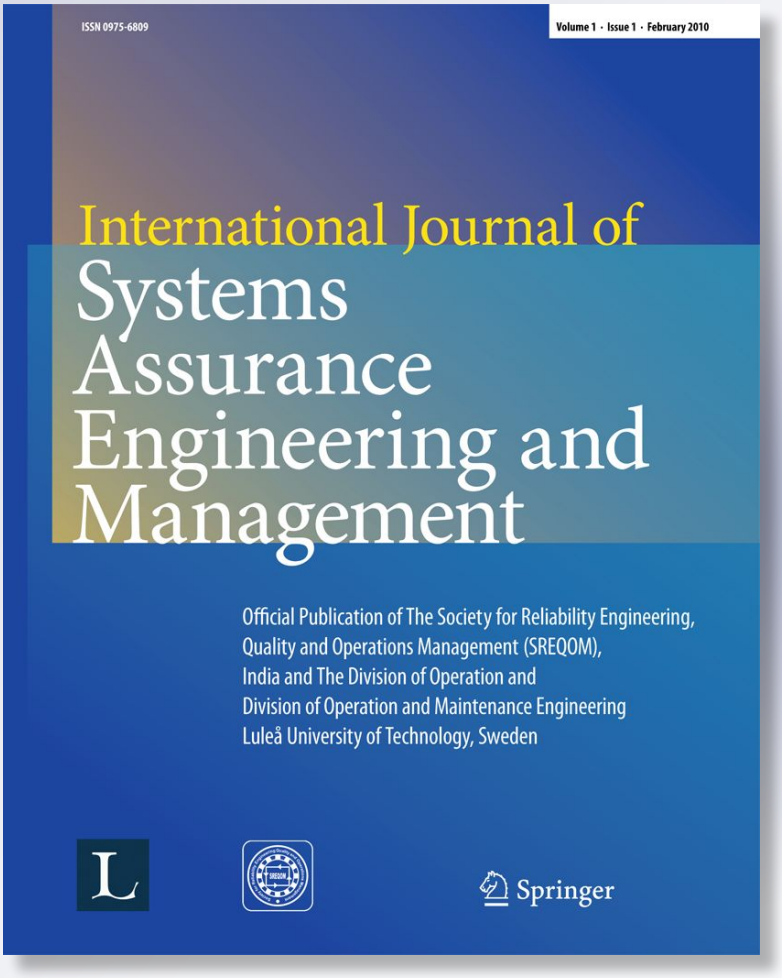
HEAT: a combined approach for thread escape analysis

Qichang Chen, Liqiang Wang & Zijiang Yang

International Journal of System Assurance Engineering and Management

ISSN 0975-6809
Volume 2
Number 2

Int J Syst Assur Eng Manag (2011)
2:135-143
DOI 10.1007/s13198-011-0069-2



Springer

Int J Syst Assur Eng Manag (Apr-June 2011) 2(2):135–143
DOI 10.1007/s13198-011-0069-2

ORIGINAL ARTICLE

HEAT: a combined approach for thread escape analysis

Qichang Chen · Liqiang Wang · Zijiang Yang

Received: 14 December 2010 / Revised: 26 August 2011 / Published online: 18 September 2011

Abstract Thread escape analysis can determine whether and when a variable becomes shared by multiple threads, which is a foundation for many other program analysis and software testing techniques. Most existing escape analysis tools are either purely dynamic or static analyses. Static analysis, which considers all possible behaviors of a program, may produce false positives; whereas dynamic approaches miss the information from unexecuted code sections 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 accesses to all variables and interprocedural information. 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 the accuracy of escape analysis compared to existing approaches and significantly reduces the performance overhead of a subsequent program analysis.

Keywords Thread escape analysis · Program analysis · Concurrent program · Software testing · Dynamic analysis

Q. Chen (✉) · L. Wang
Department of Computer Science, University of Wyoming, Laramie, WY 82071-3315, USA
e-mail: qchen2@cs.uwyo.edu

L. Wang
e-mail: wang@cs.uwyo.edu

Z. Yang
Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008-5314, USA
e-mail: zijiang.yang@wmich.edu

1 Introduction

Thread escape analysis is a program analysis technique that determines which and when objects escape from their creating threads (i.e., may be accessed by multiple threads). Thread escape analysis is a fundamental technique for many other 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 addition, many concurrent software testing techniques are dependent on the thread escape analysis on identifying the thread escape variables that are of their analysis focus. In this paper, the granularity for thread escape analysis is on the field level.

Most existing approaches for escape analysis are either purely dynamic (e.g., Lee et al. 2007; Nishiyama 2004) or purely static (e.g., Choi et al. 2003; Salcianu and Rinard 2001). Static analysis reasons over program source code without actually executing the program. Static escape analysis is conservative and can report all potential shared variables because all of the source code can be analyzed,

but it may have a high rate of false positives (alarms). Dynamic analysis reasons about behavior of a program through executing it. Generally, dynamic escape analysis is more accurate in identifying shared variables, but it may have a high rate of false negatives (i.e., some shared variables cannot be found) because it does not 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 contains two phases: in the first phase, it performs static analysis on program source code to obtain the concise static summaries about field accesses and method invocations; the second phase is a dynamic analysis: we monitor the actual field accesses during execution and perform an interprocedural synthesis on the runtime information and the static summaries to determine the escaped fields. In addition, if a field would become thread-shared eventually, our approach can also determine when a field becomes shared, thus we treat the field as thread-local before it escapes in order to avoid unnecessary monitoring overhead on it.

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 monitoring overhead of subsequent program analyses (in our experiment, specifically, a hybrid program analysis approach for checking data race). For example, many memory-intensive programs would take many hours to finish because of overwhelming number of events generated from monitoring the field accesses. Our tool identifies more unshared fields, which in turn considerably reduces the number of monitored events and allows the dynamic or hybrid program analysis to finish within a reasonable time.

To summarize, our paper makes the following contributions:

- It presents an integrated static and dynamic thread escape approach to determine whether and when a field becomes shared by multiple threads. The approach theoretically has less false positives than static analysis and less false negatives than dynamic analysis.
- Subsequent analyses can significantly benefit from the proposed escape analysis in reducing dynamic monitoring overhead and improving analysis accuracy.
- We implement the approach in Java and evaluate it on a few benchmarks including large-scale real-world programs. The experiment shows that the tool significantly reduces runtime overhead by more than 90% for several memory-intensive benchmarks for the subsequent race condition analysis.

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

2 Introduction to thread escape analysis

In multithreaded object-oriented programming, such as Java, when an object o is created, o is owned by the creating thread. Object o *escapes* from its creating thread when it can be accessed by two or more threads. Thus, o may have multiple *thread-owners*. A thread-owner of o is also the owner of its all instance fields. For a static field, all threads are its owner. 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 execution state after which t will not access field $o.f$ any more, and t' does not access $o.f$ until reaching that program state. A field $o.f$ is *thread-local* if it does not have multiple thread-owner simultaneously (i.e., only one thread-owner at the same time); otherwise, it is *shared* by multiple threads. For a field $o.f$, its *escape point* is the earliest program state where it becomes shared.

Most existing static escape analyses (Choi et al. 2003; Salcianu and Rinard 2001; Sura et al. 2005) apply points-to and interprocedural analysis on program 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 escape analysis with respect to container objects (e.g., Collections and Maps in Java). 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 be never accessed by other threads.

Dynamic escape analyses (e.g., Dwyer et al. 2004; Lee et al. 2007) monitor accesses during execution and identify the escaped objects and fields if they have been observed to be accessed by multiple threads. This is more accurate for the executed traces but suffers from the 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.

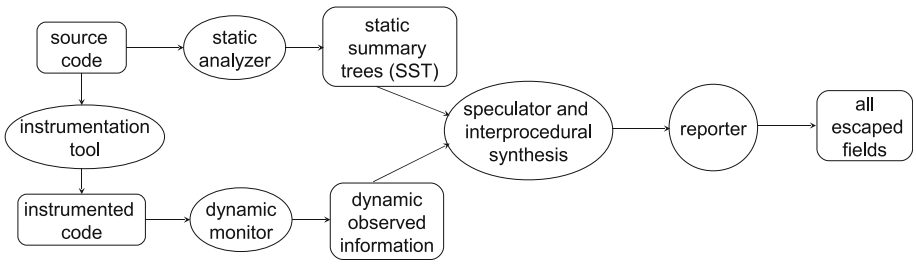
A static escape analysis (e.g., Salcianu and Rinard 2001) will report that all fields of `a1` and `a2` escape when starting `thread-2`, because static analysis is usually conservative

Fig. 1 Examples in Java demonstrating an escaped object

```
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.saving += val;
     break;
    }
   }
}
```

```
Thread-2

Withdraw(int givenID, float val){
   for(Account a: acctVector) {
    if (a.ID == givenID){
     if (a.checking >= val) {
      a.checking -= val;
     } else
     if (a.checking+a.saving>=val){
      a.checking = 0;
      a.saving -= val-a.checking;
     }
     break;
    }
   }
}
```

Fig. 2 The workflow of the tool HEAT



and cannot resolve pointer/reference accurately. If `a1.checking`, `a1.saving`, `a2.checking` and `a2.saving` are not really accessed by both `thread-1` and `thread-2` (e.g., deposit and withdraw are done on different accounts), they are just thread-local. A dynamic escape analysis (e.g., Nishiyama 2004) can resolve pointer/reference on runtime, however, it may report false negatives, because not all program code is analyzed. For example, if `thread-1` and `thread-2` deposit and withdraw from account `a1`, when `a1.checking` have enough balance, `a1.saving` and `a1.checking` are not shared by both threads; otherwise, `a1.saving` will be accessed by both threads. Thus, a dynamic escape analysis is highly sensitive to the input test cases.

The hybrid escape analysis proposed in the paper overcomes the above problems. It can speculatively approximate the unexecuted branch based on its static summary and runtime information. Specifically, the symbol `a` in the static summary is resolved using its current runtime identifier based on executed code. Thus, we can identify that the field `a.saving` is shared under some circumstances. The potential errors on `a.saving` (e.g., race condition) may be detected early.

3 Combining static and dynamic escape analyses

3.1 Overview of the hybrid approach

Figure 2 shows the workflow 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 event interception code.
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 code blocks from SSTs and the runtime observed information.
5. A reporter, which analyzes hybrid information to report all escaped fields.

3.2 Static analyzer

The static analyzer analyzes the program source code and keeps track of any possible escape fields. To be more specific, the static analyzer parses the program source code to construct *static summary trees* (SSTs) which are used in conjunction with the runtime monitor. Each SST corresponds to a brief summary of a method in a Java class. Specifically, a SST may contain nodes representing the following events.

– Read/write to `non-final`, `non-volatile` and `non-static` fields. We ignore the final fields because most subsequent concurrency analyses focus on mutable shared variables and the immutable final fields are not in their concerns. The volatile fields are ignored because they are deemed to be share variables as the keyword

"volatile" implies that the fields will be accessed and updated by multiple threads. The static fields are ignored due to the fact that any static field is always accessible to multiple threads as long as its access scope permits.

- Method invocations (interprocedural information), which are recorded in the SST for the on-the-fly interprocedural analysis in the speculation stage. In each method invocation node, we also include a link to the method definition SST node.
- Object reference assignment statements, which are recorded for the intraprocedural points-to analysis.
- Control flow structures (i.e., if/them/else, do/while/for, switch/case), which are recorded for a thorough exploration of the speculated control block in the runtime.

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

We implemented the static analyzer based on Eclipse JDT framework (Eclipse, http://www.eclipse.org/). For each source code file, it traverses the corresponding AST using JDT and iteratively analyzes each method and appends the nodes to the SSTs. Finally, the SSTs are dumped into an XML file to be used by the subsequent stage of dynamic speculation.

3.3 Dynamic monitor

HEAT uses the Eclipse JDT framework (Eclipse, http://www.eclipse.org/) to rewrite the source code of the target program. We instrument all field accesses (i.e., read and write) inside the program. Specially, we ignore accesses on those fields whose declarations are outside the scope of the program (i.e., fields from imported library). This has no effect on our escape analysis for the target program under testing since we only concern about the escape fields inside the program.

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 its owner object (i.e., `hashCode`) plus the name of that field.

To identify the escaped fields, one straightforward implementation is to collect a set of fields accessed in each thread monitor, then perform the set intersection over these sets from different thread when the program terminates. However, this approach has its drawbacks since the set of fields recorded during execution can be overwhelmingly large such that the program may run out of memory before it can finish. In addition, it is unnecessary to monitor a field which has been identified escaped.

To alleviate this problem, we insert two additional shadow fields (like an accompanying shadow) for each existing field at their definitions in the program source code. One field called `isEscaped` is to keep track of whether a field has escaped. Its initial value is `false`. When we decide the field escapes (which is discussed below), the corresponding `isEscaped` is set to be `true`, and we will skip all the subsequent observations on that field to reduce the runtime overhead. The other field called `prevThread`, which indicates the last thread that accesses the field. `prevThread` is initialized to -1 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`. The check can be phrased in a plain expression as $isEscaped = (j == prevThread?\ false{:}true)$, where j is the accessing thread of the current field access.

3.4 Speculator: interprocedural synthesis

During the runtime, we speculate every unexecuted code block using the SST generated by the static analyzer. A context-sensitive interprocedural analysis is performed on the SST at different calling contexts on the fly. Figure 4 shows the interprocedural analysis and synthesis algorithm.

When we speculate an unexecuted code block based on the corresponding SST, symbolic names in the SST are

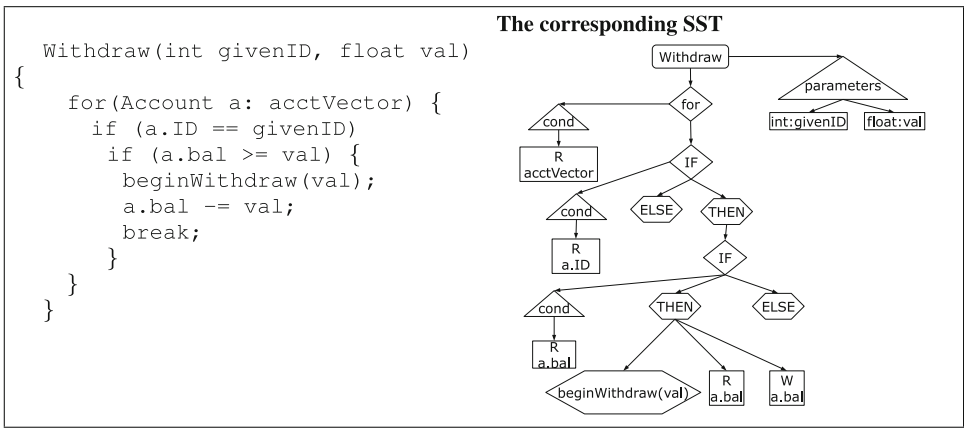


Fig. 3 An example of a static summary tree (SST) with its corresponding code

Fig. 4 The interprocedural synthesis algorithm

```
Main() {
  for (each e before the program terminates) {
    switch (e) {
      case field access:
        Let o.f be the corresponding object and field operated by e;
        CheckEscape(e, o.f);
      case object reference assignment:
          update current_BindingTable;
      case control flow condition expr:
          { Speculate(the sst for each unexecuted code block);}
    }
  }
}

CheckEscape(e, o.f) {
  Let j be the thread issuing e.
  if (e is a field initialization)
    o.f_isEscaped = false; o.f_prevThread= j;
  else
    if o.f_isEscaped == true
        return
    else
      isEscaped = (j == prevThread?false : true);
      AllEscapedFields = AllEscapedFields ∪ {o.f}.
}
Speculate(sst) {
  temp_BindingTable = current_BinddingTable;
  for (each node s in sst) {
    switch (s) {
      case a field access event, say e:
        if (the object operated by e can be resolved by current_BindingTable )
          Let o.f be the correponding object and field resolved based on temp_BindingTable for e;
          CheckEscape(e, o.f);
      case object reference assignment:
          update temp_BindingTable;
      case control flow condition expr:
          Speculate(sst for each unexecuted block);
      case method invocation, say mi:
          analyzeMethodInvocation(mi);
    }
  }
}

analyzeMethodInvocation(mi, temp_BindingTable) {
   if(mi has not been analyzed before){
    perform the parameter and argument replacements;
    for each event node n inside the SST
      if (n is a field access) {
        if we can resolve the object reference of e to some object identifier i inside temp_BindingTable
          CheckEscape(n, i.f);
        else
          Let wc.f be the field whose object name cannot be resolved when processing n. wc is
          a unique wildcard object reference with the ID "-1";
          CheckEscape(n, wc.f);
      }
      else if (n is another method invocation mi which
       has not been analyzed with the same temp_BindingTable) {
        analyzeMethodInvocation(mi, temp_BindingTable) ;
      }
    }
}
```

instantiated with their corresponding runtime object reference identifiers by querying them in the 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. Another 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 binding tables, we would replace the symbolic name with a wildcard object identifier (e.g., `Account_*.checking`). For example, Fig. 5 gives an example of how we context-sensitively analyze the method invocation method(o1). This significantly helps improve the accuracy of HEAT without resorting to any static analysis such as alias analysis (Whaley and Rinard 1999) or connection graph (Choi et al. 2003).

When we reach a method call in the SST, we expand it with the SST of its definition and perform the formal parameters and actual arguments substitution. This context-sensitive approach enables us to resolve the object references in the SST for that method invocation at different thread 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 in the static summary tree (SST). This is continued for any method invocation encountered in that expanded method invocation SST. We stop until all the method invocations have been analyzed. For recursive method calls, we only analyze its top level call because our analysis is flow-insensitive and the interprocedural analysis on the remaining calls will not yield any new escape information. When the interprocedural speculative analysis is completed, all the results from the analysis is synthesized with the runtime information in the dynamic monitor for this thread.

Once a method call has been analyzed, any of the same subsequent invocations happen in that same thread from the same calling context will be ignored by HEAT. This prevents from duplicating the events to burden our escape analysis since we only need a distinct field access from each thread to determine the escape cases.

Our unified escape analysis will report more false positives than the purely dynamic escape analysis. However, this also allows the subsequent analyses to reveal more potential errors and provide more complete diagnostic information to the developers and end users.

```
method(Object2 o1) {
// o1 = 10001999
  if (o1.f > 20)
    o1 = new Object1(); // o1 = 10002001
  else
    o1 = new Object1(); // o1 = 10002002
  o1.f = 10; // o1.f = *.f
}

// current binding table: a1 = 10001999
method(a1);
```

Fig. 5 Reference binding resolving for a code segment

4 Experiment

This section discusses the evaluation of HEAT on a collection of multi-threaded widely-used Java benchmarks: `elevator`, `tsp`, `sor`, and `hedc` are from (von Praun and Gross 2001), `moldyn` and `raytracer` are from the Java Grande forum Multi-threaded benchmark suite (Java Grande Forum, http://www.javagrande.org/); `Jigsaw` and `Apache Tomcat` are two multi-threaded real-world web services from (Jigsaw, http://www.w3c.org) and (Apache, http://tomcat.apache.org), respectively. `Elevator` is a program that uses 2 threads to simulate the elevator. `tsp` is the multi-threaded Travel Sales Person problem solver which comes with a standard set of TSP problems. Given a specific TSP test harness, the user can additionally specify the number of threads employed to solve that problem. `sor` stands for **S**uccessive **O**ver-**R**elaxation which performs 100 iterations of successive over-relaxation on a $N \times N$ grid as specified in the test harness. `Hedc` is a multi-threaded web crawler. `Jigsaw 2.2.6` is a basic HTTP server that supports both secure and unsecure communications. `Apache tomcat` is an HTTP and *J2EE* application server that supports the dynamic content generation for server-side web pages. (Smith and Bull 2001) has extensive coverages on the Java *Grande* benchmarks(namely, `Elevator`, `Tsp`, `Sor`).

We perform the experiment on a machine with Intel dual-core CPU of 1.8 GHz, 2 GB memory, Windows XP SP3, and J2SE 1.6.

Figure 6 compares the result of pure dynamic escape algorithm against our hybrid approach. "Base" is the running time of the original (uninstrumented) program. "Dummy" is the running time of the instrumented program including event intercepting but without performing any online/offline analysis.

We evaluate our hybrid escape analysis in three ways. First, we compare the runtime costs between the pure dynamic escape analysis and the hybrid one. Second, we also compare the accuracy, specifically number of escaped fields, reported by the two approaches. Third, we compare the effects of the two analyses on the performance of the subsequent data race analysis.

From Fig. 6, we can conclude that our hybrid approach reveals more escaped fields than the dynamic approach. The time difference between them is not 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 memory usage of most memory-intensive program under HEAT has not exceeded 200 MB in contrast with a memory of 30 MB for its uninstrumented version. Our HEAT tool shows that tracking all the field accesses is not only possible

Program	LOC	Threads	Base(s)	Dummy(s)	Total number of fields	Purely Dynamic Escape Analysis		Hybrid Escape Analysis(HEAT)		Second Stage Data Race Analysis(Eraser)				Data Race Analysis(Eraser) without Escape Information		code coverage
						Execution Time(s)	Unescaped Fields	Execution Time(s)	Unescaped Fields	Execution Time(s) with Dynamic Escape Information	number of reported data race locations	Execution Time(s) with Hybrid Escape Information	number of reported data race locations	Execution Time(s)	number of reported data race locations	
elevator	339	3	0.1	0.2	21	0.3	17	0.8	14	0.3	1	0.7	1	1.5	1	89.2%
tsp	519	3	0.4	3.5	36	5.7	21	7.2	15	21	3	31.2	10	157.3	10	79.7%
sor	8253	3	0.8	1.2	212	2	202	7.2	202	1.3	0	2.2	0	4.2	0	74.9%
hedc	4267	3	0.3	0.4	222	0.5	194	1.5	170	0.3	20	0.4	36	4.9	36	35.1%
jigsaw	100846	68	1.2	2.1	3907	2.7	3848	9.6	3727	50.4	18	57.3	57	83	57	8.1%
tomcat	168297	5	3	4.5	7107	4.9	7050	12	6984	17.2	37	19.3	81	45	81	13.7%
moldyn	734	3	3.5	371.5	94	719	92	883	70	5.9	3	351	8	>2 hours	8	89.60%
raytracer	852	3	4.3	377.4	64	1454	55	1468	43	1237	2	1100	4	>2 hours	4	98.90%

Fig. 6 Comparison of the purely dynamic escape algorithm and the hybrid escape algorithm in performance, accuracy and improving the subsequent Eraser data race analysis. All times are measured in seconds

in terms of time and memory space but also very feasible for most benchmarks.

Figure 6 also compares the performance among three different ways for checking data race: using the results from our hybrid escape analysis, using the results from a pure dynamic escape analysis, and without using any escape analysis (i.e., monitor all field accesses). To facilitate checking data races with the escape information, all the escaped fields reported from the first-stage escape analysis are saved in a trace file. The instrumentor selectively instruments the fields that are reported to be escaped, their enclosing methods, and relevant control structures. We perform the post-stage (offline) data race analysis after the instrumented program terminates.

As indicated by Fig. 6, the most obvious two benchmarks that benefit from this approach are `moldyn` and `raytracer`. Without the first-stage escape analysis, we have tested them for more than 2 h without termination. With the assistance of the escape information, we can easily ignore those heavily accessed but thread-local fields when checking data race. The overall time has reduced to as low as 2 min in contrast. For the other benchmarks, the performance improvements are also quite significant. Figure 7 shows the graphical comparison of the Eraser data race monitoring time with and without escape results.

In addition, Fig. 6 shows that our hybrid escape analysis results helps the subsequent data race analysis reveal more faulty locations that could be involved in the potential data race on the benchmark *tsp*. In the meantime, the Eraser data race analysis with the hybrid escape analysis results achieves the same accuracy as the original Eraser analysis while significantly reducing the performance overhead on most benchmarks.

Basen on our experimental evaluations, we can see that our hybrid escape analysis is better than the purely dynamic escape analysis in preserving the accuracy of the subsequent concurrency analysis and only trades only a small amount of overhead time for the accuracy gain over the dynamic escape analysis.

5 Related work

This paper extends our previous work (Chen et al. 2009) by formalizing our hybrid approach and presenting more experimental evaluations for the hybrid escape analysis.

Choi et al. (2003) 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 Salcianu and Rinard (2001), 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 and Hölzle (1999) and Ruf (2000) propose unification-based escape analyses and apply them to synchronization elimination. Bogda et al. proposes an static interprocedural, flow- and context-insensitive data-flow analysis which ignores the control flow to identify the thread escape objects. Ruf (2000) uses a equivalence-class based interprocedural static analysis which groups potentially aliased symbolic names into equivalence classes flow-insensitively. It builds a static call graph for the program and traverses the call graph with a initial context to identify the thread escape equivalence classes. Both their approaches will conservatively report more thread escape objects which could put more burdens on the subsequent concurrency analysis. In addition, their approach differs from ours in that they focus on identifying thread escape objects and achieving a performance gain in the program by removing the synchronization instructions from the

Fig. 7 Comparison of the data race monitoring time with the escape and without the escape results. All times are measured in seconds

program while our unified thread escape analysis aims to reduce the overhead and improve the accuracy of the subsequent concurrency analysis by identifying thread escape fields.

Compared with static escape analysis, the dynamic escape analysis in Dwyer et al. (2004) is more expensive and more precise. Dwyer et al. (2004) design both dynamic and static analysis independently to identify thread escape objects which can be removed from the partial order reduction to improve its performance. The dynamic escape analysis in Ruf (2000) considers that a particular variable is accessible(escapes) from other threads when it can be accessed from the static fields which are global to all threads. Our escape analysis is similar to them in that we also aim in identifying thread escape fields that have read/write accesses from multiple threads. In addition, our escape analysis considers more escape scenarios, such as, a variable is passed to the thread constructor. Lee et al. (2007) 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 the same to this approach except that we did not adopt the caching technique but used the state variables. Nishiyama (2004) uses an 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 (2006)'s approach is the closest one to ours 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.

There are other approaches to combine static and dynamic analyses. JPredictor (Chen et al. 2008) applies a dynamic analysis on the target programs to obtain the relevant trace and then uses the static analysis to prune irrelevant events and extract the dependency relations which can be used for checking for potential concurrency errors. Those techniques, in contrast to ours, do not use speculative synthesis. Concolic testing (Majumdar and Sen 2007) combines symbolic execution (a form of static analysis) and concrete execution to achieve high code coverage in a scalable way. Our speculative execution has a similar goal, aiming to improve code coverage of dynamic analysis in a scalable way. However, our approach is more approximate, but simpler and incurs less overhead.

6 Conclusions and future work

In this paper, we present a hybrid (i.e., combining static and dynamic) escape analysis and demonstrate its effectiveness by evaluating it on several benchmarks and real-world applications. It combines the accuracy of dynamic analysis while enhancing it with the on the fly interprocedural static analysis. The augmentation from unexecuted code blocks makes the dynamic analysis more effective at finding many subtle escape cases which otherwise would be missed by the subsequent concurrency analysis. Our experimental evaluation shows that this approach has only a slight overhead over the purely dynamic escape analysis and improves the subsequent concurrency analysis' accuracy over the purely dynamic escape analysis.

Our approach can more accurately identify thread escape variables that are the analysis focus of many concurrent program analysis and software testing techniques as the experimental data show that the proposed hybrid approach is more effective at finding many subtle escape cases. Furthermore, it can be adopted to boost the performance of subsequent program analysis (e.g., detecting race conditions and atomicity violation) and significantly lower the runtime overhead for testing memory-intensive programs.

In our 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.

Acknowledgment This work was supported in part by ONR under Grant N000140910740.

References

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

Bogda J, Hölzle U (1999) Removing unnecessary synchronization in java. SIGPLAN Not 34(10):35–46 http://doi.acm.org/10.1145/320385.320388

Chen F, Serbanuta TF, Rosu G (2008) jpredictor: a predictive runtime analysis tool for java. In: ICSE '08: proceedings of the 30th international conference on Software engineering, pp 221–230. ACM, New York, NY, USA. http://doi.acm.org/10.1145/1368088.1368119

Chen Q, Wang L, Yang Z (2009) HEAT: a combined static and dynamic approach for escape analysis. In: 33rd annual IEEE international Computer Software and Applications Conference (COMPSAC2009). IEEE Press, Seattle, USA

Choi JD, Gupta M, Serrano MJ, Sreedhar VC, Midkiff SP (2003) Stack allocation and synchronization optimizations for java using escape analysis. ACM Trans. Program Lang Syst 25(6):876–910. http://doi.acm.org/10.1145/945885.945892

Dwyer MB, Hatcliff J, Robby, Ranganath VP (2004) Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. Form Method Syst Des 25(2–3):199–240

Eclipse. Available from http://www.eclipse.org/

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

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

Lee K, Midkiff SP (2006) A two-phase escape analysis for parallel java programs. In: PACT '06: proceedings of the 15th international conference on Parallel architectures and compilation techniques. ACM, New York, NY, USA, pp 53–62. http://doi.acm.org/10.1145/1152154.1152166

Lee K, Fang X, Midkiff SP (2007) Practical escape analyses: how good are they? In: VEE '07: proceedings of the 3rd international conference on virtual execution environments. ACM, New York, NY, USA, pp 180–190. http://doi.acm.org/10.1145/1254810.1254836

Majumdar R, Sen K (2007) Hybrid concolic testing. In: Proceedings of the 29th International Conference on Software Engineering (ICSE). Institute of Electrical and Electronics Engineers

Nishiyama H (2004) 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. USENIX Association, Berkeley, CA, USA, pp 10–10

Ruf E (2000) Effective synchronization removal for Java. In: Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI). ACM Press, pp 208–218

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

Smith LA, Bull JM (2001) A multithreaded java grande benchmark suite. In: Proceedings of the third workshop on java for high performance computing, pp 97–105

Sura Z, Fang X, Wong CL, Midkiff SP, Lee J, Padua D (2005) Compiler techniques for high performance sequentially consistent java programs. In: PPoPP '05: proceedings of the tenth ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, New York, NY, USA, pp 2–13. http://doi.acm.org/10.1145/1065944.1065947

von Praun C, Gross TR (2001) Object race detection. In: Proceedings of 16th ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), SIGPLAN Notices. ACM Press, vol 36(11):70–82. http://www.inf.ethz.ch/ praun/

Whaley J, Rinard M (1999) Compositional pointer and escape analysis for Java programs. In: Proceedings of ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). ACM Press, pp 187–206. Appeared in ACM SIGPLAN Notices 34(10)