HEAT: An Integrated Static and Dynamic Approach for Thread Escape Analysis [*]

Qichang Chen and Liqiang Wang
Department of Computer Science
University of Wyoming
{qchen2, wang}@cs.uwyo.edu

Zijiang Yang
Department of Computer Science
Western Michigan University
zijiang.yang@wmich.edu

Abstract

Thread escape analysis, which determines whether and when a variable becomes shared by multiple threads, is a foundation for many other program analyses. Most existing escape analysis tools are either purely dynamic or static. Static analysis, which considers all possible behaviors of a program, may produce false positives; whereas dynamic approaches cannot analyze unobserved behaviors of a program.

This paper presents a hybrid approach that integrates static and dynamic analyses to address this problem. We first perform static analysis to obtain succinct summaries of program source code. Dynamic analysis is then used to confirm variable sharing; for unexecuted code, we determine the sharing of variables by performing an interprocedural synthesis based on the runtime information and static summaries. Compared to dynamic analysis, the hybrid approach is able to determine the escape property of variables in unexecuted code. Compared to static analysis, the hybrid approach produces fewer false alarms. We implemented this hybrid escape analysis in Java. Our experiments on several benchmarks and real-world applications show that the hybrid approach improves accuracy of escape analysis compared to existing approaches and significantly reduces overhead of subsequent program analyses.

1 Introduction

Thread escape analysis, a program analysis technique that determines which objects escape from their creating threads (*i.e.*, can be accessed by multiple threads), is important for subsequent program analyses. For example, it can determine unnecessary synchronizations for thread-local objects; it can reduce the runtime overhead when dynamically detecting concurrency-related errors, such as race conditions, atomicity violations and deadlocks, since all accesses to thread-local variables can be ignored. For object-oriented programming, even if an object escapes from its creating thread, some fields may never be accessed by multiple threads. In this paper, the granularity for thread escape analysis is on the field level.

[*]The work was supported by ONR Grant N000140910740.

Most existing approaches for escape analysis are either purely dynamic (*e.g.* [9, 7]) or purely static (*e.g.* [3, 11]). Static analysis reasons over program source code without actually executing the program. Although it can potentially report all potential shared variables, static escape analysis suffers a high rate of false positives (alarms). Dynamic analysis reasons about behavior of a program through executions. Generally, dynamic escape analysis is more accurate than static escape analysis in identifying shared variables. However, it suffers a high rate of false negatives (*i.e.*, some shared variables cannot be found) because the approach cannot analyze unexplored behavior of programs.

This paper presents a novel hybrid approach that extends a dynamic escape analysis by incorporating static analysis. Our hybrid approach consists of two phases: in the first phase, it performs static analysis on program source code to obtain the concise static summaries about accesses to all fields, assignments to reference variables, and method invocations; the second phase contains a dynamic analysis and a speculation for unexecuted code: we monitor the actual field accesses during execution and perform an interprocedural synthesis based on the runtime information and the static summaries. The static summaries are instantiated with the runtime values to speculatively approximate the behaviors of unexecuted code. A field escapes from the creating thread if it has ever been accessed by multiple threads during the real execution or speculation.

We implement our analysis for Java programs in a tool called HEAT (Hybrid Escape Analysis for Thread) and evaluate it on several benchmarks and real-world applications. The experiment shows that the hybrid approach improves accuracy of escape analysis compared to existing approaches and significantly reduces overhead of subsequent program analyses (in our experiment, specifically, a hybrid approach for checking atomicity violations).

To summarize, our paper makes the following contributions: (1) It presents an integrated static and dynamic thread escape approach to determine whether and when a field becomes shared by multiple threads. The approach reports less false positives than static analysis and less false negatives than dynamic analysis. (2) Subsequent analyses can significantly benefit from the proposed escape analysis in reducing overhead and/or improving accuracy. (3) We implement the approach in Java and evaluate it extensively. The experiment shows that the tool reduces a large portion of runtime overhead for several memory-intensive benchmarks in detecting atomicity violations.

The rest of this paper is organized as follows. Section 2 introduces escape analysis. Section 3 presents the design and implementation details. Our experiments are presented in Section 4. Section 5 discusses related work. Section 6 gives conclusions and the future work.

2 Thread Escape Analysis

When an object o is created, all its instance fields are owned by the creating thread. Its static fields (if there are) are owned by all threads. A field $o.f$ *thread-escapes* when it can be accessed by two or more threads. Thread-ownership can be transferred. The thread ownership of a field $o.f$ is said to be *transferred* from a thread t to another thread t' if there exists a program state after which t will not access field $o.f$ any more, and t' does not access $o.f$ until that program state. A field $o.f$ is *thread-local* if it does not have multiple thread-owner simultaneously; otherwise, it is *shared*.

Most existing static escape analyses (*e.g.*, [3, 13, 11]) apply points-to and interprocedural program analysis on source code or byte code to identify thread-local objects and fields. They are usually very expensive and tend to report many false positives due to the difficulty of reconciling the symbolic references with the actual memory locations. To our best knowledge, none of them can deal with the container (*i.e.*, Collections and Maps) escape case. For example, if a container object escapes, then all objects contained inside this object are considered escaped by static analysis, which might be false positives, since some objects may never be accessed by other threads.

Dynamic escape analyses (*e.g.*[4, 7]) monitor accesses to objects and fields during execution and identify the escape objects and fields when they have been observed to be accessed by multiple threads. This is more accurate but suffers from incompleteness due to the fact that not all code will be executed.

Figure 1 shows an example where both static and dynamic escape analyses are inaccurate in identifying thread-local fields. In `thread-1`, two objects `a1` and `a2` of `Account` are created and saved in a vector `acctVector`, then `thread-2` is created and started. We assume that every `Account` object has a unique identifier. Now we consider how the Eraser dynamic race detector [12] is affected by escape analysis. A static escape analysis (*e.g.*[11]) will report both objects `a1` and `a2` escape when creating and starting `thread-2`; the Eraser dynamic race detector will check all accesses on both objects for potential race conditions, even if some object is not really shared by multiple threads. Thus, large overhead may be incurred by the high rate of false positives in static escape analysis.

```
Thread-1

Initialize(){
   Account a1 = new Account();
   Account a2 = new Account();
   acctVector.add(a1);
   acctvector.add(a2);
   (new Thread-2(acctVector)).start();
}

Deposit(int givenID, float val){
   for(Account a:  acctVector)
     if (a.ID == givenID){
       a.bal += val;
       break;
     }
}

Thread-2

Withdraw(int givenID, float val){
   for(Account a:  acctVector)
     if (a.ID == givenID) {
       if (a.bal >= val)
         a.bal -= val;
       break;
     }
}
```

Figure 1. Examples in Java demonstrating escaped objects.

In dynamic escape analysis (*e.g.*[9]), if some object is not really accessed by multiple threads, it will be considered thread-local. But a dynamic escape analysis may have false negatives. In this example, in the method `withdraw` of `thread-2`, if `a.bal < val`, the statement "`a.bal -= val`" will not be executed. Thus, the Eraser dynamic race detector may not find the race condition on the field `bal` if `thread-2` starts after the method `deposit` is called in `thread-1`, because the detector is still on the state of "shared" for the field `bal`.

The hybrid escape analysis proposed in the paper avoids the above problems. It speculatively approximates the unexecuted branch in `withdraw` based on its static summary and runtime information. Specifically, the symbol `a` in the static summary is resolved using its current runtime identifier. Thus, we can identify that the field `bal` is shared. The detector will enter into the state of "shared-modified", and the race condition can be detected.

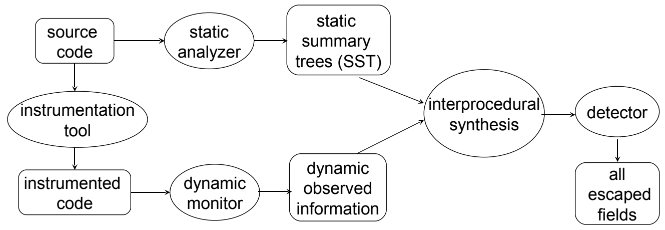


Figure 2. The architecture of the tool `HEAT`.

```
Withdraw(int givenID, float val) {
  for(Account a:  acctVector) {
    if (a.ID == givenID)
      if (a.bal >= val) {
        beginWithdraw(val);
        a.bal -= val;
        break;
      }
  }
}
```

The corresponding SST

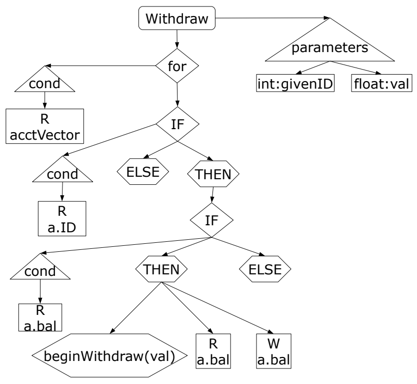


Figure 3. An example of a static summary tree (SST) with its corresponding code block, where `R` and `W` in the nodes denote "read" and "write", respectively.

3 Integrated Static and Dynamic Escape Analysis

3.1 Overview of The Hybrid Approach

Figure 2 shows the architecture of our tool `HEAT`, which consists of five components: (1) A static analyzer, which parses the source code to generate *static summary trees (SSTs)*; (2) An instrumentation tool, which inserts code for intercepting events during execution. (3) A dynamic monitor, which intercepts events and records them during execution. (4) A speculator, which performs interprocedural synthesis to combine the static summaries for unexecuted branches and loops from SSTs and the runtime observed information. (5) A detector, which analyzes hybrid information to report all escaped fields.

3.2 Static Analyzer

The static analyzer parses program source code to construct *static summary trees* (SSTs). Each SST corresponds to a brief summary of a method in a Java class. Specifically, a SST may contain nodes representing: (1) read/write to non-`final` and non-`volatile` fields; (2) method invocations (interprocedural information). (3) Object reference assignment statements. (4) Control flow structures. (*e.g.*, if/then/else, do/while/for/switch) (5) synchronization statements. (6) field access to array object in the form of $o[i].f$.

Figure 3 shows an example of a code block and its corresponding SST.

3.3 Dynamic Monitor

HEAT uses Eclipse JDT framework to rewrite the source code of the target program. We instrument all field accesses (*i.e.*, read and write) inside the program, except the accesses on those fields whose declarations are outside the scope of the program (*i.e.*, field from imported library).

3.4 Interprocedural Synthesis

We speculate every unexecuted code block with the SST generated by the static analyzer by performing a context-sensitive interprocedural analysis on the SST for different calling contexts. Figure 4 shows the algorithm.

For each running thread in the program, we have a corresponding monitor to observe all field accesses occurring in that thread. For each field, we identify it using the unique identifier of that object (`hashCode` of the class object for static fields) plus the name of that field.

When we speculate an unexecuted code block based on the corresponding SST, symbolic names in the SST are instantiated by querying binding tables. A binding table is maintained for each object; it stores the mappings between symbolic names and runtime values of all reference fields and local reference variables under the context of the object. A binding table is maintained for each class with static reference fields. Binding tables are updated when assignments to reference variables are executed. During speculative execution, assignments to reference variables in SSTs trigger updates on temporary copies of binding tables, instead of the original ones. If the runtime binding of the object reference in the speculation is unable to be determined based on

```
Main() {
  while (the program has not terminated) {
    switch (the current executing statement) {
      case access to field $o.f$: CheckEscape($o.f$);
      case object reference assignment:
          update the corresponding binding table;
      case branch point:
          Speculate(each unexecuted branch);
    }
  }
}

CheckEscape(field $o.f$) {
  if ($o.f_{prevThread} == -1$) return;
  else if ($o.f_{prevThread} \neq$ the current thread)
      { $o.f_{prevThread} = -1$;}
}

Speculate(SST) {
  for (each statement $s$ in SST) {
    switch ($s$) {
      case access to field $o.f$:
        if ($o$ can be resolved based on the binding table)
          CheckEscape($o.f$);
        else CheckEscape($O\_*.f$); /* O is the class name */
      case object reference assignment:
          update the corresponding temporary binding table;
      case branch point:
          Speculate(each branch);
      case method call:
          if (some actual arguments can be resolved) {
            substitute formal parameters with actual arguments;
            Speculate(method body);
          }
}}}
```

Figure 4. The algorithm for interprocedural synthesis and escaped field detection.

binding tables, we would replace the symbolic name with a wildcard object identifier (*e.g.*, `Account_*.bal`).

When we reach a method call during speculation, we expand it with the SST of its declaration body and perform the formal parameters and actual arguments substitution. This context-sensitive approach enables us to resolve object references in the SST for that method invocation at different calling sites. Inside the expanded method invocation, we use the binding table from dynamic monitor to resolve the bindings of as many as symbolic object reference names from the static summary tree (SST) when we perform speculation on it. This is continued for all method calls including nested ones in SST, except that all actual arguments with wildcards as object identifiers (*i.e.*, the runtime binding cannot be determined). We do not process recursive calls in the current implementation. Thus, all the information from the static analysis is synthesized with the runtime information in the dynamic monitor.

We skip method calls whose declaration bodies have no corresponding SST (*e.g.*, native methods not implemented in Java or methods defined in the library whose source code are not available). This does not affect our escape analysis for the target programs under testing since we are only concerned about the fields defined in the programs.

3.5 Detecting Escaped Fields

To identify the escaped fields, one straightforward approach is to collect a set of fields with object identifiers in each thread monitor till the end of the execution and then perform intersection over these sets from different thread monitor. However, this approach has its drawbacks since the set of fields recorded during execution can be overwhelmingly large. In addition, in this approach, a field is always monitored in the entire execution, which is unnecessary.

To alleviate this problem, we insert an additional shadow field (like an accompanying shadow) for each existing field in the corresponding class definition of program source code. The shadow field `prevThread` indicates the last thread that accesses the field. `prevThread` is initialized to 0 if no thread has accessed it. To determine whether a field has escaped, we check whether the current accessing thread on that field is same as the `prevThread` when accessing the field during execution or speculation. When a field is identified to be escaped, its accompanied shadow field `isEscaped` is set to -1. Thus, we will drop all the subsequent observations on that field, since it is already identified to be escaped. Figure 4 shows the algorithm.

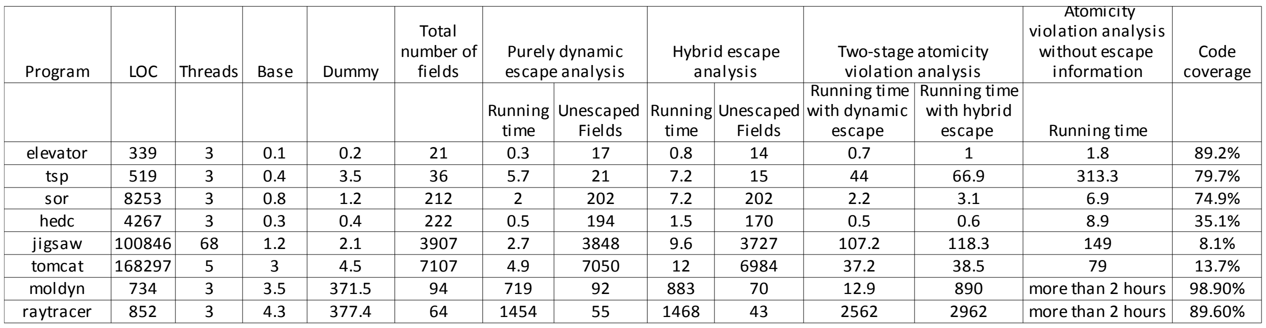
4 Experiment

We evaluated HEAT on a collection of multi-threaded Java applications: `elevator`, `tsp`, `sor`, and `hedc` are from [15], `moldyn` and `raytracer` are from the Java Grande forum Multi-threaded benchmark suite [5]; `Jigsaw` [6] and `Apache tomcat` [14] are two multi-threaded web services. We performed the experiment on a machine with Intel dual-core CPU of 1.8 GHz, 2 GB memory, Windows XP SP3, J2SE 1.6.

Figure 5 compares the result of pure dynamic escape analysis algorithm against our hybrid approach. “Base” is the uninstrumented program’s running time. “Dummy” is the instrumented program’s running time plus interception time without performing any online/offline analysis.

We evaluate our hybrid escape analysis in two ways. First, we compare the runtime costs between the pure dynamic escape analysis and the hybrid one. Second, we compare the effects of the two analyses on the performance of subsequent atomicity violation analysis.

From Figure 5, we can see that the hybrid approach reveals more potentially escaped fields than the dynamic approach. The running time difference between them are not



Program	LOC	Threads	Base	Dummy	Total number of fields	Purely dynamic escape analysis		Hybrid escape analysis		Two-stage atomicity violation analysis		Atomicity violation analysis without escape information	Code coverage
						Running time	Unescaped Fields	Running time	Unescaped Fields	Running time with dynamic escape	Running time with hybrid escape	Running time	
elevator	339	3	0.1	0.2	21	0.3	17	0.8	14	0.7	1	1.8	89.2%
tsp	519	3	0.4	3.5	36	5.7	21	7.2	15	44	66.9	313.3	79.7%
sor	8253	3	0.8	1.2	212	2	202	7.2	202	2.2	3.1	6.9	74.9%
hedc	4267	3	0.3	0.4	222	0.5	194	1.5	170	0.5	0.6	8.9	35.1%
jigsaw	100846	68	1.2	2.1	3907	2.7	3848	9.6	3727	107.2	118.3	149	8.1%
tomcat	168297	5	3	4.5	7107	4.9	7050	12	6984	37.2	38.5	79	13.7%
moldyn	734	3	3.5	371.5	94	719	92	883	70	12.9	890	more than 2 hours	98.90%
raytracer	852	3	4.3	377.4	64	1454	55	1468	43	2562	2962	more than 2 hours	89.60%

Figure 5. Comparison of the purely dynamic escape analysis algorithm and the hybrid algorithm in performance and accuracy. All times are measured in seconds.

very significant for most of benchmarks, which indicates that our hybrid analysis improves the accuracy and completeness of escape analysis without sacrificing much runtime overhead. For most of the benchmarks, the memory remains under realistic limits. The largest one has not exceeded 200 MB in contrast with the memory cost of 30 MB for the uninstrumented version. Our approach shows that tracking all the field accesses is not only possible in terms of time and memory space but also very feasible for most benchmarks.

Figure 5 also shows the performance improvement when checking atomicity violations using escape information against the approach without escape analysis (*i.e.*, monitor all field accesses). To facilitate checking atomicity violations with the escape information, we collect the fields reported from the first-stage escape analysis in a trace file. The trace file is then analyzed by our instrumentor to selectively instrument the fields that are determined to be escaped. We perform the post-stage atomicity violation analysis after the instrumented program terminates. The results of atomicity violation analysis remain the same for both approaches.

As indicated from Figure 5, the most obvious two benchmarks that benefit from this approach are `moldyn` and `raytracer`. Without the first-stage escape analysis, they run for more than 2 hours without termination. With the assistance of the escape information, we can easily ignore those heavily accessed but thread-local fields when checking atomicity violation, since they can not be involved in concurrency-related errors. The overall time has reduced to as low as 2 minutes in contrast. For the other benchmarks, the performance improvements are not significant partially because they are not as intensive on memory accesses as `moldyn` and `raytracer`.

5 Related Works

J. Choi *et al.* [3] present a static interprocedural escape analysis framework that incorporates both the thread-escape and method-escape analyses. The escape analysis is based on a connection graph which statically builds the relationship between object references and objects. In [11], Rinard *et al.* propose a static pointer and escape analysis that uses parallel interaction graphs to analyze the interactions between threads and provides precise points-to, escape and action ordering information. Our tool differs from them in that we combine the accuracy of dynamic analysis with the completeness of static analysis. Bogda *et al.* [1] and Rub [10] propose unification-based escape analyses and apply them to synchronization elimination.

Compared with static escape analysis, the dynamic escape analysis in [4] is more expensive and more precise. [7] introduces a dynamic analysis technique that caches all possible escaping objects at runtime and then performs a set intersection between cached escaping objects from different threads to obtain the escaped objects. They also perform an empirical study on several escape analysis techniques. The dynamic phase of our approach is almost same to it except that we did not adopt the caching technique but use the state variables. [9] uses a on-the-fly read-barrier-based dynamic escape analysis that eliminates the thread-local memory locations from being checked by the data race detector thus improves the performance of lock-set based data race detection.

Static and dynamic analyses have been combined for multi-threaded programs. Lee [8] is the closest to HEAT in that they present a two-phase static/dynamic interprocedural and inter-thread escape analysis. Both approaches perform an offline static analysis followed by a more accurate and faster online dynamic analysis which integrates the information from static analysis. However, their approach uses the level summaries obtained from dynamic analysis to improve the connection graph built in the offline stage and thus improve the accuracy. Integrated static and dynamic analysis has also been used to detect atomicity violations [2].

6 Conclusions and Future Works

In this paper, we present a tool HEAT for integrated static and dynamic escape analysis and demonstrate its effectiveness by evaluating it on several benchmarks and real-world applications. HEAT combines the accuracy of dynamic analysis while supplementing it with the interprocedural static analysis. The augmentation from unexecuted branches in the program makes the dynamic analysis more effective at finding many subtle escape cases.

Our experiments show that the proposed hybrid approach is more effective at finding many subtle escape case. Furthermore, it can be adopted to boost the performance of subsequent program analyses (*e.g.*, detecting race conditions and atomicity violations) and significantly lower the runtime overhead for memory-intensive programs.

In the future work, we will extend the interprocedural analysis to improve the approach's accuracy and investigate other ways to improve its performance. In addition, we will apply it to analyze more concurrency-related program properties.

References

[1] J. Bogda and U. Hölzle. Removing unnecessary synchronization in java. *SIGPLAN Not.*, 34(10):35–46, 1999.

[2] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Integrated dynamic and static analysis for atomicity violations. In *Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 5503 of *LNCS*, pages 425–439. Springer, 2009.

[3] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.

[4] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Formal Methods in System Design*, 25(2-3):199–240, 2004.

[5] Java Grande Forum. Java Grande Multi-threaded Benchmark Suite. version 1.0. Available from http://www.javagrande.org/.

[6] Jigsaw, version 2.2.6. Available from http://www.w3c.org.

[7] K. Lee, X. Fang, and S. P. Midkiff. Practical escape analyses: how good are they? In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 180–190, New York, NY, USA, 2007. ACM.

[8] K. Lee and S. P. Midkiff. A two-phase escape analysis for parallel java programs. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 53–62, New York, NY, USA, 2006. ACM.

[9] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[10] E. Ruf. Effective synchronization removal for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 208–218. ACM Press, June 2000.

[11] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proc. ACM SIGPLAN 2001 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, 2001.

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

[13] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 2–13, New York, NY, USA, 2005. ACM.

[14] Apache tomcat, version 6.0.16. Available from http://tomcat.apache.org.

[15] C. von Praun and T. R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, Oct. 2001.