A Java Dialect Free of Data Races and Without Annotations

Luis Mateu
Universidad de Chile
Computer Science Department
lmateu@dcc.uchile.cl

Abstract

We introduce a dialect of Java, JShield, for concurrent object oriented programming, whose primary design goal is robustness. JShield preserves the Java syntax and the semantics of sequential Java programs. It modifies the semantics of concurrent programs to completely avoid data races without relying on the programmer ability, and without requiring special annotations. This is achieved by combining Hoare's monitors with remote method invocations of Java to ensure the proper request of a lock before manipulating shared data. We show that this can be done with a reasonable overhead in execution time compared to Java.

1 Introduction

Data races are common, insidious errors associated with multi-threaded programming [8]. A data race occurs when a shared data structure is manipulated simultaneously by two threads without any synchronization. This situation is dangerous because it may lead the shared data structure to an inconsistent state. Data races are devilishly hard to track down because they only become apparent randomly through program crashes or even incorrect results. A language feature enforcing avoidance of data races is of paramount importance. Modern concurrent programming languages, as Java, rely on the abilities of the programmer to avoid data races : he or she must ensure that every shared data structure is protected by a mutual exclusion lock. Unfortunately, in large programs it is rather easy to neglect to ask a required lock, thus opening the door to data races.

We introduce a dialect of Java, JShield, and its implementation. JShield is based on the model of safe-threads [11] following the ideas of Brinch Hansen [4] and Hoare [9]. They state that concurrent programming languages should be designed to be data race free, that is, they should enforce programmers to write code free of data races in the same way that a strictly typed language enforces programmers to write type safe code. JShield preserves the syntax of Java and the semantics of sequential Java programs. It differs from Java on the semantics of multi-threaded programs. While in a Java application all threads share the same objects, a JShield application consists of a set of sequential processes not sharing any kind of data. Each process can only access the data stored in objects previously created by itself, thus avoiding any kind of data races. We call these processes *shielded components* because they are shielded against data races. A shielded component may communicate to other components through a mechanism similar to remote method invocations of Java. When multiple remote invocations target the same shielded component, JShield ensures that they are served sequentially by protecting the component with a monitor [9].

JShield makes concurrent programs robust because it enforces programmers to think about synchronization in the design phase of the application. Debugging in JShield is easier than in Java because programmers will never have to track down data races. However avoiding data races is not free in the same way that garbage collection is not free. Parallel programming in JShield is less efficient than multi-threaded programming in Java. Hence, from a performance point of view, JShield cannot compete with Java when the bottleneck is the processor. However JShield has a niche for applications where the bottleneck is the disk or the network, but not the processor. For example, Internet applications need parallel programming to deal, for example, with several user connections. A frequently used approach to avoid the difficulties of multi-threaded programming is to work with independent Java heavy weight processes. The good news is that programming in JShield offers the same simplicity than programming with heavy weight processes, but the former is more efficient than the latter because shielded components are implemented as light weight processes, i.e. they live in the same address space. This ideas were tested with the current JShield implementation.[1]

There are several proposals to avoid data races in programming languages (see Section 5). The closest approach

[1]The implementation of JShield is available at `http://www.dcc.uchile.cl/~lmateu/jshield/`.

to ours is Guava [2]. Like JShield, Guava is a concurrent programming language, based on Java and designed to be data race free. Programming in Guava is more complex than programming in JShield because programmers must include special annotations when writing code, as for example in formal parameters and method declarations. These annotations were introduced to minimize the overheads of Guava compared to Java. Therefore, Guava is a good choice when processor efficiency is a primary goal and JShield is good when programming simplicity is more important than efficiency.

The structure of the paper is as follows: We present JShield in Section 2. Section 3 shows the main issues of the implementation of JShield and Section 4 gives performance measurements showing the overhead in execution time of JShield compared to Java. Section 5 compares our research to related works. The conclusions are in Section 6.

2 The JShield Language

JShield is inspired by the following metaphor: a JShield application is like a distributed system consisting of several heavy-weight processes communicating through remote object invocations such as those provided by Java/RMI. Therefore, in what follows, we will make frequent comparisons with Java/RMI.

Shielded components. A JShield application must be decomposed on a system of dynamic *shielded components*. A shielded component –or component for short– is represented by a set of objects. An object always belongs to one and only one component. A component is said to be shielded because its objects are protected from the accesses of other components. A component can be viewed as all the objects contained by one JVM process in our inspiring metaphor. For example, let us consider an application consisting of the following components: a bank containing accounts and two clients which query the accounts and do cash movements (see Figure 1).

We say that *a thread runs or executes on a given component* if and only if the most recent method in the chain of invocations has been invoked on an object of that component. For example, in Figure 2, the thread T runs first on the client component. When it invokes `doMovement`, the execution proceeds on the bank and when it finishes with `doMovement` the execution returns back to the client.

We say that *a component creates an object* when a thread running on it creates an object. In JShield *objects always belong to the component that creates them.* The objects cannot migrate later to other components. Method invocations are always executed by the component owning the target object.

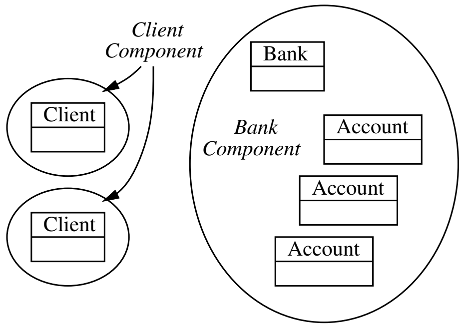

Figure 1. An example of components

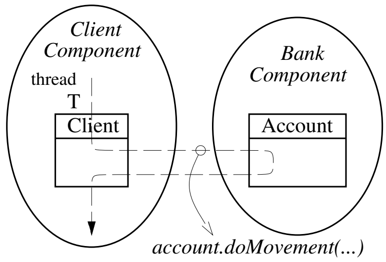

Figure 2. An example of method invocation

We call *local objects* of a component to all the objects contained or owned by that component. Objects of other components are called *foreign objects*. In Figure 1, the account objects are local to the bank and foreign to the clients.

Classes of shielded components are declared by extending the class ShieldedComBase. In our example the implementation of the Bank component is:

```
public class BankImpl
       extends ShieldedComBase
       implements Bank {
  Map accountMap= new HashMap();
                  // ids -> accounts
  public Account getAccount(String accId) {
    return (Account)accountMap.get(accId);
  }
  public void doBackup(
         ObjectOutputStream backupStream) {
    backupStream.writeObject(this);
  }
  public Account makeAccount(String accId) {
    Account account= new AccountImpl(accId);
    accountMap.put(accId, account)
    return account;
  }
}
```

This declaration is syntactically identical to a class declaration of Java but its semantics is different because JShield will ensure that other components will not have access to the instance variables of BankImpl (even though `accountMap` was not declared private) and will only be able to invoke the methods declared in the interface Bank.

Shared objects. In JShield, foreign objects are restricted to be *shared objects*. These are objects belonging to the class SharedObj or any of its subclasses. Non shared objects can only be referenced locally. A component has no access to the instance variables of a foreign object and it can only invoke the methods declared in a *shareable interface*. The shareable interface is any interface extending Shareable and is equivalent to the interface Remote of Java/RMI.

These restrictions in JShield are fundamental to avoid that two components access the same object concurrently without synchronization. Programmers cannot violate these restrictions because the JShield compiler/runtime ensures that foreign objects are always referenced by variables or expressions whose type is a shareable interface. An example of verifications carried out by the runtime are cast checks : when casting an expression to the implementation class of a shared object, an exception is thrown if the cast reference is not local.

Classes of shared objects are declared by extending the class SharedObj. In our example, the accounts are implemented by the following class :

```
public class AccountImpl
       extends SharedObj
       implements Account {
  String accId; // Identifies this account
  int accNum;
  double balance= 0.0;
  static int serial= 0;  // static!

  public AccountImpl(String accId) {
    this.accId= accId;
    this.accNum= serial++;
  }
  public double query() {
    return balance;
  }
  public boolean doMovement(double amount) {
    if (balance+amount<0.0)
      return false;
    balance+= amount;
    return true;
  }
}
```

This class has declared only one shareable interface : Account. Therefore the client components can only reference accounts through variables of the type Account. The JShield compiler/runtime will reject any attempt of a client to reference an account by its implementing class AccountImpl.

A shareable interface is declared in JShield by extending the interface Shareable. For example the interface Account is declared as :

```
public interface Account extends Shareable {
  double query();
  boolean doMovement(double amount);
}
```

The only operations that client components can invoke on accounts are those specified in the shareable interface, i.e. the methods `query` and `doMovement`. In particular, clients cannot access instance variables of `AccountImpl` through :

```
  Account account= bank.query("John");

  // Invalid!
  ((AccountImpl)account).balance= 1000000;
```

For the instruction above, JShield will reject the cast to `AccountImpl` signaling at execution time that the object referenced by account is not local to the client component. Something similar occurs in Java/RMI when a remote object is cast to its implementation class.

Mutual exclusion. In JShield there is no parallelism inside a component : at most one thread can be executed at any moment. This is achieved by associating a controlling monitor to each component. This monitor ensures the mutual exclusion of threads running inside a component and is requested automatically when entering any method declared on a shareable interface.

When a thread invokes a shareable method, the monitor is granted if the monitor is free or if it is already owned by the calling thread. The thread must otherwise wait until the monitor is free. Then the thread owns the monitor until it returns or frees the monitor to wait for a condition. The thread owning the monitor is said to be the active thread and a component having an active thread is called an active component.

This is different from Java/RMI, where if programmers believe data races may occur then they must explicitly synchronize remote methods. Otherwise, the method will not be synchronized.[2]

For example, the method `doMovement` may be invoked on a bank account while `doBackup` is running on the bank. In such a case, the thread invoking the former method must wait until the latter finishes. Note that this is true even though the methods are invoked on different objects (but both belonging to the same component).

[2]Note that in JShield the attribute *synchronized* in a method has no meaning.

Parameter passing. The Java parameter passing policy for method invocations of foreign objects is modified in JShield. This is necessary because the caller component may pass object references to the callee component. JShield must avoid creating improper references : a variable whose type is not a shareable interface referencing a foreign object.

The JShield parameter-passing policy is the following : (i) values of primitive types are passed by value just like in Java, (ii) shared objects are passed by reference, and (iii) non shared objects are passed by *deep-copy*. The JShield compiler also restricts the formal type of shared object parameters to be a shareable interface. The same policy applies to the value returned by a method and the parameters received by a constructor of a shielded component.

Deep-copy in JShield is similar to the serialization of parameters for remote method invocations in Java/RMI. Non shared objects that are referenced by a copied object must also be deep-copied. If a copied object references a shared object, the latter is not deep-copied. However the runtime throws an exception if a foreign object is referenced by an instance variable whose type is not a shareable interface.

Static variables. We intentionally included the static variable `serial` in the declaration of the class AccountImpl to underline its semantics in JShield. May a data race occur when two instances of AccountImpl are created concurrently on *different* banks? The answer is that data races associated with static variables will never occur in JShield because each component will have its own set of static variables for all classes it uses, just as multiple JVMs will have independent sets of static variables.

In other words, in JShield, static variables serve to store information which is considered to be global to the component, but not to the entire application.

Shielded components as shared objects. For convenience, in JShield, a shielded component is also a shared object owned by itself. This shared object implements the shareable interface ShieldedCom, but the programmer can implement other shareable interfaces as well. In our example, the bank component implements the following interface :

```
public interface Bank extends Shareable {
  Account getAccount(String accId);
  Account makeAccount(String accId);
  void doBackup(
        ObjectOutputStream backupStream);
}
```

Creating new components. Components are created by using the `new` operator as with normal objects. For example :

```
Bank bank= new BankImpl();

// Invalid!
BankImpl bankImpl= new BankImpl();
```

However the semantics of the `new` operator cannot be the same as in Java. In JShield, when the class being instantiated is a subclass of ShieldComBase, the type of the *new expression* is not the subclass, but an interface which extends all the shareable interfaces implemented by the component. Thus the first statement above is valid, but the second statement is invalid and JShield rejects it. If we allowed this kind of assignments, a client component could access the instance variables of the bank and then there could be data races.

Active shielded components. A shielded component like the bank is a passive shielded component because it exists only to serve requests coming from other components. On the other hand, a client is an active shielded component. It is created to have its own behavior. Classes of active components are declared by extending the class ActiveComBase, as for example :

```
public class Client
       extends ActiveComBase {
  Bank bank;
  Client(Bank bank) {
    this.bank= bank;
    start();
  }
  public void run() {
    for (...) {
      Account acc= bank.getAccount(...);
      double money= acc.query();
      ...
      acc.doMovement(...);
} } }
```

An active component starts its execution as an independent thread of execution when the method `start` is called. The behavior of the component must be put in the method `run`.

3 Implementation Issues

We implemented JShield through a preprocessor which takes programs written in JShield and translates them to Java. The preprocessor is written in Java and uses SableCC[3] to recognize the grammar of JShield (which is the same as Java). The preprocessor has 6000 lines of code, without considering the parser and libraries.

We will sketch the most important changes made to programs in JShield while translating them to Java. This will

[3] SableCC is available at `http://www.sablecc.org/`

help the reader to get an idea of the overhead added by JShield compared to Java. There are other changes which due to space constraints will not be explained here.

Translation of Shareable Methods. A shareable method is a method that appears in a shareable interface. These methods in JShield can be compared to the synchronized methods of Java. The JShield preprocessor translates these methods by adding the following code :

- At the beginning of the method a test is made to check if the invocation is foreign. If the invocation is foreign, the monitor associated with the owner component is requested and the formal parameters are deep-copied when needed.
- At return, if the invocation is foreign, the return value is deep-copied if needed, and the monitor is released.

Translation of accesses to static variables. In Java there is exactly one instance of each static variable. Thus an access to this kind of variables can be compiled just as a load/store instruction from/to an absolute address. In JShield there can be several instances of a static variable, one for each component using the variable, so such an implementation does not apply here.

In JShield the conceptual static variables of a class A are implemented as instance variables of a separated class A_s. Each component keeps a hash table mapping classes, like the class A, to objects containing the conceptual static variables in an instance of A_s. The JShield preprocessor translates each access of a static variable like $A.v$ to an expression of the form :

```
A.getStatics(A.class).v;
```

where `getStatics` looks up the class A in the hash table associated with the running component.

4 Performance

In this section we compare the performances of three approaches to developing parallel programs on a shared memory machine : (i) programming multi-threaded programs in Java, (ii) programming with multiple active shielded components in JShield, and (iii) programming with multiple heavy weight processes, where each process is a single-threaded Java program communicating through remote method invocations.

We did all measurements on a single processor : a notebook with a 333 MHz Pentium II processor and 192 MB of PC100 DRAM memory. We used Java version 1.4.0_02-b02 under Linux Suse 7.2. This is fair enough for our purposes, that is, measuring the overhead of JShield and Java/RMI compared to Java. We plan to make measurements in a shared-memory multi-processor as future work.

Raw overhead. In JShield some operations are more expensive than their equivalents in Java because JShield has to ensure that they will not open the door for future data-races. We wanted to measure the raw overhead of such operations compared to Java. The operations considered are the following :

- The time taken to increment by one an integer static variable. As stated in Section 3 in JShield static variables are not shared by components and they must be looked up in a hash table. In Java and Java/RMI they are a direct access to an absolute address.
- The time taken to invoke a method, without arguments, on a shared object. For Java, we measured the time taken to invoke a synchronized method, since this is the way used by programmers to ensure the mutual exclusion of threads. For JShield we measured the time to invoke a foreign method. And for Java/RMI we measured the time taken to execute a remote method invocation when the caller JVM and the callee JVM live in the same machine.
- The time taken to invoke a method receiving a circular linked list of 1, 2 or three elements. In Java, the runtime just passes a pointer to such an object. In JShield, the runtime will deep-copy the entire list. And in Java/RMI, the runtime will serialize the entire list through a socket.

There are some other operations which also introduce some overhead, but it is marginal when compared with the operations mentioned above. The following table shows our measurements :

Operation	Java nanosecs.	JShield nanosecs.	Java/RMI nanosecs.
static variable access	18	820	18
shared method call:			
with 0 arguments	157	1001	677000
with 1 element list	139	4616	1300000
with 2 element list	139	6120	1328000
with 3 element list	139	7356	1374000

We repeated each operation several times on a loop and measured the mean execution time of each iteration. Before any measurement we ensured that the JIT-compiler had compiled the code to be benchmarked.

As expected, accessing a static variable in JShield is considerably more expensive than in Java. Also, invoking foreign methods in JShield is quite expensive compared to invoking a synchronized method in Java, specially when deep copying is necessary. However foreign method invocation is several orders of magnitude less expensive than invoking a remote method in Java/RMI. This is explained by the latency of changing virtual address spaces in Unix.

Application performance. For comparing the performances of JShield with respect to Java and Java/RMI we separated the measurements in two groups: (i) sequential programs and (ii) parallel programs. First we consider sequential programs.

We took two sequential Java programs and compiled them in both Java and JShield. Given that JShield is compatible with Java sequential programs, no change to the source code was needed. As stated in Section 3, JShield introduces some changes in the code, as for example : static variable access is translated to a look up in a hash table. Therefore we were interested in measuring how much overhead produced those changes.

The first program is Linpack[4], a standard benchmark to measure floating point performance. The second program is the JShield preprocessor which is written in Java. It takes programs written in JShield and translates them to Java. We measured how much time it took to translate a program of 6977 lines. We got the following results :

Application	Java	JShield (overhead in %)
Linpack	36.0 MFlops/sec.	36.7 MFlops/sec. (-2%)
Preprocessor	24803 millisecs.	28344 millisecs. (14%)

For the Linpack benchmark, one expects to find no difference in performance because Linpack does not use static variables or any other feature needing special translation. However JShield appears to be marginally faster than Java. We concluded that the difference is due to random variables like the behavior of the cache or the jit-compiler.

For the preprocessor, we expected less overhead than the 14% obtained. This is explained because the JShield preprocessor uses the *Byte Code Engineering Library* [6] which uses some constant arrays declared as final static variables. JShield is unable to optimize those accesses and has to look up them through a hash table, just as normal static variables. We expect that JShield programmers will be more conscious about the cost of static variables and reduce their use. [5]

As parallel programs, we considered parallel multiplication of matrices and parallel computation of factorial. For each problem we programmed three different versions of the same algorithm, for Java, JShield and Java/RMI. The first version achieved the parallelism by using multiple threads, the second version used multiple active components and the third used multiple heavy weight JVMs communicating through remote method invocations on the same machine. The first program computes the product of two matrices of 256 rows by 256 columns. The second program computes the factorial of 10000 by using big integers. The following table shows the results :

Program and num. of threads, comp. or jvm's	Java millisecs.	JShield millisecs. (overhead)	Rmi millisecs. (overhead)
Matrix:			
1	2154	2866 (33%)	3740 (74%)
2	2114	2915 (38%)	4065 (92%)
4	2144	2962 (38%)	4645 (117%)
8	2223	3019 (36%)	5748 (159%)
16	2206	3179 (44%)	8495 (285%)
Factorial:			
1	1344	1348 (0%)	1453 (8%)
2	1302	1389 (7%)	1462 (12%)
4	1323	1393 (5%)	1515 (9%)
8	1328	1409 (6%)	1603 (21%)
16	1345	1432 (6%)	1789 (33%)

For parallel matrix multiplication, JShield is less efficient than Java because one of the matrices must be deep copied several times when passed to each one of the active components. It can be observed that the multiple JVMs version is by far the most inefficient. For the case of parallel factorial computation, the overhead of JShield is small compared to Java because this time the active components receive small primitive values as parameters. Nevertheless, when an active component finishes a part of the work, it must transmit the partial result to its creator. This result consists of a big integer object which is represented as a byte array and has a size of 1 to 8 kilobytes. Therefore the overhead is explained because this array is deep copied when transmitted.

In general, parallel programming in JShield will not be efficient on shared memory multi-processor machines for fine-grained parallelism or when components need to exchange too much information, as the matrix multiplication example. On the other hand, JShield will be very useful for applications where the bottleneck is the disk or the network, but not the processor.

5 Related Work

The first proposal of an abstraction for the design of data race free languages was the *critical region* of Brinch Hansen [3]. The `synchronized` statement of Java is very close to the critical region concept. Later, Hoare improved on critical regions and introduced in [9] the *monitor* abstraction. In Java, the objects of a class are monitors when the methods of the class are declared with the `synchronized` attribute.

Brinch Hansen pointed out in [4] that although Java claims to have monitors, they are not true monitors, at least not in the sense that monitors were originally conceived.

[4] Available at
`http://www.netlib.org/benchmark/linpackjava/`.

[5] JShield does optimize the access of final static variables of a primitive type or the type String.

A key aspect of monitors was that a language with monitors would be data race free as long as its compiler ensured that (i) shared data declared inside a monitor could only be manipulated by the monitor's procedures, and (ii) data declared outside of a monitor could only be accessed by a single thread. The problem with this approach is that it requires that programming languages do not offer pointers to dynamically allocated structures. Given that this requirement is too constraining, especially for object oriented languages, all current languages are not data race free and rely on the programmer's abilities to avoid data races. This is the case of Java. JShield was designed to be data race free while still being strongly based on pointers.

Another approach to avoid data races is dynamic analysis of programs. This approach is used in Eraser [16], a tool that rewrites the target program to track every shared memory reference and verify that consistent locking behavior is observed. This tool reports warning messages during the execution of the program consisting of the file and line number where a potential data race exists.

Static analysis of programs has also been explored. Flanagan and Freund presented in [8] a tool called Rccjava for statically detecting races, which tracks at compile time the protecting lock for each field in an object and reports every time a field is accessed without first requesting the appropriate lock. This tool relies on the programmer to aid the process of analysis by providing a small number of additional type annotations, stating that a field is protected by a given lock. This mechanism was shown to be an effective approach for eliminating data races. However it does not ensure that programs will be data race free, since the programmer may annotate –by mistake– a field with the wrong lock.

The advantage of Eraser and Rccjava over JShield is that they do not restrict the concurrency patterns the programmer can use, nor impose an overhead in execution time to the final application. In the worst case, they could report false data races, but the programmer can ignore them if he is confident about the correctness of his program.

Other works on static detection of data races are [17, 10, 1]. They are discussed in [8]. Previous works on dynamic detection are [7, 12, 13, 15].

The closest approach to ours is Guava [2]. Like JShield, Guava is a concurrent programming language, based on Java and designed to be data race free. Guava is more complex than JShield because programmers must include special annotations when writing code, as for example in formal parameters and method declarations. These annotations are required to minimize the overhead of Guava compared to Java. Therefore, Guava is a good choice when processor efficiency is a primary goal but JShield is good when programming simplicity is more important than efficiency.

Another difference between Guava and JShield is the treatment of static variables. In Guava static variables are shared by monitors (equivalent to our shielded components), but their access is automatically synchronized through a lock to avoid the problems associated with memory consistency models. The problem with this approach is the following : a desirable feature for a language is that code written for sequential programs, with no synchronization, should work when used by a parallel program. This is not true in Guava, not even in Java, when the library uses static variables. For this reason, in JShield static variables are not shared by shielded components.

Finally, Guava does not allow a monitor to reference an object owned by another monitor. Only monitors can be referenced from other monitors. In JShield a shielded component can reference an object owned by another shielded component if it is a shared object. We think that this feature is important because sometimes, to simplify programming, it is useful to group several shared objects inside the same shielded component. As the group will be controlled by the same lock, all operations will be served sequentially. In Guava, those entities must be put on separated monitors, constraining programmers to deal with the additional parallelism, which perhaps is not required from the point of view of performance.

Isolates [5] are also close to shielded components. They will be introduced in J2SDK version 1.5. Isolates are components not sharing their static variables and running on the same JVM, just like shielded components. However isolates have different goals than shielded components. The main goal of isolates is to provide strict isolation between different applications, typically requested by different users. Each application is run on an isolate and if it fails, other applications running on the same JVM are ensured to continue to work properly. On the other hand, shielded components are meant to be the building blocks for a single user application, and therefore strict isolation is not required. Another difference is that, unlike shielded components, isolates are not data race free because multiple threads are allowed to run concurrently on the same isolate.

Incommunicado [14] is an improvement of isolates adding fast communication between isolates. This communication is modeled on Java/RMI, just like communication in JShield. However, communication is more efficient in JShield because when a shielded component calls another component, the call is served by the same thread making the request, avoiding unnecessary thread context-switches if the component is free. On the other hand, when an isolate calls another isolate, two thread context-switches are mandatory because the request is always served on a different thread running on the called isolate. This is a constraint mandated by the strict isolation provided by isolates.

6 Conclusions

We introduced a dialect of Java, JShield, having the following features : (i) is data race free, (ii) has a simple semantics as compared to other languages being data race free, (iii) preserves Java syntax and the semantics of sequential programs.

The main contribution of this work is to show that it is feasible to design and implement a concurrent programming language based on the metaphor consisting in combining Hoare's monitors, process not sharing memory, and remote method invocations to enforce programmers to write programs which are data race free.

A promising aspect of JShield, which we have not explored yet, is running programs on clusters of processors not sharing memory. The shielded components of JShield offer a uniform semantics when they are executed on shared memory multi-processors as well as when they are executed on non shared memory multi-processors. We plan to develop a JShield implementation for clusters where the processor allocation of components will be decided through an aspect language.

Another interesting property of shielded components is that specifying a semantics for the migration of active shielded component is straight-forward. Hence the difficulty for such a migration in JShield is transferred to the implementations issues, such as for example, modifying the JVM to get access to thread stacks. Note however that the problem in JShield is much more affordable than trying to migrate Java threads, where it is even impossible to specify a clear semantics.

Acknowledgements. I would like to thank Claudio Gutiérrez for his contribution during the preparation of this paper.

References

[1] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 243–354, 1998.

[2] D. Bacon, R. Strom, and A. Tarafdar. Guava: A dialect of java without data races. *ACM SIGPLAN Notices*, 35(10), October 2000. In *OOPSLA'00 Conference Proceedings: Object-Oriented Systems, Languages, and Applications*.

[3] P. Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[4] P. Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.

[5] G. Czajkowski. Application isolation in the java(tm) virtual machine. In *Proceedings of the ACM conference on Object-oriented programming, systems, languages, and applications*, pages 354–366, Minneapolis, Minnesota, USA, Oct. 2000.

[6] M. Dahm. Byte code engineering. In *Proceedings of Java-Informations-Tage*, pages 267–277, Sept. 1999.

[7] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Seattle, Wash.)*, pages 1–10, 1990.

[8] C. Flanagan and S. N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, May 2000. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, PLDI 2000 (Vancouver, Canada).

[9] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–577, October 1974.

[10] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. Technical Report 1999-002, Compaq Systems Research Center, 1999. Also appeared in *Formal Techniques for Java Programs*, workshop proceedings, Bart Jacobs, Gary T. Leavens, Peter Muller and Arnd Poetzsch-Heffter, editors., Technical Report 251, Fernuniversitat Hagen, 1999.

[11] L. Mateu and J. M. Piquer. Safe-threads: a new model for object-oriented multi-threaded languages. In *Proceedings of the XIX International Conference of the Chilean Society of Computer Science*, Talca, Chile, 1999.

[12] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the Supercomputer Debugging Workshop (Albuquerque, N. Mex.)*, pages 1–16, 1991.

[13] R. H. B. Netzer. *Race Condition Detection for Debugging Shared-Memory Parallel Programs*. Ph.D. thesis, Univ. of Wisconsin-Madison, Madison, Wisconsin, 1991.

[14] K. Palacz, J. Vitek, G. Czajkowski, and L. Daynas. Incommunicado: Efficient communication for isolates. In *Proceedings of the ACM conference on Object-oriented programming, systems, languages, and applications*, pages 262–274, Seattle, Washington, USA, 2002.

[15] D. J. Perkovic and P. Keleher. Online data race detection via coherency guarantees. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (Seattle, Washington)*, pages 47–58, 1996.

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[17] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the USENIX Winter Technical Conference*, pages 97–106, 1993.