Correctness of Partial Escape Analysis for Multithreading Optimization

Dustin Rhodes
UC Santa Cruz
dustin@soe.ucsc.edu

Cormac Flanagan
UC Santa Cruz
cormac@ucsc.edu

Stephen N. Freund
Williams College
freund@cs.williams.edu

Abstract

Compilers often use escape analysis to elide locking operations on thread-local data. Similarly, dynamic race detectors may use escape analysis to elide race checks on thread-local data. In this paper, we study the correctness of these two related optimizations when using a partial escape analysis, which identifies objects that are currently thread-local but that may later become thread-shared.

We show that lock elision based on partial escape analysis is unsound for the Java memory model. We also show that race check elision based on a partial escape analysis weakens the precision of dynamic race detectors. Finally, we prove that race check elision based on a partial escape analysis is sound with respect to this weakened, but still useful, notion of precision.

CCS Concepts **•Theory of computation → Concurrency;** *Design and analysis of algorithms;* **•Computing methodologies →Optimization algorithms;**

Keywords Escape Analysis, Race Detection, Lock Elision

ACM Reference format:
Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. 2017. Correctness of Partial Escape Analysis
for Multithreading Optimization. In *Proceedings of FTFJP'17, Barcelona , Spain, June 18-23, 2017,* 6 pages.
DOI: 10.1145/3103111.3104039

1 Introduction

When reasoning about heap-allocated objects, compilers and other analyses must, in general, assume that concurrent threads can make arbitrary changes to any object. This uncertainty makes it difficult to reason about the possible behavior of code. For thread-local objects (that is, objects only accessible by a single thread), concurrent modifications are possible only after that object has *escaped* out of its allocating thread. An object escapes its allocating thread when it is assigned to a field of a thread-shared object. Many compilers and analyses make use of an *escape analysis* to determine which objects escape their allocating thread [7].

Escape analyses fall into two major categories. A *total escape analysis* determines if an object is *always* thread-local (i.e. it never escapes its allocating thread). On the other hand, a *partial escape analysis* determines if an object has not *yet* escaped its allocating thread [19]. We explore two applications for escape analysis:

1) Synchronization elimination: Optimizing compilers, such as Hotspot [14], use an escape analysis to elide expensive synchronization operations on thread-local objects. We refer to lock elision based on total and partial escape analyses as *total* and *partial lock elision* respectively.

Total lock elision has been proved sound [4]. Partial lock elision has also been proposed in the literature (see [19]); but we show that partial lock elision is unsound in that it can introduce additional behaviors that are not permitted under the Java memory model.

2) Dynamic race detection: Dynamic race detectors also leverage information about thread-local data. In particular, a dynamic race detector typically performs a *race check* every time a thread of the target program reads or writes to an object. These race checks can be elided for accesses to thread-local objects. We refer to race check elision based on total and partial escape analyses as *total* and *partial race check elision* respectively. Eliminating race checks on memory identified as thread-local by a total escape analysis does not change the precision of the detection analysis. However, as we show in this paper, utilizing partial escape analysis does weaken the precision guarantees provided by a dynamic race detector. Partial race check elision never causes a race detector to miss the first data race in a program, but may cause it to miss subsequent data races.

Contributions: In summary, this paper shows:

- partial lock elision is unsound for compilers (Section 3);
- partial race check elision may cause a race detector to miss some races in an execution (Section 4); and
- partial race check elision will never cause a race detector to miss the first race in an execution (Section 5).

2 Race Conditions: Definition and Precision

Race detectors and other analyses use the concept of a trace to analyze a specific execution of a given program. We define a program trace α as a sequence of actions performed by the various threads. These actions include reads and writes of object fields, lock acquire and releases, and forking a new thread.

The happens-before relation $<_\alpha$ for a trace α is the smallest transitively-closed relation over the actions in the trace such that the relation $a <_\alpha b$ holds whenever action a occurs before action b in the trace and one of the following holds:

- Program order: The two actions are performed by the same thread.
- Locking: The two actions acquire or release the same lock.
- Fork: One action forks a new thread and the other action is by that new thread.

If two actions in a trace are not related by the happens-before relation, then they are considered *concurrent.* Two memory access *conflict* if they both access (read or write) the same address, and at least one of the actions is a write. Using this terminology, a trace has a *race condition* on a particular address if it has two concurrent conflicting accesses to that address.

FTFJP'17, Barcelona , Spain

978-1-4503-5098-3/17/06…$15.00
DOI: 10.1145/3103111.3104039

```
                 Before Lock Elision
      Thread1                  |  Thread2
-------------------------------+---------------------------
 1  Ref r0 = new Ref();        |
 2  synchronized(r0){          |
 3    r0.f = 11;               |
 4    r0.f = 12;               |
 5  }                          |
 6  p.o = r0;                  |
                             7 | int r2;
                             8 | Ref r1 = null;
                             9 | while(r1 == null){
                            10 |   r1 = p.o;
                            11 | }
                            12 | synchronized(r1){
                            13 |   r2 = r1.f;
                            14 | }
                            15 | assert(r2 != 11);
```

```
                 After Lock Elision
      Thread1                  |  Thread2
-------------------------------+---------------------------
16  Ref r0 = new Ref();        |
17                             |
18  r0.f = 11;                 |
19  r0.f = 12;                 |
20                             |
21  p.o = r0;                  |
                            22 | int r2;
                            23 | Ref r1 = null;
                            24 | while(r1 == null){
                            25 |   r1 = p.o;
                            26 | }
                            27 | synchronized(r1){
                            28 |   r2 = r1.f;
                            29 | }
                            30 | assert(r2 != 11);
```

Figure 1. Example of Partial Lock Elision (p is a shared object and p.o is null initialized, no fields are volatile)

Data races lead to unexpected behavior, as in Java [11, 17], or undefined behavior, as in C++ [3]. As such, compilers are not allowed to introduce data races when optimizing code.

For a race detector to be *precise* it must not miss data races. More specifically, we say a dynamic race detector is

- *trace precise* if it correctly reports whether a program trace has a race, and
- *address precise* if it correctly reports all addresses in a trace that have race conditions.

Thus, any address precise race detector is also trace precise, but the converse does not hold.

We show that partial race check elision weakens the precision of a dynamic race detector from address precise to trace precise; in particular, one race in a trace may prevent subsequent races on different addresses from being detected. We prove, however, that the first race in a trace is always detected and so partial race check elision is still trace precise, which is sufficient for many applications.

3 Partial Lock Elision is Unsound

A partial escape analysis marks the point at which an object escapes its allocating thread. Partial lock elision uses this information to remove all acquire and release actions on the object before it escapes, as described in [19].

To illustrate why partial lock elision is unsound, consider the program shown in Figure 1 (left). Thread 1 allocates a new Ref object, initializes its f field with 11 and then 12 inside a synchronized block, and then shares its address via p.o with Thread 2. Thread 2 busy waits until p.o is non-null and then reads f inside a synchronized block. Clearly, the two threads race on p.o. However, the accesses to field f are race-free due to synchronization, and so Thread 2 can never read the partially-initialized value of 11. Consequently, the assertion on line 15 never fails.

Because the synchronization in Thread 1 happens before the reference escapes, partial lock elision as described in [19] removes it, resulting in the code in Figure 1 (right). In this code, there is no happens before edge between the writes to f by Thread 1 and the read by Thread 2 because Thread 1's synchronization has been removed. Since these actions are now in a race condition, the Java memory model allows either value, 11 or 12, to be read by Thread 2 allowing the assertion to fail. Thus, partial lock elision (that is, lock elision based on a partial escape analysis) introduces a data race on f and an assertion violation that was not present in the original program, so is clearly an unsound optimization.

4 Partial Race Check Elision is not Address Precise

We now explore partial race check elision and show that using partial information has consequences in this domain as well. A total race check elision algorithm removes race checks on accesses to memory that never escape its allocating thread. A partial race check elision algorithm, on the other hand, removes race checks on accesses to memory that has not escaped yet (see [6]).

To illustrate why partial race check elision is not address precise, consider the program in Figure 2. Here, Thread 1 creates a new Ref object, writes to f, and then shares the object by writing its address to p.o. Thread 2 reads the address of the Ref from p.o and writes to f. A trace of this program reveals two races: the first of which occurs on p.o on lines 33 and 34. As p is a shared object this race will be caught, since no race checks will be elided on these lines.

However, there is a second race in the program between the two writes to f on lines 32 and 35. In this case, the first access on line 32 happens before r1 has escaped and so the corresponding race check would be elided by partial race check elision. The race check on line 35 is not elided because by this time the Ref has escaped. However, the race on f will not be detected due to the previously elided race check. That is, partial race check elision would cause a

Thread 1	Thread 2
`Ref r1 = new Ref();`	
`r1.f = 11;`	
`p.o = r1;`	
	`Ref r1 = p.o;`
	`r1.f = 12;`

Figure 2. Example of Partial Race Check Elision (p is a shared object)

dynamic race detector (such as [6]) to miss the race on f, and so the detector would no longer be address precise.

5 Partial Race Check Elision is Trace Precise

In the example of Figure 2 above, the race on f is only missed due to a previous race on p.o. That is, partial race check elision can cause a dynamic race detector to miss later races in a trace, but never causes it to miss the first race, and so the optimized race detector is still trace precise.

5.1 Idealized Language EscapeJava

We formalize our proof of this property in terms of the idealized language EscapeJava shown in Figure 3. A program P consists of a sequence class definitions D (containing methods and fields) as well as a main expression e. Expressions can create new objects (new), read from fields ($e.f$), assign to fields ($e.f = e$), create temporary variables (let $x = e$ in e), call methods ($e.m(\overline{e})$), acquire and release locks (acq e and rel e), and fork new threads (fork e).

Figure 3 also shows the semantics for this language. A running program has a heap H and a thread set T. The heap maps locations to values and locks to the thread holding them (or to $\perp$ if the lock is not held). Values v are addresses p and null. A location $l = p.f$ is an object address p along with a field f. The thread set maps thread identifiers to expressions. This semantics includes actions a that are emitted for every evaluation step.

A program starts with an empty heap $\emptyset$, and a thread set $T = [t := e]$ with a single thread e with thread identifier t. A single evaluation step

$$P \vdash H, T \rightarrow^{a} H, T$$

produces an action a. Taken in sequence these actions form a trace α. We include P in the evaluation relation to facilitate method look-up, and we assume method names are unique.

5.2 Partial Race Check Elision Algorithm

Figure 4 shows a partial race check elision algorithm. This algorithm performs a dynamic thread escape analysis, recording in G all addresses reachable by multiple threads. The judgement

$$P \vdash G, H, T \rightarrow^{a}_{b} G', H', T'$$

performs a single evaluation step

$$P \vdash H, T \rightarrow^{a} H', T'$$

and also extends the set G' of global (a.k.a. escaped) addresses appropriately. The first judgement above produces two actions a and b. In this judgement action b is a no-op if the action a is a field access whose race check can be elided; otherwise b is simply the action a of the target program. Combining multiple steps of this judgement yields a run of the race check elision algorithm

$$P \vdash G, H, T \longrightarrow^{\alpha}_{\beta} G', H', T'$$

where α is the full trace of the target program, and the trace β is a subsequence of α that elides accesses to thread-local objects.

Figure 4 contains rules for all actions that the semantics emits. Acquire and release actions are never elided because of the unsoundness of partial lock elision, as shown in Section 3. Reads and writes are elided if the address being read/written is not in G (they have not escaped), but are kept if the address is in G (reachable by multiple threads). A write to an object in G expands G to include this new object as well as all objects reachable from it given the current heap. Finally, a fork is never elided, and also expands the global set to include all items reachable from the forked thread (as they are also reachable from the *forking* thread).

Thus, this race check elision judgement

1. runs the program,
2. performs a dynamic partial escape analysis, and
3. uses this information (in G) to elide accesses to thread-local data.

5.3 Partial Race Check Elision is Trace Precise

The key correctness property we want to show is that if

$$P \vdash \emptyset, \emptyset, [t := e] \longrightarrow^{\alpha}_{\beta} G, H, T$$

then α has a race *if and only if* β has a race. In other words, race check elision never converts a racy trace α into a race-free trace β.

We start by formalizing the notion of *reachability* with respect to a given heap. We say an address p' is *reachable* from p in heap H if:

- $p' = p$, or
- for some field f, p' is reachable from $H(p.f)$

Moreover, we say an address p is *reachable* from an expression e in a heap H if p is reachable from some addresses q in e in heap H. We use $reachable(e, H)$ to denote the set of addresses reachable from expression e in H.

We say a state G, H, T is *valid* if G contains all references reachable by multiple threads in T with heap H.

Definition 5.1. G, H, T is *valid* if $\forall t_1, t_2 \in Tid$. if $t_1 \neq t_2$ then $reachable(T(t_1), H) \cap\ reachable(T(t_2), H) \subseteq G$

The set of addresses an action accesses is defined as follows:

$$\begin{array}{rcl} addr(t : \texttt{write}\ q.f\ v) & = & \{q, v\} \cap ObjAddr \\ addr(t : \texttt{read}\ q.f\ v) & = & \{q, v\} \cap ObjAddr \end{array}$$

Additionally, each access has a target address.

$$\begin{array}{rcl} target(t : \texttt{write}\ q.f\ v) & = & q \\ target(t : \texttt{read}\ q.f\ v) & = & q \end{array}$$

Finally, the function tid extracts the thread of an action:

$$tid(t : a) = t$$

$$\begin{array}{lllll}
P & \in & \mathit{Program} & ::= & \overline{D}\ e \\
D & \in & \mathit{Class} & ::= & \texttt{class}\ c\ \{\overline{f}, \overline{\mathit{method}}\} \\
\mathit{method} & \in & \mathit{Method} & ::= & m(\overline{x})\{e\} \\
l & \in & \mathit{Location} & ::= & p.f \\
p, q & \in & \mathit{ObjAddr} & & \\
v & \in & \mathit{Value} & ::= & p \mid \texttt{null} \\
e & \in & \mathit{Expression} & ::= & \texttt{new}\ c() \mid x \mid v \mid e.f \mid e.f = e \mid e.m(\overline{e}) \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{acq}\ e \mid \texttt{rel}\ e \mid \texttt{fork}\ e \\
 & & & & \\
E & \in & \mathit{Context} & ::= & E.f \mid E.f = e \mid p.f = E \mid E.m(\overline{e}) \mid p.m(\overline{v}, E, \overline{e}) \mid \texttt{acq}\ E \mid \texttt{rel}\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e \\
H & \in & \mathit{Heap} & ::= & (\mathit{ObjAddr} \to \mathit{Tid}_\bot)\ \text{and}\ (\mathit{Location} \to \mathit{Value}) \\
T & \in & \mathit{ThreadSet} & ::= & \mathit{Tid} \to e \\
t & \in & \mathit{Tid} & ::= & \text{(Thread identifiers)} \\
a, b & \in & \mathit{Action} & ::= & t : \texttt{acq}\ p \mid t : \texttt{rel}\ p \mid t : \texttt{read}\ l\ v \mid t : \texttt{write}\ l\ v \mid t : \texttt{fork}\ p\ t' \mid t : \texttt{no-op} \\
\alpha, \beta & \in & \mathit{Trace} & ::= & \overline{a}
\end{array}$$

$$\boxed{P \vdash H, T \to^{a} H, T}$$

$$\begin{array}{llll}
P \vdash H, T[t := E[p.m(\overline{v})]] & \to^{t:\texttt{no-op}} & H, T[t := E[e[\overline{x} := \overline{v}, \texttt{this} := p]]] & \text{if } p \text{ contains } m(\overline{x})\{e\} \\
P \vdash H, T[t := E[\texttt{let}\ x = v\ \texttt{in}\ e]] & \to^{t:\texttt{no-op}} & H, T[t := E[e[x := v]]] & \\
P \vdash H, T[t := E[p.f = v]] & \to^{t:\texttt{write}\ p.f\ v} & H[p.f := v], T[t := E[v]] & \\
P \vdash H, T[t := E[p.f]] & \to^{t:\texttt{read}\ p.f\ v} & H, T[t := E[v]] & H[p.f] = v \\
P \vdash H, T[t := E[\texttt{acq}\ p]] & \to^{t:\texttt{acq}\ p} & H[p := t], T[t := E[\texttt{null}]] & H[p] = \texttt{null} \\
P \vdash H, T[t := E[\texttt{rel}\ p]] & \to^{t:\texttt{rel}\ p} & H[p := \texttt{null}], T[t := E[\texttt{null}]] & \\
P \vdash H, T[t := E[\texttt{fork}\ p]] & \to^{t:\texttt{fork}\ p\ t'} & H, T[t' := p.\texttt{run}(), t := E[\texttt{null}]] & t' \text{ is fresh} \\
P \vdash H, T[t := E[\texttt{new}\ c()]] & \to^{t:\texttt{no-op}} & H, T[t := E[p]] & \text{where } p \text{ is fresh}
\end{array}$$

Figure 3. EscapeJava: Syntax and Semantics

$$\boxed{P \vdash G, H, T \to^{a}_{a} G, H, T}$$

$$\frac{P \vdash H, T \to^{t:\texttt{no-op}} H, T'}{P \vdash G, H, T \to^{t:\texttt{no-op}}_{t:\texttt{no-op}} G, H, T'} \qquad \frac{P \vdash H, T \to^{t:\texttt{acq}\ p} H, T'}{P \vdash G, H, T \to^{t:\texttt{acq}\ p}_{t:\texttt{acq}\ p} G, H, T'} \qquad \frac{P \vdash H, T \to^{t:\texttt{rel}\ p} H, T'}{P \vdash G, H, T \to^{t:\texttt{rel}\ p}_{t:\texttt{rel}\ p} G, H, T'}$$

$$\frac{\begin{array}{c} P \vdash H, T \to^{t:\texttt{write}\ p.f\ v} H', T' \\ p \notin G \end{array}}{P \vdash G, H, T \to^{t:\texttt{write}\ p.f\ v}_{t:\texttt{no-op}} G, H', T'} \qquad \frac{\begin{array}{c} P \vdash H, T \to^{t:\texttt{write}\ p.f\ v} H', T' \\ G' = G \cup \mathit{reachable}(H, v) \\ p \in G \end{array}}{P \vdash G, H, T \to^{t:\texttt{write}\ p.f\ v}_{t:\texttt{write}\ p.f\ v} G', H', T'} \qquad \frac{\begin{array}{c} P \vdash H, T \to^{t:\texttt{fork}\ p\ t'} H, T' \\ G' = G \cup \mathit{reachable}(H, p) \end{array}}{P \vdash G, H, T \to^{t:\texttt{fork}\ p\ t'}_{t:\texttt{fork}\ p\ t'} G', H, T'}$$

$$\frac{\begin{array}{c} P \vdash H, T \to^{t:\texttt{read}\ p.f\ v} H, T' \\ p \notin G \end{array}}{P \vdash G, H, T \to^{t:\texttt{read}\ p.f\ v}_{t:\texttt{no-op}} G, H, T'} \qquad \frac{\begin{array}{c} P \vdash H, T \to^{t:\texttt{read}\ p.f\ v} H, T' \\ p \in G \end{array}}{P \vdash G, H, T \to^{t:\texttt{read}\ p.f\ v}_{t:\texttt{read}\ p.f\ v} G, H, T'}$$

Figure 4. Dynamic Escape Analysis and Race Check Elision Algorithm

We next prove that our analysis preserves validity.

Lemma 5.2 (Preservation). *If G, H, T is valid and $P \vdash G, H, T \rightarrow^{a}_{b} G', H', T'$ then G', H', T' is valid.*

Proof. Case analysis of $P \vdash G, H, T \rightarrow^{a}_{b} G', H', T'$ □

The proof of our desired correctness property in one direction is straightforward.

Theorem 5.3. *If G, H, T is valid and $P \vdash G, H, T \longrightarrow^{\alpha}_{\beta} G', H', T'$ and β has a race then α has a trace.*

Proof. Suppose β has a race between two concurrent conflicting accesses b_1 and b_2, then b_1 and b_2 also appear in α. By case analysis on $P \vdash G, H, T \longrightarrow^{\alpha}_{\beta} G', H', T'$, no acquire, release, or fork actions are elided so these actions remain the same in both α and β. Therefore b_1 and b_2 are also concurrent and conflicting in α, and so α has a race. □

To prove the other implication we must show that no accesses involved in the first race in α have been elided in β.

Theorem 5.4. *If G, H, T is valid and $P \vdash G, H, T \longrightarrow^{\alpha}_{\beta} G', H', T'$ and α has a race then β has a race.*

Proof. Let a_1, a_2 be the first race in α where $p = target(a_1)$ and $p = target(a_2)$. By induction on the length of α, without loss of generality assume $\alpha = a_1.\alpha'.a_2$ where $a_1.\alpha'$ is race-free:

- If $a_1 \in \beta$ then $p \in G$ so $a_2 \in \beta$ (since G is increasing by Lemma 5.6 below) so β has a race.
- If $a_1 \notin \beta$ then we have:
  $\alpha = a_1.\alpha'.a_2$
  $p \notin G$
  $p \in addr(a_1)$
  $p \in addr(a_2)$
  α' is race-free
  By lemma 5.5 below, $a_1 <_\alpha a_2$ and we have a contradiction.

□

We next prove two auxiliary lemmas required by the above proof: First, if two actions a_1 and a_2 access the same address p, where p is not in G at the time of a_1, and no race occurs between a_1 and a_2, then a_1 and a_2 are ordered by happens-before. Intuitively, this ordering arises from the race-free transmission of p from $tid(a_1)$ to $tid(a_2)$. There must exist some pair of actions a'_1 and a'_2 which write p to a shared object and read p from that object in a race-free manner. As a'_1 and a'_2 are ordered by happens-before, this same ordering applies to a_1 and a_2.

Lemma 5.5. *Suppose*

$$\begin{array}{l} G, H, T \textit{ is valid} \\ P \vdash G, H, T \longrightarrow^{\alpha}_{\beta} G', H', T' \\ \alpha = a_1.\alpha'.a_2 \\ p \notin G \\ p \in addr(a_1) \\ p \in addr(a_2) \\ \alpha' \textit{ is race-free} \end{array}$$

Then $a_1 <_\alpha a_2$

Proof. By induction on the length of α. Let $t_1 = tid(a_1)$ and $t_2 = tid(a_2)$. We proceed by case analysis on how thread t_2 recieved the address p.

- If $t_1 = t_2$ then $a_1 <_\alpha a_2$ by program order.
- If α contains t_1 : `fork` p t_2 then $a_1 <_\alpha a_2$ by the fork ordering.
- Otherwise t_2 read p from some shared location $q.f$ previously written by some thread t_3 (which may or may not be t_1). Thus $\alpha' = \alpha_1.a'_1.\alpha_2.a'_2.\alpha_3$ where
  - $a'_1 = t_3$: `write` $q.f$ p and
  - $a'_2 = t_2$: `read` $q.f$ p

  then:

$p \notin G, p \in addr(a_1)$	Given
$p \in addr(a'_1)$	By Construction
Let $\alpha' = \alpha_1.a'_1.\alpha_2.a'_2.\alpha_3$	
$a_1.\alpha_1$ is race-free	Because α' is race-free
$\lvert\alpha_1\rvert < \lvert\alpha\rvert$	
$a_1 <_\alpha a'_1$	By induction on $\lvert\alpha\rvert$
$a'_1 <_\alpha a'_2$	By α' is race-free
$a'_2 <_\alpha a_2$	By program order
$a_1 <_\alpha a_2$	By transitivity

□

Theorem 5.4 relies on the fact that G is monotonically increasing.

Lemma 5.6. *If $P \vdash G, H, T \rightarrow^{a}_{b} G', H', T'$ then $G \subseteq G'$.*

Proof. By case analysis on $P \vdash G, H, T \rightarrow^{a}_{b} G', H', T'$. □

Our main correctness result is then a straightforward combination of the above two theorems.

Theorem 5.7 (Trace Precision). *If G, H, T is valid and*

$$P \vdash G, H, T \longrightarrow^{\alpha}_{\beta} G', H', T'$$

then α has a race if and only if β has a race.

Proof. By Theorem 5.4 and Theorem 5.3. □

6 Related Work

A memory model describes what behaviors are permitted by a program, and consequently what optimizations and program transformations a compiler may perform. Sequential consistency [1] is a simple memory model but it prohibits many common and desirable optimizations. The Java Memory Model [11] is a weaker memory model, and therefore enables more optimizations.

Ferrara [8] describes the consequences of the Java memory model for static analysis, including total escape analysis. Unfortunately, the allowed optimizations under the Java memory model are complex, Sevcík and Aspinall [17] detail a variety of unsound compiler optimizations dealing with race conditions.

Compilers use escape analysis for a variety of optimizations. The two most common being lock elision for thread-local locations and allocating temporary objects on the stack as opposed to the heap. Choi et al. [4, 14] provide the original total escape analysis implemented in the Java Hotspot compiler. They describe a variety of optimizations that can be performed with this analysis and provide a proof of correctness for their total escape analysis and optimizations. They later improve on this work with a faster analysis that does not lose precision [5].

A variety of papers extend this initial analysis to either add functionality, improve speed, or improve precision [2, 9, 22]. Salcianu and Rinard [15] use a modified total escape analysis for allocating memory in a region based manner in order to aid garbage collection. Stadler et al. [19] extend the total escape analysis computation into a partial escape analysis. Their analysis extends the total escape analysis to be flow sensitive in order to allow for optimizations of objects that only escape on some paths. They call this analysis a partial escape analysis. Unfortunately, this added precision is unsound when applied to lock elision, as shown in Section 3. They also make use of inlining to improve their partial escape analysis, a technique also used by Shankar et al. [18].

In addition to compilers, many race detection algorithms use escape analyses. Naik et al. [12] use a total escape analysis in their static race detector, as do Voung et al. [21]. Their analysis performs multiple passes, with a final expensive lockset pass at the end to compute a set of locations where races may occur. Their earlier total escape analysis pass removes any variables that do not escape into another thread from the analysis. This race check elision pass does not reduce their precision beyond that of their other optimization passes.

This technique of using a fast, static escape analysis to improve the speed of another, later analysis is also used by von Praun and Gross [20]. They add a total escape analysis to the dynamic lockset algorithm Eraser [16]. They use this total escape analysis to perform race check elision for memory that never escapes into multiple threads. This optimization does not weaken their correctness guarantees.

Nishiyama [13] uses partial escape analysis in a dynamic race detector. He implements a low level lockset-based algorithm in the Hotspot Java virtual machine. His analysis uses a partial escape analysis based on a read barrier to produce a subset of objects that must be instrumented. Objects are not included in the analysis until the read barrier detects reads from multiple threads on the same address. The use of the read barrier, as opposed to a write barrier, means this analysis is not trace precise.

Likewise, Christiaens and Bosschere make use of a similar partial escape analysis to filter results for a vector clock based dynamic race detector [6]. They implement the vector clock checks at a Java machine code level. They also implement a partial escape analysis at this level that matches our own closely. In addition, they integrate the analysis with the Java garbage collector in order to remove previously global objects from the global set as they become unreachable. Their use of partial race check elision makes the analysis trace precise, instead of address precise. Partial escape analyses can also be implemented at a higher level than the Java machine code as done by Harrington and Freund [10].

7 Conclusion

Compared to a total escape analysis, a partial escape analysis *seems* better in that it characterizes more objects as “thread-local”, but particular care is needed when using this “currently thread-local” information to drive optimizations. We show how partial lock elision optimizations in the literature are in fact unsound. Conversely, we also prove that partial race check elision optimizations *are* sound, albeit with a weaker notion of precision.

Acknowledgment

This work was supported, in part, by NSF Grants 1337278, 1421051, 1421016, and 1439042.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29, 12 (1996), 66–76.

[2] Bruno Blanchet. 1999. Escape Analysis for Object-Oriented Languages: Application to Java. In *OOPSLA*. 20–34.

[3] Hans-Juergen Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *PLDI*. 68–78.

[4] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. 1999. Escape Analysis for Java. In *OOPSLA*. 1–19.

[5] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. 2003. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.* 25, 6 (2003), 876–910.

[6] Mark Christiaens and Koen De Bosschere. 2001. TRaDe: A Topological Approach to On-the-Fly Race Detection in Java Programs. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. 105–116.

[7] Steve Dever, Steve Goldman, and Kenneth Russell. 2006. New Compiler Optimizations in the Java HotSpotffl Virtual Machine. In *JavaOne Conference*.

[8] Pietro Ferrara. 2008. Static Analysis Via Abstract Interpretation of the Happens-Before Memory Model. In *TAP*. 116–133.

[9] David Gay and Bjarne Steensgaard. 2000. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *Compiler Construction, 9th International Conference*. 82–93.

[10] Emma Harrington and Stephen N Freund. 2014. Using Escape Analysis in Dynamic Data Race Detection. *Williams College Technical Report CSTR 201401* (2014).

[11] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *POPL*. 378–391.

[12] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *PLDI*. 308–319.

[13] Hiroyasu Nishiyama. 2004. Detecting Data Races Using Dynamic Escape Analysis Based on Read Barrier. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*. 127–138.

[14] Michael Paleczny, Christopher A. Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*.

[15] Alexandru Salcianu and Martin C. Rinard. 2001. Pointer and escape analysis for multithreaded programs. In *PPOPP*. 12–23.

[16] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (1997), 391–411.

[17] Jaroslav Sevcík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*. 27–51.

[18] Ajeet Shankar, Matthew Arnold, and Rastislav Bodík. 2008. Jolt: lightweight dynamic analysis and removal of object churn. In *OOPSLA*. 127–142.

[19] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial escape analysis and scalar replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 165.

[20] Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. In *OOPSLA*. 70–82.

[21] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: static race detection on millions of lines of code. In *SIGSOFT*. 205–214.

[22] John Whaley and Martin C. Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *OOPSLA*. 187–206.