High-Level Data Races

Cyrille Artho[1], Klaus Havelund[2], and Armin Biere[1]

[1] Computer Systems Institute, ETH Zurich, Switzerland
[2] Kestrel Technology, NASA Ames Research Center,
Moffett Field, California USA

Abstract. Data races are a common problem in concurrent and multi-threaded programming. They are hard to detect without proper tool support. Despite the successful application of these tools, experience shows that the notion of data race is not powerful enough to capture certain types of inconsistencies occurring in practice. In this paper we investigate data races on a higher abstraction layer. This enables us to detect inconsistent uses of shared variables, even if no classical race condition occurs. For example, a data structure representing a coordinate pair may have to be treated atomically. By lifting the meaning of a data race to a higher level, such problems can now be covered. The paper defines the concepts *view* and *view consistency* to give a notation for this novel kind of property. It describes what kinds of errors can be detected with this new definition, and where its limitations are. It also gives a formal guideline for using data structures in a multi-threading environment.

1 Introduction

Multi-threaded (concurrent) programming is becoming increasingly popular in enterprise applications and information systems [2,13]. The Java programming language [1], for example, explicitly supports this paradigm [11]. Multi-threaded programming, however, provides a potential for introducing intermittent concurrency errors that are hard to find using traditional testing. The main source of problem is that a multi-threaded program may execute differently from one run to another due to the apparent randomness in the way threads are scheduled. Since testing typically cannot explore all schedules, some bad schedules may never be discovered. One kind of error that often occurs in multi-threaded programs is a *data race*. In this paper we shall go beyond the traditional notion of a data race, and introduce a higher level notion of data races, together with an algorithm for detecting such. The algorithm has been implemented in the Java PathExplorer (JPaX) tool [9], which provides a general framework for instrumenting Java programs, and for monitoring and analyzing execution traces. The principles and theory presented are, however, universal and apply in full to concurrent programs written in languages like C and C++ as well [14].

The traditional definition of a data race is as follows [15]: *A data race occurs when two concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous*. Consider for example two threads T_1 and T_2, that both access a shared object containing a counter variable *x*, and assume that both threads call an *increase()*

method on the object, which increases *x* by 1. The *increase()* method is compiled into a sequence of bytecode instructions (load *x* to the operand stack, add 1, write back the result), which by the Java Virtual Machine (JVM) is executed non-atomically. Suppose the two threads call *increase()* at nearly the same time and that each of the threads execute the *load* instruction first, which loads the value of *x* to the thread-local operand stack. Then they will both add 1 to the original value, which results in a combined increment of 1 instead of 2. We shall refer to this traditional notion of data race as a *low-level data race*, since it focuses on a single variable.

The standard way to avoid low-level data races on a variable is generally to protect the variable with a lock: all accessing threads must acquire this lock before accessing the variable, and release it again after. In Java, methods can be defined as `synchronized` which causes a call to such a method on an object to lock the object. Return from the method will release the lock. Java also provides an explicit statement form: `synchronized(`*obj*`){`*stmt*`}`, for taking a lock on the object *obj*, and executing statement *stmt* protected under that lock. If the above mentioned *increase()* method is declared `synchronized`, the low-level data race cannot occur.

Several algorithms and tools have been developed for analyzing multi-threaded programs for low-level data races. The Eraser algorithm [15], which has been implemented in the Visual Threads tool [7] to analyze C and C++ programs, is an example of a dynamic algorithm that examines a program execution trace for locking patterns and variable accesses in order to predict potential data races. The Eraser algorithm maintains a *lock set* for each *variable*: the set of locks protecting the variable. In this paper we shall turn this around and study the *variable set* associated to a *lock*. This notion makes it possible to detect what we shall refer to as *high-level data races*. The original inspiration for this problem was due to an example provided by Doug Lea in [12], and which is presented in modified form in Section 2. It defines a simple class representing a coordinate pair with two components *x* and *y*. All accesses are protected by synchronization on `this`, using `synchronized` methods. Therefore, data race conditions on a low level are not possible. As will be illustrated, there can, however, in this example be data races on a higher level, and this can be detected as inconsistencies in the granularity of *variable sets* associated to locks. The algorithm for detecting high-level data races is a dynamic execution trace analysis algorithm like the Eraser algorithm [15].

Beyond Eraser, several static analysis tools exist that examine a program for low-level data races. The Jlint tool [2] is such an example. The ESC [5] tool is also based on static analysis, or more generally on theorem proving. It, however, requires annotation of the program, and does not appear to be as efficient as the Eraser algorithm in finding low-level data races. Dynamic tools have the advantage of having more precise information available in the execution trace. More heavyweight dynamic approaches include model checking, which explores all possible schedules in a program. Recently, model checkers have been developed that apply directly to programs (instead of just on models thereof), for example the Java PathFinder system (JPF) developed by NASA [8,17], and similar systems [6,4,10,3,16]. Such systems, however, suffer from the state space explosion problem. Furthermore, a data race, low-level as well as high-level, can be hard to find even with model checking since it typically needs to cause a violation of some explicitly stated property.

The paper is organized as follows. Section 2 introduces the problem of high-level data races. Section 3 presents the concepts for detecting high-level data races. Section 4 describes the implementation and experiments carried out. Section 5 outlines future work and Section 6 concludes the paper.

2 The Problem of High Level Data Races

Lock protection for a shared field ensures that no concurrent modification is possible. However, this only refers to low-level access of the fields, not their entire use or their use in conjunction with other fields. The remainder of this paper assumes detection of low-level data races is covered by the Eraser algorithm [15], which can be applied in conjunction with our analysis.

```
class Coord {
  double x, y;
  public Coord(double px, double py) { x = px; y = py; }
  synchronized double getX() { return x; }
  synchronized double getY() { return y; }
  synchronized Coord getXY() { return new Coord(x, y); }
  synchronized void setX(double px) { x = px; }
  synchronized void setY(double py) { y = py; }
  synchronized void setXY(Coord c) { x = c.x; y = c.y; }
}
```

Fig. 1. The `Coord` class encapsulating points with x and y coordinates.

A program may be inconsistent even when it is free of low-level data races. Fig. 1 shows a class implementing a two-dimensional coordinate pair with two fields *x*, *y*, which are guarded by a single lock. If only `getXY`, `setXY`, and the constructor are used by any thread, the pair is treated atomically. However, it is obvious that the versatility offered by the other accessor (`get`/`set`) methods is dangerous: if a thread only uses `getXY` and `setXY` and relies on complete atomicity of these operations, threads using the other accessor methods falsify this assumption.

Imagine a case where one thread reads both coordinates while another one sets them to zero. If the write operation occurs in two phases, `setX` and `setY`, the other thread may read an *intermediate result* which contains the value of *x* already set to zero but still the original *y* value. This is clearly an undesired and often unexpected behavior. We will use the term *high-level data race* to describe this kind of scenario.

Nevertheless, there exist scenarios where some of the other access methods are allowed and pair-wise consistency is still maintained. The novel concept of *view consistency* captures this notion of consistency while allowing partial accesses. In previous work, only the use of *locks for each variable* has been considered. The opposite direction, the use of *variables under each lock*, is the core of our new idea.

Fig. 2 shows another example with three threads, which is abbreviated for better readability. View consistency does not need to distinguish between read and write ac-

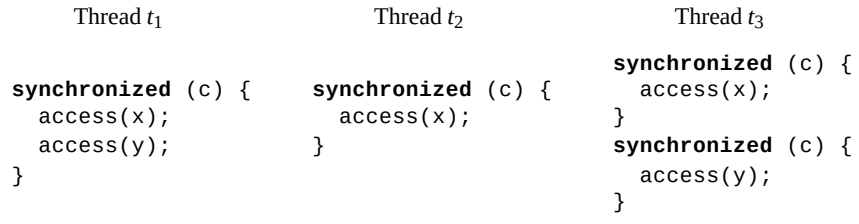

Fig. 2. One thread using a pair of fields and two other threads accessing components individually.

cesses, as will be shown after introducing the new example. Reading and writing are abstracted as `access(f)`, where `f` is a shared field. Calls of synchronized methods offering access protection are represented using `synchronized (lock) { access(f); }` as an abstraction of the inlined method. Thread creations are not shown. Any control structures within each thread are hidden as well. Furthermore, it is assumed that each field accessed by a thread is a reference to a shared object, visible to all threads.

Initially, we only consider the first two threads t_1 and t_2. It is not trivial to see whether an access conflict occurs or not. As long as t_2 does not use y as well, it does not violate the first thread's assumption that the coordinates are treated atomically. Even though t_1 accesses the entire pair $\{x,y\}$ atomically and t_2 does not, the access to x alone can be seen as a partial access or partial update. A read access to x may be interpreted as reading $\{x,y\}$ and discarding y; a write access may be seen as writing to x while leaving y unchanged. So both threads t_1 and t_2 behave in a consistent manner.

Each thread is allowed to use only a part of the coordinates, as long as that use is consistent. Inconsistencies arise with thread t_3, which uses x in one operation and y in another operation, releasing the lock in between. If for example thread t_3 *reads* its data in two parts, with another thread like t_1 writing to it in between, t_3 may obtain partial values corresponding to two *different* global states. If on the other hand thread t_3 *writes* its data in two parts, other threads, like t_1, may read data corresponding to an *intermediate* state. The difficulty in analyzing such inconsistencies lies in the wish to still allow partial accesses to sets of fields, like the access to x of thread t_2.

3 A Solution Based on View Consistency

This section defines *view consistency*. It lifts the common notion of a data race on a single shared variable to a higher level, covering sets of shared variables and their uses.

3.1 Views

A lock *guards* a shared field if it is held during an access to that field. A lock may guard several shared fields. Views express what fields are guarded by a lock. Let I be the set

of object instances generated by a particular run of a Java program. Then F is the set of all fields of all instances in I.

A *view* $v \in \mathbf{P}(F)$ is a subset of F. Let l be a lock, t a thread, and $B(t,l)$ the set of all `synchronized` blocks using lock l executed by thread t. For $b \in B(t,l)$, a view *generated by* t with respect to l, is defined as the set of fields accessed in b by t. The *set of generated views* $V(t) \subseteq \mathbf{P}(F)$ of a thread t is the set of all views v generated by t. In the previous example in Fig. 2, thread t_1 using both coordinates atomically generates view $v_1 = \{x,y\}$ under lock $l = \mathtt{c}$. Thread t_2 only accesses x alone under l, having view $v_2 = \{x\}$. Thread t_3 generates two views: $V(t_3) = \{\{x\},\{y\}\}$.

3.2 Views in different threads

A view generated by a thread is a *maximal view*, $v \in M(t)$, iff it is maximal with respect to set inclusion in $V(t)$:

$$v_m \in M(t) \quad \text{iff} \quad \forall v \in V(t)\ [v_m \subseteq v \rightarrow v_m = v]$$

Only two views which have fields in common can be responsible for a conflict. This observation is the motivation for the following definition. Given a set of views $V(t)$ generated by t and a view v' generated by another thread, the *overlapping views* of t with v' are all non-empty intersections of views in $V(t)$ with v':

$$\text{overlap}(t,v') \equiv \{v' \cap v \mid v \in V(t) \wedge v \cap v' \neq \emptyset\}$$

A set of views $V(t)$ is *compatible* with another thread's maximal view v_m, iff all overlapping views of t with v_m form a chain:

$$\text{compatible}(t,v_m) \quad \text{iff} \quad \forall v_1, v_2 \in \text{overlap}(t,v_m)\ [v_1 \subseteq v_2 \vee v_2 \subseteq v_1]$$

View consistency is defined as mutual compatibility between all threads: A thread is only allowed to use views that are compatible with the maximal views of all other threads.

$$\forall t_1 \neq t_2,\ v_m \in M(t_1)\ [\text{compatible}(t_2,v_m)]$$

In the example, we had $V(t_1) = M(t_1) = \{\{x,y\}\}$, $V(t_2) = M(t_2) = \{\{x\}\}$, $V(t_3) = M(t_3) = \{\{x\},\{y\}\}$. There is a conflict between $t_1$and t_3 as stated, since $\{x,y\} \in M(t_1)$ intersects with the elements in $V(t_3)$ to $\{x\}$ and $\{y\}$, which do not form a chain.

The problem shown above can be generalized to sets of fields and locks. If there are several locks taken in nested `synchronized` blocks protecting a field, the lock in the outermost `synchronized` block is relevant for the atomicity of the field accesses. This is because inner locks belong to the access implementation of that data, but not to the high-level usage of it.

3.3 Incompleteness of this approach

Essentially, this approach tries to infer what the developer intended when writing the multi-threaded code. It can discover inconsistencies in the code, but an inconsistency

does not automatically imply a fault in the software. *False positives* (spurious warnings) are still possible if a thread uses a coarser locking than actually required by operation semantics. This may make the code shorter or achieve a better performance, since locking and unlocking can be expensive. Releasing the lock between two independent operations requires splitting one `synchronized` block into two blocks. *False negatives* (missed faults) are possible if all views are consistent, but the locking is still insufficient. Assume a set of fields that must be accessed atomically, but is only accessed one element at a time by every thread. Then no view of any thread includes all variables as one set, and the view consistency approach cannot find the problem.

4 Experiments

The experiments were all made with JPaX [9], a run-time verification tool consisting of two parts: an instrumentation module and an observer module. The instrumentation module produces an instrumented version of the program, which when executed generates an event log with the information required for the observer to determine the correctness of the examined properties. The observer of the events used here only checks for high-level data races. For these experiments, a new and yet totally un-optimized version of JPaX was used. It instruments every field access, regardless of whether it can be statically proven to be thread-safe. Because of this, some data-intensive applications created log files which grew prohibitively large (> 0.5 GB) and could not be analyzed.

Four applications were analyzed. Those applications include a discrete-event elevator simulator, and two task-parallel applications: SOR (Successive Over-Relaxation over a 2D grid), and a Travelling Salesman Problem (TSP) application. The latter two use worker threads [11] to solve the global problem. Many thanks go to Christoph von Praun who kindly provided these examples, which were referred to in [18]. In addition, a Java model of a NASA planetary rover controller, named K9, was analyzed. The original code is written in C++ and contains about 35,000 lines of code, while the Java model is a heavily abstracted version with 7,000 lines. Nevertheless, it still includes the original, very complex, synchronization patterns.

Table 1 summarizes the results of the experiments. All experiments were run on a Pentium III with a clock frequency of 750 MHz using Sun's Java 1.4 Virtual Machine, given 1 GB of memory. Only applications which could complete without running out of memory were considered. It should be noted that the overhead of the built-in Just-In-Time (JIT) compiler amounts to 0.4 s, so a run time of 0.6 s actually means only about 0.2 s were used for executing the Java application. The Rover application could not be executed on the same machine where the other tests were run, so no time is given there.

It is obvious that certain applications using large data sets incurred a disproportionally high overhead in their instrumented version. Most examples passed the view consistency checks without any warnings reported. For the elevator example, two false warnings referred to two symmetrical cases. In both cases, three fields were involved in the conflict. In thread t_1, the views $V(t_1) = \{\{1,3\},\{3\},\{2,3\}\}$ were inconsistent with the maximal view $v_m = \{1,2,3\}$ of t_2. While this looks like a simple textbook example, the interesting aspect is that one method in t_1 included a *conditional* access to field 1. If that branch had been executed, the view $\{2,3\}$ would actually have been $\{1,2,3\}$,

Application	Run time, uninstrumented [s]	Run time, instrumented [s]	Log size [MB]	Warnings issued
Elevator	16.7	17.5	1.9	2
SOR	0.8	343.2	123.5	0
TSP, very small run (4 cities)	0.6	1.8	0.2	0
TSP, larger run (10 cities)	0.6	28.1	2.3	0
NASA's rover controller K9	-	-	-	1

Table 1. Analysis results for the given example applications.

and there would have been no inconsistency reported. Since not executing the branch corresponds to reading data and discarding the result, the warning is a false positive.

One warning was also reported for the NASA K9 rover code. It concerned six fields which were accessed by two threads in three methods. The responsible developer explained the large scope of the maximal view with six fields as an optimization, and hence it was not considered an error.

5 Future work

There are many areas in which this work can be expanded. They can be classified into technical and theoretical problems.

On the technical side, there are still issues with the run-time analysis tool JPaX. The code instrumentation and event generation does not always provide a reliable identification of objects. It relies on name, type, and hash code of objects. The latter can change during execution, which causes difficulties in the observer. Nonetheless, the hash code is the best identification which is easily obtainable in Java.

Furthermore, the instrumentation has to be optimized with respect to statically provable thread-safety. For instance, read-only or thread-local variables do not have to be monitored. Apart from that, the observer analysis could run on-the-fly without event logging. This would certainly eliminate most scalability problems. Additionally, the current version reports the same conflict for different instances of the same object class.

On the theoretical side, it is not yet fully understood how to properly deal with nested locks. The views of the inner locks cause conflicts with the larger views of the outer locks. These conflicts are spurious. Finally, the elevator case study has shown that a slightly different, control-flow independent definition of view consistency is needed. Perhaps static analysis may be better suited to check such a revised definition.

6 Conclusions

Data races denote a concurrent access to shared variables where an insufficient lock protection can lead to a corrupted program state. Classical, or low-level, data races concern accesses to single fields. Our new notion of high-level data races deals with accesses to sets of fields which are related and should be accessed atomically.

View consistency is a novel concept considering the association of variable sets to locks. This permits detecting high-level data races that can lead to an inconsistent program state, similar to classical low-level data races. Experiments on a small set of applications have shown that developers seem to follow the guideline of view consistency to a surprisingly large extent. We think this concept, which is now formally defined, captures an important underlying idea in multi-threading design.

References

1. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
2. C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-threaded Java Programs. In D. Grant, editor, *Proc. 13th ASWEC*, pages 68–75. IEEE Computer Society, 2001.
3. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Italy, 2001.
4. J. Corbett, M. B. Dwyer, J. Hatcliff, C. S. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proc. 22nd International Conference on Software Engineering*, Ireland, 2000. ACM Press.
5. D. L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, California, USA, 1998.
6. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, France, 1997.
7. J. Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In *7th SPIN Workshop*, volume 1885 of *LNCS*, pages 331–342. Springer, 2000.
8. K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
9. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proc. First International Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*, pages 97–114, France, 2001. Elsevier Science.
10. G. Holzmann and M. Smith. A Practical Method for Verifying Event-Driven Software. In *Proc. ICSE'99, International Conference on Software Engineering*, USA, 1999. IEEE/ACM.
11. D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.
12. D. Lea. Personal e-mail communication, 2000.
13. Sun Microsystems. Java 2 Platform Enterprise Edition Specification. `http://java.sun.com/j2ee`.
14. B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly, 1998.
15. 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, 1997.
16. S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.
17. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proc. ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, 2000.
18. C. von Praun and T. Gross. Object-Race Detection. In *OOPSLA*, pages 70–82. ACM, 2001.