Using Block-local Atomicity to Detect Stale-value Concurrency Errors

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

[1] Computer Systems Institute, ETH Zürich, Switzerland
[2] Kestrel Technology, NASA Ames Research Center, USA

Abstract. Data races do not cover all kinds of concurrency errors. This paper presents a data-flow-based technique to find stale-value errors, which are not found by low-level and high-level data race algorithms. Stale values denote copies of shared data where the copy is no longer synchronized. The algorithm to detect such values works as a consistency check that does not require any assumptions or annotations of the program. It has been implemented as a static analysis in JNuke. The analysis is sound and requires only a single execution trace if implemented as a run-time checking algorithm. Being based on an analysis of Java bytecode, it encompasses the full program semantics, including arbitrarily complex expressions. Related techniques are more complex and more prone to over-reporting.

1 Introduction

Multi-threaded, or concurrent, programming has become increasingly popular, despite its complexity [2]. The Java programming language [1] explicitly supports this paradigm while leaving the programmer a lot of freedom for utilizing it [16]. Multi-threaded programming, however, provides a potential for introducing intermittent concurrency errors that are hard to find using traditional testing. The main source of this 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,* as defined below. Traditionally this term has denoted unprotected field accesses, which will be referred to as *low-level data races.* However, the absence of low-level data races still allows for other concurrency problems, such as *high-level data races* [3] and *atomicity violations* [11,22].

Both high-level data races and previously presented atomicity checks suffer from the fact that they show violations of common conventions, which do not necessarily imply the presence of a fault. High-level data races further suffer from the fact that data flow between protected regions (`synchronized` blocks in Java) is ignored. Our approach complements low-level and high-level data races by finding additional errors, while still being more precise than previous atomicity-based approaches.

The algorithm was designed to avoid the need of a specification or an exhaustive search of the program state space [10]. It detects stale-value errors as defined by Burrows and Leino [8] and augments existing approaches concerning low-level and high-level data races and can be employed in conjunction with these analyses. The algorithm is fully automated, requiring no user guidance beyond normal input. The actual fault analyzed does not have to occur in the observed execution trace, which is why the algorithm is more powerful than traditional testing techniques. The algorithm is very modular and thus suitable for static analysis. Compared to other atomicity-based approaches, it is simpler yet more precise because it captures data flow and thus models the semantics of the analyzed programs more precisely. It is related to [8] but models the full semantics of Java bytecode, including arithmetic expressions. The checking algorithm is implemented as a dedicated algorithm in JNuke [4]. Preliminary experiments show that it is about two orders of magnitude faster than Burrows' prototype.

1.1 Low-level Data Races

The traditional definition of a (low-level) data race is as follows [19]:

> A data race can occur 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.

Without synchronization, it is theoretically possible that the effect of a write operation will never be observed by other threads [14]. Therefore it is universally agreed that low-level data races must be avoided. Several algorithms and tools have been developed for detecting low-level data races, such as the Eraser algorithm [19], which has been implemented in the Visual Threads tool [15].

The standard way to avoid low-level data races on a variable is to protect it 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 to lock the current object instance. Return from the method will release the lock. Java also provides an explicit form of specifying the scope and lock of synchronization using `synchronized{`*lock*`}{`*stmt*`}`, for taking a lock on object *lock*, and executing statement *stmt* protected under that lock. If the above unprotected methods are declared `synchronized`, the low-level data race cannot occur.

1.2 High-level Data Races

A program may contain a potential for concurrency errors, even when it is free of low-level data races. The notion of high-level data races refers to sequences in a program where each access to shared data is protected by a lock, but the program still behaves incorrectly because operations that should be carried out *atomically* can be interleaved with conflicting operations [3].

```
public void swap() {
  int oldX;
  synchronized (lock) {
    oldX = coord.x;
tent state (0, y)
    coord.x = coord.y; // swap X
    coord.y = oldX;    // swap Y
  }
}
```

```
public void reset() {
  synchronized (lock) {
    coord.x = 0;
  } // inconsis-

  synchronized (lock) {
    coord.y = 0;
  }
}
```

Fig. 1. A high-level data race resulting from three atomic operations.

Figure 1 shows such a scenario. While the `swap` operation on the two coordinates x and y is atomic, the `reset` operation is not. Because the lock is released after setting x to 0, other threads may observe state $\langle 0, y\rangle$, an intermediate state, which is inconsistent. If `swap` is invoked by another thread before `reset` finishes, this results in final state $\langle y, 0\rangle$. This is inconsistent with the semantics of `swap` and `reset`. The view consistency algorithm finds such errors [3].

1.3 Atomic Sequences of Operations

The absence of low-level and high-level data races still allows for other concurrency errors. Figure 2 shows such an error: The increment operation is split into a read access, the actual increment, and a write access. Consider two threads, where one thread has just obtained and incremented the shared field. Before the updated value is written back to the shared field, another thread may call method `inc` and read the *old* value. After that, both threads will write back their result, resulting in a total increment of only one rather than two.

The problem is that the entire method `inc` is not *atomic*, so its outcome may be unexpected. Approaches based on reduction [11,22] detect such atomicity violations. The algorithm presented here uses a different approach but also detects the error in the example. Moreover, it is conceptually simpler than previous atomicity-based approaches and at the same time more precise.

```
public void inc() {
  int tmp;
  synchronized (lock) {
    tmp = shared.field;
  } // lock release
  tmp++;
  synchronized (lock) {
    shared.field = tmp;
  }
}
```

Fig. 2. A non-atomic increment operation.

1.4 Outline

Section 2 gives the intuition behind our algorithm. Section 3 formalizes the property to be checked, and Section 4 extends the algorithm to nested locks and recursion. The precision of this new algorithm is discussed in Section 5. Section 6 discusses related work. Section 7 shows initial experiments, and Section 8 concludes.

2 Our Data-flow-based Algorithm

The intuition behind this algorithm is as follows: Actions using shared data, which are protected by a lock, must always operate on *current* values. Shared data is stored on the heap in shared fields, which are globally accessible. Correct synchronization ensures that each access to such shared fields is exclusive. Hence shared fields protected by locks always have current values.

These values are accessed by different threads and may be copied when performing operations such as an addition. Storing shared values in local variables is common practice for complex expressions. However, these local variables retain their original value even when a critical (`synchronized`) region is exited; they are not updated when the global shared field changes. If this happens, the local variable will contain a *stale* value [8] which is inconsistent with the global program state.

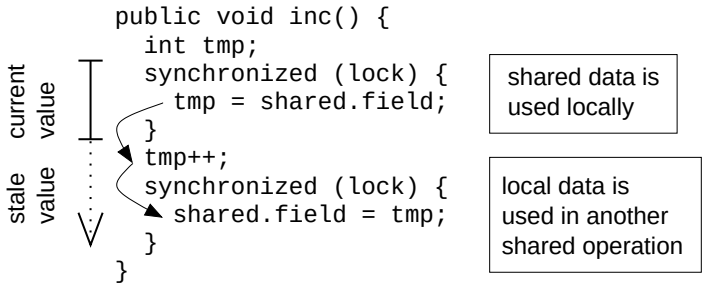

Fig. 3. Intuition behind our algorithm.

Figure 3 shows how the error from the previous example is discovered by our new algorithm. A shared field is assigned to a local variable `tmp`, which is again used later, outside the `synchronized` block. The value of the shared field thus "escapes" the `synchronized` block, as indicated by the first arrow. While the lock is not held, other threads may update the shared field. As soon as the original thread continues execution (in computations, method calls, or assignments), effects of its actions may depend on a stale value. The second arrow indicates the data flow of the stale value.

Note that we use an uncommon notion of escape analysis. Usually escape analysis is concerned with *references* escaping from a certain scope or region [5,7,9,23]. In our algorithm, escaping *values* are considered, not just references, and the scope of interest are `synchronized` blocks.

The lack of a single synchronization scope for the entire sequence of operations is responsible for having stale values. Hence, if the entire method had been `synchronized`, it would have consisted of a single block, which would have executed atomically. Our algorithm uses existing `synchronized` blocks to verify whether shared data escapes them. It therefore requires synchronization to be present for accesses to shared fields. The assumption that each field access itself is properly guarded against concurrent access can be verified using Eraser [19].

Like Eraser and the high-level data race algorithm [3], our new algorithm only requires one execution trace if implemented as a run-time verification algorithm. Furthermore, the property is entirely thread-local. A static implementation of the algorithm is therefore straightforward. If aliases of locks are known, method-local static analysis can verify the desired property for each method while requiring only summary information about other methods. Static analysis has the advantage of being able to symbolically examine the entire program space.

A dynamic analysis on the other hand has precise information about aliases of locks. However, a particular execution typically cannot cover the entire behavior of a program. Even though the probability of actually observing erroneous states in a multi-threaded program is small, dynamic analysis algorithms are often capable of detecting a potential error even if the actual error does not occur [3,19]. The reason is that the property which is checked against (such as locking discipline) is stronger than the desired property (e.g. the absence of a data race). The algorithm presented here also falls into that category.

3 Formalization of Our Algorithm

This section gives a precise formalization of our algorithm. The algorithm is explained without going into details about nested locks and method calls. These two issues are covered in the next section.

In Java, each method invocation frame contains an array of variables known as its *local variables* and a fixed-size stack holding its *stack variables.* These two kinds of variables are always thread-local [14]. Both kinds of variables will be referred to as registers r. A shared field f will denote a field of a dynamic object instance which is accessed in a shared context, using lock protection.

A *monitor block* encompasses a range of instructions: Its beginning is the lock acquisition (`monitorenter`) of a new lock. Its end is marked by the corresponding lock release (`monitorexit`). It is assumed that lock acquisitions and releases are nested as required by the Java semantics [1]. Each monitor block has a unique ID b distinguishing individual lock acquisitions. Reentrant lock acquisitions and releases have no effect on mutual exclusion and are ignored.

A register is *shared* when it contains the value of a shared field f and *unshared* otherwise. When shared, the monitor block in which the shared value originated

is also recorded. The *state* $\mathrm{s}(r) = \langle sh, b \rangle$ of a register comprises its shared status $sh \in \{0, 1\}$ and its monitor block ID b. The *current monitor block* b_{curr} is the block corresponding to the latest non-reentrant lock acquisition.

At the beginning of execution, all registers are unshared. There are two possibilities to obtain a shared value: First, a `getfield` instruction within a monitor block will produce a shared value. Second, a method invocation may return a shared value. Shared values have state $\langle 1, b_{curr} \rangle$. We will use two auxiliary functions returning the first and second part of a state s, respectively: shared(s) and monitorblock(s).

Each assignment of a value will carry over the state of the assigned value. Operations on several values will result in a shared value if any of the operands was shared. A register r is *used* by an instruction i, $r \in \mathrm{used}(i)$, if it is read by it. If r is a stack element, the corresponding stack argument is consumed when it is read, according to the Java bytecode semantics [14]. If r is a local variable, reading it does not have any further effect. Note that this definition of usage includes expressions and arithmetic operations. In expression `tmp2 = tmp1 + tmp0`, the result `tmp2` is shared if any of the operands `tmp1` or `tmp0` is shared.

A *stale value* is a value of a shared register that originated from a different monitor block than where it is used. This can be formalized as follows: A program uses no stale values iff, for each program state and each register r used by current instruction i, the following holds: monitor block of that register, $s(r)$, must be equal to the current monitor block:

$$\forall i,\, r \cdot (r \in \mathrm{used}(i) \wedge \mathrm{shared}(s(r)) \rightarrow (\mathrm{monitorblock}(s(r)) = b_{curr})$$

If a single operation uses several shared values with different monitor block IDs b, then at least one of them must be a stale value. This property is then violated, and the result of that operation is again a shared value. [3] We will refer to this property as *block-local atomicity*. If it holds for the entire program, then actions based on shared data will always operate on current data.

4 Extension to Nested Locks and Recursion

The assumption behind dealing with nested locks is that any locks taken beyond the first one are necessary to ensure mutual exclusion in the nested `synchronized` blocks. This is a natural assumption arising from the program semantics: nested locks are commonly used to access shared fields of different objects, which use different locks for protection. Let l_{outer} and l_{inner} denote an outer and an inner lock, respectively. Assume a thread acquires l_{inner} when already holding l_{outer}. It then accesses a shared field f holding both locks. After releasing l_{inner}, the shared field is no longer protected by that nested lock and may thus be updated

[3] In our implementation we marked the result of any such operation as unshared. The operation already generates a warning. Resetting the state of that register prevents generating more than one warning for any stale value.

by other threads. This usage of stale values outside the nested lock l_{inner} violates block-local atomicity.

Low-level data race detection like Eraser misses this error, because each field access operation is properly protected. Block-local atomicity detects that the shared value becomes stale outside the inner monitor block. The following treatment of nested locks covers such errors: The algorithm declares a separate monitor block for each nested lock. If any operation outside the inner block uses a shared value such as f, this will be detected by the consistency check defined in the previous section.

Using the shared data from f outside the inner block would only be safe if l_{inner} was superfluous: If l_{inner} was always used only in conjunction with l_{outer}, then l_{inner} would not contribute to protection against concurrent access. Instead the extra lock would constitute an overhead that should be eliminated, and the warning issued by our algorithm can help to identify this problem.

Because a new monitor block is used with each lock acquisition, the total number of locks held when acquiring a new lock l_{inner} is not relevant. Thus the idea generalizes to a set of outer locks L_{outer} instead of a single outer lock l_{outer}.

When dealing with *method calls,* only the effect of data flow and `synchronized` blocks has to be considered. In run-time analysis, this is implemented trivially as method calls do not have to be treated specially.[4] In static analysis, method calls are essentially inlined, using only summary information of called methods. If no new synchronization is used by the called method, the method call has no special effect and behaves like a local operation. Otherwise, if a new (non-reentrant) lock is used by the callee, the return value will be shared with a new unique monitor block ID. Hence the return value of a call to a `synchronized` method is shared, unless the caller itself used the same lock during the call, which would make the inner lock merely reentrant.

Because of this treatment of nested locks, handling inter-method data flow is quite natural and very efficient. The analysis does not have to consider calling contexts other than the lock set held. A context-insensitive variant of the algorithm is easily created: One can simply assume that any locks used in called methods are distinct. The algorithm will still be sound but may emit more false warnings. The same assumption can be used if the effect of a called method is unknown, e.g. when a method is native.

Finally, in a static implementation of the algorithm, the temporary lock release in a `wait()` operation has to be modeled explicitly [8]. For run-time verification in JNuke [4], the lock release event is implicitly generated by its run-time verification API [4].

5 Precision and Limitations of Our Algorithm

If a program is free of data races, our algorithm finds all stale values but may issue false warnings. Atomicity-based approaches, including this one, are sometimes

[4] A call to a synchronized method is treated like a block using `synchronized(this)`.

too strict because certain code idioms allow that the globally visible effect of a non-atomic operation corresponds to an atomic execution. *Serializability* is a more precise property, but even non-serializable programs can be correct.

5.1 Soundness and Completeness

Our algorithm assumes that no low-level data races are present. This kind of error can be detected by algorithms like Eraser [19]. If a program is free of (low-level) data races then our static analysis algorithm is sound; no faults are missed. In a static approximation of this analysis, however, the alias information about locks is not always known. If one assumes each lock acquisition utilizes a different lock, the algorithm remains sound but becomes more prone to overreporting. Furthermore, soundness is also preserved if it is assumed that any unknown method called returns a shared value belonging to a monitor block of its own. If the algorithm is implemented dynamically, then soundness depends on the quality of a test suite and can usually not be guaranteed.

False positives may be reported if too many distinct monitor blocks are created by the analysis. A possible reason is the creation of more locks than actually necessary to ensure mutual exclusion. However, assuming that synchronization primitives are only used when necessary, then the algorithm will not report false positives, in the following sense: each reported usage of a shared value in a different monitor block actually corresponds to the use of a stale value.

5.2 Precision Compared to Previous Atomicity-based Approaches

Block-local atomicity is more precise than method-level atomicity as used by previous approaches [11,13,18,22]. These approaches check for the atomicity of operations and assume that each method must execute atomically. This is too strict. Non-atomic execution of a certain code block may be a (welcome) optimization allowing for increased parallelism. Our algorithm detects whether such an atomicity violation is benign or results in stale values. Furthermore, it does not require assumptions or annotations about the desired scope of atomicity.

Our algorithm uses data flow to decide which regions must necessarily be atomic. At the same time, the analysis determines the size of atomic regions. Therefore block-local atomicity reports any errors found by earlier atomicity-based approaches but does not report spurious warnings where no data flow exists between two separated atomic regions.

Figure 4 shows an example that illustrates why our algorithm is more precise. A program consists of several threads. The one shown in the figure updates a shared value once a second. For instance, it could read the value from a sensor and average it with the previously written value. It then releases the lock, so other threads can access and use this value. A reduction-based algorithm will (correctly) conclude that this method is not atomic, because the lock is released during each loop iteration. However, as there is no data flow between one loop iteration and the next one, the program is safe. Our algorithm analyzes the program correctly and does not emit a warning.

```
void sensorDaemon() {
  while (true) {
    synchronized (lock) {
      value = shared.field; // acquire latest copy
      value = func (value);
      shared.field = value; // write back result
    }
    sleep(1000); // wait
  }
}
```

Fig. 4. The importance of data flow analysis for `synchronized` blocks.

5.3 Limitations of Atomicity-based Approaches

The strict semantics of atomic operations and block-local atomicity are not always required for a program to be correct. This creates a potential for warnings about benign usages of stale values. An example is a logging class using lax synchronization: It writes a local copy of shared data to its log. For such purposes, the most current value may not be needed, so block-local atomicity is too strict.

Finally, conflicts may be prevented using higher-level synchronization. For instance, accesses can be separated through thread `start` or `join` operations [15]. This is the most typical scenario resulting in false positives. Note that other atomicity-based approaches will always report a spurious error in such cases as well. The segmentation algorithm can eliminate such false positivies [15].

5.4 Serializability

Even without higher-level synchronization, block-local atomicity is sometimes too strong as a criterion for program correctness. Serializability is a weaker but still sufficient criterion for concurrent programs [10]. Nevertheless, there are cases involving container structures where a program is correct, but neither atomic nor serializable. Consider Figure 5, where a program reads from a buffer, performs a calculation, and writes the result back. Assume `buffer.next()` always returns a valid value, blocking if necessary. After a value has been returned, its slot is freed, so each value is used only once. Method `buffer.add()` is used to record results. The order in which they are recorded does not matter in this example.

The reason why the program is correct is because the calculation does *not* depend on a stale *shared* value; "ownership" of the value is transferred to the current thread when it is consumed by calling `buffer.next()`. Thus the value becomes thread-confined and is no longer shared. This pattern is not captured by our data flow analysis but is well-documented as the "hand-over protocol" [16]. It could be addressed with an extension to the approach presented here, which checks for thread-local confinement of data.

```
public void work() {
  int value, fdata;
  while (true) {
    synchronized (lock) {
      value = buffer.next();
    }

    fdata = f(value); // long computation

    synchronized (lock) { // Data flow from previous block!
      buffer.add(fdata);  // However, the program is correct because
    }                     // the buffer protocol ensures that the
  }                       // returned data remains thread-local.
}
```

Fig. 5. A correct non-atomic, non-serializable program.

6 Related work

Our algorithm builds on previous work on data races. It has been designed to detect errors that are not found by data race analysis. The algorithm is related to previous work on atomicity violations but is an independent approach to that problem. The data flow analysis used in our algorithm is at its core an escape analysis, although it uses different entities and scopes for its analysis.

6.1 Data races

Low-level data races denote access conflicts when reading or writing individual fields without sufficient lock protection [19]. For detecting data races, the *set of locks* held when accessing shared fields is checked. High-level data races turn this idea upside down and consider the *set of fields* accessed when holding a lock. View consistency serves as a consistency criterion to verify whether these accesses are semantically compatible [3].

Block-local atomicity is a property which is independent of high-level data races. Figure 1 in the introduction showed that certain faults result in high-level data races but do not violate block-local atomicity. However, the reverse is also possible, as shown in Figure 2, where no high-level data races occur, but stale values are present in the program. Hence the two properties are independent [22]. Both high-level data races and block-local atomicity build on the fact that the program is already free of underlying low-level data races, which can be detected by Eraser [19]. The intent behind block-local atomicity is to use it in conjunction with low-level and high-level data race analyses, because these notions do not capture atomicity violations.

6.2 Atomicity of Operations

Atomicity of operations is not directly concerned with data accessed within individual critical (`synchronized`) regions, but with the question whether these regions are sufficiently large to guarantee atomic execution of certain operations. Atomicity is a desirable property in concurrent programs [11,13,18,22]. In conjunction with the absence of data races, program correctness with respect to concurrently accessed data can be guaranteed.

The key idea is to reduce sequences of operations to serializable (atomic) actions based on the semantics of each action with respect to Lipton's reduction theory [17]. In Figure 2, the actions of the entire increment method cannot be reduced to a single atomic block because the lock is released within the method. The reduction-based atomicity algorithm verifies whether an entire shared method is atomic. Recent work includes a run-time checker that does not require annotations [11]. A different approach to verify the atomicity of methods extends the high-level data race algorithm with an extra scope representing the desired atomicity of a method [18]. Block-local atomicity is more precise than such previous approaches, as shown in section 5. At the same time it is conceptually simpler, because modeling the data flow of instructions is much simpler than deciding whether a sequence of instructions is atomic.

Atomicity only by itself is not sufficient to avoid data corruption.[5] However, augmenting data race checks with an atomicity algorithm finds more errors than one approach alone.

6.3 Stale Values

The kind of error found by our algorithm corresponds to stale values as defined by Burrows and Leino [8] but is an independent approach to this question. Our algorithm compares IDs of monitor blocks to verify whether a register contains stale shared data. Their algorithm uses two flags *stale* and *from_critical* instead, which must by updated whenever a register changes. Unlike their approach, which is based on source code annotation, we model the semantics of Java bytecode directly. This covers the full semantics of Java, including method calls and arithmetic expressions. This allows us to discover potential non-determinism in program output, when registers are written to an output. Their approach misses such an error as it involves the use of a register in a method call. Furthermore, we have a dedicated checker for this property, which is orders of magnitude faster than their prototype which uses the ESC/Java [12] framework that was targeted to "more heavy-weight checking" [8].

6.4 Serializability

Atomicity is sometimes too strong as a desired property. Atomic blocks are always serializable, but correct programs may be serializable but not atomic

[5] Flanagan and Qadeer ignored the Java memory model when claiming that low-level data races are subsumed by atomicity [13].

[10]. Serializability, while weaker than atomicity, still suffices to guarantee the consistency of thread-local and global program states. Code idioms exist where operations are performed on outdated values but still yield the same result as if they had been performed on the current value, because of double-checking.

```
public void do_transaction() {
  int value, fdata;
  boolean done = false;
  while (!done) {
    synchronized (lock) {
      value = shared.field;
    }

    fdata = f(value); // long computation

    synchronized (lock) {
      if (value == shared.field) {
        shared.field = fdata;
        // The usage of the locally computed fdata is safe because
        // the shared value is the same as during the computation.
        // Our algorithm and previous atomicity-based approaches
        // report an error (false positive).
        done = true;
      }
    }
  }
}
```

Fig. 6. A code idiom that cannot be analyzed with block-local atomicity.

Figure 6 derived from [10] shows a code idiom suitable for long computations: A shared value is read and stored locally. A complex function is then computed using the local copy. When the result is to be written back, the writing thread checks whether the computation was based on the current value. If this was the case, the result is written; otherwise the computation is repeated with a new copy of the shared value. Note that even in a successful computation, the shared value may have been changed in between and re-set to its original value. Thus this operation is non-atomic but still serializable, and therefore correct.

Atomicity-based approaches, including this one, will report an error in this case [11,21,22]. Flanagan's definition of atomicity only entails visible effects of an operation; in this sense, the program is atomic but irreducible [10]. On the other hand, the program violates block-local atomicity because its *action* is not atomic. It is merely wrapped in a (potentially endless) loop that creates the impression of atomicity. There is currently no approach other than model checking [20] to decide whether a program is serializable. This observation does not diminish the

value of our algorithm because model checking is still much too expensive to be applied to large programs [4,10,20].

6.5 Escape Analysis

Our data flow analysis is related to escape analysis, see [5,7,9,23] and the more recent [6], in the sense that it determines whether some entity escapes a region of interest. In our case entities are values (primitive as well as references), and the regions are synchronization sections. For example, if the content of a field x is 5 and this value is read inside a synchronized section, and then later used outside this region, then that value has escaped. In traditional escape analysis on the other hand, typically entities are references to heap-allocated objects (not primitive values, such as an integer) and regions are methods or threads. In our case, the analysis is simpler because modeling the effect of each instruction on the stack and local variables is straightforward.

7 Experiments

A preliminary version of a static analyzer that checks for block-local atomicity has been implemented in JNuke [4]. So far it can only analyze small examples because the analyzer cannot yet load multiple class files. Due to this, relevant parts of benchmark packages had to be merged into one class. The check for lock use in called methods was not yet implemented and was performed manually in trivial cases. These limitations will be overcome in the final version.

Benchmark	Size [LOC]	PraunGross [18]	FlanaganFreund [11]	Block-local atomicity
Elevator	500	2	2	2
SOR	250	0	0	0
TSP	700	1	7	1

Table 1. Comparison to other approaches: Number of warnings reported.

Besides a few hand-crafted examples used for unit testing, three benchmark applications [18] were analyzed: A discrete-event elevator simulator and two task-parallel applications, SOR (Successive Over-Relaxation over a 2D grid) and the Travelling Salesman Problem (TSP). Table 1 shows the results. The three warnings issued by all approaches are benign: In the elevator example, the two warnings refer to a case where a variable is checked twice, similarly to the example in Figure 6. For TSP, the warning refers to an access inside a constructor, where the data used is still thread-local.

Our approach necessarily reports fewer atomicity violations than the run-time checker from Flanagan and Freund [11]. This can be expected since block-local atomicity implies method-local atomicity, and thus the number of violations

of method-local atomcity constitutes an upper bound to the number of block-local atomicity violations. It remains to be seen how much the reports from our algorithm will differ on larger benchmarks in comparison to [18].

Compared to a previous prototype checker for stale values [8], our checker is significantly faster. Burrows reported 2000 source lines (LOC) per minute on unspecified hardware. JNuke checked a binary resulting from 500 lines in 0.02 s, on a Pentium 4 with a clock frequency of 2.8 GHz. Accounting for different hardware, a difference of about two orders of magnitude remains.

8 Conclusions and Future Work

We have presented a data-flow-based algorithm to detect concurrency errors that cannot be detected by low-level [19] or high-level [3] data races. Previous atomicity-based approaches were entirely based on the atomicity of operation sequences, but ignored data flow between `synchronized` blocks [11,18,22]. This results in cases where correct non-atomic methods are reported as faulty. The algorithm presented in this paper detects stale values [8]. This conceptually simpler and more precise property captures data flow between `synchronized` blocks. The property can be checked in a thread-modular, method-local way. It can be implemented as a static analysis or as a run-time checking algorithm.

Future work includes investigating the relationship to Burrows' algorithm in more depth. Our algorithm currently issues a warning when a stale register is used even though the use of such a snapshot may be benign. Burrows' more relaxed reporting could be more useful for pratical purposes. Extensions to the algorithm include coverage of thread-locality of data and higher-level segmentation [15] of events. It remains to be seen how easily the algorithm translates into a run-time implementation. A major challenge for a run-time analysis of this algorithm is the fact that each instruction has to be monitored, creating an impossibly large overhead for instrumentation-based analysis. However, our JNuke framework is capable of handling efficient low-level listeners that could make a run-time algorithm feasible [4]. A static analysis may still be preferable because aliasing information of locks can usually be approximated easily [2].

Acknowledgements

Thanks go to Christoph von Praun for the benchmark applications and for quickly answering questions about the nature of the atomicity violations in them.

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, multithreaded Java programs. In D. Grant, editor, *Proc. 13th ASWEC*, Canberra, Australia, 2001. IEEE Computer Society.

3. C. Artho, K. Havelund, and A. Biere. High-level data races. *Journal on Software Testing, Verification & Reliability (STVR)*, 13(4), 2003.
4. C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: efficient dynamic analysis for Java. In R. Alur and D. Peled, editors, *Proc. CAV '04*, Boston, USA, 2004. Springer.
5. B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proc. OOPSLA '99*, pages 20–34, Denver, USA, 1999. ACM Press.
6. B. Blanchet. Escape analysis for java, theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, November 2003.
7. J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proc. OOPSLA '99*, pages 35–46, Denver, USA, 1999. ACM Press.
8. M. Burrows and R. Leino. Finding stale-value errors in concurrent programs. Technical Report SRC-TN-2002-004, Compaq SRC, Palo Alto, USA, 2002.
9. J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. OOPSLA '99*, pages 1–19, Denver, USA, 1999. ACM Press.
10. C. Flanagan. Verifying commit-atomicity using model-checking. In *Proc. SPIN Workshop (SPIN'04)*, volume 2989 of *LNCS*, Barcelona, Spain, 2004. Springer.
11. C. Flanagan and S. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004.
12. C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI 2002*, pages 234–245, Berlin, Germany, 2002. ACM Press.
13. C. Flanagan and S. Qadeer. Types for atomicity. In *Proc. Workshop on Types in Language Design and Implementation (TLDI'03)*, New Orleans, USA, 2003. ACM Press.
14. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 2000.
15. J. Harrow. Runtime checking of multithreaded applications with Visual Threads. In *Proc. SPIN Workshop (SPIN'00)*, volume 1885 of *LNCS*, Stanford, USA, 2000. Springer.
16. D. Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 1999.
17. Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
18. C.v. Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. In *Proc. Formal Techniques for Java-like Programs*, volume 408 of *Technical Reports from ETH Zürich*. ETH Zürich, 2003.
19. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. on Computer Systems*, 15(4), 1997.
20. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.
21. C. von Praun and T. Gross. Object-race detection. In *OOPSLA 2001*, Tampa Bay, USA, 2001. ACM Press.
22. L. Wang and S. Stoller. Run-time analysis for atomicity. In *Proc. Run-Time Verification Workshop (RV'03)*, volume 89(2) of *ENTCS*, Boulder, USA, 2003. Elsevier.
23. J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. OOPSLA '99*, pages 187–206, Denver, USA, 1999. ACM Press.