrigstad et al. [25] proposed Loci, a simple type system that lets programmers express their intentions with respect to thread locality, and statically checks that the expressed intentions are consistent throughout the application. By allowing thread-locality to be expressed in a syntactically traceable way, thread-locality of a datum is easily propagated throughout a program and made easily visible to a programmer or static analyzer.

To ensure soundness of Loci, no values are labeled thread-local unless the system can prove that through static analysis. If it is not possible to statically prove thread-locality of certain values Loci would conservatively consider them shared, to ensure safety in the sense that programmers do not treat shared data as thread-local. Furthermore, Loci is designed to be simple and minimal, which might lead to deeming parts of data that are thread-local in practice as shared, either due to lack of expressive power, or to keep soundness. For example, static fields of a class are always considered shared in Loci, but it is not the case that every static field will be accessed by multiple threads at runtime. All these factors reduce the accuracy of Loci in capturing all thread-local data.

The goal of this work is to evaluate the design of Loci annotations with respect to their ability to capture the actual thread-locality of a system. Concretely, we set out to measure the extent to which Loci can be used to capture existing thread-locality of a system without substantial refactoring,

and the annotation overhead required. In particular, we investigate whether it is possible to use Loci on only parts of a system without propagating annotation requirements, and still capture substantial thread-locality.

We quantified the amount of thread-locality Loci fails to capture, by detecting actual thread-locality on object-level in running Java programs, and comparing it to Loci's view of locality for each object. Loci could correctly capture thread-locality rate in the range 83.5-99%, using roughly 5 annotations per 1000 lines of code. In addition to the main goal, important information were gathered about thread-locality properties of Java applications, as thread-locality rate for all benchmarks we ran was over 84%, when considering only user-threads. This work also led to some changes to Loci's design, which are further explained in Chapter 5.

The rest of this chapter is organized as follows: Section 1.2 briefly introduces the Loci system, Section 1.3 discusses methods used to evaluate Loci from different perspectives, Section 1.4 lists contributions of this work, and Section 1.5 outlines the rest of this thesis report.

1.2 The Loci System

The Loci system is a pluggable type checker [17] for Java, originally proposed by Wrigstad et al. [25], and extended by Sherwany and Wrigstad [24]. It allows programmers to add thread-locality information to Java language constructs through a set of statically checkable annotations. The checks performed by Loci at compile time ensure that the expressed programmer intentions on thread-locality are consistent throughout the program, and that thread-local data will never be touched by more than one thread.

To reduce annotation burden, and to avoid cluttering code, there are default annotations elaborated where they are not expressly inserted, and part of this work evaluates the choice of defaults.

The first version of Loci (Loci 1.0) was implemented as an eclipse plug-in, on-top of the standard type checker presented in Java 6. The underlying framework was not powerful enough to fully express the intended design, or fully support all Java language features. Evaluation of the implementation suggested improvements to the design of Loci, in order to make it more expressive, flexible and compatible with the way Java code is usually written.

The design of Loci was refined, and it was reimplemented by Sherwany and Wrigstad [24], leading to the second version of Loci (Loci 2.0). Due to design changes and extended support of Java, it was not straight forward to generalize results obtained from evaluating Loci 1.0 to Loci 2.0, and the evaluation procedure was repeated. During evaluation of Loci 2.0, more design decisions were changed, leading to Loci 3.0, which is the Loci version we re-evaluate in this study.

The second chapter of this thesis overviews the Loci system design and

implementation, and gives various usage examples. The next section describes how both Loci 1.0 and Loci 2.0 were evaluated, the shortcomings of the evaluation process, and then describes the comprehensive procedure we followed to evaluate Loci 3.0.

1.3 Evaluating Loci

1.3.1 Previous Work

An evaluation of the syntactic-overhead of Loci 1.0 was carried out before by Wrigstad et al. in [25]. Loci 1.0 annotations were added manually to a number of benchmarks, and automatically using a type inferencer to the standard Java library. The results showed that the design of Loci 1.0 is compatible with the way Java programs are written, and the syntactic-overhead was relatively small. In addition to that, the rate of thread-locality for selected Java benchmarks was measured, using an instrumented JVM to report shared and thread-local objects in running Java applications. 69% of all objects created when running the entire DaCapo benchmark suite [15] were found thread-local. Furthermore, All objects annotated thread-local were indeed accessed by a single thread, and this showed that the implementation of Loci 1.0 is correct.

Loci 2.0 was evaluated in terms of syntactic-overhead in a similar manner. Since the annotated code base of Loci 1.0 was not compatible with the new rules of Loci 2.0, the annotation work had to be redone. Some design flaws were uncovered and fixed only after a major part of the evaluation was conducted. Many parts of the annotated benchmarks were no longer compatible with the refined design, which adversely affected the accuracy of the evaluation process. Also, due to lack of resources, Loci 2.0 was not evaluated in terms of ability to express thread-locality as present during runtime.

1.3.2 Our Evaluation Goals

In this work, we conducted a thorough evaluation of the current design and implementation of Loci 3.0. The following aspects were investigated:

- *Ability to capture a high rate of thread-locality*: Loci is designed to minimise the number of annotations necessary, and uses a minimalist design, which may affect precision negatively. This work sets out to study the precision of Loci annotations to determine the validity of this fundamental design choice. We wish to know the percentage of a program's actual thread-locality that can be captured by moderate addition of Loci annotations.

- *Syntactic-overhead*: Loci's design and choice of defaults aims to allow programmers express their intentions with as few annotations as possible. It was shown during the evaluation of previous versions of Loci that it is possible to only annotate parts of a program and still express intended thread-locality. However, the annotation-overhead required to capture a *maximum rate* of thread-locality was not measured. The latter is important to understand the extent Loci could be used to *e.g.* improve performance through optimizations based on thread-local values.

- *Finding possible flaws in Loci's implementation*: We want to check that under the current implementation of Loci, all objects annotated thread-local were indeed accessed by a single thread at runtime. This aids finding and fixing implementation bugs.

1.3.3 Evaluation Steps

To go about measuring the amount of thread-locality Loci fails to capture, we need to measure actual thread-locality on object-level in running Java programs, and compare it to Loci's view of locality for each object. We have built a Java profiler to report, for a number of multi-threaded Java benchmarks, the rate of thread-locality in terms of object percentage and memory-in-bytes percentage, and recorded thread-locality for each object allocated from the source code[1]. We also extended the Loci system with functionalities that facilitate reporting thread-locality, as assumed by Loci for each object allocation. After that, we generated runtime-thread-locality reports using the profiler for existing systems, then we used Loci to annotate them, with the aim to express as much possible of thread-locality as reported. Finally, we compared for each object, its actual and Loci-assigned thread-locality, and reported the percentage of objects for which these two views did not agree.

1.4 Contributions

This work yields the following contributions:

C1: Design and implementation of a thread-locality profiler An agent that can be attached to any Java virtual machine that supports JVMTI (Java Virtual Machine Tool Interface [10]), to monitor and report object allocations and thread accesses to objects during a run of a Java program.

[1] There are also objects allocated by the compiler or virtual machine, and these are irrelevant to our evaluation goal.

C2 Extending Loci to pass its view of thread-locality to runtime The Loci type checker was extended to facilitate passing thread-locality information for each object creation from source code to instrumented byte code. The instrumented byte-code provides an interface for external tools to get a reference to a newly allocated object and a flag indicating its statically inferred thread-locality. We used this interface to provide our thread-locality profiler with object allocation events along with each object's statically inferred thread-locality.

C2 Changing some design decisions of Loci: We changed some design decisions to reduce annotation overhead when dealing with primitive arrays. We also established that enumeration types are always shared.

C3 Annotating a large code base with Loci 3.0 annotations We annotated more than 85,000 lines of code, according to latest Loci specifications (Loci 3.0).

C4 Evaluation of Loci's design and implementation Our tests show that Loci was able to capture thread-locality rate in the range 83.5-99%, with a low syntactic over head, roughly 5 annotations per 1000 lines of code, which can be attributed to careful design decisions to good choice of defaults.

A notable result is that for all the benchmarks we tested, the rate of thread-locality was higher than 84% in terms of number of allocated objects. It should be noted that this rate does not include objects allocated outside the source code of a running application, such as object allocation instructions inserted by the compiler or objects allocated by the virtual machine to support its own operations. In addition to that, only threads visible during compile-time are monitored, meaning that the effect of other threads created by the the virtual machine (*e.g* garbage collector) is not considered. This is important since it suggests that thread-locality based optimizations can have a high impact on improving the performance of Java applications.

1.5 Possible Applications

The ability to statically express and check thread-locality using Loci annotations allows programmers to grasp useful knowledge on how their concurrent programs operate, and it becomes easier to reason about in certain aspects of parallel programming, such as identifying shared data and coordinating threads accesses to it.

Tools facilitating thread-locality based optimizations can use information expressed by programmers through Loci, instead of being solely guided by

automatic tools [18]. An interesting application we are currently investigating is using Loci's type information in the simplification of cache coherence protocols. Blas Cuesta, Alberto Ros et al. have shown [20] that the effectiveness of directory caches can be increased by deactivating coherence for private memory blocks. It was shown in simulations that this contributes to either shorten the runtime of parallel applications by 15% keeping directory cache size, or maintain system performance while using directory caches 8 times smaller. The dynamic technique proposed to identify those private blocks is based on analysis of operating system pages, and it detects that for most parallel programs, around 60-70% of memory blocks used are private. However, this technique is not accurate, and can miss many private memory blocks. In this study, we have shown that 80-99% of thread-locality can be detected using Loci statically. These information can be used to detect a larger percentage of private memory blocks, compared to the dynamic approach, which is expected to deliver better runtime, and/or better cache utilization for parallel Java applications. We are working with Alberto Ros to devise a technique that enables utilizing Loci's type information by the cache proposed in [20].

1.6 Thesis Outline

This thesis consists of six chapters. The second chapter overviews the Loci system: its core design, current implementation and provides some usage examples. The third chapter covers measuring thread-locality of Java programs, using the thread-locality profiler we developed, and shows results obtained from profiling selected benchmarks from DaCapo [15] and SPECjvm2008 [11] benchmarks suites. Chapter Four describes how Loci was extended to generate instrumented bytecode, and the interface provided to external tools to capture Loci's thread-locality information at runtime. Chapter Five describes the process of adding Loci annotations to Xalan [15] and Ray Tracer [5] Java benchmarks, and how this process lead to changes in the design of Loci. This chapter also contains the results and realizations of this study. The last chapter concludes and outlines future work.

Chapter 2

Overview of the Loci System

In chapter one, the Loci system was briefly introduced. We now overview Loci in more depth. We start with a literature survey of proposals for detecting and enforcing thread-locality, and how Loci compares to them. Then we present Loci's design, and describe it's current implementation. Finally we demonstrate how to use Loci, and present some examples[1]

2.1 Studies on Thread-Locality

Thread-locality can be detected/monitored/enforced statically or dynamically, and has been studied for a variety of purposes. Domani et al. [21] propose thread-local heaps where each thread is given its own chunk of memory in which to allocate objects. The goal is to remove locking from the Garbage collector for thread local objects. They use dynamic analysis at runtime to track thread-locality, and do not enforce it. Choi et al. [18] propose a data flow algorithm applied at compile-time for escape analysis of objects in Java programs to determine if an object can be allocated on the stack or if an object is accessed only by a single thread during its lifetime, so that synchronization operations on that object can be removed. Other researchers [16] have employed compile-time escape analysis to identify local and shared objects, and some JVMs perform similar analysis under the hood [14].

The above mentioned proposals mainly target compiler optimizations and memory management, through detection of thread-locality in Java programs. The Loci system, originally proposed by Wrigstad et al. [25] targets thread-locality from another perspective, it allows programmers to express their intentions in regard to thread-locality by attaching thread-locality information to data as type information, and statically checks whether these

[1] Most of the material presented in this chapter is adapted from the second chapter of [24].

intentions are consistent throughout a program (e.g. a shared type is not assigned to a local type). A program using Loci's annotations is easier to reason about in certain aspects; for instance, programmers can safely drop synchronization code or locks for objects typed as thread-local, or make sure that shared objects have proper access guards. Also, tools facilitating thread-locality based optimizations can use information expressed by programmers through Loci, instead of being solely guided by automatic tools as in [18], which may allow detecting higher rates of thread-local objects.

The rest of this chapter further describes the Loci system. The following section lists design goals and properties of Loci.

2.2 Properties of Loci

Loci is a pluggable, ownership-based type checker [17] of thread-locality, for Java and Java-like programming languages. It was designed to meet the following design goals [24]:

G1: Conservative checking (soundness) It detects all leaks that occur or might occur. False positives might ensue.

G2: Optional checking Loci is a pluggable type system, meaning it can be applied only to some modules, and be turned on and off during different stages of development.

G3: Simplicity The system has a small number of annotations and a set of simple rules, to make the system practical and easy to learn and use.

G4: Low syntactic-overhead Each type is assigned a default annotation in the absence of explicit annotations, based on Loci's ownership type rules (*e.g.* for an instance type it is the same as the enclosing instance), in order to minimize the amount of manually-added annotations.

There are other tools similar in spirit to Loci, such as the tool proposed by Clarke, Dave and Wrigstad [19]. Loci mainly differs from other tools in being minimal and focused on thread-local data.

We focus in this work on the Java specific implementation of Loci, from this point on, all discussions refer to this implementation.

2.3 Loci for Java

Loci is applicable to object-oriented class-based languages similar to Java. Currently, the only implementation is for Java. This implementation of Loci aims to achieve two more design goals:

G5: Backward-compatibility It should be possible and feasible to port legacy Java code to use Loci without requiring extensive refactoring.

G6: Fully defined The type checker covers all features of the Java programming language.

Loci is implemented for Java as a compiler plug-in. The additional types of Loci are expressed as annotations [2]. The Java platform provides an API for annotation processing. However, Loci permits annotating certain Java constructs for which this API does not provide annotation support (*e.g.* generics or class instantiation), which limits expressing the intended design. To overcome this limitation, Loci 2.0 was implemented using the Checker Framework [22], which provides extended support for annotation processing. This framework consists of an API that permits building compiler plug-ins to perform optional type checking, a number of ready to use plug-ins, referred to as "checkers", a custom Java compiler, and a number of supporting specifications, file formats and tools.

In the following section, Loci's core annotations are presented. It should suffice for the reader to understand the core annotations in order to be able to follow this work. For a complete description of all Loci annotations, consult the Loci tool manual [26].

2.4 Annotations

Loci extends Java with three annotations:

`@Local` which denotes a thread-local value; meaning that the value is (or all instances of a class will be) thread-local

`@Shared` which denotes a value that can be arbitrarily shared between threads; and

`@ThreadSafe` denotes an object whose thread-locality is not known and must therefore be treated conservatively.

Loci requires that data flow preserves the thread-locality of a value in a variable, except for assignments into `@ThreadSafe`. This dictates what must hold if the contents of a variable y is stored into a variable, field or parameter x (by assignment, argument passing, etc.). For clarity, a summary is found in Figure 2.1.

The use of these annotations is governed by Loci's view of memory, which will be explained in the following section.

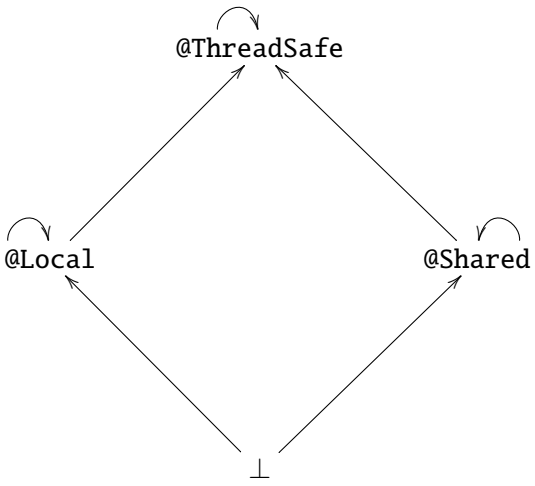

Figure 2.1: Annotation lattice capturing dataflow constraints. Each edge denotes permitted dataflow. ⊥ denotes "free" objects (objects with no annotation attached).

2.5 Logical View of Memory

Loci *logically* divides the heap into a number of "heaplets". Each thread has its own heaplet, and there is a one-to-one mapping between heaplets and threads, with the exception of the so-called shared heaplet which is accessible by all threads. The Loci annotation system enforces the following simple properties on Java programs, shown in Figure 2.2[2]:

1. References from one heaplet into another are not allowed ($\longrightarrow$).
2. References from heaplets to the shared heap are unrestricted ($\dashrightarrow$).
3. References from the shared-heap into a heaplet must be stored in a thread-local field ($\bullet\dashrightarrow$).

The third property above ensures that a thread-local piece of data (ρ_i) resides in a field which is only accessible by the thread i to which ρ_i belongs. If the j^{th} thread wants to access the same field, it will get the last value stored in the field by the j^{th} thread, or `null` if no such value exists. Logically, there is a copy of the field for each thread.

Together, these simple properties make heaplets effectively thread-local, and objects in the heaplets are thus safe from race conditions and data races [25].

[2]The figure is taken from Wrigstad et al. [25].

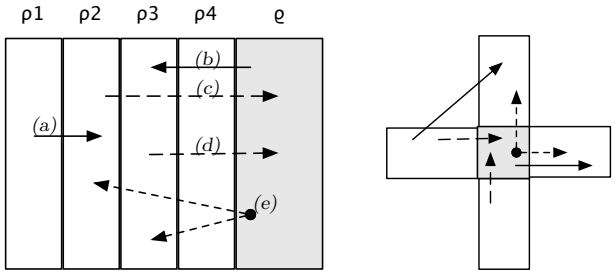

Figure 2.2: Thread-Local Heaplets and a Shared Heap. The gray area (ϱ) is the shared heap, white areas ($\rho_1..\rho_4$) represent the thread-local heaplets. Solid arrows are invalid and correspond to Property 1 in Section 2.5, dashed arrows are valid pointers into the shared heap (Property 2), respectively from the shared heap into heaplets (Property 3, when "anchored" in a bullet). The right-most figure is a Venn diagram-esque depiction of the same figure to illustrate the semantics of the shared heap.

2.6 Loci Example

The following is a very simple example (taken from the Loci Manual), to give the reader an insight on how Loci is applied to Java code:

```
1 import loci.quals.*;
2
3 @Local public class BadHelloWorld{
4   @Shared Object b = new @Local Object();
5   public static void main(String... args){
6     System.out.println("Example");
7   }
8 }
```

Listing 2.1: A short example that shows how Loci performs and constrains Java programs.

Note how, at the 4th line, an attempt to assign a `@Local` object to a `@Shared` reference is made, which is a violation according to Loci's rules. When attempting to compile the above program, the following error message is reported:

```
BadHelloWorld.java:3: incompatible types.
@Shared Object b = new @Local Object();
found : @Local Object
required: @Shared Object
1 error
```

2.7 Integration with Core Java Concepts

In this section, the most important rules of applying Loci's annotations to various Java constructs, and Loci's default annotations are presented.

2.7.1 Classes and Types

Classes in Loci can be `@Local` or `@Shared`. `@Shared` classes can only be instantiated in the shared heaplet, and `@Local` classes can only be instantiated in the heaplet of the current thread. A class is `@Local` (or `@Shared`) if the class, its super class or one of its implemented interfaces is annotated `@Local` (or `@Shared`). A class that has no explicit annotations and has no direct or indirect `@Shared` or `@Local` superclass (or implemented interfaces) can be used to create both shared and thread-local instances. These classes are useful for libraries and other situations where flexibility is desirable.

`@Local` and `@Shared` classes may only be subclassed by `@Local` and `@Shared` classes respectively. The `Object` root class is not annotated, which is an important design choice to allow maximum flexibility, since it is the parent of all Java classes.

If `c` is a field, variable, return type or parameter and is declared without having an explicit annotation, and is an instance of class `T` that is also unannotated, then `c`'s thread-locality will be the same as the current instance/object that holds it, and we say that it has an `@Owner` type. Consider the following example:

```
class Foo {
  Object f;
}

@Local Foo a = ...; //A thread-local instance of Foo
@Shared Foo b = ...; //A shared instance of Foo
```

Here, `a.f` is thread-local, and `b.f` is shared

2.7.2 Fields and Thread-Locality

`@Local` or `@ThreadSafe` fields are not allowed inside `@Shared` or unannotated classes. This is necessary since the enclosing object might be shared across threads making the field effectively shared too. One way to have them is by implementing them through a `java.lang.ThreadLocal` indirection, which degrades performance.

```
@Shared class Foo{
    @Local Object a; //Invalid
    ThreadLocal<@Local Object> b; //OK
}
```

2.7.3 Statics and Thread-Locality

Class objects are shared by default (completely ignorant of class loaders, etc.). In Java, the enclosing object of a static context is a class object. Therefore, every unannotated type in a static context defaults to `@Shared`

2.7.4 `@Owner` and Subtyping

As we mentioned in Section 2.7.1 an `@Owner` inherits its thread-locality from the object that contains it [26]. Therefore, an `@Owner` inside a `@Shared` object is `@Shared`, and an `@Owner` inside a `@Local` object is `@Local`.

```
class Foo{
  Object value; //Implicitly @Owner

  //The parameter type is implicitly @Owner
  void setValue(Object value){
    this.value = value;
  }

  //The return type is implicitly @Owner
  Object getValue(){
    return value;
  }
}

...
@Shared Foo aShared = new Foo();
@Local Foo bLocal = new Foo();

aShared.value = new @Local Object(); //Not OK, value is @Shared
bLocal.value = new @Shared Object(); //Not OK, value is @Local
```

Loci cannot guarantee the "actual" thread-locality of any `@ThreadSafe` type[3]. Therefore, Loci cannot determine the inherited thread-locality of any occurrence of `@Owner` types which are enclosed by a `@ThreadSafe` object. Imagine that we treat `@ThreadSafe` as `@Shared` and `@Local`, which means that all `@Owner` variables enclosed by a `@ThreadSafe` instance are `@ThreadSafe`, this makes the following program legal, despite leading to a serious leak.

```
class B{
  Object value; //value is implicitly @Owner
}

@Shared B b = ...; //b.value should be @Shared
@ThreadSafe B c = b; //if we suppose that c.value is @ThreadSafe
c.value = new @Local Object(); //Leak!
```

[3] At run time, everything is either `@Shared` or `@Local`.

To solve this, we follow the same path as wildcards in Java [8]. A wildcard in generics is a readable but not writable property:

```
class Cell{
  Object value;
  void set(Object t){...}
  Object read(){...}
}
...
@Shared Cell b = ...;
@ThreadSafe Cell c=b;//Java(as well as Loci) loses type information
@ThreadSafe Object b = c.read(); //OK

b.value = new @Shared Object(); //OK
c.value = new @Local Object(); //value is not writable
c.set(new Object());//Is not OK
```

A similar solution was suggested by Lu et al. [23] for their "Dynamic Exposure", for preventing exposing of an object by downgrading its dynamic ownership [23].

2.7.5 Arrays and Generics

Loci requires arrays to have the same thread-locality as their enclosed elements, with exception to arrays of primitive types[4]. In generics, annotations are bound similar to how types are bound by the extends clause.

```
class A<@ThreadSafe T> {}
class B<@Shared K> {}
```

The type parameter `T` in class `A` is bound by a `@ThreadSafe Object`, which means it can be bound to any thread-locality when the type is instantiated. However, class `B` only accepts `@Shared` instances as type arguments.

If a bound on an annotation is not explicitly defined, the annotation on the (possibly implicit) upper class bound is used.

```
class A<T extends @ThreadSafe Object> {}
class B<K extends @Shared Object> {}
```

We can pass any thread-locality to `T`, as it is bound by a `@ThreadSafe Object`. However we can only pass `@Shared` type argument to `K`.

[4]For instance, it is possible to have a `@Shared` array of `@Local` elements. In fact, this was not permitted in Loci versions preceding this study. In Section 5.6.2, we explain why we changed Loci's treatment of primitive arrays.

2.8 Annotating Java's Thread API

The `java.lang.Thread` and `java.lang.Runnable` are both annotated `@Shared`. This is natural, as instances of `Thread` or of classes implementing `Runnable` are always shared between at least two threads: the creating thread and the thread it represents [26].

2.9 Conclusion

Loci provides a simple way for programmers to statically express thread-locality. It is important for the system to be useful (especially for thread-locality based optimizations) that the annotation-overhead corresponds to detecting a high rate of thread-local objects in practice. To be able to evaluate Loci in this respect, we need to have both the rate of thread-locality at run-time, and the rate of thread-locality as expressed by Loci. In Chapter Three, we describe how we measure thread-locality at runtime using a profiling tool we built, and we show the results we got from profiling some Java benchmarks using it. Chapter Four presents an extension we developed to Loci to enable reporting thread-locality expressed by Loci for each object allocation.

Chapter 3

Measuring Thread-Locality of Java Programs

One of the main goals of this study is to examine if Loci's annotations can be applied to Java programs in a way to statically capture all/most thread-local data present at runtime. To be able to perform this examination, we need a way to capture runtime thread-locality information. In this chapter, we present a tool we designed and implemented for this purpose. We start by motivating the need for such a tool, then we overview its implementation, and finally we show the results of applying it on a number of Java benchmarks.

3.1 Detecting Thread-Locality at Runtime

To be able to measure the extent to which Loci is able to capture thread-locality, we must know the actual thread-locality of a program. Since the Java runtime does not monitor such things, and there were no third-party tools to report all information we need, we constructed our own monitoring tool. In addition to measuring thread-locality in a system, our tool also records the sequence of threads accessing a shared object and its class, total memory occupied and the fraction occupied by thread-local objects, and lists all Java threads[1] that ran during program execution and all loaded classes. Having this information is very useful to drive the annotation of existing programs, especially when the intention is to partially annotate them. It is possible, for instance to start by annotating classes that generate a large number of objects, all of which have the same thread-locality, and avoid annotating classes in the system that are not loaded.

It is important to note that we do not monitor objects created by the vir-

[1] As opposed to native and system threads.

tual machine to support its internal operation, nor native or system threads. These are irrelevant, since they are not visible at compile time when Loci operates, and should not affect a programmer's reasoning about thread-locality on source-code level. Moreover, code that is annotated `@Local` and passes Loci checks can safely forgo analysis that tries to locate synchronization errors or data races. We only treat the garbage collector thread specially, because we wish to know how it affects thread-locality. It should be noted that if Loci is to be used for thread-locality based optimizations, we need to find a way to manage the effect of system and VM threads.

3.2 Previous and Related Work

It was desired to know which fraction of objects are effectively thread-local since the first proposal of Loci, because having this knowledge influences its design and choice of defaults. This was measured by Wrigstad et al. [25] through instrumenting revision 15.182 of Jikes RVM [9] to report the fraction of live objects that have been used from multiple threads. Detecting object accesses was done using a read barrier. These measurements also included objects used by the VM, which itself is heavily multi-threaded, hence even for single-threaded benchmarks the rate was not 100%. In the entire DaCapo suite, 69% of all objects were reported as thread-local.

The main issue with the the JVM instrumentation approach is that objects used by the VM were also monitored, while they are not relevant to this study for the reasons mentioned earlier. Other than that, in order to monitor any Java application, it should be run on that specific instrumented JVM; any other parties that wish to use this tool must obtain this altered JVM. Furthermore, extending the instrumentation is cumbersome, as it involves going through the large code base of the JVM, and a detailed understanding of the general JVM specifications and Jikes code specifications.

We wanted to implement similar functionality in a way that would be portable, easier to use by other parties, and open for future extension as the Loci system evolves and needs to be evaluated from other perspectives. It is important to allow other parties to use this tool easily, because it gives a valuable insight on what goes on inside a concurrent application, and can be used to support assumptions of programmers about thread-locality and sharing, and also ease applying Loci annotations to legacy code.

3.3 Monitoring Thread-Locality Through Profiling

Gathering dynamic information during program execution is a profiling task. Profiling can be done in various ways (*e.g.* instrumentation, event monitoring), and many Java profiling frameworks are available. We built our

profiling tool as a JVMTI[2] pluggable profiling agent. The implementation was guided by the heap-tracker demo[4], as our agent's basic operations are similar. The interested reader can refer to Appendix A for implementation and usage details.

3.4 General Operation of Thread-Locality Agent

The most basic operation of the agent is to monitor each allocated object for thread accesses. All objects are considered local when first allocated (with a few exceptions, such as the objects of class `Thread`). If during runtime, an object is touched by more than one thread, it is marked as shared. We consider reading or writing to a field of an object to be "touching" an object. Calling a method of an object might also be considered touching, because for each instance method, a copy of 'this' reference is implicitly passed at the first formal parameter. This, however is irrelevant, as it does not change the state of the object, and if the state changes by writing to a field in the body of a method, this will be detected.

3.5 Results of Profiling Selected Java Benchmarks

Multi-threaded benchmarks from the Dacapo benchmark suite were run with the agent attached. These benchmarks are heavily multi-threaded, and contain a range of real-world applications. They were profiled before in the initial study of Loci, and we wanted to see how the rate detected by our profiler compares to the rates detected before. This suite scales the number of threads it creates according to the number of available cores. We also ran benchmarks from SPECjvm2008 benchmark suite [11]. The peak compliant run for each benchmark from this suite was profiled. We skipped the warm-up phase, executed one iteration that performed one operation, as we are not interested in measuring performance. We also disabled report generation, and restricted the maximum number of hardware threads to 16. These tests were run on a Dell PowerEdge R820 machine, with four Intel Xeon E5-4650 2.70GHz CPUs, 128GB of RAM and 64 available cores, running Debian wheezy (kernel 3.2.0-3-amd64).

Table 3.1 shows profiling results. It is notable that the minimum thread-locality rate reported is 84.76%, and the average rate in these benchmarks is 98.40%. This is significantly larger than the rate of 69% detected in the initial study, and suggests that the Java runtime contributed around 30% of the data sharing rate.

[2]The Java Virtual Machine Tool Interface [10]

[4]The source code of this demo can be found with any installation of Oracle JDK higher than 5.0 or OpenJDK

Benchmark	Locality Rate (# objects)	Shared Memory	Threads	Original Runtime(s)	Runtime with Agent
DaCapo:					
Avrora	84.76%	16.18%	8	5.098	40m 26s
Batik	99.98%	0.019%	10	2.938	55s
Eclipse	97.59%	2.9%	295	29.045	29m 8s
Sunflow	99.99%	0.0%	133	1.442	2h 48m 48s
Lusearch	99.99%	0.0%	66	1.610	29m 57s
Luindex	99.97%	0.03%	3	1.138	2m 11s
H2	97.94%	1.8%	66	8.582	41m 53s
Tomcat	98.68%	1.33%	205	2.934	6m 5s
PMD	99.96%	0.046%	67	4.914	7m 30s
Xalan	99.76%	0.27%	66	3.967	25m 6s
SPECjvm 2008:					
Compress	99.11%	1.84%	20	1.895	8h 9m 29s
Crypto	99.89%	0.13%	54	6.298	3h 11m 28s
Serial	99.99%	0.00%	19	22.254	4h 12m 7s
Compiler	99.78%	0.22%	37	12.250	6h 45m 35s
Derby	99.87%	0.17%	37	25.836	3h 2m 58s
MPEGaudio	98.75%	2.02%	20	2.745	2h 20m 16s
Scimark	99.59%	1.08%	173	56.322	32h 39m 10s

Table 3.1: Results of profiling benchmarks from DaCapo and SPECjvm2008 suites

For more detailed results, please refer to Appendix C to view profiling summaries generated for these benchmarks.

3.6 Conclusions and Future Work

The thread-locality profiler provides a simple way for programmers to understand thread-locality and sharing behavior of their applications, and can ease adding Loci annotations to legacy code. It is easy to attach to a running program and results of profiling can be displayed in various ways. Profiling selected Java benchmarks shows that thread-locality rates tend to be very high.

One drawback of our profiling tool is major runtime overhead, as seen in Table 3.1. Some benchmarks take several hours to run. In future, we will attempt to improve the performance of the profiler.

Chapter 4

Measuring Loci's view of Thread-Locality

In the previous chapter, we described how we go about reporting thread-locality information as present at runtime. Now we describe how Loci's view of thread-locality for each allocated object can be recorded to be compared with the object's actual thread-locality, which facilitates evaluating Loci's precision. The reader may refer to Appendix B for implementation details and issues.

4.1 Introduction

Loci extends Java types with thread-locality information through annotations. Annotations are typically used at compile time, and are not retained in bytecode or available at runtime. However, to be able to measure thread-locality rate as inferred by Loci, we need to keep this information attached to each object instantiation. It is possible since Java 5.0 to use the reflection API to pass annotation data to runtime, but we could not use it for our purpose since Loci is built using the Checker Framework, which uses non-standard annotation formats, and does not yet extend the reflection API to support them. Another problem is that Loci can infer thread-locality of an object based on the annotation of its owner, even if the object does not have an explicit annotation. The inferred information is maintained only during compilation, which means that merely checking explicit annotations on object allocation statements is not enough to tell its thread-locality.

In this chapter, we describe the solution we implemented to be able to preserve Loci's type information and make it available at runtime. In summary, we extended Loci to allow storing thread-locality annotations for all object creations into bytecode, reading annotations from bytecode and replacing them with "allocation events" that provide a reference to a newly allocated object, along with its Loci-inferred thread-locality to be used by

external tools. After that, we modified our thread-locality profiler to monitor these events, and append actual thread-locality information to each allocated object, in addition to its Loci-inferred thread-locality.

4.2 Propagating Loci Annotations

Based on ownership rules, Loci temporarily adds annotations to code constructs lacking them when it performs type-checking during compilation. This means that merely looking at object allocation statements after compilation is not enough to determine an object's thread-locality. Exploiting thread-locality information can be done much easier if we store these temporary annotations, and make them available in source or bytecode beyond compilation.

4.2.1 Implicit Annotations in Loci

Loci operates on abstract syntax trees, as represented by the Java compiler API [6]. During type checking in Loci, each abstract syntax tree node is visited, and an associated annotated type is created. We wish to keep these annotated types beyond the type checking process. It is not straightforward to insert them into source or generated bytecode directly at this point, because the compiler API does not provide support for such operations. So we needed to store implicit annotation information externally. One tool that comes as a part of the Checkers Framework, named Annotation File Utilities [1], provides this functionality. It allows external storage of annotations according to the annotation file format specification [1].

4.3 Using Annotation Files to Instrument Bytecode

We use annotation files to instrument bytecode. We first insert annotation information into bytecode using a command from the Annotation File Utilities. We do so because it is easier to instrument if annotations are present in the bytecode, rather than an external file. This will become clear when we present the instrumentation tool. The annotation file utilities provides two commands to insert annotations to source code or byte code from an annotation file. We use the latter to insert annotations to bytecode.

Annotations are stored in bytecode as code attributes. If an attribute is unknown to the virtual machine, it is simply ignored. We mentioned before that the Checker framework supports annotating certain constructs that are not supported by standard Java. This causes source code that uses Checker-compatible annotations to be incompatible with standard Java compilers. For this reason, the Checker framework has its own compiler. However, the

bytecode generated from the compiler is compatible with any JVM, since annotations are inserted as non-standard attributes, which are ignored at runtime.

Once annotations are inserted into bytecode, they can be read and interpreted by a BCI framework as non-standard attributes. We used ASM [3] to read thread-locality information from attributes, and insert an allocation event after each object allocation instruction.

4.4 An Interface to Use Instrumented Bytecode

To ease generating instrumented bytecode, we made the instrumentation code an extentsion to Loci. If a programmer desires to compile directly into instrumented code, he or she can request this by passing an option to the Loci plug-in during compilation. Client code can capture allocated objects along with their thread-locality by providing an implementation to `LociSampler` interface, a part of Loci, that contains the following definition:

```
public void newobj(Object obj, boolean isShared);
```

Calls to this method are inserted after each object allocation in bytecode, the value of `isShared` depends on the annotation of the object.

We used this interface in combination with our thread-locality profiler to track both Loci-thread-locality and runtime-thread-locality for each object. We provided a native implementation to `LociSampler.newobj(Object obj, boolean isShared)` in the profiler's source code. Whenever an object is allocated in the instrumented code, our native implementation is called. This code appends a structure to each object that, in addition to holding Loci-thread-locality, enables tracking its thread-locality information. At the end of execution, we compare actual and Loci thread-locality. This way we can measure Loci's precision in capturing actual thread-locality.

4.5 Conclusion

The tool described in this chapter provides a simple interface for client code to detect and exploit Loci's thread-locality information associated with each allocated object. We used it to extend our profiler so we can record true thread-locality and Loci-thread-locality in one run.

Chapter 5

Evaluating Loci: Procedure and Results

The previous chapters presented the Loci system, and two tools we developed to facilitate its evaluation in terms of precision at capturing thread-locality. This chapter covers the process of adding Loci annotations to the Ray Tracer benchmark from Java Grande [5], and Xalan from DaCapo [15] benchmark suite. We describe how the selected benchmarks were annotated according to the latest Loci specification, and list the results we got in terms of annotation overhead, and the effectiveness of Loci's annotations in capturing thread-locality.

5.1 Introduction

Changes to the design of Loci and its underlying implementation made previously annotated code base incompatible with the current Loci specification, and so it was necessary to repeat the process of annotation. In this chapter, we describe the process of annotating two Java benchmarks: Ray Tracer from Java Grande [5] and Xalan from DaCapo [15] benchmark suite. We discuss some criteria and guidelines we set and followed during different stages of adding the annotations, mention some challenges we faced, due to implementation bugs or design decisions, how some design decisions were changed based on observations made during typing to ease the use of Loci, and finally show the results of comparing Loci's view of thread locality to actual thread-locality during various stages of the typing process, analyze the results and draw conclusions.

5.2 Annotation Considerations

There is no right way or wrong way to adding Loci annotations to a program. Loci can be used to check an entire application or selected parts, and the programmer is free to express his/her intentions regarding thread-locality of various Java constructs. There are, however, several factors to keep in mind. Sometimes, considering future uses of code is important. For instance, when annotating a library, it might be restrictive if some classes are annotated `@Local`, while client code needs to use instances of these classes in a shared context, without resorting to `ThreadLocal`, which normally degrades performance. On the other hand, if some classes are annotated `@Shared`, it can be the case that the client code would use them in a thread-local context, but they would still be treated as shared values, this, for example, reduces the effectiveness of thread-locality based optimizations. One may argue that these issues are solved if class-level annotations are not used, or if library classes are minimally annotated. It should be kept in mind that class-level annotations can greatly reduce the annotation overhead, which is an important factor in itself. It is possible to annotate every object instantiation with the desired thread-locality, which is expected to capture a high rate of thread-locality, but is extremely inefficient, and does not reflect the way Loci is meant to be used. Or, we could put many class level annotations, based on what is observed during a specific run of a program for a specific usage, which again is likely to capture a good rate of thread-locality, but would not reflect how code would be annotated to permit flexibility in future usage.

For the purpose of this study, the main concern is to capture as much thread-local objects present during runtime as possible, while minimizing annotation overhead. It is hard to objectively satisfy the previous considerations. When annotating code, it is crucial to achieve a reasonable balance between rate of captured thread-locality and annotation overhead.

5.3 The Ray Tracer Benchmark:

The Ray Tracer benchmark, part of the Java Grande benchmark suite is a multi-threaded ray tracing program of 1436 lines of code. It was previously annotated when evaluating both Loci 1.0 and Loci 2.0[1]. The annotated version of this benchmark from Loci 3.0 contained a total of 15 annotations, 3 `@Shared` and 12 `@Local`, and it did not need to be changed to obey the current Loci rules[2]. However, Loci inferred that a shared instance (as observed at runtime) of class `JGFRayTracerBench` was `@Local`. This reflected

[1] The reader may refer to [24] to see the impact of changing the design and implementation of Loci on the way this benchmark was annotated.

[2] Less annotations could be used, but it was desired to know if all classes can be annotated `@Local`.

a serious problem, since it violates the soundness design goal of Loci. The wrong inference occurred in the following code snippet, contained in method `JGFapplication()` from the class under discussion:

```
1 thobjects[i] = new Runnable() {
2   public void run() {
3     RayTracerRunner temp = new RayTracerRunner(j,width,height,br);
4     temp.run();
5   }
6 };
```

The `Runnable` instance created at the 3rd line of code is shared, because the class `Runnable` is annotated `@Shared`. Hence, All objects owned by this instance are also `@Shared`. `width` and `height` at the same line are fields of type `int` in class `JGFRayTracerBench`, which is local because it extends the `@Local` class `RayTracer`. Since these two fields are touched by the `Runnable` instance, they become shared, and the containing `JGFRayTracerBench` must be `@Shared` not `@Local`.

Loci failed to detect this violation, because when it performs its type checks, it does not process primitives, which makes it miss such cases. This issue is now fixed in Loci 3.0.

5.3.1 Thread-Locality Comparison Results

The thread-locality profiler detected a total of 36955 object allocations, only 6 of which were shared across threads. The benchmark creates a number of worker threads (we specified two when profiling), and splits data to be rendered among them. This justifies the high rate of thread-locality, because there isn't much shared data among worker threads. Loci, on the other hand, detected 8 shared objects, the other two were arrays of types `Runnable` and `Thread`. The reason is that Loci does not allow arrays to have thread-locality different from the contained elements, and both `Runnable` and `Thread` classes are annotated `@Shared`.

5.3.2 Conclusion

For this benchmark, Loci detected a thread-locality rate of 99.9%, almost identical to the actual thread-locality rate. The fact that Loci does not allow arrays to have thread-locality different from the contained elements made it miss only two local objects out of 36947.

Analyzing this small benchmark showed that Loci could be used to detect a high rate of thread-locality using a very small number of annotations. It also demonstrated an example of how Loci's conservative analysis might decrease the rate of detected thread-locality, and uncovered a bug in Loci which was immediately fixed.

The small size of the benchmark eased analyzing the results of profiling and annotating it. The downside is that it does not give an accurate view of annotation-overhead when Loci is used with larger code base, that has more dependencies among classes, and is thus expected to require more elaboration during adding annotations.

5.4 The DaCapo Xalan Benchmark

Xalan-Java is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. We annotated classes from `xml` package loaded during running Xalan benchmark from the DaCapo [15] benchmark suite, as well as the test harness from Dacapo loaded when running this benchmark. Annotating this benchmark required more elaboration compared to the Ray Tracer, mainly due to the large code base, and frequent need of code refactoring.

To see how different annotations strategies affect the rate of detected thread-locality and annotation overhead, we decided to add annotations iteratively and see how our decisions in each iteration affect annotation-overhead and accuracy of capturing runtime thread-locality.

5.5 First Annotation Iteration

At the first iteration, we added the minimal amount of annotation needed to eliminate Loci warnings concerning thread-safety violations. We were careful to consider future uses of classes, and avoid using class-level annotations, while still trying to converge to the rate of thread-locality present during runtime. We used the following self-set criteria to guide the annotation process:

- We tried to converge on the rate of thread-locality present at runtime.
- We used the results of profiling thread-locality of the benchmark to guide annotating it. We decided to annotate classes of objects that appear to be shared as `@Shared`. It might not be the case that every instance of such class will be `@Shared`, but we wanted to test the assumption on which Loci was built, that thread-locality is a property of class by design, that is, if we annotate a class `@Shared`, then all/most instances it creates are also shared in practice.
- Future uses of classes were considered. Except for the previous criteria, we avoided using class-level annotations on classes exposed to client code in code libraries, while we used them freely for classes used to internally support the operations of a library.

- Many times, the documentation of a class facilitated adding class-level annotations. It is often mentioned if a certain algorithm or data structure is thread-safe, or if it is designed specifically to support multi-threading.

- When possible, we avoided using the `ThreadLocal` API, because it degrades performance. Some libraries had their own implementation of certain data structures which are available in the Java library, for the sake of speed. When optimization was achieved by using non-thread-safe code, it made little sense to use the `ThreadLocal` API.

We needed a total of 433 annotations to annotate 85,000 lines of code in 712 source files. 213 of them `@Shared`, 79 `@Local` and 141 `@ThreadSafe`. We have many `@Shared` annotations, because we tried to use `@Local` on class level as often as possible, which meant we needed to explicitly mark many constructs `@Shared` frequently, in order to avoid using `@Shared` on class level. The annotation overhead is 4 annotations per 1000 lines of code, which is relatively small.

In the following sections, we list some issues, observations and limitations we came across during the annotation process in the first iteration, and how we handled them.

5.5.1 Refactoring Collections

The benchmark uses collections as they were implemented before the introduction of Java Generics, which tend to confuse Loci regarding annotation inference. These had to be be refactored to use generic definition. Collections of Strings were frequently used, which is natural as Xalan is an XML transformation library. Due to a limitation in Loci, contained strings can only be `@ThreadSafe`, even though in practice they are local in the majority of cases, because they are immutable structures. The refactoring overhead was major, we did refactor the most part, but eventually neglected warnings issued by Loci concerning String types, and decided to mark String object occurrences as `@Local` when encountered on the bytecode level instead. We later noticed that there were not as many string objects as we expected, because the compiler we used either allocated them as character constants on a thread's stack, or transformed them into `StringBuilder` objects, which we considered `@Shared`, because the documentation of the class `StringBuilder` mentioned its instances are not thread-safe.

5.5.2 Changing Treatment of Primitive Arrays

The benchmark contains a large amount of arrays, because it implements its own data structures which are based on variable-size arrays (arrays are created with an initial size and extended by certain chunks when needed). The

purpose of these custom data structures is to improve performance, through using non-thread safe code and implementing other optimizations. An example of such data structure is `FastStringBuffer` [4]. The documentation of this class, and some others explicitly states that their implementation is not thread-safe. Such classes were troublesome when they contained arrays of primitive types; primitives are considered `@Local` in Loci, and they force containing arrays to be considered `@Local` as well. When these arrays are members of "flexible" or `@Shared` classes, Loci issues errors declaring they should be `@Shared`. This however, makes little sense, since in practice, the primitives will not be shared, having to annotate them `@Shared` is not accurate, and using `ThreadLocal` API to allow each thread retrieve a copy of this data causes unnecessary performance overhead, and defies the purpose of implementing these custom data structures. To overcome the previous issue, we changed the treatment of primitive arrays to make them hold the annotations of their `@Owner`, instead of the annotation of their contents as the case of non-primitive arrays.

5.5.3 Miscalculating Thread-Locality of `for-each` Loops

Consider the following code snippet:

```
for (@Shared Object bm : line.getArgList())
```

Neither `line`, nor `getArgList()` have explicit annotations, and so `bm` should be `@Shared`, however, Loci still issued warnings for such code. The only way to fix it was to replace the `for-each` loop with an ordinary one. This issue will be fixed in future Loci releases.

5.5.4 First Iteration Results

Profiling objects allocated in annotated code parts reports a thread-locality rate of 99.9%. There were 283 shared objects out of 259134. Loci-detected thread-locality rate was 38.6%. However, in terms of memory, 13.9% of memory was found shared by Loci, opposed to 0.013% shared memory detected by profiling.

5.6 Second Annotation Iteration

We wanted to know why the thread-locality rate in terms of number of objects was low. We expected this was due to Loci's inability to infer annotations for all object creation statements from the minimal set of annotations. This lead to the second iteration of the annotation process. We made Loci output the source code locations of object creation statements, that hold the annotation `@Owner`, meaning Loci could not bind them to either `@Shared` or `@Local` during type checking. 6 more class-level annotations were added,

Iteration	# Annotations	Locality Rate (# Objects)	Shared Memory
1	433	38.6%	13.9%
2	439	55.1%	10.45%
3	451	83.5%	2.8%

Table 5.1: Results of annotating Xalan at different phases. During the first phase, minimal annotations were added to eliminate Loci's warnings. In the second phase, objects with `@Owner` annotation were resolved. Finally at phase 3, thread-locality mismatches were analysed and some of them were resolved.

and they were enough to resolve all object creation statements. This time, thread-locality rate in terms of number of objects was 55.1%, while shared memory rate was 10.45%. It was remarkable that a few more annotations made a significant difference.

5.7 Third Annotation Iteration

During this iteration, we combined information from runtime and static analysis for each object. We wanted to see what are the classes of objects for which loci's inferred thread-locality does not match actual thread-locality. There were mismatches for 42296 instances of class `ELemContext`. Which was annotated `@Shared` on the class level. Changing this annotation to `@Local` raised thread-locality rate to 76%. Performing such fixes, and adding more annotations (in total 12 annotations) raised the rate to 83.5%, with shared memory of only 2.8%. Other objects for which mismatches occurred were mainly input/output streams, readers/writers, and files. There were 5137 mismatched `String` instances, and very notably, 13 `ThreadLocal` instances as well. Also, we did not detect objects for which Loci infers `@Local` while actual thread-locality is `@Shared`, which supports the correctness of Loci's design and implementation.

Table 5.1 shows Loci-detected thread-locality rates at different annotation phases.

5.8 Recommended Annotation Procedure

Based on our experience on employing runtime information during the process of adding annotations, we recommend users of Loci to follow a procedure similar to the one we followed when annotating their old programs, which can be summarized in the following steps:

- Profile the application, if any class has shared instances, never attempt to annotate it or any of its instances `@Local`. Furthermore, you can

start by annotating the class `@Shared`.

- Resolve all Loci warnings. You can use profiling traces to get an insight on how to fix them. It is important to never annotate something that appears to be shared as `@Local`.
- After resolving all warnings, run the application, if the detected rate of thread-locality is close to ideal, or satisfies the needs of your application, you can stop at this point.
- You can proceed by checking which object-creation statements still have an `@Owner` annotation, meaning Loci could not bind it to either `@Shared` or `@Local`, and attempt to annotate them explcitely, or add class-level annotations that enables Loci to infer `@Shared` or `@Local` for them.
- If the rate of thrad-locality is still low, you can profile the application again to see where mismatches of thread-locality occur, and then attempt to resolve them.

5.9 Conclusions

According to the results of our evaluation procedure, Loci does well in expressing thread-locality. We were able to express more than 99.9% of thread-locality correctly using Loci annotations for the Ray Tracer benchmark. For Xalan, the rate was 83.5% in terms of number of objects, and 97.1% in terms of memory.

During the process of annotating benchmarks, results of dynamic detection of thread-locality using our profiler helped reduce the time needed to annotate legacy code, and guided the annotation process. It is worth mentioning that annotations on the selected code parts enable tracking 72.7% of total memory used by Xalan, and thread-locality rates we report are related to this tracked area. Untracked area has unknown thread-locality.

We uncovered some bugs of Loci during this evaluation process, that were either reported or fixed. And a design decision was changed to make Loci better express actual thread-locality and easier to use.

Chapter 6

Conclusion and Future Work

The previous chapters presented the Loci system, and the framework we built to evaluate the accuracy of its annotations in reflecting thread-locality present at runtime. We covered the process of adding Loci annotations to selected benchmarks, and the results of evaluation were presented and discussed. In this final chapter, we conclude this study and suggest future work and improvements to Loci.

6.1 Results

Results obtained from this study show that Loci can be used to capture a substantial amount of thread-locality with a relatively low annotation-overhead. We annotated two benchmarks, and were able to get thread-locality rates of 99.9% and 83.5%, in terms of number of objects. These results show that Loci's design is working. Class-level annotations in many cases were precise enough to capture correct thread-locality for most instances, which supports the assumption of thread-locality being a property of a class by design. Some implementation bugs were uncovered, and they are either reported or fixed. The bugs do not greatly affect the usability of Loci, and can be worked around by the user, as explained in Chapter 5. During this study a design decision was changed to make Loci easier to use, and more expressive, as discussed in Section 5.6.2.

6.2 Future Work

The number of benchmarks annotated was small, this was mainly due to lack of time. Annotating more benchmarks would better support our results. In future releases of our tools, we will attempt to improve the performance on the thread-locality profiler described in Chapter 2, port both Loci and the thread-locality instrumenter extension to a newer and more stable version of

the Checker Framework, and fix bugs addressed in Appendix B.4. We will also continue working with Alberto Ros to devise a technique that enables utilizing Loci's type information by the cache coherence protocol proposed in [20].

Appendix A

The Thread-locality Profiler

The thread-locality profiler is a tool we developed to monitor and measure thread-locality rate of Java applications at runtime. Here we describe the underlying framework, implementation details and how to use this tool.

A.1 JVMTI and Profiling Agents

We built our profiling tool as a JVMTI[2] pluggable profiling agent. JVMTI is a programming interface used by development and monitoring tools. It provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine. JVMTI is a two-way interface. A client of JVMTI, called an agent, can be notified of interesting occurrences through events. JVMTI can query and control the application through many functions, either in response to events or independent of them. An agent is usually deployed as a native dynamic library[3], and may be started at JVM startup by specifying the agent library name using a command line option.

This framework was chosen because it supports our implementations goals due to the following:

- It is supported by a number of Java virtual machines and comes with both OpenJDK and the standard JDK.
- It is mature, well documented and relatively easy to use, which facilitates maintenance and future extension.
- The resulting tool is easy to distribute and use.

[2]The Java Virtual Machine Tool Interface [10]

[3]For example, on Windows, an agent library is a "Dynamic Linked Library" (DLL). On the Solaris, an agent library is a shared object (.so file).

- JVMTI is a native interface, which gives flexibility in accessing low-level features.

The drawback of choosing this framework was major runtime overhead, as shown in Table 3.1. In the following section, we describe the design and implementation of our tool as a pluggable profiling agent on top of JVMTI framework.

Our implementation was guided by the heap-tracker demo[4], as our agent's basic operations are similar.

A.2 The Life-Cycle of an Agent

An agent passes through a number of phases triggered by events as follows[5]:

1. Startup phase: Triggered by the JVM loading the agent library. An agent must export a start-up function, that would normally initialize the agent by requesting capabilities for monitoring events of interest, enabling event generation and providing callback functions to be invoked when an event of interest occurs. This phase ends when the start-up function returns.

2. Live Phase: Triggered by VM Init event, which indicated that the virtual machine is fully initialized. During this phase, the agent is free to call all JVMTI and JNI functions. Most events will be captured and processed during this phase. Note that the JVM in phases between returning from the start-up function and sending VMInit is not yet in a stable state, and so the agent may not call all JVMTI and JNI functions.

3. Shutdown Phase: This optional phase is triggered by a platform-specific agent unload event. The agent may perform any finalization and housekeeping routines during this phase.

Understanding the general life-cycle of an agent provides basis for understanding how the thread-locality agent works.

A.3 Events Monitored by the Thread-Locality Profiler

The following is a list of all monitored events, reasons for monitoring them and how they are handled.

[4]The source code of this demo can be found with any installation of Oracle JDK higher than 5.0 or OpenJDK.

[5]There are actually more phases, but the most important and relevant ones are highlighted here, for a complete list of phases, see [10].

- Class prepare: In order to generate field access and modification events, a watch must be set on each field we wish to monitor. This is done after 'class prepare' event is sent. Class fields are scanned when a class is about to be loaded, and watches are set on some of them. We later describe the criteria for deciding whether to watch a field.

- Thread start: When a thread touches an object, we need a way to tell if it was the same thread that touched it before or a new one. So at this point, we append additional information to the thread to be able to identify it.

- Field access and modification: When an instance field is accessed or modified, i.e. a field attached to an object not to a class, if the object containing it is being touched for the first time, a structure is appended to it that will contain thread-locality information collected later. we then run a routine that checks if the object owning this thread is being touched by a thread different from the one that created it or accessed it before. If this was the case, the object is marked shared.

- Object free: We wish to distinguish objects marked shared only because of being touched by the garbage collector. Objects that are marked thread-local until the occurrence of this event, are marked as gc-shared. If the finalizer thread operates on objects before they are freed, this is reported as well.

  During the shutdown phase, information about shared objects, such as their class, the sequence of thread accesses are written to disk as binary files, and a summary of the profiling process is printed.

A.4 Implementation Issues

In this section, we highlight issues that required special attention, and justify some implementation decisions.

A.4.1 Appending Thread-Locality Information to Objects

We now explain how we extend each object with thread-locality information. JVMTI allows values, called tags to be associated to objects. Tags can be used to simply mark an object or to store a pointer to more detailed information. We associate a structure to each object to identify it, and hold its thread-locality and identifiers of thread(s) that accessed it. The tag value can be used to retrieve the tagged object, along with its associated structure at any point of runtime, as long as the object is not freed.

A.4.2 Reducing Memory and Runtime Overhead

Thread-locality information associated with shared objects is written to persistent storage during profiling in binary format. It is interesting to check the class of a shared object and the sequence of threads that accessed it, however, it is expensive to keep them in memory throughout execution[8]. Thus, thread-locality structures associated to shared objects are tracked and freed as soon as an object is freed. The written information are parsed offline using a separate program, to reduce runtime overhead and save memory during execution.

Watching fields for access and modification events is expensive as well. We do not add watches to fields added by the Java compiler (referred to as synthetic fields), because we want to detect thread-locality as can be inferred by looking at the source code, which is enough for evaluating Loci[9]. Static fields are not watched either, because they are not associated to objects instances, but to the single object representing their class, which will always be shared, and is marked shared upon allocation. Thread objects are also marked shared upon allocation.

Our decision not to monitor method invocations on an object also improved runtime by two to three orders of magnitude.

A.4.3 Managing Concurrency

Even though the agent code itself is apparently not concurrent, the JVM is heavily concurrent, and so are most benchmarks we profiled. This means that in many times, events we monitor occur concurrently. A thread-locality structure associated with an object is shared among all threads accessing it, and data it holds is prone to races. Operations of creating, reading and writing thread-locality structures are protected with a locking mechanism provided by JVMTI.

A.4.4 Threads not Visible in Source Code

Other than threads created and managed in source code, the virtual machine runs other threads under the hood to support runtime operations, and JVMTI interface might hide their operation from the agent. Some threads that our agent detected and were not present in the source code are the garbage collector thread, the finalizer thread, which is used to to call finalize methods on objects, and the destroyVM thread, used to unload the virtual machine near shutdown. There might be other threads, based on the virtual machine implementation.

[8] Keep in mind that millions of object might be created during execution.

[9] In all the benchmarks we ran, synthetic fields did not occur.

A.5 Viewing Profiling Results

When program execution is finished, a summary of profiling results is displayed, and thread access information of shared objects are written to disk in binary format. To parse the binary data, a simple utility program that comes with the agent may be used. For details on using the agent and the utility, please refer to `java.net/projects/loci`.

The following is a sample summary generated when running the agent on Xalan benchmark, a part of the DaCapo benchmark suit. Please refer to Appendix C to view other summaries generated for benchmarks we profiled in this study.

```
Total number of objects touched: 1362046

Number of local objects: 1359254 (99.795%)
Number of shared objects: 2792 (0.204986%)
Number of objects marked shared due to
garbage collection: 1351758 (99.2447%)
Number of objects marked shared due to
finalization:32 (0.00234941%)

Total touched objects memory occupied in bytes: 57759008
Shared memory of touched objects occupied in bytes:
115360 (0.199726%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
13456: Thread-0
13457: Thread-1

Runtime: 0h 17m 12secs
```

Appendix B

The Thread-locality Instrumenter

The thread-locality instrumenter, presented in Chapter Four is an extension to Loci that allows storing thread-locality annotations for all object creations into bytecode as "allocation events". In this appendix, we describe an important part of its supporting framework, the Annotation File utilities, then we show implementation details of handling implicit annotations in Loci. We show after that how annotations are stored in bytecode as attributes, and how allocation events are inserted to correspond to each annotation. Finally we list implementation issues and challanges, and how we handled them.

B.1 The Annotation File Utilities

Sometimes, it is convenient to store annotations outside both source code and bytecode. For example, if the source code is not available, annotation can be specified by writing them in an external file, such file is called a Java Annotation Index File. The complete specifications of the format of this file are available online [1]. We now present a subset of this specification that is of interest to our purpose.

Annotations on object instantiations using `new` keyword are identified by their relative index from the start of the method containing the `new`, referred to as the "offset". Note that this also applies to direct assignments to class fields, because these actually take place inside each declared constructor in the bytecode. It is important to understand bytecode representation to be able to generate a correct annotation file. For example, method signatures in an annotation file should be in VM format [7]. In addition to that, some methods may not appear in the source but the bytecode (e.g `clinit`, which will be explained soon), and need to be considered properly.

Figure B.1 shows a class, with object instantiations using `new` in various

```
1  package test;
2  import loci.quals.*;
3
4  @Local public class JaifTest {
5    int foo;
6    @Local Object o1 = new Object();
7    @Shared Object o2;
8
9    @Shared static Object o3 = new Object();
10
11   public JaifTest() {
12     o2 = new @Shared Object();
13   }
14
15   public JaifTest(int foo) {
16     this.foo = foo;
17   }
18
19   public void doSomething() {
20     Object o4 = new Object();
21     @Shared Object o5 = new Object();
22   }
23 }
```

Figure B.1: Annotated class, with object instantiations in various code locations

code locations, Figure B.2 shows the corresponding annotations file generated using Loci. Note how method signatures are converted to VM format. Creating the instance assigned to `o1` at line 6 actually takes place in both `init()` and `init(I)` methods, each corresponding to a constructor in source. Also notice the presence of `<clinit>()` method, which initializes the class. The contained `new *0:@Shared` corresponds to instantiation at line 9 in the source code, because the instance created there is assigned to a static field.

B.2 Extending Loci to Store Implicit Annotations

Loci operates on abstract syntax trees, as represented by the Java compiler API [6]. During type checking in Loci, each abstract syntax tree node is visited, and an associated annotated type is created. We wish to keep these annotated types beyond the type checking process.

A hook was added to the method in Loci that implicitly annotates syntax tree nodes. When a tree of kind `NEW_CLASS` or `NEW_ARRAY` is visited, its annotation is fetched, along with its index, containing method and class. After that, a string representing the annotated node is constructed according

```
1  package loci.quals:
2  annotation @Shared: @java.lang.annotation.Retention(value=RUNTIME)
3
4  package loci.quals:
5  annotation @Local: @java.lang.annotation.Retention(value=RUNTIME)
6
7  package test:
8  class JaifTest:
9  method doSomething()V:
10 new *0:@Local
11
12 package test:
13 class JaifTest:
14 method doSomething()V:
15 new *1:@Shared
16
17 package test:
18 class JaifTest:
19 method <init>(I)V:
20 new *0:@Local
21
22 package test:
23 class JaifTest:
24 method <init>()V:
25 new *0:@Local
26
27 package test:
28 class JaifTest:
29 method <init>()V:
30 new *1:@Shared
31
32 package test:
33 class JaifTest:
34 method <clinit>()V:
35 new *0:@Shared
```

Figure B.2: Annotated class, with object instantiations in various code locations

to the annotation file specification, and written to an index file, named the same as the containing class name. Loci has many annotation types, instances may only be annotated `@Shared` or `@Local`. Loci will try to infer either of these if the instantiation tree is not annotated, if it could not, then the annotation remains `@Owner`. At runtime, everything is either local or shared, we assume that what remains `@Owner` is `@Shared` at runtime, to conserve soundness of the system.

Constructor methods need to be handled in a slightly different way, when

direct assignments of new object instances to class fields are present [1]. These should be added to the beginning of every declared constructor, in the order of their appearance in the source code. If no constructors are declared, they should be added to the default constructor[2]. Hence, indices of any new trees within a constructor should be shifted, and the shift distance is the number of direct field assignments. Thus, strings representing new tree nodes in constructors and field assignments cannot be fully constructed and written until all other fields are visited. Construction of such strings is completed after each class visit is ended, and at that point they are written to the corresponding annotation file.

A question in this case is, in what order are the direct assignment trees added to each constructor? The Java compiler specification does not enforce any order, but the intuitive answer to this question is to add them in the order they appear in the source code. We observed that this was the case for the compiler we used, and it makes sense to add them in this way, any other way will require scanning the source code multiple times. If the programmer needs to guarantee correct ordering, then all assignment of instances to class fields should happen in the body of constructors, or a single static initializer.

Similarly, direct assignments to static fields should be added to `clinit` method, which performs initialization for the class.

At this point, all objects allocated using `new` have known explicit annotations, stored in an external annotation file. We have described in Section 4.3 how these files are utilized to instrument bytecode with object allocation events that carry thread-locality information.

In the following section, we describe the structure of code attributes that represent non-standard annotations of the Checker framework in Java bytecode.

B.3 Checker-Supported Annotations in Bytecode

Standard Java annotations are stored in bytecode as variable length attributes in the attributes table of a `ClassFile`, `field_info`, or `method_info` structures[3]. Checker-supported annotations are stored in bytecode the same way, and their structure is similar to standard Java annotations. Both standard and Checker-supported annotations can be runtime visible or invisible. When they are visible, the JVM must make these annotations available so they can be returned by the appropriate reflective APIs, otherwise, it ignores them. Loci annotations are runtime-visible, even though there is currently

[1] In the context of this description, we will refer to them as direct assignments from now on.

[2] In the Java programming language, if a constructor is not declared in a class, a default constructor that takes no arguments is declared by the compiler.

[3] Consult The 'class' File Format document [12] for detailed description of these structures and Java class file format

no reflection support for Checker annotations as this might change in future, and we wish to ease forward-compatibility.

We now describe the structure of Checker-annotations attribute as stored in bytecode, when the target of an annotation is an object/array instantiation and the annotation is runtime visible, as described in the type annotations specification of November 25, 2009 [13]. This attribute is named `RuntimeVisibleTypeAnnotati- ons`.

RuntimeVisibleTypeAnnotations has the following structure:

```
RuntimeVisibleTypeAnnotations_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u2 num_annotations;
   extended_annotation annotations[num_annotations];
}
```

The items of RuntimeVisibleTypeAnnotations structure are as follows:

- `attribute_name_index`: This value of this item is an index to the constant_pool [12] table. The entry at that index must represent the string `"RuntimeVisibleTypeAnnotations"`. With this item, this attribute can be identified when processing bytecode.
- `attribute_length`: Indicates the length of the attribute, excluding the initial six bytes [4].
- `num_annotations`: this item gives the number of runtime-visible annotations represented by the structure.
- `extended_annotation`: Each value of the annotations table represents a single runtime visible annotation on a program element. The number of annotations is equal to the value of `num_annotations`, the previous item.

When the target of an annotation is an object or array creation instruction, it has the following structure:

```
extended_annotation {
    u2 type_index;
    u1 target_type;
    target_info {
        u2 offset;
    };
}
```

Items of `extended_annotation` are as follows:

[4] u2 means that the length of the item is two bytes, u4 is four bytes, these are the initial 6 bytes occupied by the `attribute_name_index` and `attribute_length`.

- `type_index`: an index to the constant_pool. The entry at that index is a String representing the name of the annotation, in the case of Loci, it is either `Lloci/quals/Local;` for `@Local` or `Lloci/quals/Shared;` for `@Shared`.

- `target_type`: Denotes the type of program element that the annotation targets, for object or array instantiation, it is `0x04` or `0x05`, respectively.

- `target_info`: A structure that varies by target type, and contains enough information to uniquely identify the target of a given annotation. For object/array instantiation, it is an offset value, representing the distance in byte from the start of code in a method. Due to a bug in the Annotation file utilities, annnotations inserted using the insert-annotations command will always be `-1`. This affected the way we perform instrumentation; it did not, however, affect the correctness or overhead of instrumentation.

B.4 From Bytecode Attributes to Runtime Events

Once we have annotation information in the form of attributes inside a bytecode, we need to perform the following procedure for each class we intend to instrument:

1. Go over all methods inside a class, check if they have a `RuntimeVisibleTypeAnnotations` attribute.

2. For each attribute, check the number of annotations, and go over them all.

3. If the annotation target is object/array creation, check their annotation value.

4. For each object/array creation encountered, insert a method call that takes a reference to the object, and boolean value true when the object is `@Shared`, false when the object is `@Local`. This method call is implemented by client code as desired, to exploit Loci-inferred thread-locality information.

To implement this procedure, ASM [3] bytecode instrumentation framework was used.

In the following sections, we discuss interesting implementation details, some issues and bugs out of our control, and how we handled them.

B.4.1 Wrong Offset Values

Normally, an object allocation instruction is associated with an annotation using the offset value. However, insert-annotations command always incorrectly assigns -1 as an offset value. To overcome this problem, We make sure that there are as many annotations as object allocation instructions. When a `RuntimeVisibleTypeAnnotations` property is interpreted, an ordered list of boolean values corresponding to thread-locality is created. Whenever an object allocation instruction is encountered, an element from the list is consumed. This way we keep correspondence between annotations and instructions.

B.4.2 Allocations Not Present in Source

A Java complier might allocate objects in bytecode that are not present in the original source code. This is a compiler-specific issue. For example, when Strings are created without a `new` operator in source, the compiler we use allocates them as `StringBuilder` instances, or, when assert keyword is present, an `AssertionError` instance is allocated. This rises a number of issues, the most severe is that handling such objects is compiler-specific, which makes our tool less portable across different compilers. Another issue was to come up with a Loci and runtime consistent thread-locality information for these instances. For the previous cases, it was simple, `StringBuilder` objects are not thread-safe, and thus should always be considered shared. For `AssertionError`, Loci's default for all Error and Exception subtypes is `@Local`, so we consider them thread-local. A safe approach for other cases is to consider them `@Shared`, to preserve soundness.

B.4.3 Skipping Annotations on Local Variables

When an annotation referring to a target other than object/array allocation is encountered, its bytes are skipped. As mentioned, the number of bytes in an `extended_annotation` item depends on the target type, which defines the size of `target_info`. In the case of a `local_var` target type, which denotes an annotation on a local variable decleration, the size of `target_info` is variable. For an unknown reason, the Checker compiler version we use generates wrong values for the length of this item, and it is not possible to tell the number of bytes to skip in many cases.[5] The Checker framework developers do not support versions we use of the compiler or the Checker framework, which we are obliged to use since Loci works only with those versions.[6] We expect this issue to be solved when Loci is ported to a newer

[5] We came up with an ad-hoc formula to correctly calculate the number of bytes to skip, which works many times, and notifies the user when it does not.

[6] It is not straight-forward to port Loci to the newest version because specifications and implementation of the framework change in an extremely backward-incompatible fashion.

version of the Checker framework. Until then, when skipping the bytes of a `local_var` target fails, the user is notified about the error location, and must remove the corresponding annotation from the source to solve it. If the user does not do so, the method is not instrumented.

Appendix C

Results of Profiling Thread-locality of Selected Java Benchmarks

C.1 Note:

When the number of threads exceeds 16, thread details are truncated. To view complete summaries, please visit `java.net/projects/loci`.

C.2 Benchmarks from the Dacapo Suite

C.2.1 Avrora

```
Total number of objects touched: 1535928

Number of local objects: 1301919 (84.7643%)
Number of shared objects: 234009 (15.2357%)
Number of objects marked shared due to garbage collection: 1976
    (0.128652%)
Number of objects marked shared due to finalization:25
    (0.00162768%)

Total touched objects memory occupied in bytes: 46229928
Shared memory of touched objects occupied in bytes: 7483200
    (16.1869%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
14803: node-0
20262: node-3
22590: node-5
```

```
17934: node-1
19098: node-2
21426: node-4

Runtime: 0h 40m 26secs
```

C.2.2 Batik

```
About to transcode 3 SVG file(s)

Converting mapWaadt.svg to /home/nosheenzaza/thesis/
    ThreadLocality_JVMAgent/java_test_code/classes/./scratch/
    mapWaadt.png ... ... success
Converting mapSpain.svg to /home/nosheenzaza/thesis/
    ThreadLocality_JVMAgent/java_test_code/classes/./scratch/
    mapSpain.png ... ... success
Converting sydney.svg to /home/nosheenzaza/thesis/
    ThreadLocality_JVMAgent/java_test_code/classes/./scratch/
    sydney.png ... ... success

Total number of objects touched: 550540

Number of local objects: 550470 (99.9873%)
Number of shared objects: 70 (0.0127148%)
Number of objects marked shared due to garbage collection: 2036
    (0.369819%)
Number of objects marked shared due to finalization:37
    (0.00672067%)

Total touched objects memory occupied in bytes: 22003416
Shared memory of touched objects occupied in bytes: 4256
    (0.0193425%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
6271: Thread-0
131740: Java2D Disposer
315993: Thread-2
368336: Thread-3
544792: Thread-4
545279: Thread-5
545453: Thread-6
64222: Thread-1

Runtime: 0h 0m 55secs
```

C.2.3 H2

```
Using scaled threading model. 64 processors detected, 64 threads
    used to drive the workload, in a possible range of [1,4000]
....
Completed 4000 transactions
  Stock level ............. 155 ( 3.9%)
  Order status by name .... 108 ( 2.7%)
  Order status by ID ...... 65 ( 1.6%)
  Payment by name ......... 1050 (26.2%)
  Payment by ID ........... 661 (16.5%)
  Delivery schedule ....... 167 ( 4.2%)
  New order ............... 1774 (44.4%)
  New order rollback ...... 20 ( 0.5%)
Resetting database to initial state

Total number of objects touched: 33638018

Number of local objects: 32945620 (97.9416%)
Number of shared objects: 692398 (2.05838%)
Number of objects marked shared due to garbage collection:
    29324416 (87.1764%)
Number of objects marked shared due to finalization:31 (9.21576e
    -05%)

Total touched objects memory occupied in bytes: 1213538576
Shared memory of touched objects occupied in bytes: 21885840
    (1.80347%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
4934: Thread-1
30673564: OE_Thread:0
30673565: OE_Thread:1
.
.
.
30673620: OE_Thread:56
30673622: OE_Thread:58
30673621: OE_Thread:57
30673623: OE_Thread:59
30673624: OE_Thread:60
30673626: OE_Thread:62
30673625: OE_Thread:61
30673627: OE_Thread:63
4751: Thread-0

Runtime: 0h 41m 53secs
```

C.2.4 PMD

```
Total number of objects touched: 4452941

Number of local objects: 4451240 (99.9618%)
Number of shared objects: 1701 (0.0381995%)
Number of objects marked shared due to garbage collection: 8994
    (0.201979%)
Number of objects marked shared due to finalization:43
    (0.000965654%)

Total touched objects memory occupied in bytes: 150146744
Shared memory of touched objects occupied in bytes: 69232
    (0.0461096%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
9546: PmdThread 1
9560: PmdThread 2
9578: PmdThread 3
9606: PmdThread 4
.
.
.
13998: PmdThread 55
14097: PmdThread 56
14174: PmdThread 57
14269: PmdThread 58
14364: PmdThread 59
14451: PmdThread 60
14550: PmdThread 61
14620: PmdThread 62
14677: PmdThread 63
14724: PmdThread 64
2697: Thread-0

Runtime: 0h 7m 30secs
```

C.2.5 Eclipse

```
Unzip workspace
Initialize workspace ...................
Index workspace ......................
Build workspace
Search .. 4,207 references for default constructor in workspace
      .. 1,957 references for method 'equals' in workspace
Type hierarchy tests
AST tests
Completion tests .........
Format tests ..............
Model tests ................
Delete workspace

Total number of objects touched: 13825650

Number of local objects: 13492816 (97.5926%)
Number of shared objects: 332834 (2.40737%)
Number of objects marked shared due to garbage collection:
    13367840 (96.6887%)
Number of objects marked shared due to finalization:4243
    (0.0306893%)

Total touched objects memory occupied in bytes: 684662448
Shared memory of touched objects occupied in bytes: 19919864
    (2.90944%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
17667: State Saver
17880: Framework Event Dispatcher
17947: Refresh Packages
18418: Start Level Event Dispatcher
29368: Refresh Packages
50548: Worker-0
52243: Java indexing
52382: Worker-1
58603: process reaper
58675: Output Stream Monitor
58677: Output Stream Monitor
58683: Input Stream Monitor
58689: Process monitor
59379: process reaper
59393: Output Stream Monitor
59394: Input Stream Monitor
59397: Output Stream Monitor
59392: Process monitor
```

```
67506: Worker-2
146358: Worker-3
146362: Worker-4
3921175: Compiler Source File Reader
3921176: Compiler Source File Reader
.
.
.
6199279: Compiler Processing Task
6228219: Compiler Source File Reader
6228227: Compiler Source File Reader
6228228: Compiler Source File Reader
6228229: Compiler Source File Reader
6228230: Compiler Source File Reader
6228231: Compiler Source File Reader
6228232: Compiler Source File Reader
6228234: Compiler Source File Reader
6234311: Compiler Processing Task
6280226: Worker-6
12669272: Worker-7
12674779: Worker-8

Runtime: 0h 29m 8secs
```

C.2.6 Tomcat

```
Using scaled threading model. 64 processors detected, 64 threads
    used to drive the workload, in a possible range of [1,64]
Server thread created
Loading web application
Creating client threads
Waiting for clients to complete
Client threads complete ... unloading web application
Server stopped ... iteration complete

Total number of objects touched: 1313747

Number of local objects: 1296532 (98.6896%)
Number of shared objects: 17215 (1.31037%)
Number of objects marked shared due to garbage collection: 882744
    (67.1928%)
Number of objects marked shared due to finalization:2127
    (0.161903%)

Total touched objects memory occupied in bytes: 56938768
Shared memory of touched objects occupied in bytes: 762096
    (1.33845%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
8914: NioBlockingSelector.BlockPoller-1
28634: ContainerBackgroundProcessor[StandardEngine[Catalina]]
29309: http-7080-ClientPoller-0
29329: http-7080-ClientPoller-1
29345: http-7080-ClientPoller-2
29362: http-7080-ClientPoller-3
29379: http-7080-ClientPoller-4
29397: http-7080-ClientPoller-5
29413: http-7080-ClientPoller-6
.
.
.
113932: http-7080-exec-60
114058: http-7080-exec-61
114313: http-7080-exec-62
114323: http-7080-exec-63
230951: http-7080-exec-64
2611: Thread-0

Runtime: 0h 6m 5secs
```

C.2.7 Lusearch

```
Using scaled threading model. 64 processors detected, 64 threads
    used to drive the workload, in a possible range of [1,64]
4 query batches completed
8 query batches completed
12 query batches completed
16 query batches completed
20 query batches completed
24 query batches completed
28 query batches completed
32 query batches completed
36 query batches completed
40 query batches completed
44 query batches completed
48 query batches completed
52 query batches completed
56 query batches completed
60 query batches completed
64 query batches completed

Total number of objects touched: 4889096

Number of local objects: 4888989 (99.9978%)
Number of shared objects: 107 (0.00218854%)
Number of objects marked shared due to garbage collection: 4152580
     (84.9355%)
Number of objects marked shared due to finalization:156
    (0.00319077%)

Total touched objects memory occupied in bytes: 196005936
Shared memory of touched objects occupied in bytes: 12240
    (0.00624471%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
3581: Query0
3582: Query1
3583: Query2
3587: Query3
.
.
.
3730: Query61
3731: Query62
3733: Query63

Runtime: 0h 29m 57secs
```

C.2.8 Luindex

```
Total number of objects touched: 113233

Number of local objects: 113206 (99.9762%)
Number of shared objects: 27 (0.0238446%)
Number of objects marked shared due to garbage collection: 2595
    (2.29173%)
Number of objects marked shared due to finalization:28
    (0.0247278%)

Total touched objects memory occupied in bytes: 4256848
Shared memory of touched objects occupied in bytes: 1280
    (0.0300692%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
41419: Lucene Merge Thread #0

Runtime: 0h 2m 11secs
```

C.2.9 Sunflow

```
Using scaled threading model. 64 processors detected, 64 threads
    used to drive the workload, in a possible range of [1,256]

Total number of objects touched: 60220386

Number of local objects: 60219897 (99.9992%)
Number of shared objects: 489 (0.000812017%)
Number of objects marked shared due to garbage collection: 4759
    (0.00790264%)
Number of objects marked shared due to finalization:25 (4.15142e
    -05%)

Total touched objects memory occupied in bytes: 2464716216
Shared memory of touched objects occupied in bytes: 13360
    (0.00054205%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
2295: Java2D Disposer
7025: Thread-2
7026: Thread-3
7031: Thread-4
.
.
.
44680: Thread-123
44993: Thread-124
45402: Thread-125
45651: Thread-126
45981: Thread-127
46334: Thread-128
46631: Thread-129
2420: Thread-1
60220380: Thread-0

Runtime: 2h 48m 48secs
```

C.2.10 Xalan

```
Using scaled threading model. 64 processors detected, 64 threads
    used to drive the workload, in a possible range of [1,100]
Normal completion.

Total number of objects touched: 1389362

Number of local objects: 1386051 (99.7617%)
Number of shared objects: 3311 (0.238311%)
Number of objects marked shared due to garbage collection: 1292462
     (93.0256%)
Number of objects marked shared due to finalization:82
    (0.00590199%)

Total touched objects memory occupied in bytes: 58515112
Shared memory of touched objects occupied in bytes: 162648
    (0.277959%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
14009: Thread-0
14010: Thread-1
14011: Thread-2
14012: Thread-3
14013: Thread-4
.
.
.
14127: Thread-56
14129: Thread-57
14132: Thread-58
14133: Thread-59
14135: Thread-60
14137: Thread-61
14139: Thread-62
14140: Thread-63

Runtime: 0h 25m 6secs
```

C.3 Benchmarks from SPECjvm2008

C.3.1 Compress

```
SPECjvm2008 Peak
  Properties file: none
  Benchmarks: compress

  WARNING: Run will not be compliant.
  Property specjvm.run.initial.check must be true for publication.
  Property specjvm.create.xml.report must be true for publication.
  Not a compliant sequence of benchmarks for publication.
  Property specjvm.run.type must be 2 for publication.
  Property specjvm.run.checksum.validation must be true for
      publication.
  Property specjvm.fixed.operations not allowed in publication run
      .
--- --- --- --- --- --- --- --- ---
  Benchmark: compress
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Wed Nov 14 22:29:53 CET 2012
Iteration 1 (1 operation) ends: Thu Nov 15 06:39:21 CET 2012
Iteration 1 (1 operation) result: 0.03 ops/m

Valid run!
Score on compress: 0.03 ops/m


Total number of objects touched: 7912

Number of local objects: 7842 (99.1153%)
Number of shared objects: 70 (0.884732%)
Number of objects marked shared due to garbage collection: 0 (0%)
Number of objects marked shared due to finalization:0 (0%)

Total touched objects memory occupied in bytes: 232080
Shared memory of touched objects occupied in bytes: 4288
    (1.84764%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
3215: Program Runner for compress
```

```
3350: BenchmarkThread compress 3
3348: BenchmarkThread compress 1
3349: BenchmarkThread compress 2
3352: BenchmarkThread compress 5
3351: BenchmarkThread compress 4
3353: BenchmarkThread compress 6
3356: BenchmarkThread compress 9
3355: BenchmarkThread compress 8
3354: BenchmarkThread compress 7
3359: BenchmarkThread compress 12
3358: BenchmarkThread compress 11
3357: BenchmarkThread compress 10
3360: BenchmarkThread compress 13
3361: BenchmarkThread compress 14
3362: BenchmarkThread compress 15
3396: BenchmarkThread compress 16
7905: DestroyJavaVM

Runtime: 8h 9m 29secs
```

C.3.2 Crypto

```
SPECjvm2008 Peak
  Properties file: none
  Benchmarks: crypto.aes crypto.rsa crypto.signverify

  WARNING: Run will not be compliant.
  Property specjvm.run.initial.check must be true for publication.
  Property specjvm.create.xml.report must be true for publication.
  Not a compliant sequence of benchmarks for publication.
  Property specjvm.run.type must be 2 for publication.
  Property specjvm.run.checksum.validation must be true for
      publication.
  Property specjvm.fixed.operations not allowed in publication run
      .
--- --- --- --- --- --- --- --- ---
  Benchmark: crypto.aes
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Thu Nov 15 13:09:09 CET 2012
Iteration 1 (1 operation) ends: Thu Nov 15 13:55:22 CET 2012
Iteration 1 (1 operation) result: 0.35 ops/m

Valid run!
Score on crypto.aes: 0.35 ops/m
```

```
--- --- --- --- --- --- --- --- ---
  Benchmark: crypto.rsa
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Thu Nov 15 13:55:25 CET 2012
Iteration 1 (1 operation) ends: Thu Nov 15 14:17:00 CET 2012
Iteration 1 (1 operation) result: 0.74 ops/m

Valid run!
Score on crypto.rsa: 0.74 ops/m
--- --- --- --- --- --- --- --- ---
  Benchmark: crypto.signverify
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Thu Nov 15 14:17:03 CET 2012
Iteration 1 (1 operation) ends: Thu Nov 15 16:20:37 CET 2012
Iteration 1 (1 operation) result: 0.13 ops/m

Valid run!
Score on crypto.signverify: 0.13 ops/m


Total number of objects touched: 1406740

Number of local objects: 1405289 (99.8969%)
Number of shared objects: 1451 (0.103146%)
Number of objects marked shared due to garbage collection: 16321
    (1.1602%)
Number of objects marked shared due to finalization:2
    (0.000142173%)

Total touched objects memory occupied in bytes: 43633664
Shared memory of touched objects occupied in bytes: 59160
    (0.135583%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
3215: Program Runner for crypto.aes
3872: BenchmarkThread crypto.aes 1
3874: BenchmarkThread crypto.aes 3
```

```
3920: BenchmarkThread crypto.aes 16
.
.
.
1218541: BenchmarkThread crypto.signverify 5
1218547: BenchmarkThread crypto.signverify 11
1218549: BenchmarkThread crypto.signverify 13
1218550: BenchmarkThread crypto.signverify 14
1218585: BenchmarkThread crypto.signverify 16
1218551: BenchmarkThread crypto.signverify 15
1406733: DestroyJavaVM

Runtime: 3h 11m 28secs
```

C.3.3 Serial

```
SPECjvm2008 Peak
  Properties file: none
  Benchmarks: serial

  WARNING: Run will not be compliant.
  Property specjvm.run.initial.check must be true for publication.
  Property specjvm.create.xml.report must be true for publication.
  Not a compliant sequence of benchmarks for publication.
  Property specjvm.run.type must be 2 for publication.
  Property specjvm.run.checksum.validation must be true for
      publication.
  Property specjvm.fixed.operations not allowed in publication run
      .
--- --- --- --- --- --- --- --- ---
  Benchmark: serial
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Thu Nov 15 08:41:01 CET 2012
Iteration 1 (1 operation) ends: Thu Nov 15 12:53:03 CET 2012
Iteration 1 (1 operation) result: 0.06 ops/m

Valid run!
Score on serial: 0.06 ops/m


Total number of objects touched: 51136673

Number of local objects: 51135192 (99.9971%)
Number of shared objects: 1481 (0.00289616%)
```

```
Number of objects marked shared due to garbage collection:
    50165349 (98.1005%)
Number of objects marked shared due to finalization:2 (3.91109e
    -06%)

Total touched objects memory occupied in bytes: 1571854072
Shared memory of touched objects occupied in bytes: 62976
    (0.00400648%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
3215: Program Runner for serial
9236: BenchmarkThread serial 1
9241: BenchmarkThread serial 6
9240: BenchmarkThread serial 5
9237: BenchmarkThread serial 2
9244: BenchmarkThread serial 9
9242: BenchmarkThread serial 7
9243: BenchmarkThread serial 8
9239: BenchmarkThread serial 4
9238: BenchmarkThread serial 3
9245: BenchmarkThread serial 10
9246: BenchmarkThread serial 11
9248: BenchmarkThread serial 13
9284: BenchmarkThread serial 16
9250: BenchmarkThread serial 15
9249: BenchmarkThread serial 14
9247: BenchmarkThread serial 12
51136666: DestroyJavaVM

Runtime: 4h 12m 7secs
```

C.3.4 Compiler

```
SPECjvm2008 Peak
  Properties file: none
  Benchmarks: compiler.compiler compiler.sunflow

  WARNING: Run will not be compliant.
  Property specjvm.run.initial.check must be true for publication.
  Property specjvm.create.xml.report must be true for publication.
  Not a compliant sequence of benchmarks for publication.
  Property specjvm.run.type must be 2 for publication.
  Property specjvm.run.checksum.validation must be true for
      publication.
  Property specjvm.fixed.operations not allowed in publication run
      .


--- --- --- --- --- --- --- --- ---

  Benchmark: compiler.compiler
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Thu Nov 15 23:24:35 CET 2012
Iteration 1 (1 operation) ends: Fri Nov 16 01:08:35 CET 2012
Iteration 1 (1 operation) result: 0.15 ops/m

Valid run!
Score on compiler.compiler: 0.15 ops/m


--- --- --- --- --- --- --- --- ---

  Benchmark: compiler.sunflow
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Fri Nov 16 01:09:28 CET 2012
Iteration 1 (1 operation) ends: Fri Nov 16 06:07:53 CET 2012
Iteration 1 (1 operation) result: 0.05 ops/m

Valid run!
```

```
Score on compiler.sunflow: 0.05 ops/m


Total number of objects touched: 129275246

Number of local objects: 128995827 (99.7839%)
Number of shared objects: 279419 (0.216143%)
Number of objects marked shared due to garbage collection:
    74157149 (57.3638%)
Number of objects marked shared due to finalization:54 (4.17713e
    -05%)

Total touched objects memory occupied in bytes: 4063726792
Shared memory of touched objects occupied in bytes: 9039528
    (0.222444%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
3215: Program Runner for compiler.compiler
2576115: BenchmarkThread compiler.compiler 1
2576335: BenchmarkThread compiler.compiler 3
2577325: BenchmarkThread compiler.compiler 12
2577105: BenchmarkThread compiler.compiler 10
2576995: BenchmarkThread compiler.compiler 9
.
.
.
40495956: BenchmarkThread compiler.sunflow 8
40495736: BenchmarkThread compiler.sunflow 6
40496066: BenchmarkThread compiler.sunflow 9
40495846: BenchmarkThread compiler.sunflow 7
40495516: BenchmarkThread compiler.sunflow 4
40496286: BenchmarkThread compiler.sunflow 11
40496176: BenchmarkThread compiler.sunflow 10
40495626: BenchmarkThread compiler.sunflow 5
129275239: DestroyJavaVM

Runtime: 6h 45m 35secs
```

C.3.5 Derby

```
SPECjvm2008 Peak
  Properties file: none
  Benchmarks: derby

  WARNING: Run will not be compliant.
  Property specjvm.run.initial.check must be true for publication.
  Property specjvm.create.xml.report must be true for publication.
```

```
  Not a compliant sequence of benchmarks for publication.
  Property specjvm.run.type must be 2 for publication.
  Property specjvm.run.checksum.validation must be true for
      publication.
  Property specjvm.fixed.operations not allowed in publication run
      .


--- --- --- --- --- --- --- --- ---

  Benchmark: derby
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Fri Nov 16 13:34:27 CET 2012
Iteration 1 (1 operation) ends: Fri Nov 16 15:47:09 CET 2012
Iteration 1 (1 operation) result: 0.12 ops/m

Valid run!
Score on derby: 0.12 ops/m


Total number of objects touched: 176488340

Number of local objects: 176259700 (99.8705%)
Number of shared objects: 228640 (0.12955%)
Number of objects marked shared due to garbage collection:
    148460015 (84.1189%)
Number of objects marked shared due to finalization:32 (1.81315e
    -05%)

Total touched objects memory occupied in bytes: 6657886008
Shared memory of touched objects occupied in bytes: 11431528
    (0.171699%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
3215: Program Runner for derby
199838: derby.antiGC
205870: Timer-0
207337: derby.rawStoreDaemon
307612: Thread-1
24571650: derby.rawStoreDaemon
24572778: derby.rawStoreDaemon
24573829: derby.rawStoreDaemon
```

```
24574833: BenchmarkThread derby 1
24574841: BenchmarkThread derby 9
24574840: BenchmarkThread derby 8
24574839: BenchmarkThread derby 7
24574838: BenchmarkThread derby 6
24574837: BenchmarkThread derby 5
24574835: BenchmarkThread derby 3
24574836: BenchmarkThread derby 4
24574881: BenchmarkThread derby 16
24574842: BenchmarkThread derby 10
24574845: BenchmarkThread derby 13
24574844: BenchmarkThread derby 12
24574846: BenchmarkThread derby 14
24574847: BenchmarkThread derby 15
24574834: BenchmarkThread derby 2
24574843: BenchmarkThread derby 11
176488333: DestroyJavaVM

Runtime: 3h 2m 58secs
```

C.3.6 MPEGaudio

```
SPECjvm2008 Peak
  Properties file: none
  Benchmarks: mpegaudio

  WARNING: Run will not be compliant.
  Property specjvm.run.initial.check must be true for publication.
  Property specjvm.create.xml.report must be true for publication.
  Not a compliant sequence of benchmarks for publication.
  Property specjvm.run.type must be 2 for publication.
  Property specjvm.run.checksum.validation must be true for
      publication.
  Property specjvm.fixed.operations not allowed in publication run
      .


--- --- --- --- --- --- --- --- ---

  Benchmark: mpegaudio
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Fri Nov 16 16:36:38 CET 2012
Iteration 1 (1 operation) ends: Fri Nov 16 18:56:50 CET 2012
Iteration 1 (1 operation) result: 0.11 ops/m
```

```
Valid run!
Score on mpegaudio: 0.11 ops/m


Total number of objects touched: 12083

Number of local objects: 11933 (98.7586%)
Number of shared objects: 150 (1.24141%)
Number of objects marked shared due to garbage collection: 0 (0%)
Number of objects marked shared due to finalization:0 (0%)

Total touched objects memory occupied in bytes: 375896
Shared memory of touched objects occupied in bytes: 7616
    (2.02609%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
3215: Program Runner for mpegaudio
3273: BenchmarkThread mpegaudio 3
3272: BenchmarkThread mpegaudio 2
3271: BenchmarkThread mpegaudio 1
3274: BenchmarkThread mpegaudio 4
3275: BenchmarkThread mpegaudio 5
3276: BenchmarkThread mpegaudio 6
3277: BenchmarkThread mpegaudio 7
3278: BenchmarkThread mpegaudio 8
3279: BenchmarkThread mpegaudio 9
3280: BenchmarkThread mpegaudio 10
3282: BenchmarkThread mpegaudio 12
3319: BenchmarkThread mpegaudio 16
3285: BenchmarkThread mpegaudio 15
3284: BenchmarkThread mpegaudio 14
3281: BenchmarkThread mpegaudio 11
3283: BenchmarkThread mpegaudio 13
12076: DestroyJavaVM

Runtime: 2h 20m 16secs
```

C.3.7 Scimark

```
SPECjvm2008 Peak
  Properties file: none
  Benchmarks: scimark.fft.large scimark.lu.large scimark.sor.large
       scimark.sparse.large scimark.fft.small scimark.lu.small
      scimark.sor.small scimark.sparse.small scimark.monte_carlo

  WARNING: Run will not be compliant.
```

```
  Property specjvm.run.initial.check must be true for publication.
  Property specjvm.create.xml.report must be true for publication.
  Not a compliant sequence of benchmarks for publication.
  Property specjvm.run.type must be 2 for publication.
  Property specjvm.run.checksum.validation must be true for
      publication.
  Property specjvm.fixed.operations not allowed in publication run
      .


--- --- --- --- --- --- --- --- ---

  Benchmark: scimark.fft.large
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Fri Nov 16 22:52:00 CET 2012
Iteration 1 (1 operation) ends: Fri Nov 16 23:13:41 CET 2012
Iteration 1 (1 operation) result: 0.74 ops/m

Valid run!
Score on scimark.fft.large: 0.74 ops/m


--- --- --- --- --- --- --- --- ---

  Benchmark: scimark.lu.large
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Fri Nov 16 23:13:41 CET 2012
Iteration 1 (1 operation) ends: Sat Nov 17 03:46:41 CET 2012
Iteration 1 (1 operation) result: 0.06 ops/m

Valid run!
Score on scimark.lu.large: 0.06 ops/m


--- --- --- --- --- --- --- --- ---

  Benchmark: scimark.sor.large
  Run mode: static run
  Test type: multi
```

```
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Sat Nov 17 03:46:41 CET 2012
Iteration 1 (1 operation) ends: Sat Nov 17 05:14:57 CET 2012
Iteration 1 (1 operation) result: 0.18 ops/m

Valid run!
Score on scimark.sor.large: 0.18 ops/m


--- --- --- --- --- --- --- --- ---

  Benchmark: scimark.sparse.large
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Sat Nov 17 05:14:57 CET 2012
Iteration 1 (1 operation) ends: Sat Nov 17 08:01:05 CET 2012
Iteration 1 (1 operation) result: 0.10 ops/m

Valid run!
Score on scimark.sparse.large: 0.10 ops/m


--- --- --- --- --- --- --- --- ---

  Benchmark: scimark.fft.small
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Sat Nov 17 08:01:05 CET 2012
Iteration 1 (1 operation) ends: Sat Nov 17 08:18:18 CET 2012
Iteration 1 (1 operation) result: 0.93 ops/m

Valid run!
Score on scimark.fft.small: 0.93 ops/m


--- --- --- --- --- --- --- --- ---

  Benchmark: scimark.lu.small
```

```
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Sat Nov 17 08:18:18 CET 2012
Iteration 1 (1 operation) ends: Sat Nov 17 08:49:03 CET 2012
Iteration 1 (1 operation) result: 0.52 ops/m

Valid run!
Score on scimark.lu.small: 0.52 ops/m


--- --- --- --- --- --- --- --- ---

  Benchmark: scimark.sor.small
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Sat Nov 17 08:49:03 CET 2012
Iteration 1 (1 operation) ends: Sat Nov 17 09:07:05 CET 2012
Iteration 1 (1 operation) result: 0.89 ops/m

Valid run!
Score on scimark.sor.small: 0.89 ops/m


--- --- --- --- --- --- --- --- ---

  Benchmark: scimark.sparse.small
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Sat Nov 17 09:07:05 CET 2012
Iteration 1 (1 operation) ends: Sat Nov 17 09:49:12 CET 2012
Iteration 1 (1 operation) result: 0.38 ops/m

Valid run!
Score on scimark.sparse.small: 0.38 ops/m


--- --- --- --- --- --- --- --- ---
```

```
  Benchmark: scimark.monte_carlo
  Run mode: static run
  Test type: multi
  Threads: 16
  Iterations: 1
  Run length: 1 operation

Iteration 1 (1 operation) begins: Sat Nov 17 09:49:12 CET 2012
Iteration 1 (1 operation) ends: Sun Nov 18 07:31:09 CET 2012
Iteration 1 (1 operation) result: 0.01 ops/m

Valid run!
Score on scimark.monte_carlo: 0.01 ops/m


Total number of objects touched: 97377

Number of local objects: 96980 (99.5923%)
Number of shared objects: 397 (0.407694%)
Number of objects marked shared due to garbage collection: 3526
    (3.62098%)
Number of objects marked shared due to finalization:2
    (0.00205387%)

Total touched objects memory occupied in bytes: 2552656
Shared memory of touched objects occupied in bytes: 27680
    (1.08436%)

Thread IDs and Names (During live phase):
1: Signal Dispatcher
2: main
3215: Program Runner for scimark.fft.large
3291: BenchmarkThread scimark.fft.large 1
3294: BenchmarkThread scimark.fft.large 4
3292: BenchmarkThread scimark.fft.large 2
3295: BenchmarkThread scimark.fft.large 5
3293: BenchmarkThread scimark.fft.large 3
.
.
.
96572: BenchmarkThread scimark.monte_carlo 16
96526: BenchmarkThread scimark.monte_carlo 3
96536: BenchmarkThread scimark.monte_carlo 13
96538: BenchmarkThread scimark.monte_carlo 15
96529: BenchmarkThread scimark.monte_carlo 6
97370: DestroyJavaVM

Runtime: 32h 39m 10secs
```

Bibliography

[1] Annotation File Utilities. [Online]. Available: http://types.cs.washington.edu/annotation-file-utilities/

[2] Annotations. [Online]. Available: http://docs.oracle.com/javase/6/docs/technotes/guides/language/annotations.html

[3] ASM. [Online]. Available: http://asm.ow2.org/

[4] FastStringBuffer class of Xalan-J 2.7.1. [Online]. Available: http://svn.apache.org/viewvc/xalan/java/tags/xalan-j_2_7_1/src/org/apache/xml/utils/FastStringBuffer.java?revision=578285&view=markup

[5] The Java Grande Forum Multi-threaded Benchmarks. [Online]. Available: http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/contents.html

[6] Java SE Core Technologies - Java Compiler. [Online]. Available: http://java.sun.com/javase/technologies/core/toolsapis/javac/index.jsp

[7] Java VM Type Signatures. [Online]. Available: http://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/types.html#wp276

[8] Java wildcards. [Online]. Available: http://download.oracle.com/javase/tutorial/extra/generics/wildcards.html

[9] Jikes RVM homepage. [Online]. Available: http://jikesrvm.org/

[10] JVM(TM) Tool Interface. [Online]. Available: http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html

[11] SPECjvm2008 (Java Virtual Machine Benchmark). [Online]. Available: http://www.spec.org/jvm2008/

[12] The 'class' File Format. [Online]. Available: http://docs.oracle.com/javase/specs/jvms/se5.0/html/Overview.doc.html#32310

[13] Type Annotations Specification (JSR 308). [Online]. Available: http://types.cs.washington.edu/jsr308/specification/java-annotation-design-20091125.html

[14] (2002, 8) Understanding the IBM Java Garbage Collector. [Online]. Available: http://www.ibm.com/developerworks/ibm/library/i-garbage1/

[15] S. M. Blackburn, R. Garner, C. Hoffman, K. S. Khan, A. M.and McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, M. Hosking, A.and Jump, H. Lee, J. E. B. Moss, D. Phansalkar, A.and Stefanović, T. VanDrunen, and B. von Dincklage, D.and Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[16] B. Blanchet, "Escape analysis for object-oriented languages: application to java," *SIGPLAN Not.*, vol. 34, no. 10, pp. 20–34, Oct. 1999. [Online]. Available: http://doi.acm.org/10.1145/320385.320387

[17] G. Bracha, "Pluggable type systems," in *OOPSLA04 Workshop on Revival of Dynamic Languages*, Vancouver, Canada, 2004.

[18] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff, "Stack allocation and synchronization optimizations for java using escape analysis," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 6, pp. 876–910, Nov. 2003. [Online]. Available: http://doi.acm.org/10.1145/945885.945892

[19] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen, "Minimal ownership for active objects," in *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, ser. APLAS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 139–154. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89330-1_11

[20] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 93–104, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024723.2000076

[21] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald, "Thread-local heaps for java," in *Proceedings of the 3rd international symposium on Memory management*, ser. ISMM '02. New York, NY, USA: ACM, 2002, pp. 76–87. [Online]. Available: http://doi.acm.org/10.1145/512429.512439

[22] M. Ernst. (2008–present) The checker framework: Custom pluggable types for Java. [Online]. Available: http://types.cs.washington.edu/checker-framework/

[23] Y. Lu, J. Potter, and J. Xue, "Ownership downgrading for ownership types," in *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, ser. APLAS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 144–160. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10672-9_12

[24] A. Sherwany, "The design, implementation and evaluation of a pluggable type checker for thread-locality in java," 2011.

[25] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek, "Loci: Simple thread-locality for Java," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 445–469. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_21

[26] T. Wrigstad and A. Sherwany, *Loci 0.1 Manual*, 2011. [Online]. Available: http://loci.java.net/manual