USENIX Association

Proceedings of the Third Virtual Machine Research and Technology Symposium

San Jose, CA, USA
May 6–7, 2004

Detecting Data Races using Dynamic Escape Analysis based on Read Barrier

Hiroyasu Nishiyama
Systems Development Laboratory, HITACHI, Ltd.
1099, Ohzenji, Asao-ku, Kawasaki-shi, 215-0013 Japan
nisiyama@sdl.hitachi.co.jp

Abstract

In multi-threaded programs, a data race results in extremely hard to locate bugs because of its non-deterministic behavior. This paper describes a novel dynamic data race detection method for object-oriented programming languages. The proposed method is based on the lockset algorithm. It uses read-barrier-based dynamic escape analysis for reducing number of memory locations that must be checked at runtime for detecting data races.

We implemented the proposed data race detection method in HotSpot Java[1] VM. The results of an experimental evaluation show a significant performance improvement over the previous write-barrier-based method and also that the proposed method can perform data race detection with a relatively small runtime overhead.

1 Introduction

Compared with traditional sequential applications, multi-threaded applications are prone to errors arising from the concurrent nature of program execution. A *data race*[22] is one of the primary causes of unpredictable behavior in multi-threaded programs. Data races occur if two parallel threads access a same memory location without using implicit or explicit synchronization that prevents simultaneous access of the data; and if at least one thread is write access. A program involving data races is sensitive to the dynamic state of its runtime environments such as in thread scheduling or user interactions. This leads to unpredictable behavior of the program. Unfortunately, the cause of this problem is difficult to pinpoint.

Java[16, 28] is a widely used object-oriented programming language that includes support for multi-threading. Java provides thread functionality as a means for describing concurrently executing entities. Threads in Java are actively utilized by applications such as GUI applications and Internet servers.

Java provides language-level synchronization constructs based on the *monitor* as a means for serializing accesses from concurrently executing threads. Language-level integration of the monitor in Java simplifies the specification of critical sections. Although this may decrease the occurrence of synchronization programming errors, the complete prevention of synchronization errors is still hard to achieve for large-scale multi-threaded applications.

Extensive studies on optimizations to eliminate useless synchronizations[15] and on reduction in overhead for uncontended cases[4, 5] have been made; however, synchronization specifications are usually considered to be a source of severe performance overhead. Consequently, programmers sometimes are conservative about using synchronization. This may introduce additional unintended synchronization errors.

Many works have treated automatic detection of data races; and there exists several commercial or open-source products that detects data races[13, 17]. From a theoretical perspective, Netzer and Miller proposed a classification of data races[22]: *actual race*, *feasible race*, and *apparent race*. They have also shown that a general data race detection is NP-hard. Past research on automatic data race detection can be classified as ones involving static, dynamic, or combined methods. The static approaches

[1] Java and other names are trademarks or registered trademarks of SUN Microsystems Inc. in the United States and other countries.

include an approach for adding annotations to a program and proving the correctness with a theorem prover[18], a post-mortem approach that analyzes events log of a program execution[3], and an approach using a language extension incorporating a type system for static race detection[10]. Dynamic approaches examine the access pattern to shared locations on-the-fly verifying its correctness according to some criteria such as the *locking discipline*[6] or *happens-before* relation[19].

In this paper, we propose a novel on-the-fly data race detection method for object-oriented languages based on the *lockset algorithm*. The lockset algorithm dynamically verifies the locking discipline to detect synchronization errors in a given program.

Although the lockset algorithm has been applied to real-life applications, it incurs a severe performance overhead. Our method reduces this overhead by exploiting the intuitive characteristic of program behavior that most objects in a typical object-oriented application do not incur shared access from multiple threads.

We use this property for reducing the number of objects to be examined while a program is being executed. The number of examined objects is limited by using *dynamic escape analysis* based on the *read barrier*. Dynamic object reachability tracking using the read barrier enables a reduction in the number of shared objects compared with the previously proposed write barrier based method.

The rest of the paper is organized as follows. Section 2 summarizes related works on data race detection based on the lockset algorithm. Section 3 describes the new data race detection approach using the read barrier. Section 4 describes its implementation in HotSpot Java VM. Section 5 describes our experimental results on a set of benchmark programs.

2 Detection of Data Races

Modern concurrent or multi-threaded programming frameworks provide structured synchronization primitives such as monitor or communication channel instead of much simpler ones such as spin lock or semaphore.

A monitor employed by Java is a programming language construct that encapsulates shared resources and its access functions. The monitor of Java can represent an exclusive program region using an acquisition and release of an object specified by the synchronized attribute of a method or a synchronized statement.

The lockset algorithm detects synchronization errors in a monitor-based concurrent program. This algorithm assumes that a correct program keeps the following locking discipline:

> *When some thread accesses shared data, the thread must hold at least one monitor. Furthermore, the intersection of the sets of all monitors held when accessing the data must not be empty.*

Let $L(v)$ represent a lockset for a shared memory location v, $E(v)$ represent a set of times when read or write reference events for the memory location v occurred, $T(v, i)$ represent a set of threads that accessed the memory location v at time i, and M_{ti} represent a set of locks that is held by thread t at time i. Then, $L(v)$ can be defined as follows:

$$L(v) = \forall_{i \in E(v)} \forall_{t \in T(v,i)} \cap M_{ti}$$

If the set $L(v)$ becomes empty, we can recognize that at least two accesses from different threads have been performed with nonuniform synchronization. Thus, we can suspect the occurrence of data races at the memory location v.

As an example of a program involving a data race, we will consider the program shown in Figure 1. The `Account` class in this figure defines the method `inc` which increments the field `balance` defined at (a), and also defines the method `dec` which decrements the field `balance`. Consider a case where thread #1 calls `inc` and thread #2 calls `dec`. The call for the `inc` method obtains a monitor with the synchronized statement at (b). Accordingly, the monitorset at the update point (c) of the `balance` is written as {`this`}. In regard to the call for `dec` by thread #2, since the method `dec` does not perform synchronization, the monitor set at update point (d) of the `balance` becomes empty. Calculating the lockset from the monitor set of each thread yields an empty set. Thus, we can suspect the possible occurrence of a data race for the field `balance` in this program.

```
                                                    Monitor Set
                                                    Thread#1   Thread#2   #1∩#2
class Account {
    private double balance;          // (a)         {this}     φ          φ
    ...
    void inc(double diff) {
        synchronized(this) {         // (b)
            balance += diff;         // (c)         {this}
        }
    }
    void dec(double diff) {
        balance -= diff;             // (d)                    φ
    }
}

Thread#1: account.inc(...)
Thread#2: account.dec(...)
```

Figure 1: Example of a program containing a data race.

Eraser[26] is an example of a data race detection tool for pthreads based multi-threaded programs using the lockset algorithm. Eraser detects data races by using a binary rewriting technique to modify all memory reference instructions in a target program, and by monitoring memory references dynamically at runtime by calculating their associated monitor sets.

Since Eraser's target language, C or C++, can access any memory location through pointers, distinguishing data structures that need exclusive access from ones that only need thread private accesses is difficult. Thus, Eraser associates a data structure called *shadow memory* that exists in parallel with the program's accessible memory regions. In addition, it tracks all memory accesses dynamically in the shadow memory. This leads to a huge space and time overhead[2].

An object oriented programming language such as Java abstracts a main memory as a collection of objects instead of a simple sequence of bytes. Several studies have been done on using this characteristic to reduce the overhead of data race detection.

One example is the Object Race Detection[23] of Praun and Gross. It restricts units of data race detection at the level of objects instead of field elements. However, this restriction on race detection leaves open the possibility of false negative or false positive reports because of the ambiguity of field accesses or array element accesses. They also use static escape analysis[1] to decrease dynamic monitoring costs by excluding objects that can be proved as inaccessible from more than one thread.

The escape analysis is a technique for determining that an object is only accessible by a single thread. The escape analysis is also useful for optimizations such as synchronization removal[25], thread-local memory management[8], stack allocation of objects[11] and object inlining[7].

The data race detection system of Choi et al.[2] combines static escape analysis with compiler optimizations. For reducing detection costs, they restrict the detectable combination of data races. This means that their system does not report all access pairs that participate in data races, but guarantees that at least one access is reported.

The TRaDe system of Christiaens et al.[20] is a purely dynamic system based on happens-before relation. It calculates the set of objects that may be referenced from multiple threads by using dynamic escape analysis based on the write barrier.

In the related field of program analysis, Dwyer et al. described an combined approach of dynamic and static escape analysis of state-space reduction for model-checking object-oriented programs[9]. The model-checking is a software verification technique by exploring all possible execution sequences. Although the model-checking can provide more precise information of a program than the lockset algorithm, enumerating execution sequences is much heavy-weight operation.

[2]Savage et al. reported a 10X to 30X execution overhead[26].

3 Reduction in Data Race Detection Cost using Read Barrier

A data race detection method using the lockset algorithm monitors accesses to shared memory locations updating their monitor sets; it checks whether multiple threads accessed the location without obtaining common locks. However, monitoring all field references results in a severe performance penalty, as in Eraser. We reduce the penalty of reference monitoring using escape analysis by checking the reachability of an object from multiple threads. This reduction is accomplished by only tracking locksets for objects that are reachable from multiple threads.

We employed the dynamic method similar to TRaDe for our race detection system. The reason for using dynamic method is as follows: First, the dynamic method does not require complex whole program analysis and optimization framework. Second, the dynamic method can provide more accurate results than the pure static method because the static analysis can not follow exact data flow information. Finally, the dynamic class loading mechanism of Java makes an application of the static analysis difficult because it may invalidates assumption of the analysis. For program optimization, the deoptimization[12] can be used to cancel the incorrect compile time assumptions. In contrast, since a data race detection requires monitoring the history of a program execution, the invalidation of the assumption may result in false positive or false negative errors.

3.1 Dynamic Escape Analysis

An object in the Java virtual machine is only accessible from its allocating thread immediately after its allocation except for special objects such as class objects. Stack and method local variables are thread private resources and are not accessible from other threads[28].

Thus, an object O owned by a single thread becomes accessible from more than one thread when an O's reference or a reference to some object that can reach O by following reference chains is assigned to an object that can be referenced from multiple threads.

The data race detection method of TRaDe assigns each object one of the following two attributes: (a) a global attribute meaning that the object can be reached from multiple threads, and (b) a local attribute meaning that the object can be reached from only one thread. A globally reachable object at the point of its allocation such as class object is marked with a global attribute when it is allocated. When a reference R is assigned to an object of a global attribute, the global attribute is also assigned to the object that is pointed to by R and all objects that can be reached by following references from R.

The method of TRaDe can be regarded as a *write barrier*[24] based dynamic escape analysis method, which tracks object reachability at the point of its reference assignment.

We employed a read-barrier-based dynamic escape analysis method, which updates the reference attribute of each object when its reference is obtained from the object field.

This method reduces the data race detection overhead by calculating reference attributes of an object as follows:

1. A local reference attribute is assigned to an object after its allocation. A creating thread ID is also assigned to each object.
2. For each reference field in an object, if a reference attribute of its pointing object is local and its thread ID is not identical to the referencing thread, the reference attribute of the pointed object is changed to global.

The read barrier and write barrier are techniques that are usually used for maintaining intergeneration references in generational garbage collectors[24]. The write barrier is generally preferred for implementing a garbage collector because write references are less frequent than read references.

When using read or write barrier as a means for dynamic escape analysis, the costs of monitoring data races for global objects must be considered in addition to barrier overhead. Taking the following points into consideration, we can expect that the read barrier based methods will be superior to the write barrier based method:

- **Reduction in global attribute maintenance costs**

When a local object O_l is assigned to a reference field of a global object O_g, the write barrier based method needs to recursively traverse objects that can be reachable from O_l changing reference attributes of the traversed objects to global ones. This requires a high execution overhead on reference assignment and also requires extra memory overhead on the traversal.

The read barrier based method only needs to change the attribute of O_l to global when a thread which is not the owner thread of O_l has obtained a reference to O_l, in other words, when the object O_l has escaped to another thread. In contrast to the write barrier based method, the read barrier based method can perform data race detection with a much lower and definite cost. It also requires no recursive object traversals. Thus, we can expect lower runtime memory requirements.

- **Reduction in monitoring cost**

  The write barrier based method regards objects that can be reachable from global objects as potential candidates of data races. In contrast, the read barrier based method recognizes objects from which references are obtained by multiple threads, that is, objects reached by multiple threads through reference traversals, as potential candidates.

  Therefore, the target objects of the read barrier based method comprise a subset of the write barrier based method. As a result, we can expect a reduction in monitoring overhead of data race detection by using the read barrier.

3.2 Array Objects

A program such as a scientific application frequently uses a parallel processing idiom that divides arrays into multiple regions and assigns each region to separate worker threads. Since each worker thread of such an idiom usually accesses disjoint array regions, no data races can occur. However, since Java treats an array as a special kind of object, previous object based escape analysis consider the whole array as a target of the data race detection even for such an array-dividing idiom.

We extend the data race detection for an array to an array sub-block level to deal with such an idiom. An array is divided into fixed length sub-blocks and locality detection is applied to each array sub-block.

3.3 State Model

The basic idea of the read barrier based method is to classify objects and array sub-blocks into escaped and non-escaped groups at their reference field reading points. Our method follows Eraser[26] and Object Race Detection[23] and introduces an extended state model which mixes objects, array sub-blocks, and object fields as detection units.

Figure 2 shows the state model for an object, an array sub-block, and an object field. This model consists of the following seven states:

Object-specific states:

owned An object can be accessed from its allocating thread only. In the common case, when an object is created, it can be referenced from its creating thread only. Thus, the initial state of a normal object is owned.

escaped A reference to an object is obtained by threads other than its owner thread, that is, ones that have escaped to another thread. Fields of an escaped object should be monitored for race detection by the lockset algorithm.

The sub-block level states for an array are introduced after an array object changes its state from owned to escaped.

Field-specific states:

shared read A field in an escaped object is read by more than one thread. Since concurrent read requests for a field in a shared read state do not cause any data races, no conflict is reported.

shared modified A field in an escaped state is modified by some thread. No conflict is reported for a field with a non-empty lockset in the shared modified state because thread synchronization is expected to have been properly performed.

conflict A field of an escaped object is written by more than two threads or read and written by different threads with an empty lockset. A conflict is reported for the field, since a synchronization error for the field is suspected.

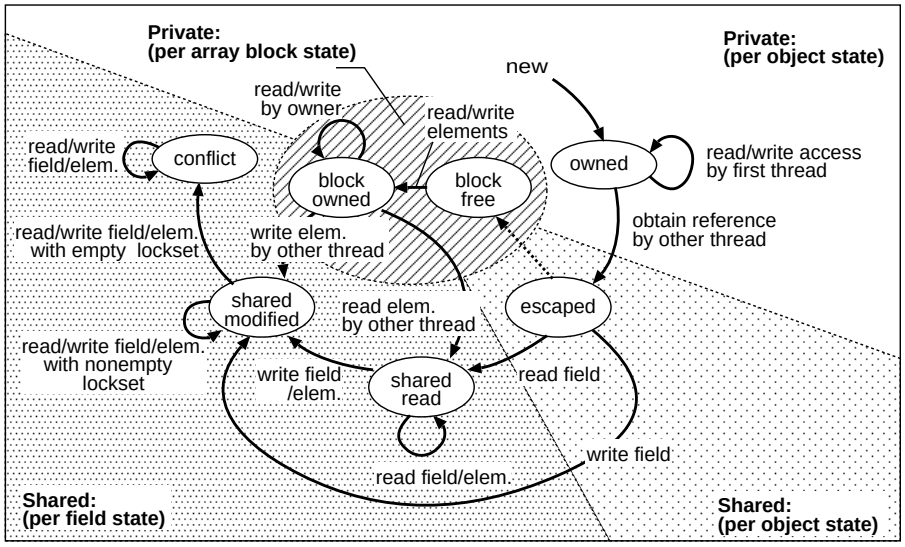

Figure 2: State Model for Objects, Array Blocks, and Fields

Array sub-block specific states:

block free This is the initial state of an array sub-block representing that an array object belonging to it has escaped to a non allocating thread. Since each sub-block element is free from accesses at the point of escape, the sub-block does not yet become subject to race detection by the lockset algorithm.

block owned An array sub-block in a block free state is referenced by some thread. The referencing thread becomes the owner of the sub-block. Following references for the array sub-block in the block-owned state by its owner thread does not initiate element-level data race detection. If a non owner thread accesses a sub-block in the block-owned state, the state of the block changes to the shared read or shared modified state and an element-level race detection by the lockset algorithm is initiated.

The read barrier based race detection method starts lockset computations for fields of an object when more than one thread can reach the object by traversing object references. Hence races may fail to be detected depending on dynamic thread scheduling conditions. However, as Savage et al. pointed out[26], many programs initialize objects without obtaining locks then pass their references to other threads. Therefore, starting a lockset computation after multiple threads reach an object can eliminate many false data race reports originating in object initialization.

4 Implementation

In this section, we describe an implementation of the proposed race detection method on the HotSpot Java virtual machine[21].

4.1 Object Format

Our implementation extends the object format of the base virtual machine, adding an extra field that is used for data race detection. Figure 3 shows the modified object format. The object format of the HotSpot Java VM consists of a two-word header: (a) a mark field that represents object age, hash code, and other information, and (b) a klass field that points to a class object. The modified object format is extended to a three-word header, adding a state field that represents reference attributes and external information about the object. The state field denotes two per object states (owned and escaped), two per array sub-block states (block free and block owned), and a special state for thread synchronization (locked). The owned and block-owned states are represented by making the LSB of the state field zero and recording thread ID to the remaining bit field. The difference between an owned

and a block owned state is distinguished by object's class or the object kind implicitly recognized by the bytecode execution state. The escaped object state is represented by setting the LSB of the state field to one and the remaining state field to a pointer to the external per field information. The state field representation for an escaped array object is similar to the escaped object but it points to an array of sub-block information. Each element of the array keeps similar information on the state field for a normal object.

Three per field reference states (shared read, shared modified, and conflict) are recorded in an external data structure pointed to from the state field of an escaped object or the information array of an array sub-block.

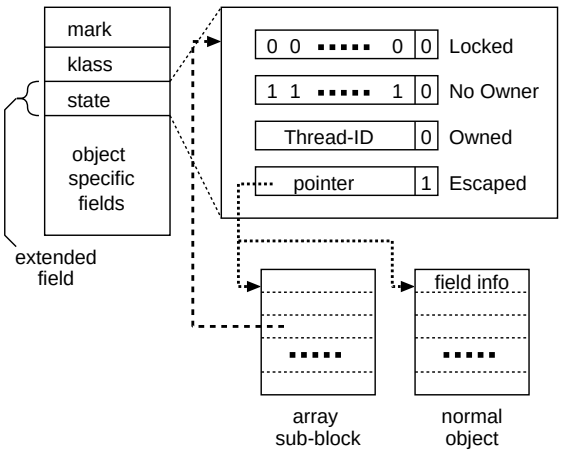

Figure 3: The extended object format including the state field which represents object reference state or points to external data structure.

4.2 Code Sequences

Bytecode execution of HotSpot VM is performed using indirect threading interpreter and dynamic JIT compiler. The threaded interpreter uses machine code sequences that is generated from code templates at interpreter startup.

To detect data races, we modified the machine code templates and compiler generated code sequences for the following bytecodes:

- `monitorenter/monitorexit`

  Machine code sequences for `monitorenter` and `monitorexit` bytecode are modified to record owned monitors in the per thread monitor stack. For the sake of efficiency, the conversion of a monitor stack into a monitor set is delayed until some object enters the escaped state.

- `getfield/getstatic/[ilfdabc]aload`, `putfield/putstatic/[ilfdabc]astore`

  The reading of the reference field by using `getfield` or `getstatic` bytecode is modified to check whether its target object has escaped or not.

  For each field reference or modification, if the target object has escaped attribute, the lockset algorithm is applied for detecting data races on the object. Result of the set calculation between locksets are cached for efficiency as is done by Eraser[26].

We also modified the field references by runtime system in the Java VM and the field references from native methods[3] through the JNI (Java Native Interface).

4.3 Dealing with Special Threads

A finalizer method may be defined in Java to allow work to be performed before an object is reclaimed by the garbage collector. In the HotSpot VM implementation, a finalizer is executed by a special thread called a finalizer thread[4]. The finalizer method is usually defined without explicit synchronization assuming implicit execution ordering through GC. This may increase possibility of false positive reports. Hence, in our implementation, we have excluded references from these special threads from the target of data race detection by default. This exclusion can be turned-off using an runtime option.

5 Experimental Results

In this section, we describe an experimental comparison of our read barrier based method and the write barrier based dynamic method. We used 64 bytes as sub-block size of an array for calculating the shared state.

[3]This modification of field reference includes accesses by undocumented `sun.misc.unsafe` class.

[4]There are other special threads, such as a reference handler thread that deals with weak references.

Application	Description	# threads
compress	Data compression program from SPEC JVM98	1
jess	Java expert shell system from SPEC JVM98	1
db	Memory resident database program from SPEC JVM98	1
javac	Java compiler program from SPEC JVM98	1
mpegaudio	MPEG audio decompression program from SPEC JVM98	1
mtrt	Multi-threaded ray-trace program from SPEC JVM98	3
jack	Java parser generator program from SPEC JVM98	1
SPECjbb	Java business benchmark program	42
Crypt	IDEA encryption program from Java Grande Benchmark	2
LUFact	LU factorization program from Java Grande Benchmark	2
SOR	Successive over-relaxation program from Java Grande Benchmark	2
Series	Fourier coefficient analysis program from Java Grande Benchmark	2
Sparse	Sparse matrix multiplication program from Java Grande Benchmark	2

Table 1: Benchmark Programs

Table 1 lists the benchmarks[14, 27] we used for the evaluation. The '# threads' column indicates the number of dynamically generated threads except system threads such as the finalizer thread. The Java Grande benchmarks are numerical benchmarks, and they perform many array operations on large shared array objects. Since they use the barrier based synchronization method, their manner of synchronization is not standard. However, they can reveal the overhead of race detection for array intensive applications.

Among these benchmarks, our system successfully found a data race on `RayTrace.threadCount` in mtrt benchmark. As reported in [2], this data race does not affect the correctness of the program execution.

All evaluations are performed using the HotSpot Java virtual machine ported to an HP-UX operating system. The environment of the experimental evaluation was HP9000/C3000(PA-8500 400MHz, 250MB main memory) with HP-UX11.0.

Figure 4 shows normalized execution time of the write barrier based method and the read barrier based method[5] compared to normal execution[6].

[5]The implementation of the write barrier based method is similar to the one for read barrier based method except that it recursively check object escape attributes at reference assignment instead of object reference.

[6]HotSpot Java VM optimizes execution of synchronization primitives for uncontended cases. However, since we need to maintain monitor stack at each monitor related bytecode execution, our read barrier and write barrier implementation does not use this optimization.

The write barrier based method requires a large execution overhead (69.7% to 3389.7% for the SPECjvm/jbb benchmarks, and 0.6% to 6777.0% for the Java Grande benchmarks) for detecting data races. In contrast, the read barrier based method can perform data race detection with a much lower overhead (57.8% to 735.7% for the SPECjvm/jbb benchmarks, and 0.6% to 6012.5% for the Java Grande benchmarks). We can see significant performance improvements of the read barrier based method over the write barrier based method on the mpegaudio and jack benchmarks. The performance improvements for these benchmarks indicate an increase in the number of dynamically examined objects because the write barrier based method deals with all globally reachable objects as targets of data race detection. The overheads for detecting data races for several of the Java Grande benchmarks are high compared with the SPEC benchmarks since these benchmarks frequently access shared arrays among threads with disjoint indexes.

To investigate the details of the field references, we obtained a breakdown of read/write references of the object fields. The result is shown in Figure 5. The WB and RB after each benchmark name indicate the write barrier based method and the read barrier based method, respectively. We can see from Figure 5 that the ratio of shared fields references of the write barrier based method are high for six SPEC benchmarks (compress, jess, javac, mpegaudio, jack, and SPECjbb) that have improved performance with the read barrier based method. In particular, nearly 40% of references are treated as targets of data race detection for the mpegaudio benchmark, for which the read barrier based method

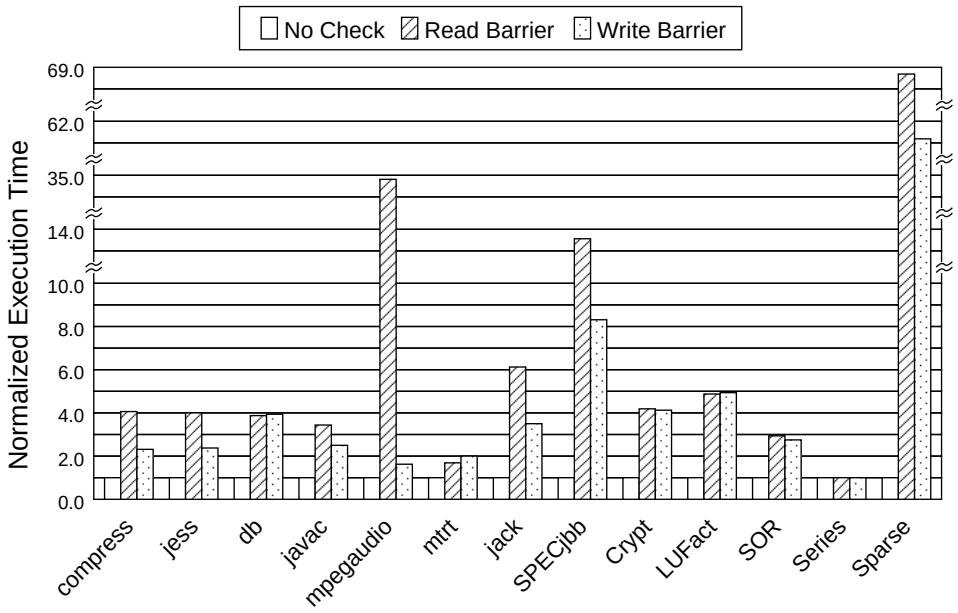

Figure 4: Execution time of the write barrier based method and the read barrier based method normalized to normal execution.

significantly improved performance.

Some SPEC benchmarks such as mtrt decreased the performance slightly by using the read barrier based method. We can see from Figure 5 that shared access ratio of these benchmarks does not decrease by the proposed method. This means that the performance differences arise from the read barrier overhead.

Compared with the SPEC benchmarks, the performance improvements for the Java Grande benchmarks are not as large. This is because the Java Grande benchmarks adopt a programming style that performs many accesses to small numbers of large shared arrays.

To confirm the effectiveness of array sub-block division, we compared the performance of benchmarks with and without array sub-block division. The result is shown in Figure 6. We can see that array sub-block division significantly improved performance of the SOR and Sparse benchmarks. It also made moderate improvements on the benchmarks such as SPECjbb, Crypt, or LUFact.

Division of arrays alone is effective for reducing overheads. For example, the shared field reference ratio of the write barrier based method without array division was almost 100% for mpegaudio, LUFact, and SOR benchmarks. Dividing arrays reduced this ratio to less than half, 1/117 for SOR, of the ratio without division.

Figure 7 shows the per object memory overhead of each race detection method including state field, monitor sets, monitor stacks, and data structures pointed to from the state field. The results of the write barrier based method do not include implicit memory overhead for object traversal on reference assignments. Note that the vertical axis of this graph is a logarithmic scale. The proposed method requires only a small number of bytes per object for the SPEC benchmarks. In contrast, the write barrier based method requires a large memory overhead for SPEC benchmarks such as compress and mpegaudio. Since the Java Grande benchmarks use large array objects, per object extra memory overhead is larger than with the SPEC benchmarks.

6 Conclusions and Future Work

In this paper, we proposed a novel data race detection method for object-oriented multi-threaded languages using dynamic escape analysis based on the read barrier. Data races are sources of errors

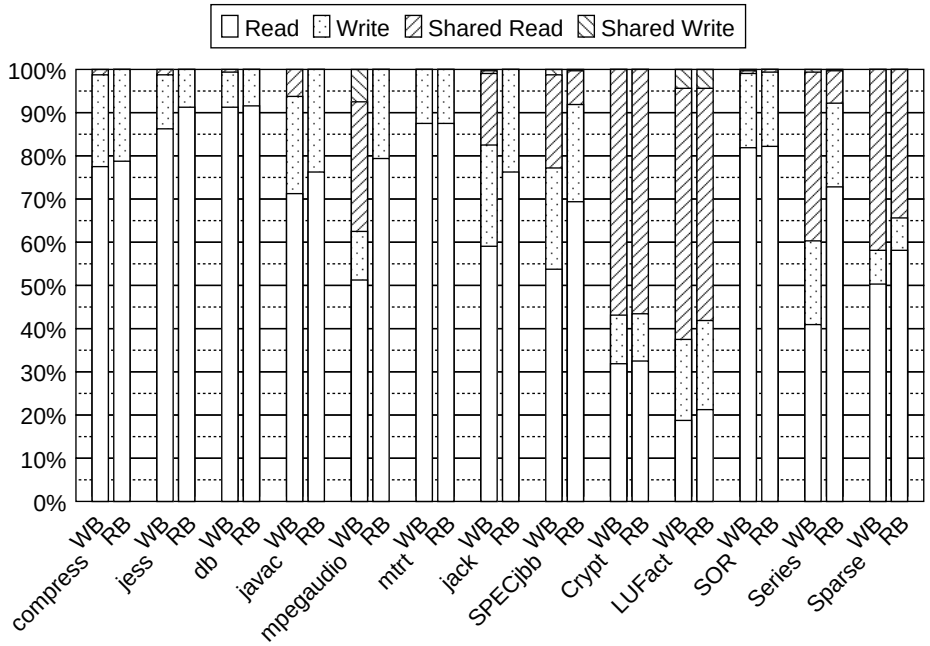

Figure 5: Breakdown of Object References

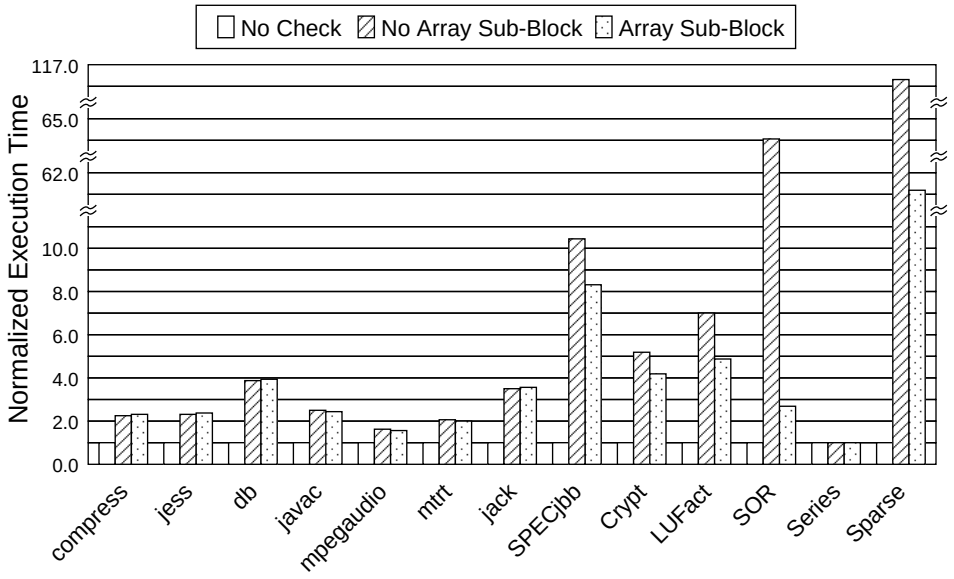

Figure 6: Relative Execution Time of Array Sub-Block Division

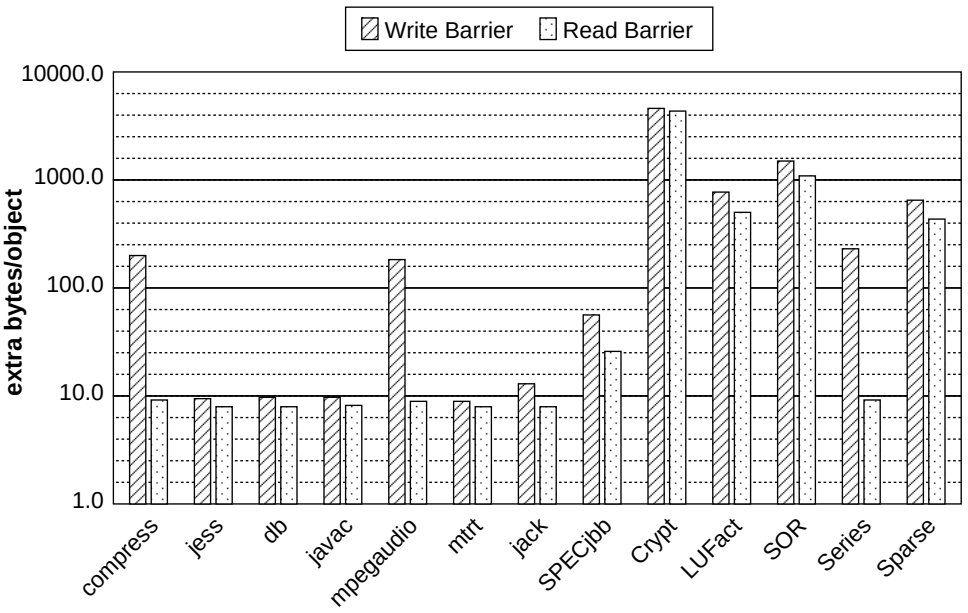

Figure 7: Extra Memory Overhead of Each Detection Method per Object

that are difficult to dissolve. Our method verifies whether a reference to an object is obtained by more than one thread at the point where reference fields are read out. By restricting the targets of data race detection to only escaped objects using the read barrier, we have successfully reduced data race detection overhead.

Our experimental results for a modified HotSpot Java virtual machine show that our method performs data race detection more effectively than the previous write barrier based method, and its detection overhead is small compared with normal execution. In addition to read barrier based filtering of objects, we developed a method that divides an array into sub-blocks as units of race detection. This division of arrays are also improves performance. Using these techniques, we can look for the occurrence of data races while incurring a relatively low overhead, 57.8% to 735.7% for SPEC benchmarks and 0.6% to 6012.5% for Java Grande benchmarks.

In the future, we intend to using static compiler optimizations for improving the execution performance. Static optimization such as common subexpression elimination can remove redundant checking of object/field status. Combination of static/dynamic escape analysis can also improve the performance by decreasing the number of objects that need dynamic checking.

Acknowledgments

We would like to thank the anonymous referees. Their comments were very useful for revising this paper.

References

[1] J.D. Choi. Escape Analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[2] J.D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2001.

[3] J.D. Choi, B. Miller, and R. Netzer. Techniques for Debugging Parallel Programs with Flowback Analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, 1991.

[4] D.Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the SIGPLAN*

Conference on Programming Language Design and Implementation, pages 258–268, 1998.

[5] D. Dice. Implementing Fast Java Monitors with Relaxed-Locks. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2001.

[6] A. Dinning and E. Schonberg. Detecting Access Anomalies in Programs with Critical Sections. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, 1991.

[7] J. Dolby and A. Chien. An Automatic Object Inlining Optimization and its Evaluation. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[8] T. Domani, G. Goldshtein, E.K. Kolodner, and E. Lewis. Thread-Local Heaps for Java. In *Proceedings of the 2002 Internationl Symposium on Memory Management*, 2002.

[9] M.B. Dwyer, J. Hatcliff, V.R. Prasad, and Robby. Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs. Technical Report SAnToS-TR2003-1, SAnToS Laboratory, Kansas State University, 2003.

[10] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[11] D. Gay and B. Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *2000 International Conference on Compiler Construction*, 2000.

[12] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, 1992.

[13] J. Seward and N. Nethercote and J. Fitzhardinge. Valgrind. `http://valgrind.kde.org/`, 2003.

[14] Java Grande Forum. Java Grande Forum Benchmark. `http://www.epcc.ed.ac.uk/javagrande/javag.html`, 2003.

[15] J.Bogda and U. Hölzle. Removing Unnecessary Synchronization in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 35–46, 1999.

[16] J.Gosling, B.Joy, and G.Steele. *The Java Language Specification*. Addison Wesley, 1999.

[17] KAI Software. Tutorial: Using Assure for Threads. `http://www.intel.com/software/products/assure/assuret_tutorial.pdf`, 2001.

[18] K.Rustan, M.Leino, and G. Nelson. An Extended Static Checker for Modula-3. In *Proceedings of 7th International Conference on Compiler Construction, LNCS1383*, 1998.

[19] L. Lamport. Time, clock, and the orderling of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[20] M.Christiaens and K.D.Bosschere. TRaDe, A Topological Approach to On-the-fly Race Detection in Java Programs. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2001.

[21] Sun Microsystems. Java HotSpot Technology. `http://java.sun.com/products/hotspot/`, 2003.

[22] R. Netzer and B. Miller. What Are Race Condition? - Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1), 1992.

[23] C.v. Praun and T.Gross. Object Race Detection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[24] R.Jones and R.Lins. *Garbage Collection - Algorighms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[25] E. Ruf. Effective Synchronization Removal for Java. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalarro, and T.E. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. *ACM Transactions on Computer Systems*, 15(4):391, 411 1997.

[27] Standard Performance Evaluation Corp. SPEC benchmarks. `http://www.spec.org`, 2003.

[28] T.Lindholm and F.Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2000.