Scalable Thread Sharing Analysis

Jeff Huang
Parasol Laboratory
Texas A&M University
jeff@cse.tamu.edu

ABSTRACT

We present two scalable algorithms for identifying program locations that access thread-shared data in concurrent programs. The static algorithm, though simple, and without performing the expensive whole program information flow analysis, is much more efficient, less memory-demanding, and even more precise than the classical escape analysis algorithm. The dynamic algorithm, powered by a location-based approach, achieves significant runtime speedups over a precise dynamic escape analysis. Our evaluation on a set of large real world complex multithreaded systems such as Apache Derby and Eclipse shows that our algorithms achieve unprecedented scalability. Used by client applications, our algorithms reduce the recording overhead of a record-replay system by 9X on average (as much as 16X) and increase the runtime logging speed of a data race detector by 32% on average (as much as 52%).

1. INTRODUCTION

A major difficulty of concurrent programming lies in the complex interaction among threads on shared data. Distinguishing shared data accesses from those thread-local is vital to virtually all concurrency-related problems, from program comprehension, compiler optimization, to testing, debugging, and verification. As examples, if an access is not to shared data, synchronizations surrounding it can be safely eliminated [8, 41], a race detection technique can ignore it as it cannot be involved in any race condition [21], and a replay technique does not need to track it because it cannot introduce non-determinism [20].

Problem. We shall call the problem of localizing shared data accesses, *i.e.*, determining if a program statement can read or write thread-shared data, as *Thread Sharing Analysis* (TSA). Unfortunately, sound and complete TSA is generally undecidable. A classical approach to this problem is *escape analysis* [8, 41], which approximately determines if an allocated object may escape from the method or thread creating it. Escape analysis has attracted more than a decade of active research [8, 41, 31, 9, 12, 28] and has been proven valuable in practice. For instance, it has been used in Oracle JVM for performance optimization [2]: if an object does not escape a method, it will be allocated on the method's stack frame, and if an object does not escape a thread, the synchronization associated with it will be eliminated.

Limitations of Escape Analysis. We argue that escape analysis does not suite the TSA problem well, the focus of which differs in several aspects. First, a thread-escaped object *may not be accessed by multiple threads*. For example, static variables are often considered as escaped, and any object that is reachable from static variables is classified as thread-escaped. However, a static variable may only be accessed by a single thread. Second, an object is thread-escaped *does not mean all data associated with the object are shared*. In object-oriented programs, different threads may access different fields or only part of the fields is shared. Third, standard escape analysis algorithms [8, 41, 18] *do not directly work for array accesses*. Because escape analysis has no information about *array aliases*, it cannot determine if an access `a[x]` is to shared data or not, even though `a` may refer to an escaped array object. Fourth, a thread-escaped object may be *immutable* that it is only read after initialization.

More importantly, escape analysis is expensive (in both time and space) that it is difficult to scale to large programs. The classical algorithm [8, 41] needs to construct an interprocedural *Information Flow Graph* (IFG) for the whole program, and propagates the shared nodes in the graph (initially all static variables and *Runnable* objects are shared nodes). This process is slow as the propagation continues until reaching a fix point, and is highly memory-demanding as it needs to create a summary IFG for every reachable method. For instance, TLOA [18], a state-of-the-art escape analysis implemented in Soot [24], cannot even finish analyzing a toy program in 25 minutes in our experiment.

Our Solution. In this paper, we present two TSA algorithms (a static and a dynamic) for object-oriented programs (exemplified by Java) that scale to real world large multithreaded systems. The static algorithm is *much faster*, less memory-demanding, and even *more precise* than the classical escape analysis [8, 41]. It is sound, object-sensitive, field-sensitive, and works for array objects (but does not distinguish between individual array elements). The algorithm leverages two key insights. First, a field-sensitive analysis can significantly improve precision. Rather than reporting which objects escape a thread and thus might be concurrently accessed, it reports which mutable fields of those objects might be accessed by multiple threads. Sec-

ICSE '16, May 14-22, 2016, Austin, TX, USA

 ISBN 978-1-4503-3900-1/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884811

ond, building an IFG is unnecessary for TSA, instead, the call graph and points-to analysis (which are commonly available in compiler frameworks and are also required by escape analysis) can be directly leveraged to perform TSA. This seems counter-intuitive because existing points-to analyses (*e.g.*, SPARK [25]) often sacrifice precision for efficiency by applying flow-insensitive, context-insensitive algorithms. However, as we will show empirically (Section 6) with extensive evaluation, even built upon context-insensitive points-to analysis and the simple CHA call graph, our algorithm significantly outperforms escape analysis with interprocedural information-flow analysis, for the reasons mentioned earlier.

The limitation of static TSA (as for any static analysis) is that when certain code is not known statically (*e.g.*, in the presence of reflection or dynamic code generation), it may produce unsound results because the call graph is incomplete. The dynamic TSA algorithm is proposed to complement static TSA by tracking shared data accesses dynamically. The key insight is to represent dynamic memory locations in a field-sensitive, but *object-insensitive* way, such that for any program location it suffices to analyze at most two memory accesses performed at that location; and if a program location is marked as shared, analysis of future memory accesses performed at that location can be skipped. Powered by a location-based approach, our algorithm improves the efficiency of a precise dynamic escape analysis algorithm [31, 9] by an order of magnitude, but still achieving close precision (within 20% difference on average).

Highlights of Results. We demonstrate the efficiency and effectiveness of our algorithms with extensive evaluation on a wide range of real world large complex multithreaded programs. By comparing with a state-of-the-art static escape analysis (TLOA) [18] and a precise dynamic escape analysis (DEA) [31, 9], we show that our static TSA algorithm is able to analyze these large programs in a few minutes, whereas TLOA runs either out of memory or timeout in an hour. And our dynamic algorithm is 1.6X–14X faster than DEA while maintaining good precision, whereas DEA cannot even finish by running out of memory on some programs. We have also applied our algorithms in two client applications: a deterministic record-replay system LEAP [20] and a dynamic race detector RVPredict [21]. Our algorithms reduce the recording overhead of LEAP by 9X on average (as much as 16X), speed up the runtime logging of RVPredict by 32% on average (as much as 52%), and enable RVPredict to detect 206 more data races in these large programs.

Our implementations are open source and are publicly available at: https://github.com/parasol-aser/tsa

Contributions. To sum up, our contributions are threefold:

- We present a scalable static thread sharing analysis that significantly improves classical escape analysis.
- We present a scalable dynamic thread sharing analysis that significantly improves precise dynamic escape analysis while maintaining good precision.
- We demonstrate the practicality and scalability of our algorithms with extensive evaluation on real world large systems as well as two client applications.

Roadmap. The rest of the paper is organized as follows: Section 2 presents a motivating example to illustrate the TSA problem and to identify the limitations of escape analysis; Section 3 presents our static and dynamic TSA algorithms; Section 4 describes our implementations of the two algorithms. Section 5 presents a case study of our static TSA algorithm and Section 6 reports our detailed experimental results. Section 7 discusses further improvements of our algorithms. Section 8 reviews the related research and Section 9 concludes the paper.

2. MOTIVATING EXAMPLE

Figure 1 shows a simple example illustrating the TSA problem. The program contains three threads accessing a shared object, `s`, with two instance fields, `x` and `y`, a static field, `z`, and an array reference, `a`. There are in total nine reads/writes to these field/array variables, marked with numbers from ❶ to ❾. The goal of TSA is to find out which of these nine accesses are to thread-shared data. We shall call such an access as shared access point (SAP) in this paper. This information is useful in many applications, for example, when all SAPs in a program are known, a dynamic race detector can eliminate the instrumentations on those non-SAPs, which improves the runtime performance. Note that here the identification of SAPs is *context insensitive*: if an access has multiple runtime instances, as long as any one of them reads or writes thread-shared data, this access is classified as a SAP. Although context sensitive identification may produce more precise results, it is also more expensive and harder to scale, which is not the focus of this paper.

Since all these nine accesses are to the shared object `s`, an escape analysis will conclude that they are all SAPs, because `s` escapes its creating thread `T1`. However, only ❷❸❹❼❾ are true SAPs. ❶ (read `y`) is not a SAP because `y` is immutable (not modified after initialization). ❺ (write `z`) is not a SAP because though `z` is static it is only accessed by thread `T1`. ❻ and ❽ (both read `a`) are not SAPs because `a` is immutable (only read by threads `T1` and `T3` after initialization).

Another major problem with escape analysis is that it relies on interprocedural information flow analysis, which is expensive to compute. For example, for this program, the state-of-the-art escape analysis implementation TLOA [18] in Soot [24] runs out of memory in 25 minutes on a 3.60GHz machine with JDK 7 and 4GB heap. In practice, to use TLOA, JDK libraries are typically excluded from the analysis. However, this is unsound in general (see more detailed discussion in Section 5). For instance, when JDK libraries are excluded, TLOA can finish in a second for this program, but it will classify ❷ ❸ ❹ as non-SAPs, which is wrong. The reason is that without analyzing Java `HashSet` in this example, one cannot determine if there is an information flow from `s` to `o`, and hence would miss the call to the `hashCode` method on `s` by `T2`.

Furthermore, escape analysis does not directly handle array accesses. It can only determine if an array object can escape or not, but not for array indexing operations. As a result, escape analysis may miss shared array accesses. For instance, TLOA identifies the SAP ❼ (`b[0]=1`) as thread-local, even though `b` aliases `a` and ❼ conflicts with ❾ (`a[0]=2`).

3. THREAD SHARING ANALYSIS

In this section, we present our static and dynamic TSA algorithms, and discuss their usages and characteristics.

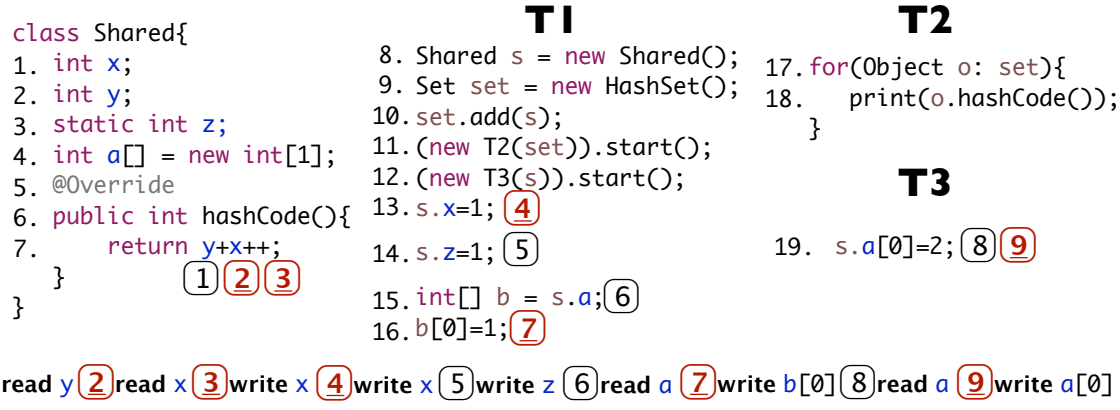

Figure 1: All field and array accesses are numbered. Accesses to shared data are in red.

Algorithm 1 **StaticSharingAnalysis**

```
1: Input: CG - program call graph; // global state
2:        PointsTo - points-to analysis. //global state
3: threads ← FindStaticThreads();
4: for t ∈ threads do
5:   m ← t.entryMethod;
6:   VisitMethod(t, m, ∅);
```

3.1 Static Algorithm

Algorithm 1 outlines our static TSA algorithm. From a high level view, it leverages the call graph and points-to analysis (PTA) of the program and traverses the field and array access statements reachable from each static thread to identify SAPs. It consists of two steps: identifying thread-shared data, and answering the SAP queries. To correctly classify those accesses to thread-shared but *immutable* data, our algorithm also distinguishes between reads and writes.

Specifically, we first find all the static threads in the program that may be executed once or multiple times (**line 3**). This is a standard step also employed by escape analysis such as TLOA [18]. Finding static threads is not difficult in practice because threads are almost always explicitly defined, either at the language level or through common APIs such as PThreads and OpenMP. In cases where threads are not explicit, such as customized user-level threads, developers might still be able to provide annotations for the analysis, since customized threads are likely to be an important aspect in the target program.

We then traverse the statements of each static thread starting from its entry method (*i.e.*, the main method or the run method of a *Runnable* class). In Java, there are four kinds of statements that are relevant to TSA: instance field access, static field access, array access, and method invocation, because only these statements may access or influence thread-shared data. We next describe how they are handled in the procedure **visitMethod** (Algorithm 2) to identify thread-shared data.

Instance field access `o.f` (lines **4-7**). We query PTA to find all the possible abstract objects that the base reference `o` may point to. Depending on the context sensitivity of PTA, the abstract objects may be represented in different ways, for example, it is represented by the object allocation site in the default context-insensitive PTA [25] provided in Soot. Nevertheless, our algorithm is oblivious to the specific representation. For each such abstract object `o'`, we call the procedure **CheckSharing**(t,o,f,$isWrite$) (Algorithm 3) to check if the field `f` of `o'` is shared or not. In **CheckSharing**, we maintain for each object field `o.f` a set of write threads and a set of read threads, retrieved by **GetReadThreads** and **GetWriteThreads**, respectively. If a field f of an object o is accessed by more than one static thread (or by only one static thread that may run more than once[1]), and with at least one of them is a write, we mark `o.f` as shared (via the procedure **MarkObjectFieldShared**(o,f)).

Static field access `f` (lines **9-10**). Static field accesses are handled in a similar way as to that for instance field accesses, except that we set the object `o` to `null`. Note that `f` is represented by its full signature (including the class name), such that each static field is unique.

Algorithm 2 **VisitMethod**(t, m, $visitedMethods$)

```
1: visitedMethods.add(m);
2: for s ∈ m.statements do
3:   switch (s)
4:     case o.f (read/write instance field):
5:       objects ← FindPointsToObjects(o);
6:       for o' ∈ objects do
7:         CheckSharing(t,o',f,read/write);
8:       break;
9:     case f (read/write static field):
10:      CheckSharing(t,null,f,read/write);
11:      break;
12:    case a[i] (read/write array):
13:      objects ← FindPointsToObjects(a);
14:      for o' ∈ objects do
15:        CheckSharing(t,o',null,read/write);
16:      break;
17:    case m(args) (call a new method):
18:      methods ← FindCalleeMethods(CG,m(args));
19:      for m ∈ methods do
20:        if !visitedMethods.contains(m) then
21:          VisitMethod(t, m, visitedMethods);
22:    default: break;
23:  end switch
```

[1]In our implementation, we use a simple `MultiRunStatementsFinder` in Soot based on the CHA call graph to compute `runMoreThanOnce`.

Array access `a[i]` (lines **12-15**). For array accesses, we do not distinguish between different array indexes (unless all array accesses in the program have a constant index value). Although there exists a large body of work (we will discuss in Section 8) that can infer the content of arrays, in general, the value of `i` is undecidable statically. Hence, for practicality, we abstract every array with a single field (represented simply by `null`) and consider all array accesses as to that single field. We then handle array accesses similar to that of instance field accesses.

Method invocation `m(args)` (lines **17-21**). For method invoke statement (the receiver object is also included in the arguments `args`), we first use the call graph to get the possible target methods and then traverse the statements of each of them following the same procedure **visitMethod**.

After all thread-shared data is identified, the analysis result can then be easily queried by client analyses to answer if a statement contains a SAP or not. Algorithm 4 shows our algorithm **QuerySAP** which takes a statement, s, and a set of shared object and field data, D, and returns if s contains a SAP. It checks for three kinds of memory accesses that s may contain:

- Instance field access `o.f`: it first finds all the possible abstract objects that `o` may reference using points-to analysis. For each such object `o'`, it checks if `o'.f` has been marked as shared ($o'.f \in D$). If any of `o'.f` is shared, it returns true.
- Static field access `f`: similar to instance field access `o.f` excepts that `o` is set to *null*.
- Array access `a[i]`: similar to instance field access `a.f` excepts that `f` is set to *null*.
- If s contains none of the above accesses, it simply returns false.

Discussion. Our static TSA algorithm presented above is mostly straightforward. However, we were not able to find it presented in previous literature or implemented in popular compiler frameworks, partly because that the key focus of TSA is different from classical escape analysis. More importantly, being practical and at the same time sound (*i.e.*, no SAP will be mis-classified), our algorithm features in the following ways:

- It does not need to perform the expensive information flow analysis.
- It works at the granularity of field rather than object.
- It handles array accesses as well, *i.e.*, it will not miss any SAP to array object (but does not distinguish different array indexes).
- It does not assume that static field accesses are SAPs by default, but handles them in the same way as instance field accesses.

Note that our algorithm assumes that the call graph and points-to analysis of the whole program are constructed beforehand. For static analysis, this is a reasonable assumption and actually adds to the practical aspect of thread sharing analysis, since call graph and points-to analysis serve as the basis for most whole program analyses and optimization (including escape analysis), and are available in standard compilers such as Soot[2], WALA[3], and LLVM[4]. Moreover, as we will show empirically in Section 6, building call graph and points-to analysis is much less expensive (*i.e.*, faster and consumes less memory) than information flow analysis, which is required by escape analysis.

Algorithm 3 **CheckSharing(t,o,f,$isWrite$)**

```
1: Input: t - current static thread;
2:        o - accessed object; null means static field access
3:        f - accessed field; null means array access
4:        isWrite - true means write and false means read.
5: if isWrite then
6:   if t.runMoreThanOnce then
7:     MarkObjectFieldShared(o,f);
8:   else
9:     wthreads ← GetWriteThreads(o,f);
10:    if wthreads ≠ ∅ && !wthreads.contains(t) then
11:      MarkObjectFieldShared(o,f);
12:    else
13:      rthreads ← GetReadThreads(o,f);
14:      if rthreads ≠ ∅ && !rthreads.contains(t) then
15:        MarkObjectFieldShared(o,f);
16:      else
17:        wthreads.add(t);
18: else
19:   wthreads ← GetWriteThreads(o,f);
20:   if wthreads ≠ ∅ && !wthreads.contains(t) then
21:     MarkObjectFieldShared(o,f);
22:   else
23:     rthreads ← GetReadThreads(o,f);
24:     rthreads.add(t);
```

Algorithm 4 **QuerySAP(s, D)**

```
1: Input: s - a program statement;
2:        D - a set of thread-shared object and field data.
3: Output: true – s contains a SAP.
4: switch (s)
5:   case o.f (access instance field):
6:     objects ← FindPointsToObjects(o);
7:     for o' ∈ objects do
8:       if o'.f ∈ D then
9:         return true;
10:    break;
11:  case f (access static field):
12:    if f ∈ D then
13:      return true;
14:    break;
15:  case a[i] (access array):
16:    objects ← FindPointsToObjects(a);
17:    for o' ∈ objects do
18:      if o' ∈ D then
19:        return true;
20:    break;
21:  default: return false;
22: end switch
23: return false;
```

[2]http://sable.github.io/soot/
[3]http://wala.sourceforge.net/
[4]http://llvm.org/

3.2 Dynamic Algorithm

When call graph and points-to analysis are not available, or their quality is impaired, *e.g.*, by the increasing use of reflection in libraries and frameworks, dynamic TSA could be more favorable in some client applications. For example, for dynamic concurrency bug detection [21, 4], a lightweight dynamic TSA can be run first to identify SAPs for a given program input, and then those SAPs are used in a second run with the same input to find races or atomicity violations. Though the program needs to be run twice, because those non-SAPs are not tracked, the overall performance can often be improved [4].

However, dynamic analysis in general faces two challenges. Besides the well-known unsoundness due to input sensitivity, a complete dynamic analysis often incurs prohibitive runtime overhead (both time and space). Dynamic TSA is no exception. Although it seems obvious to perform TSA when all thread memory access information is available at runtime, precisely tracking every memory access to determine if it accesses thread-shared data is expensive and does not scale to large real world programs. For instance, in our experiments, the precise dynamic escape analysis [31, 9] can slow down the program by as much as 82X and even run out of memory for some programs (*e.g.*, *H2* and *Sunflow*) after a few minutes.

Our dynamic TSA algorithm addresses the runtime overhead problem by making the following observations: although a statement may be executed multiple times at runtime, as long as one of its execution instances accesses thread-shared data, the statement contains a SAP and should be instrumented (*e.g.*, by a dynamic race detector [21] or atomicity violation checker [4]). Hence, our algorithm remembers the program location of each memory access, and uses this information to filter out most redundant checks on accesses to the same program location. This enables two optimizations. First, if a program location is marked as shared, we can skip checking any future memory access performed at that location. Second, we orchestrate all objects by their type and ignore different array indexes, and only analyze for each program location one or two memory accesses performed at that location. After the orchestration, it is sufficient to track memory accesses performed at each program location at most twice. The reason is that the type and field of memory accesses performed at a program location are invariant. Since determining if a program location accesses shared data can involve at most two memory accesses performed at that location, it is hence sufficient to analyze the first two and skip any future memory accesses performed at the location. As we will show empirically in Section 6, this optimization significantly improves the efficiency of TSA in practice, while still achieving good precision.

Algorithm 5 shows our dynamic TSA algorithm. It runs on every dynamic heap access. For each program location, we only track *twice* to check the field or array it accesses, regardless of the runtime object of the field and index of the array. If the same field (or array object) is accessed by two different threads from two different program locations, or twice from the same program location, with at least one write, the field (or array) is marked as shared. For each shared field or array object, all statements accessing it are classified as SAPs. We use a common interface **DynamicSharingAnalysis** (t,loc,o,f,$isWrite$) to track both field and array accesses at runtime on `o.f` by thread t at location loc. For field accesses, `o` is set to *null* as we do not distinguish between different object instances. For array accesses, `o` is the accessed array object and `f` is set to *null* as we do not distinguish different array indexes. If `o.f` is determined shared, we mark all statements that have accessed `o.f` as SAPs (via the procedure **MarkSAPs**(o,f)). For client analyses to use our dynamic TSA result, we can simply save the SAPs into a file upon the termination of the program execution.

Algorithm 5 DynamicSharingAnalysis

Run on every heap access

```
1: Input: t - current thread;
2:        loc - accessed program location;
3:        o - accessed array object; null means field access
4:        f - accessed field; null means array access
5:        isWrite - true means write and false means read.
6: if loc.accessedLessThanTwice then
7:   if !isObjectFieldShared(o,f) then
8:     wthreads ← GetWriteThreads(o,f);
9:     if wthreads ≠ ∅ && !wthreads.contains(t) then
10:      MarkObjectFieldShared(o,f);
11:      MarkSAPs(o,f);
12:      return;
13:    rthreads ← GetReadThreads(o,f);
14:    if isWrite then
15:      if rthreads ≠ ∅ && !rthreads.contains(t) then
16:        MarkObjectFieldShared(o,f);
17:        MarkSAPs(o,f);
18:      else
19:        wthreads.add(t);
20:    else
21:      rthreads.add(t);
```

Discussion. It is worthing noting that in our algorithm each program location is tracked at most twice for all field and array accesses. Because the field `f` is unique to each location, if the array reference `o` at a program location is also insensitive (*i.e.*, always references the same array object), which is often true in practice, there is no need to track the accesses from the same location more than twice.

We also note that our dynamic TSA algorithm is *object-insensitive*, which is crucial for scalability. Although it is unsound (*i.e.*, may miss SAPs in other executions due to the nature of dynamic analysis) and less precise than the precise dynamic escape analysis [31, 9] (*i.e.*, may mis-classify thread-local accesses to different object instances of the same class as shared), it improves its efficiency by an order of magnitude (see Section 6). Our dynamic algorithm is quite promising to be used together with a precise analysis (*e.g.*, in a double-run mode) in applications such as replay [20] and dynamic race detection [21]. We will show in Section 6 that our algorithm reduces the recording overhead of replay by as much as 16X and produces as much as 52% speedup for race detection in our experiments.

4. IMPLEMENTATION

We have implemented our algorithms for Java based on Soot [24] and ASM [1]. The static TSA algorithm is implemented as a whole program analysis phase in Soot. The dynamic TSA algorithm is implemented as a Java agent.

Call graph and points-to analysis. Soot is a popu-

Table 1: Comparison between our static TSA algorithm and TLOA escape analysis on the example program in Figure 1. #SAPs denotes the total number of reported shared data accesses, and (*) denotes true SAPs.

Mode	#StaticThreads	#Total accesses	TLOA		static-TSA	
			#SAPs	Total Time	#SAPs	Total Time
With-JDK	4	19	0(**0***)	**OOM**(25min)	**5**(**5***)	30s
Without-JDK	3	18	8(**2***)	0.7s	2(**2***)	0.4s

lar static Java bytecode analysis framework with numerous built-in points-to analyses and call graph construction algorithms. We use the SPARK toolkit [25] provided in Soot to build a context-insensitive points-to analysis on the fly based on the Class Hierarchy Analysis (CHA) call graph. Nevertheless, our algorithm works for any call graph construction and points-to analysis algorithms.

Reflection and custom classloaders. Dynamic class loading through reflection or custom classloaders is common in real world large Java applications. This feature raises numerous issues for static program analysis. As it is difficult to know statically the reflective calls that the program will execute at runtime, the constructed call graph will be incomplete. To alleviate this problem, we use TamiFlex [6] to log reflective calls witnessed in concrete program executions with typical program inputs and insert the recorded reflective calls into the program to facilitate static analysis. TamiFlex is effective in resolving reflective calls for the benchmarks used in our experiments, though it is unsound in general.

5. CASE STUDY

In this section, we present a detailed case study with the simple example in Figure 1 to understand the characteristics of our static TSA algorithm and compare its performance with the state-of-the-art escape analysis implementation (TLOA [18]). All experiments were conducted on an 8-core Intel i7 3.6GHz Linux machine with Java HotSpot 1.7 and 4GB memory.

We first ran both our static TSA algorithm and TLOA on the *whole* example program, *i.e.*, with all JDK libraries used in the program included. This is needed in general to ensure the analysis soundness. Otherwise, if the JDK libraries are excluded, the call graph will be incomplete which would lead to incorrect results. The result is reported in Table 1 (Row 1). TLOA ran out of memory on this simple program after 25 minutes, and was not able to find any shared data access. Conversely, our static TSA algorithm finished in 30 seconds and correctly identified all the five SAPs. Note that the 30s includes all the analysis time including that for constructing the CHA call graph and the SPARK points-to analysis. The reason why TLOA ran out of memory is that, in addition to the example code, the JDK libraries it uses contain a large number of classes to analyze. Because TLOA has to build an information flow graph for the whole program, it uses up the memory after 25 minutes.

Unsoundness after excluding JDK libraries. In practice, to speed up the analysis, JDK libraries are often excluded by users[5], which may sacrifice the analysis correctness. To further investigate this problem, we also conducted the same comparison with JDK libraries excluded. Table 1 (Row 2) reports the result.

With JDK libraries excluded, TLOA was able to finish the analysis in 0.7 second. However, it reported eight SAPs, of which only two (❹ and ❾) are true. TLOA missed three real SAPs (❷❸❼) and reported six false alarms. Comparatively, our static TSA algorithm identified two SAPs (❼ and ❾) in 0.4 second, both of which are true SAPs, but missed the other three (❷❸❹). The reason is that the exclusion of JDK libraries makes the call graph fail to recognize the edge from the call `o.hashCode` at line 18 to the `hashCode` method of class `Shared` at line 6. Therefore, the read and write accesses to `x` at line 7 (❷ and ❸) are not analyzed, and hence our algorithm cannot determine that ❹ (the write to `x` at line 13) is a SAP. The effect of missing call graph edges is also reflected by the number of static threads and the total number of accesses found in the analysis. As reported in Table 1, the whole program contains four static threads: in addition to `T1`, `T2` and `T3`, it should also include a Reference Handler thread (used by Java garbage collector to handle References) started by the JVM. However, the Reference Handler thread is not found after excluding the JDK libraries. The reason for missing one memory access (the analysis found in total 18 memory accesses instead of 19) is similar.

6. EVALUATION

Our evaluation aims to answer three questions:

1. How efficient/effective is our static TSA algorithm compared to classical escape analysis?
2. How efficient/effective is our dynamic TSA algorithm compared to the precise dynamic escape analysis algorithm?
3. How effective are our algorithms when applied in client applications?

To answer the first question, we evaluated our static TSA algorithm and compared with TLOA on seven real world large complex multithreaded Java programs, including *Apache Derby*, *Eclipse*, *Jigsaw*, *H2*, *Avrora*, *Sunflow*, and *Lusearch*. All these benchmarks were collected from recent concurrency studies [21, 22]. The total size is more than 1.7MLOC.

To answer the second question, we evaluated our dynamic TSA algorithm on the same set of benchmarks and compared it with a precise dynamic escape analysis algorithm, DEA [31, 9]. Both our dynamic TSA algorithm and DEA are unsound (*i.e.*, their results are only valid for the observed execution, but may not hold for future executions). Nevertheless, they can still be useful in many applications in practice (see our discussion in Section 8). Because the implementation of DEA is not available, to perform a fair comparison, we also implemented DEA based on ASM (which tracks precisely every memory access at runtime).

To answer the third question, we applied both our two TSA algorithms in two client applications: (1) LEAP [20]

[5]This can be done by setting an option `-no-bodies-for-excluded` provided in Soot.

– a record and replay system for deterministically replaying multithreaded programs; (2) RVPredict [21] – a dynamic data race detector for Java programs.

All experiments were done with the same hardware and JVM configurations as the case study in Section 5. As the executions of these multithreaded benchmarks were non-deterministic, all data were averaged over three runs.

6.1 Results of Static TSA

The results of our static TSA algorithm compared with TLOA on the seven large benchmarks with and without JDK libraries are summarized in Tables 2 and 3, respectively. Overall, our static TSA algorithm is much more efficient and effective than TLOA, and scales well to these seven large programs containing over 1.7 million of lines of code.

With-JDK. Table 2 Column 3 reports the number of static threads in the whole program. Column 4 reports the total number of memory accesses (including both array and field accesses in the program). Columns 5-6 (TLOA) report the number of accesses classified as SAPs by TLOA and the total analysis time. For none of these large programs, TLOA was able to finish the analysis in an hour. Columns 7-8 (static-TSA) report the number of accesses classified as SAPs by our static TSA algorithm and the total time, including all the analysis (CHA, points-to, and TSA) and query time for all memory accesses in the program. For all these large programs, our algorithm was able to finish in less than three minutes (ranging between 34s to 141s, and on average 66s). Though being static, our algorithm is very effective in finding SAPs. For example, for the largest benchmark *Eclipse* which contains 19 static threads and over 60K total memory accesses, our algorithm finished in 141 seconds and found that 60% of the reads and writes in *Eclipse* are non-SAPs (either to thread-local or to immutable data). Overall, our algorithm pruned 49%-80% of the total memory accesses in these seven large programs to be non-SAPs.

Without-JDK. We have shown in our case study that excluding JDK libraries can improve performance, but also lead to unsound analysis results (*i.e.*, some real SAPs will be missed). As reported in Table 3, after excluding the libraries, TLOA was able to finish for most benchmarks in a minute, except for *Eclipse* which TLOA still timed out in an hour. However, the results are now unsound. For example, for *Avrora*, both TLOA and our algorithm report zero SAP, which is obviously wrong. Regardless of the unsoundness, however, our static TSA algorithm is still much faster than TLOA. For example, for *Sunflow*, while TLOA took 38s and found 4506 SAPs, our algorithm found 4527 SAPs in 15s. The result also indicates that our algorithm is more precise than TLOA. For instance, for *Jigsaw*, we manually inspected the three SAPs reported by our algorithm and found that there are all true SAPs. However, eight of the 11 SAPs reported by TLOA are false alarms.

6.2 Results of Dynamic TSA

Dynamic TSA is able to precisely determine SAPs because the address of every memory access is known. The main challenge is how to minimize the runtime overhead. The results of our dynamic TSA algorithm compared with DEA are summarized in Table 4. Column 2 reports the native execution time of each benchmark. Column 3 reports the number of dynamic threads in the execution. The number can be smaller or larger than that of static threads, because a static thread can have multiple runtime instances, and some static threads are not started due to code coverage. Column 4 reports the total number of executed static program locations that access memory. Note that these dynamic numbers are smaller than the numbers identified by static analysis, because the dynamic executions only cover a subset of program paths. Columns 5-7 report the number of program locations classified as SAPs and the analysis overhead of DEA and our dynamic TSA algorithm, respectively.

Overall, our algorithm is much more efficient than DEA. For these benchmarks, our algorithm was 1.6X to 14X faster than DEA. For example, for *Avrora*, DEA took 69s, while our algorithm took only 5.2s. Moreover, for two benchmarks (*H2* and *Sunflow*), DEA ran out of memory after a few minutes (407s and 344s respectively), whereas our algorithm finished in only 15.7s and 4.7s, respectively. The average time taken by DEA was 125s for each benchmark, while our algorithm took only 7s on average. Although being much faster, our algorithm does not lose much precision compared to DEA. For example, for *Eclipse*, out of 42504 total accesses, DEA identified 10384 SAPs, while our algorithm reported 11785 SAPs (only 4% more). On average, our dynamic TSA algorithm reported 19% (ranging between 4%–44%) more SAPs than DEA on these benchmarks.

6.3 Results on Two Client Applications

6.3.1 Application on record and replay

LEAP [20] is a record and replay system that records thread access orders local to each shared data to support deterministically reenacting past multithreaded program executions. The replay theorem underpinning LEAP guarantees that it only needs to track shared data accesses during recording. Therefore, providing a thread sharing analysis to LEAP can reduce the amount of online tracking and hence reduce the recording overhead. We ran LEAP with four different settings and measured its recording overhead: `All` (tracking all accesses), `static-TSA` (tracking only the SAPs reported by our static TSA algorithm), `dynamic-TSA` (tracking only the SAPs reported by our dynamic TSA algorithm), and DEA (tracking only the SAPs reported by the precise dynamic escape analysis algorithm).

The results are summarized in Table 5. Column 2 reports the native execution time of the corresponding program without any record-replay support. Column 3-6 report the execution time with the record-replay support of LEAP by recording all memory accesses, only the SAPs reported by our static TSA algorithm, our dynamic TSA algorithm, and by DEA, respectively. Note that the SAPs used in this experiment for our static TSA algorithm are those reported in Table 2 without excluding the Java libraries, so that the results are sound. Also, in Column 6, the execution time for DEA are not reported for *H2* and *Sunflow*, because on these two benchmarks DEA ran out of memory and was not able to report SAPs.

Overall, our TSA algorithms are highly effective for reducing the recording overhead. Recording all memory accesses (the `All` setting) can slow down the program by as much as 30X on *Avrora* and 14X on average. Compared to the `All` setting, our static TSA algorithm produced only 5.3X slowdown on average, reducing the overhead by 37% in *Derby* to as much as 1600% in *Lusearch*. For example, for *Lusearch*, the native execution took 324ms and tracking all accesses

Table 2: Results of static TSA (including JDK libraries). Our algorithm finished the analysis for all the benchmarks in 34s-141s and found that 49-80% of the accesses are to thread-local or immutable data. TLOA was not able to finish for any large program in an hour.

Program	LOC	#StaticThreads	#Total accesses	TLOA		static-TSA	
				#SAPs	Total Time	#SAPs	Total Time
jigsaw	101K	18	14075	-	**TimeOut**(1h)	5707(↓**59%**)	44s
derby	302K	3	61909	-	**TimeOut**(1h)	23527(↓**62%**)	82s
avrora	93K	9	25929	-	**TimeOut**(1h)	7745(↓**70%**)	45s
h2	189K	16	24729	-	**TimeOut**(1h)	12686(↓**49%**)	64s
sunflow	109K	10	22243	-	**TimeOut**(1h)	4527(↓**73%**)	94s
lusearch	410K	3	7886	-	**TimeOut**(1h)	2119(↓**80%**)	34s
eclipse	560K	19	60219	-	**TimeOut**(1h)	23890(↓**60%**)	141s

Table 3: Results of static TSA (excluding JDK libraries). TLOA was able to finish for most benchmarks, but produced unsound results. Our algorithm is much faster and also more precise than TLOA.

Program	#StaticThreads	#Total accesses	TLOA-No-JDK		static-TSA-No-JDK	
			#SAPs	Total Time	#SAPs	Total Time
jigsaw	4	122	11	1.5s	3	1.1s
derby	2	61908	3991	37s	0	32s
avrora	1	25928	0	14s	0	13.6s
h2	3	24728	2489	18.4s	269	16s
sunflow	5	22242	4506	38s	4527	14.7s
lusearch	2	7885	2267	14.5s	1783	8.7s
eclipse	12	60215	-	**Timeout**(1h)	20738	46s

Table 4: Results of dynamic TSA. The precise dynamic escape analysis algorithm DEA runs out of memory for two benchmarks. For the remaining benchmarks, our dynamic TSA algorithm runs 1.6X–14X faster than DEA. On average, our algorithm reports 19% more SAPs than DEA.

Program	Native	#DynamicThreads	#Total accesses	DEA		dynamic-TSA	
				#SAPs	Overhead	#SAPs	Overhead
jigsaw	423ms	12	6424	1945(**30%**)	1.6s(**278%**)	3366(**52%**)	0.86s(**103%**)
derby	1063ms	3	24645	3775(**15%**)	9.6s(**803%**)	14640(**59%**)	1.5s(**41%**)
avrora	946ms	9	6528	1283(**20%**)	69s(**6294%**)	2038(**35%**)	5.2s(**450%**)
h2	760ms	9	11015	-	407s(**OOM**)	5101(**46%**)	15.7s(**2180%**)
sunflow	394ms	17	6613	-	344s(**OOM**)	2205(**33%**)	4.7s(**1093%**)
lusearch	324ms	10	4878	459(**9%**)	27s(**8233%**)	824(**17%**)	16.8s(**5085%**)
eclipse	879ms	19	42504	10381(**24%**)	29s(**3200%**)	11785(**28%**)	4.8s(**446%**)

took 6.8s (with almost 20X recording overhead), whereas tracking only the SAPs reported by our static algorithm took 1.6s (with less than 4X overhead). The other two settings (with `dynamic-TSA` and DEA) achieved even more overhead reduction because they track fewer SAPs (since the dynamic algorithms are more precise than static analysis).

Our dynamic TSA algorithm achieved comparable overhead reduction to that of DEA, with 4.8X average slowdown (compared to 3.7X of DEA) over the native execution. For example, for *Lusearch*, our dynamic algorithm took only 1s (2X overhead) and DEA took 0.9s (1.8X overhead). However, we note that using the SAPs reported by dynamic algorithms may not guarantee the deterministic replay. Because of scheduling non-determinism, even with the same input, multiple runs of the same program may produce different SAPs. If a SAP that introduces non-determinism is not tracked, LEAP may fail to replay the program execution.

6.3.2 Application on Race Detection

RVPredict [21] is a dynamic data race detector that logs an execution trace (including shared data access events and synchronizations) of multithreaded Java programs and performs offline constraint analysis to predict races. To scale to large traces, RVPredict employs a windowing strategy that divides the trace into a sequence of fixed-size windows (typically 10K events in a window) and performs race analysis on each window separately. Providing a thread sharing analysis to RVPredict may affect the speed and race detection result of RVPredict in two ways. First, both the trace size and the logging time will be reduced as fewer non-shared data access events are logged. Second, more data races may be detected as each window will contain more shared data accesses, and the race prediction time may or may not increase depending on the complexity of the generated race constraints.

Similar to the experiments with LEAP, we ran RVPredict with four different settings (`All`, `static-TSA`, `dynamic-TSA`, and `DEA`) and compared the number of logged events in the corresponding trace, the online logging time, the offline race prediction time, and the number of detected races. The results are summarized in Tables 6 and 7. Overall, our TSA algorithms are effective in reducing the trace size and the online logging time for most of the programs excepts *Jigsaw*, reducing the number of logged events by 1%-54% (18% on average) and the logging time by 7%-30% (16%). For example, our static TSA algorithm reduced the number of logged events in *H2* from 6.7M to 3.1M, and reduced the logging time for *Avrora* from 56 minutes to 43 minutes. For *Jigsaw*, our static algorithm was less effective (reduced the trace size and the logging time by only 1% and 7%, respectively). The reason is that there are many repeated events

Table 5: Application on record and replay – reducing recording overhead (LEAP).

Program	Native time	Replay recording overhead (LEAP)			
		All	static-TSA	dynamic-TSA	DEA
jigsaw	423ms	1.8s(**325%**)	1.1s(**160%**)	1.0s(**136%**)	1.0s(**136%**)
derby	1063ms	1.7s(**69%**)	1.4s(**32%**)	1.4s(**32%**)	1.3s(**12%**)
avrora	946ms	29.2s(**2987%**)	20.6s(**2076%**)	19.4s(**1951%**)	18.6s(**1866%**)
h2	720ms	18.5s(**2469%**)	4.3s(**497%**)	4.4s(**511%**)	-
sunflow	394ms	4.1s(**941%**)	1.0s(**154%**)	0.93s(**136%**)	-
lusearch	324ms	6.8s(**1998%**)	1.6s(**394%**)	1.0s(**209%**)	0.9s(**178%**)
eclipse	879ms	11.2s(**1174%**)	4.6s(**423%**)	4.3s(**389%**)	4.2s(**378%**)
Average:	-	**14X**	**5.3X**	**4.8X**	**3.7X**

Table 6: Application on dynamic race detection (RVPredict) – the number of events and online logging time.

Program	#Events				Online logging time			
	All	static-TSA	dynamic-TSA	DEA	All	static-TSA	dynamic-TSA	DEA
jigsaw	3.77M	3.73M(↓**1%**)	2M(↓**47%**)	744K(↓**80%**)	11.5s	10.7s(↓**7%**)	6.9s(↓**40%**)	4.4s(↓**72%**)
derby	2.4M	1.9M(↓**21%**)	1.6M(↓**33%**)	1.5M(↓**38%**)	16.2s	13.8s(↓**15%**)	13.1s(↓**19%**)	11.2s(↓**31%**)
avrora	1221M	948M(↓**22%**)	924M(↓**24%**)	803M(↓**34%**)	55m51	43m3s(↓**23%**)	40m21s(↓**28%**)	36m39s(↓**34%**)
h2	6.7M	3.1M(↓**54%**)	2.6M(↓**61%**)	-	21s	14.8s(↓**30%**)	10.9s(↓**48%**)	-
sunflow	439M	402M(↓**8%**)	401M(↓**9%**)	-	16m	14m55s(↓**7%**)	14m13s(↓**11%**)	-
lusearch	266M	240M(↓**10%**)	127M(↓**52%**)	73M(↓**73%**)	12m57s	11m5s(↓**14%**)	6m15s(↓**52%**)	3m34s(↓**72%**)
eclipse	49M	43M(↓**12%**)	39M(↓**20%**)	35M(↓**29%**)	2m25s	2m(↓**17%**)	1m47s(↓**26%**)	1m37s(↓**33%**)
Average:	-	↓**18%**	↓**35%**	↓**50%**	-	↓**16%**	↓**32%**	↓**48%**

Table 7: Application on race detection (RVPredict) – offline prediction time and number of detected races. * means that RVPredict did not finish the analysis due to runtime exception caused by a resource error.

Program	Offline race prediction time				#Races			
	All	static-TSA	dynamic-TSA	DEA	All	static-TSA	dynamic-TSA	DEA
jigsaw	4.5s	3.9s(↓**13%**)	1.5s(↓**67%**)	2.6s(↓**42%**)	6	6(0)	5(**-1**)	5(**-1**)
derby	52s*****	55s(-)	57s*****(-)	306s(-)	16*****	178(**+162**)	16*****(0)	46(**+30**)
avrora	**OOM**(1h)	1001s(↓**72+%**)	996s(↓**72+%**)	742s(↓**79+%**)	42	51(**+9**)	52(**+10**)	55(**+13**)
h2	5.5s	3.9s(↓**29%**)	2.4s(↓**56%**)	-	0(0)	0(0)	0(0)	-
sunflow	424s	360s(↓**15%**)	391s(↓**8%**)	-	4	4(0)	3(**-1**)	-
lusearch	**OOM**(1h)	187s(↓**95+%**)	211s(↓**94+%**)	84s(↓**98+%**)	6	6(0)	8(**+2**)	8(**+2**)
eclipse	224s	662s(↑**196%**)	1065s(↑**375%**)	1853s(↑**727%**)	56	91(**+35**)	100(**+44**)	104(**+48**)
Total:	-	-	-	-	134	340(**+206**)	189(**+55**)	215(**+92**)

on some thread-local data in *Jigsaw* that are recognized as SAPs by our static algorithm.

Comparatively, our dynamic TSA algorithm achieved a much higher reduction ratio, reducing the number of logged events by 9%-61% (35% on average) and the logging time by 11%-52% (32% on average). Overall, the reduction ratio by our dynamic algorithm is between 5%-33% (15% on average) less than that of DEA. The DEA algorithm achieved the highest reduction ratio, with 50% on trace size and 48% on logging time on average. Nevertheless, we note that the higher reduction ratio by dynamic algorithms is achieved by sacrificing race detection capability, because the critical events on those SAPs missed by the dynamic algorithms are not logged into the trace. This fact was also revealed by the number of races detected by different settings as reported in Table 7. For example, for *Jigsaw*, while both `All` and `static-TSA` detected 6 data races, `dynamic-TSA` and `DEA` only detected 5.

For the offline race prediction, the results show that for most benchmarks our TSA algorithms enabled RVPredict to detect many more races than the `All` setting with much less time. For *Avrora* and *Lusearch*, RVPredict with the `All` setting even ran out of memory after an hour, whereas RVPredict with our two TSA algorithms both finished in less than 20 minutes and detected 44 and 54 more data races, respectively. For *Derby*, RVPredict with `static-TSA` detected 178 races in 55s, whereas that with `All` detected only 16 in 52s and did not finish the analysis due to a runtime exception caused by resource error. In total, RVPredict with `static-TSA` and `dynamic-TSA` detected 206 and 55 more races than that with `All`. For *Eclipse*, RVPredict with our two algorithms detected 91 races in 662s and 100 races in 1065s, respectively, whereas that with `All` detected 56 races in 224s. The reason why RVPredict with our algorithms took more time for *Eclipse* was that, with our algorithms applied, many non-SAP events are not logged in the trace, and therefore each window of events analyzed by RVPredict contains more potential race pairs to check.

For DEA, for some benchmarks (*Avrora* and *Eclipse*), RVPredict with it detected even more races than that with our static and dynamic TSA algorithms, because each window analyzed by RVPredict with DEA contains even more shared data accesses. However, for a couple of the benchmarks (*Jigsaw* and *Derby*), RVPredict with DEA detected less. The reason is that dynamic algorithms are unsound, which causes some SAPs that are involved in data races not logged in the trace, and hence RVPredict was not able to detect those races on the SAPs missed by DEA.

7. DISCUSSION

Our experimental results clearly demonstrate the superior scalability and effectiveness of our static and dynamic TSA

algorithms over existing escape analysis. Following we identify a few directions that may further improve our algorithms in terms of precision and performance.

More precise points-to analysis. Because our static TSA algorithm is built on existing call graph and points-to analysis, the precision can hence be affected by the underlying points-to analysis algorithms. Our experimental results reported in Section 6 are based on the simple CHA call graph and flow-insensitive, Anderson-style [3] context-insensitive points-to analysis, which are not very precise. We expect that a more precise points-to analysis with flow-sensitivity [19] and/or context-sensitivity [37] such as CFA [34], object-sensitivity [27], and type-sensitivity [36] will enable our algorithm to produce more precise results. The challenge is how to balance the efficiency while maximizing precision. We plan to investigate this direction further in our future work.

Array analysis. For scalability, both our static and dynamic TSA algorithms do not distinguish between different individual array elements. In fact, many approaches have been proposed before to infer the contents of arrays [11, 16, 17]. The `FunArray` [11] technique developed by Cousot, Cousot, Logozzo based on abstract interpretation [10] can scale the analysis of array content properties to large codebase with negligible overhead. We plan to investigate the possibility of integrating our algorithms with `FunArray` to efficiently distinguish between shared and non-shared array elements, which will improve precision.

Other object-oriented languages. Although our presentation is based on Java, our algorithms are generally applicable to object-oriented programs such as C++. However, different language features may impact the precision and efficiency of our algorithm. For example, facing function pointers and pointer arithmetic, the quality of points-to analysis in C++ might not be as good as Java. And a language like C++ makes extensive use of inlining might facilitate interprocedural analysis in a way that makes it more tractable than Java. We plan to also evaluate our algorithms for different languages in our future work.

8. RELATED WORK

Thread sharing analysis (TSA) has found many important applications in practice, such as synchronization optimization [7, 33], lock allocation [18, 38], transactional memory [35], race and atomicity detection [21, 14, 13, 4], replay debugging [20], verification [23], type system for safe parallelism [5, 42], and auto-parallelization [32]. However, this fundamental problem was not sufficiently addressed before, in particular, the most representative technique – escape analysis – does not suite this problem well. Our work presented in this paper features two practical TSA algorithms that are scalable to real world programs with millions of lines of code. We next briefly survey additional related work.

Choi *et al.* [8], Whaley and Rinard [41] pioneered the use of static escape analysis to eliminate synchronizations on non-escaped objects. Naik *et al.* used thread-escape analysis in the Chord framework [28] to compute *EscapingPairs* for static race detection. Nishiyama [31], Christiaens and Bosschere [9] used dynamic escape analysis to speed up on-the-fly race detection. Dwyer *et al.* [12] developed a dynamic object-escape analysis that improves dynamic partial order reduction [15] for model checking. As shown in this paper, escape analysis is less precise and more expensive than our proposed algorithms for TSA. Nevertheless, escape analysis has a boarder application scope than TSA. In certain applications such as performance optimization of sequential programs (*e.g.*, stack allocation), it would be desirable to run escape analysis regardless of its greater cost and lesser precision for one particular use.

Praun and Gross [40] proposed Object Use Graph (OUG) to statically approximate the happens-before relation of thread accesses to shared objects. OUG has been shown useful in many applications such as method specialization and object race detection [39]. Differently, OUG is at the granularity of object and its construction requires precise symbolic execution of static threads to track data/control flow and synchronizations to determine thread-conflicting objects.

Researchers have also proposed may-happen-in-parallel (MPH) analysis [30, 26] to determine if it is possible for execution instances of two given statements (or the same statement) to execute in parallel. MPH can be used to improve the precision of thread sharing analysis and has been shown useful in race detection [29] and lock allocation [18]. However, an ideal MPH would need a precise whole program information flow analysis and reasoning of thread synchronizations [30], the complexity of which is cubic in the size of the program and may not scale well in practice.

Although dynamic TSA is unsound in general (may miss shared data accesses), it can still be useful in many applications in practice such as bug finding. For instance, DoubleChecker [4] employs a lightweight dynamic sharing analysis for efficiently detecting atomicity violations. It achieves low overhead by running the program twice. The first run performs an imprecise analysis that tracks cross-thread dependencies among transactions only. The second run then only needs to precisely process those transactions that the imprecise analysis identified as being potentially involved in atomicity violations.

9. CONCLUSION

We have presented static and dynamic algorithms for identifying shared data accesses in multithreaded programs and demonstrated with extensive evaluation that our algorithms are highly scalable and effective in improving the performance of important concurrency analysis applications such as record-replay and dynamic race detection. Our static algorithm outperforms (much faster, less memory-demanding, and more precise than) the state-of-the-art static escape analysis, and our dynamic algorithm achieves significant speedups over a precise dynamic escape analysis while maintaining close precision. Our results suggest that these algorithms are promising for practical use in analyzing real world large multithreaded programs.

10. ACKNOWLEDGEMENTS

I would like to thank Daniel Jiménez and the anonymous reviewers for their constructive comments on earlier versions of this paper. This research is supported by faculty start-up funds from Texas A&M University, a Google Faculty Research Award, and NSF award CCF-1552935.

11. REFERENCES

[1] Asm bytecode analysis framework. http://asm.ow2.org/.

[2] Java hotspot virtual machine performance enhancements. http://docs.oracle.com/javase/7/docs/technotes/\\guides/vm/performance-enhancements-7.html.

[3] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, PhD thesis, University of Copenhagen, 1994.

[4] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. Doublechecker: efficient sound and precise atomicity checking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[5] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel Java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 97–116, 2009.

[6] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. *International Conference on Software Engineering*, 2011.

[7] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 1999.

[8] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 1999.

[9] Mark Christiaens and Koenraad De Bosschere. Trade: Data race detection for Java. In *ICCS*, 2001.

[10] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*.

[11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[12] Matthew B. Dwyer, John Hatcliff, Robby, and Venkatesh Prasad Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 2004.

[13] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[14] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[15] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[16] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[17] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[18] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT*, 2007.

[19] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization*, 2011.

[20] Jeff Huang, Peng Liu, and Charles Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 207–216, 2010.

[21] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014.

[22] Jeff Huang and Lawrence Rauchwerger. Finding schedule-sensitive branches. In *Joint European Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2015.

[23] Peter Müller K. Rustan M. Leino and Jan Smans. Verifying concurrent programs with chalice. In *Foundations of Security Analysis and Design*, 2009.

[24] Patrick Lam, Eric Bodden, and Laurie Hendren. The soot framework for Java program analysis: a retrospective, 2011.

[25] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *CC*, 2003.

[26] Lin Li and Clark Verbrugge. A practical mhp information analysis for concurrent java programs. In *LCPC*, 2004.

[27] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 2005.

[28] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[29] Mayur Hiru Naik. *Effective Static Race Detection for Java*. PhD thesis, 2008.

[30] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1999.

[31] Hiroyasu Nishiyama. Detecting data races using

dynamic escape analysis based on read barrier. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, 2004.

[32] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-time Parallelization of Loops with Privatization and Reduction Parallelization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[33] Erik Ruf. Effective synchronization removal for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[34] M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. Program Flow Analysis, 1981.

[35] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in stm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[36] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[37] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[38] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 334–345, 2006.

[39] Christoph von Praun and Thomas R. Gross. Object race detection. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2001.

[40] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[41] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 1999.

[42] Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. Cooperative types for controlling thread interference in java. In *ACM International Symposium on Software Testing and Analysis*, 2012.