

# A Two-Phase Escape Analysis for Parallel Java Programs

Kyungwoo Lee  
Purdue University  
West Lafayette, IN

kwlee@purdue.edu

Samuel P. Midkiff  
Purdue University  
West Lafayette, IN

smidkiff@purdue.edu

## ABSTRACT

Thread escape analysis conservatively determines which objects may be accessed in more than one thread. Thread escape analysis is useful for a variety of purposes – finding races in multi-threaded programs, removing useless synchronization, allocating data to thread-local heaps, and compiling to target more strict consistency models. Thread escape analyses are often interprocedural, and interprocedural analyses are generally either too slow to perform at runtime in dynamic systems, or trade-off significant amounts of precision for speed. This paper describes a two-phase offline/online interprocedural and inter-thread escape analysis that is faster and more accurate, on average, than previously published analyses. By performing an offline pre-analysis followed by a dynamic online analysis that integrates offline results with dynamic information, significant improvements in performance and accuracy are achieved. For compiling Java programs under a sequentially consistent memory model, our approach enables application executions that are, on average, 1.5 times faster than those using the previous fastest online algorithm, with only 80% of the online compilation time.

## Categories and Subject Descriptors

D.3.2 [Language Classifications]: Object-oriented languages; D.3.4 [Processors]: Compilers, optimization, incremental compilers; F.3.2 [Semantics of Programming Languages]: Program analysis

## General Terms

Languages, Performance

## Keywords

Java, escape analysis, memory models, consistency models, compilation models

This material is based upon work supported by the NSF under Grant No. CCR-0081265. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FACT'06, September 16–20, 2006, Seattle, Washington, USA.

Copyright 2006 ACM 1-59593-264-X/06/0009 ...\$5.00.

## 1. INTRODUCTION

A variety of interprocedural analyses exist in offline compilers, but are too expensive to be used in dynamic, or just-in-time, compilers. Dynamic compilers typically use local, or intraprocedural analyses to reduce the runtime compilation overhead. An important interprocedural analysis is thread escape analysis (hereafter referred to as *escape analysis*), which determines if an object is accessible from more than one thread.

Escape analysis is useful for a variety of reasons. It can be used to determine useless synchronization by identifying locks that are only accessed in a single thread [25, 7]; it can be used to determine objects that may be involved in races, and thereby reduce the overhead of race detection [9]; and it can be used to support allocators and garbage collectors that are more efficient [12, 26]. This research is part of a larger project – the Pensieve project – where escape analysis is used to efficiently compile programs under a sequentially consistent memory model, with optimizations performed where legal.

The Pensieve system performs five analyses that are of interest to this paper. The first two are other escape analyses, that of [32], which is heretofore the fastest, on average, escape analysis known, and that of [29], which is both much slower and much more accurate. The third analysis is *thread structure analysis* [29], which determines which code, in different threads, can execute at the same time as other code, and whose results are used by delay set analysis. The fourth analysis is *delay set analysis* [27, 29], which intuitively performs a form of dependence analysis that reflects the effect of memory accesses in different threads on the legality of reordering memory accesses in a method being compiled. A delay between two memory references requires both that the compiler not reorder these references at compile time, and insert a memory barrier, or fence, if necessary to ensure that the memory references are not reordered at runtime. Delay set analysis only needs to consider thread-escaping variables, and the accuracy of the escape analysis directly affects the accuracy of this analysis, and the number of optimizations that can legally be performed. The fifth analysis and transformation efficiently inserts hardware fences along each delay to enforce the delays in hardware at runtime.

This paper describes a two-phase interprocedural escape analysis with nearly the precision of slower analyses, yet with an analysis time that is lower than the previous fastest known analysis. This two-phase analysis is applied to the problem of compiling and optimizing Java programs in a dynamic compilation systems while implementing a sequen-

tially consistent memory model. To be useful in dynamic compilation systems, the analysis should be both fast and precise. We meet these conflicting demands by using a slower offline analysis to pre-compute much of the information needed to support a fast and precise online analysis. At runtime, it is necessary to perform a whole program analysis, and to properly account for changes in classes that occurred between when the classes were analyzed in the offline analysis, and when they are loaded during a particular execution. This is done by projecting the information accumulated in the offline analysis onto a simpler representation that can be rapidly interprocedurally propagated at runtime. The combination of the offline and online analyses leads to an analysis that is, on average, faster and more accurate than any other analysis known to the authors.

This paper makes the following contributions:

- It describes a new, fast (at runtime), and precise escape analysis that is safe in the presence of partial program information, and that uses information computed offline to make the online phase faster.
- It describes a lightweight data structure, the *level summary*, that compactly represents thread escape information for objects and enables a fast interprocedural analysis.
- It presents experimental data showing the precision and the runtime overhead of this algorithm compared with state-of-the-art algorithms for dynamic optimization systems when applied to the problem of compiling for sequential consistency.

The rest of the paper is organized as follows. Section 2 describes our two-phase escape analysis. Section 3 presents our experimental results. Section 4 discusses work related to our technique. Finally, Section 5 gives our conclusions.

## 2. THE TWO-PHASE ESCAPE ANALYSIS ALGORITHM

Our escape analysis consists of two phases: an expensive, but precise offline phase followed by a less expensive online phase that exploits information found in the offline phase. Information is passed from the offline to the online phase via classfile annotations.

### 2.1 The Offline Analysis

In this section we discuss the properties that any offline algorithm used by our analysis must have.

An object  $O$  is said to thread-escape if it may be accessed in two or more threads. Conservatively,  $O$  thread-escapes when it is:

1. referenced in a chain of references from a static field – the objects reachable from a static field are potentially accessible anywhere in the program, and thus may be accessed in two or more threads.
2. referenced in a chain of references from a thread object  $O_t$ , since  $O_t$  is created (and thus accessed) in a parent thread  $O_p$ , and the objects reachable from  $O_t$  may be accessed in both  $O_p$  and  $O_t$  (i.e. in two or more threads).

```

Class C {
  static S s
  ...
  void foo(C1 z, C2 w, C3 y) {
    ...
    b.fl = z;
    b.f2 = w;
    b.fl = x;
    x.fl = y;
    goo(b);
    s = x;
  }
  ...
}

```

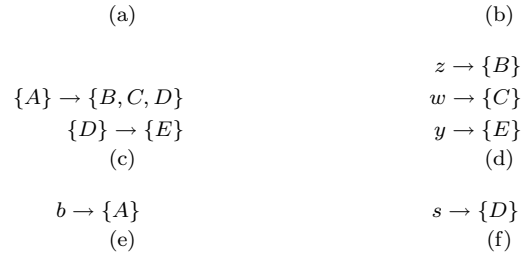
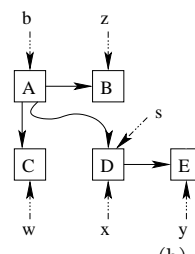


Figure 1: A Connection Graph Example

Thus determining whether  $O$  thread-escapes can be viewed as a reachability problem. Either the points-to graph developed by some analyses (e.g. [31]) or a connection graph (CG) (see, e.g. [8]) provide this reachability information.

Figure 1(a) shows a program fragment and Figure 1(b) shows an associated (simplified) CG. Each box in the graph represents an object, where objects created at different `new` sites are considered different. An edge  $(O_1, O_2)$ , from the box for  $O_1$  to the box for  $O_2$ , indicates that some field of  $O_1$  may reference  $O_2$  (the CG shown is field insensitive). Objects in the graph are pointed to (with a dashed line) by a label indicating the corresponding variable in the program which references the object. Note that if an object is referenced by  $n$  objects, it would be adjacent to  $n$  solid edges. The objects B, C, and D (referenced by  $z$ ,  $w$ , and  $x$  respectively) are referenced by fields of the object A (which is referenced by  $b$ ): this is indicated by edges (A, B), (A, C) and (A, D). Moreover the object E, referenced by  $y$ , is referenced by a field of the object D, which is indicated by the edge (D, E). Therefore the object A indirectly references the object E since a path exists from A to E. Thus, for example, by applying the first escape condition, above, to the graph, D and E can be determined to thread-escape because they are reachable by the class static field  $s$ .

Our system, uses a slightly modified version of [8] for our offline escape analysis. As in [8], an object reachable from a parameter may be represented by more than one node in the CG. It differs from [8] in two ways. First, we perform a partial program analysis, since the entire program may not be available until it is dynamically executed. The analysis, when confronted with missing methods and classes, optimistically “accounts” for the effects of the call by ignoring the effects of the call on the CG. The effects of the call will be incorporated during the runtime analysis when the called method will necessarily be available before it is compiled and executed. Second, we do not actually perform an offline escape analysis using the CG. The CG contains the information necessary to build the fast online analysis.

At the conclusion of the offline analysis, the CG for each

method of class `C` is written to `C.class` as an annotation. Figure 1(c) through Figure 1(f) show the four categories of annotations that are placed into the class file:

1. Reachability information for objects directly referenced by an object  $O$  is recorded. This is shown in Figure 1(c), where, for example, it is shown that `A` directly reaches `B`, `C`, and `D`. While this information has a worst-case space complexity of  $O(n^2)$ , where  $n$  is the number of statically identified objects, experimental data shows that in practice space is not a problem.
2. Reachability information for objects reachable from a parameter of the method. This is shown in Figure 1(d).
3. Reachability information for objects reachable from an argument to a method invocation. This is shown in Figure 1(e).
4. Reachability information for objects reachable from a static field or a field in a thread object. An example of this is shown in Figure 1(f).

All this information is straight-forwardly derived from the CG.

## 2.2 The Online Analysis

The online interprocedural escape analysis (OIPA) uses the CG that is written to the class file as annotations. The OIPA interacts heavily with the Jikes RVM [5] classloader and the Jikes RVM adaptive compiler system. Figure 2 gives an overview of the OIPA system and its interactions with the Jikes RVM.

Classes are loaded using a modified primordial classloader. The classloader examines the bytecode of the methods in the loaded class and adds call site and call target information to the *Call Graph Database*. The classloader locates the CG information in the class file annotations and adds it to the *Escape Summary Database*. The modified primordial classloader pre-loads classes when possible, but the OIPA does not require all classes to be loaded.

When a method is invoked for the first time, it is compiled by the Jikes RVM *baseline compiler*. The baseline compiler performs a rapid bytecode to native code translation of the class file. Few optimizations and analyses are performed by the baseline compiler, and in particular all objects are conservatively assumed to thread-escape.

The Jikes RVM adaptive compilation system [6] monitors executing methods, and when a sufficiently hot method is found, the Jikes RVM optimizing compiler compiles the method. The optimizing compiler makes use of the OIPA to provide escape information.

As described in Section 2.2.1, OIPA is invoked any time the call graph of the program changes. The OIPA itself involves four sub-tasks. The first builds a call graph using information in the *Call Graph Database*. This is described in Section 2.2.1. The second performs thread information analysis to get more precise information about when class static fields are accessed in different threads, and when the fields of thread objects are accessed in different threads. This is described in Section 2.2.3. The third interprocedurally propagates escape information for static fields and thread fields that was accumulated in the second step. This propagation, and the efficient, non-CG representation that supports

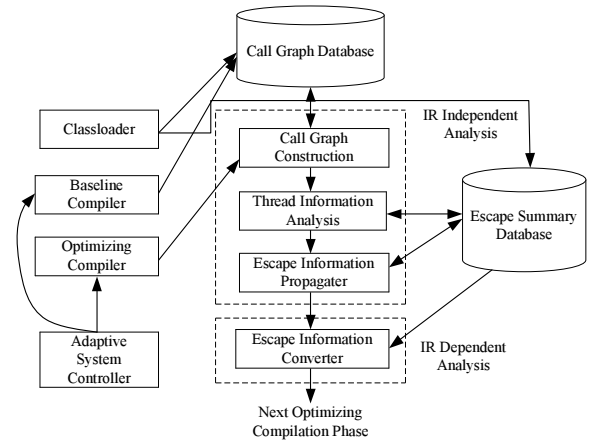


Figure 2: Online Analysis Configuration

it, is described in Section 2.2.4. Finally, an intraprocedural analysis is performed to convert the escape information into a form compatible with the Jikes RVM intermediate representation (IR), so it can be used by other, pre-existing analyses.

### 2.2.1 Call Graph Construction

The OIPA is heavily dependent on the precision of the call graph, which must approximate the targets of polymorphic calls. Because dynamic compilation systems are aware of what classes have been loaded, fast and simple algorithms such as class hierarchy analysis (CHA) [11] can be used to build precise call graphs.

Two situations cause our system to construct a new call graph. First, a call graph is built when the optimizing compiler begins its first compilation. Second, if a new class is loaded, CHA is done immediately to determine if the call graph has changed. If it has, then the new method in the call graph might invalidate the optimistic assumptions made by earlier OIPA<sup>1</sup>. In this case code invalidation is done, a new OIPA is performed, and the invalidated code is re-compiled.

### 2.2.2 The OIPA Lattice

The OIPA sub-tasks described next are performed using the lattice  $ThreadEscape < Thread_i < NoEscape$ , where  $i$  is a unique thread identifier. An object, reference or field being analyzed will be initialized to *NoEscape*. The  $Thread_i$  state denotes that the object, reference or field is reachable from the thread labeled  $i$ , and the *ThreadEscape* state denotes that the object, reference or field may be simultaneously reachable from two or more different threads, and thread-escapes. Intuitively, we perform constant propagation over  $Thread_i$ , with *ThreadEscape* indicating an escaping entity is not reached by a single thread id value  $i$ . We note that in the worst case on-stack replacement might be required [14].

At merge points in the call graph, the escape state for an object, reference or field is given by the meet function  $\sqcap$  (where  $es_i$  and  $es_j$  are different escape states):

$$\begin{aligned} es_i \sqcap es_i &= es_i \\ es_i \sqcap es_j &= es_i, \quad es_i < es_j \\ Thread_i \sqcap Thread_j &= ThreadEscape, \quad i \neq j \end{aligned} \quad (1)$$

<sup>1</sup>Code invalidation has not been necessary with the tested benchmarks.

### 2.2.3 Thread Information Analysis

As mentioned in Section 2.1, there are two ways an object  $O$  may be accessed in multiple threads:  $O$  is accessed via a chain of references from a static field or  $O$  is accessed via a chain of references from a field in a thread object  $O_t$ . Naively assuming all objects escape that are accessible from a static field or a thread field is too conservative. To be more precise, we only initialize some static fields, and fields of some threads, as *ThreadEscape*, as described below.

A chain of references that causes a static field to reach an object  $O$  will directly result in  $O$  thread-escaping only if two or more threads access the static field. Similarly, a chain of references from some thread fields will alone cause the object to thread-escape only if a chain of references from another thread object can reach the object. We now describe a dataflow analysis that allows us to apply these less conservative conditions.

#### 2.2.3.1 Determining static field initializations.

Our algorithm detects static fields that may be accessible by multiple threads similar to what is done in [25]. Unlike the algorithm of [25], we are interested in all objects, not just objects used as locks.

First, the algorithm initializes all methods and static fields to *NoEscape*, and then assigns a unique ID  $i$  to the thread allocation site, and the escape state *Thread<sub>i</sub>* to the **run** method associated with the thread allocation site. Thread allocation sites involved in cycles in the call or control flow graph of the allocating method may allocate multiple threads, therefore the **run** methods associated with these thread allocation sites are conservatively marked *ThreadEscape*. The **main** Java method runs in its own thread, and is assigned the unique escape state *Thread<sub>main</sub>*.

By traversing the call graph in top-sort order, methods that can be invoked (directly or indirectly) from a **run** method are marked with an escape state by applying the lattice meet operation (Equation 1) to the method’s current escape state and the propagated **run** method’s escape state. The traversal iterates over strongly connected components (SCC) in the graph.

Once methods are labeled with an escape state, the escape state of each static field accessed within the marked methods is updated by applying the  $\sqcap$  operator to the current state of the static field and the escape state of the method. This results in static fields that may be reached by more than one thread to be marked as *ThreadEscape*.

#### 2.2.3.2 Determining thread and thread field initializations.

Every thread  $T$ , and its fields, are accessible in at least two threads – the creating (or parent) thread, and the thread  $T$  itself, via the **this** pointer. Thus, a naive analysis would show that every thread object  $O_t$ , and all objects reachable from it, thread-escape. A more precise analysis is possible, however, by using the technique of [29, 32], where the escape state of  $O_t$  is considered to be greater than *ThreadEscape* (i.e. not thread-escaping) when the following two conditions are true.

First, if  $O_t$  is not connected to (i.e. reachable from) any parameter of the thread creating method  $m$  of the parent thread  $O_p$ , then  $O_t$  is not accessible outside of  $m$ , and hence cannot be accessed by another thread spawned outside of

$m$ , except via a static field. Static fields are not considered now because this analysis, and the just described analysis of static fields, are performed independently. The interprocedural propagation phase will account for the effect of static fields.

Second, if  $O_t$  is used only in **start()** and **join()** method invocations in  $m$  of  $O_p$ , i.e. there is no access to  $O_t$  between these two, then  $O_p$  and  $O_t$  never access  $O_t$  at the same time. If both conditions are true, the escape state of  $O_t$  is *Thread<sub>O<sub>t</sub></sub>*. Otherwise  $O_t$  is marked as *ThreadEscape* (i.e. *Thread<sub>O<sub>t</sub></sub>*  $\sqcap$  *Thread<sub>O<sub>p</sub></sub>*).

This in turn sharpens the analysis of objects reachable from the fields in  $O_t$ , since under the above conditions (i.e.  $O_t$  does not thread-escape), a field in  $O_t$  reaching an object directly causes it to thread-escape only when  $O_p$  also accesses the field in  $O_t$  (all fields in  $O_t$  are assumed to be accessed in  $O_t$  itself). Thus the above conditions allow us to examine only fields accessed in the constructor for  $O_t$ , or in the thread creating method of  $O_p$ . Those accessed fields in  $O_t$  reached from  $O_p$  are conservatively marked as *ThreadEscape* since shared objects may be assigned to those fields.

### 2.2.4 Interprocedural Escape Information Propagation

Next, the OIPA algorithm interprocedurally propagates escape information for objects referenced, directly or indirectly, from static and thread fields. Since static fields are global, and all other interprocedural interactions in Java take place through parameters (or objects reachable from them by a chain of references), it is sufficient to interprocedurally propagate the effects of arguments on parameters, and parameters on arguments, with a local propagation within each method capturing the effect of chains of references from the parameters and arguments.<sup>2</sup>

The interprocedural propagation takes place in two passes over the call graph: a bottom-up (reverse top-sort order) traversal of the graph and a top-down (top-sort order) traversal of the graph. Both iterate over SCC in the call graph. The bottom-up pass propagates the effect of the escape states of objects reachable from the callee parameter on those from the caller arguments. The top-down pass propagates the effects of the escape states of objects reachable from the caller arguments on objects reachable from the callee parameters. Propagation of information is similar in the top-down and bottom-up traversals. For brevity and simplicity, we confine our descriptions, definitions, discussion and examples to the bottom-up case, unless otherwise stated.

Note that when the adaptive compilation system optimizes a method in a class, and OIPA is performed, all classes may not be loaded due to lazy loading of classes; some loaded classes may not have been analyzed with the offline system, and may not contain a CG; some loaded classes may have changed, resulting in CGs changing after they were analyzed with other classes in the offline system; and the loaded class may be binary compatible with an analyzed class, but contain different logic and therefore different annotations than would have been present with the original class. All of these require an interprocedural updating of the CG information.

<sup>2</sup>Reflection can also cause objects to escape, but it does not occur in our benchmark suite, and we, like the analyses of [25, 32, 7, 8], do not handle reflection.

To reduce the runtime analysis cost, we use a *level summary*, which is a conservative approximation of the CG.

We now give a sequence of definitions culminating in the definition of a level summary.

DEFINITION 1. *Let there be the chain of references from object (reference)  $O_i$  to object  $O_j$ :*

$$O_i.f_1.f_2.\dots.f_n \text{ references } O_j, \quad (2)$$

where  $f_1, f_2, \dots, f_n$  are field references. Then, the (field) referencing level from  $O_i$  to  $O_j$  is  $n$ , the number of field references needed to reach  $O_j$  from  $O_i$ , i.e.  $O_j$  is reachable from  $O_i$  at (field) referencing level  $n$ .

Thus, in Figure 1(b), the referencing level from A to E is 2.

DEFINITION 2. *Let object  $O_i$  reach object  $O_j$  via  $m$  different chains of references, with reference levels  $l_1, l_2, \dots, l_m$ . Then the least referencing level from  $O_i$  to  $O_j$  is the minimum value of  $l_1, l_2, \dots, l_m$ . Similarly, the greatest referencing level is the maximum value of  $l_1, l_2, \dots, l_m$ .*

DEFINITION 3. *Let  $e_1, e_2, \dots, e_k$  be the least referencing levels from an object (reference)  $O$  to  $k$  escaping objects. The escape referencing level is the minimum of  $e_1, e_2, \dots, e_k$ . The escape referencing level for  $O$  which only reaches objects with NoEscape is  $\infty$ .*

Thus, in Figure 1(b) D and E are both escaping if we assume the static field  $\mathbf{s}$  is reachable from two or more threads. The referencing levels from A to D and E are 1 and 2 respectively, thus the escape referencing level for A, or  $\mathbf{b}$  that references A, is 1.

DEFINITION 4. *The level summary associated with an object (reference)  $O$  is a tuple  $\langle \text{level}, \text{es} \rangle$ , where level is the escape referencing level and es is the least escape state of an object reachable from  $O$ .*

Therefore, if an object  $O$  reaches two objects,  $O_1$  and  $O_2$ , with escape referencing levels  $l_1$  and  $l_2$ , and escape states  $\text{ThreadEscape}$  and  $\text{Thread}_i$ , the level summary for  $O$  will be  $\langle \min(l_1, l_2), \text{ThreadEscape} \rangle$ . In Figure 1(b) the level summary for A (or  $\mathbf{b}$ ) is  $\langle 1, \text{ThreadEscape} \rangle$ .

Intuitively, the level summary only keeps track of *where* (at which referencing level) escape happens. The OIPA conservatively assumes that an object  $O$  with the level summary  $\langle \text{level}, \text{es} \rangle$ , causes all objects reachable from  $O$  in *level* or more steps have an escape state of at least *es*. Thus in Figure 1(b) the level summary for A would cause B and C to also escape. The level summary for  $O$  can easily be computed by traversing the chains of objects in the CG.

Figure 3 shows the OIPA algorithm for interprocedurally propagating escape information in the backwards traversal using level summaries. At each call site, level summaries are constructed for parameters of the callee (lines 1-2). The caller's escape information is updated using the level summaries (lines 4-8). If the greatest referencing level from an argument of the caller,  $l_{A_i}$  (which can be also computed by traversing the chains of objects in the caller's CG), is less than the *level* (i.e.  $l_{P_i}$ ) of the level summary for the corresponding parameter (i.e. there is no corresponding referencing level in the caller), the algorithm conservatively merges the escape states of objects reachable from the argument  $A_i$  with the *es* (i.e. the escape state of the parameter  $i$ ,  $e_{P_i}$ )

```

m is the caller method, m' is the callee
Argument  $A_i$  of m corresponds to parameter  $P_i$  of m'
1 foreach  $P_i$  of the callee m'
2   Build the level summary  $\langle l_{P_i}, e_{P_i} \rangle$  for  $P_i$ 
3   if  $e_{P_i} \neq \text{NoEscape}$ 
4     Calculate the greatest referencing level  $l_{A_i}$ 
       for argument  $A_i$ 
5   if  $l_{A_i} \geq l_{P_i}$ 
6     Apply  $\sqcap$  to escape states of objects reachable
       from  $A_i$  at the level  $l_{P_i}$  and more steps with  $e_{P_i}$ 
7   else
8     Apply  $\sqcap$  to escape states of objects reachable
       from  $A_i$  at the level  $l_{A_i}$  with  $e_{P_i}$ 
9 endfor

```

Figure 3: Online Interprocedural Analysis Algorithm (Backward Traversal).

of the level summary at level  $l_{A_i}$  (lines 7-8). This allows the algorithm to safely update the escape information for the caller without updating the CG (which would be expensive) since objects at the lower referencing level still have the merged escape state of objects at the higher referencing level.

Unlike the offline interprocedural analysis algorithm, the OIPA algorithm is fast for two reasons. First, it constructs the level summaries of the callee in  $O(n)$  time, and it only needs  $O(a \cdot n)$  time to update the caller's escape information with the level summaries (where  $a$  is the number of call sites in the method, and  $n$  is the number of nodes in the CG). Second, we do intraprocedural propagation at each interprocedural step without analyzing the method body.

### 2.2.5 Escape Information Converter

The IR independent CG and level summary based information produced by the previous steps must be mapped onto the high-level intermediate representation (HIR) used by the Jikes RVM optimizing compiler. The goal of this phase is to determine the referencing level from an initial point (i.e. parameters or arguments) to each (object) reference. Once the referencing level for each reference is found, we can calculate the escape states of references using the level summary. A tuple  $\langle b, i, l, e \rangle$  is assigned to each reference in the IR:  $b$  is a bytecode index,  $i$  is the parameter or argument index,  $l$  is the referencing level, and  $e$  is the escape state. Each tuple is initialized with  $\langle \infty, \infty, 0, \text{NoEscape} \rangle$ . The first two fields,  $b$  and  $i$ , contain information about the initial point, i.e. a point external to the method from which the associated reference is reached.

The transfer functions of Table 1 are applied to the tuples for each statement in the IR as tuples are propagated from the initial points through the IR. The transfer functions (1) track the referencing level from the initial points (mentioned above) to the reference, and (2) merge escape states. When tuples from different initial points (i.e. the fields  $b$  or  $i$  are different) merge at some point in the program, the initial point of the tuple after applying the transfer function is the one with the least bytecode index. This is safe since the CG representation of the program will indicate that an object escapes regardless of the path used to reach the object.

Once the propagation is finished, each tuple is examined and the escape state of its corresponding reference is estimated. Tuples with  $\infty$  in the first field of the tuple are

Statement	Transfer Function
parameters $x_0, \dots, x_{k+1}$	<b>foreach</b> $x_i$ $X_i \leftarrow \langle 0, i, 1, NoEscape \rangle$ <b>endfor</b>
$x = y$	$X, Y \leftarrow X \sqcap Y$
$x.f = y$ $x[] = y$	$Y \leftarrow Y \sqcap X^+$
$x = y.f$ $x = y[]$	$X \leftarrow X \sqcap Y^+$
getstatic $x$ putstatic $x$	update escape state of $X$ with <i>Escape Summary Database</i>
method invocation $x_{k+1} = x_0.n(x_1, \dots, x_k)$	<b>foreach</b> $x_i$ update $X_i$ with $\langle bc\ index, i, 1, NoEscape \rangle$ <b>endfor</b>

$$\begin{aligned}
X &= \langle x_b, x_i, x_l, x_e \rangle \\
Y &= \langle y_b, y_i, y_l, y_e \rangle \\
X^+ &= \langle x_b, x_i, x_l + 1, x_e \rangle \\
Y^+ &= \langle y_b, y_i, y_l + 1, y_e \rangle
\end{aligned}$$

$$X \sqcap Y = \begin{cases} \text{if } x_b = y_b & \begin{cases} \text{if } x_i = y_i & \langle x_b, x_i, \min(x_l, y_l), x_e \sqcap y_e \rangle \\ \text{else } x_i > y_i & \langle x_b, y_i, y_l, x_e \sqcap y_e \rangle \\ \text{else } x_i < y_i & \langle x_b, x_i, x_l, x_e \sqcap y_e \rangle \end{cases} \\ \text{else } x_b < y_b & \langle x_b, x_i, x_l, x_e \sqcap y_e \rangle \\ \text{else } x_b > y_b & \langle y_b, y_i, y_l, x_e \sqcap y_e \rangle \end{cases}$$

**Table 1: Transfer Functions for Finding Reachability Information and Escape State for References**

not reachable from any initial point, but may be reachable from a chain of references from any static/thread field access site. Because every transfer function performs the meet operation on escape states, the last field of the tuple, the  $e$  (escape state) field, is directly used for these associated references. For other references (i.e. references without  $\infty$  in the first tuple position), each initial point is identified by the bytecode index and parameter or argument index, and the corresponding level summary built in the previous phase is retrieved to estimate the escape state of the reference using the  $l$  (referencing level) field of the tuple.

### 2.2.6 Soundness of the Algorithm in the Presence of Changed Classes

Our algorithm can use secure hash functions to ensure that the annotations associated with a class were generated by a trusted offline analyzer. The algorithm must also work if the trusted annotations are associated with a class  $C_{changed}$  that was originally analyzed in an offline partial program analysis with some class(es)  $C_{same}$ , but was changed<sup>3</sup>, and re-analyzed, possibly with classes other than  $C_{same}$ , afterwards. Thus the escape information for  $C_{changed}$  and  $C_{same}$  is not obviously compatible. We note that for methods being loaded online, but neither analyzed nor annotated offline, a conservative level summary (i.e.  $\langle 0, ThreadEscape \rangle$ ) for each parameter is used by the analysis.

We discuss the three elementary cases where  $C_{changed}$  might lead to incorrect analysis results, and show that this does not happen. Basically, we consider inter-thread and interprocedural effects of  $C_{changed}$  on the entire chain of

<sup>3</sup>The class  $C_{changed}$  itself can be changed, or a binary compatible class with different annotations could be loaded.

classes with these three cases. We first describe the case where  $C_{changed}$  has more parallelism than in the initial offline analysis, which may affect the escape states of thread fields or static fields. The second and third cases describe how the effect of  $C_{changed}$  is interprocedurally propagated to the unchanged  $C_{same}$  depending on caller or callee relations. An apparently different case is when both the caller and the callee change. This can be modeled as a changed caller invoking an unchanged null callee, which invokes the changed callee. Piecewise, this falls into the three cases.

$C_{changed}$  **spawns a thread that it did not spawn originally.** In this scenario, the call graph construction will detect that a new class is loaded, and a new method invocation is present (either to spawn the thread, or the constructor for the thread, or both). The OIPA will rebuild the call graph, perform thread information analysis, and invalidate code compiled using the previous OIPA results. Thus the effect of the new thread(s) on thread fields and static fields will be captured.

**Method  $m_{changed}$  in  $C_{changed}$  is invoked by a method  $m_{same}$  in some  $C_{same}$ .** Consider the backward propagation first. Let  $m_{same}$  pass one or more arguments  $a_1, a_2, \dots, a_n$  to parameters  $p_1, p_2, \dots, p_n$  of  $m_{changed}$ , and assume an object  $O_a$  escaped with the CG developed in the offline analysis, but do not with the new  $m_{changed}$ . Then the analysis is correct but conservative.

Next assume that  $O_a$  did not escape in the CG developed in the offline analysis, but should be detected as escaping with the new  $m_{changed}$ . Moreover, assume that some  $a_1, a_2$  both directly or indirectly reference  $O_a$  in  $m_{same}$ . As stated in [8], a single object  $O_a$  may be represented by two or more nodes in the CG. If  $O_a$  should be detected as escaping, a chain of references must exist from a thread field or a static field to nodes representing  $O_a$  in  $m_{changed}$ . Moreover, the level summary for at least one of  $p_1$  and  $p_2$  will cause nodes representing  $O_a$  in  $m_{changed}$  to escape. If only one node in  $m_{changed}$  represents  $O_a$ , the same level summary for both of  $p_1$  and  $p_2$  is computed and correctly propagated to  $O_a$  in  $m_{same}$  as a trivial case. If not,  $O_a$  in  $m_{same}$  will have a correct escape state by merging escape states with two level summaries for  $p_1$  and  $p_2$ . In the forward propagation,  $O_a$  reflects the level summary for both  $a_1$  and  $a_2$ . Therefore all nodes representing  $O_a$  in  $m_{changed}$  will receive the correct escape information.

**Method  $m_{changed}$  in  $C_{changed}$  invokes a method  $m_{same}$  in some  $C_{same}$ .** The argument for this is symmetrical to the above argument. From the point of view of method  $m_{changed}$  in  $C_{changed}$ ,  $m_{same}$  in  $C_{same}$  is the changed method, and  $m_{changed}$  is the unchanged method.

## 3. EXPERIMENTAL RESULTS

This section uses a collection of multi-threaded Java applications. `mtrt` is from the SpecJVM98 benchmark suite [3] and `moldyn`, `montecarlo`, and `raytracer` are from the Java Grande Forum Multi-threaded benchmark suite [1]. The rest are taken from the literature [20, 16], including the concurrent implementation of two data structures, hashmaps and queues.

### 3.1 Offline Analysis Result

We implemented our offline phase using the Soot optimizing framework [2] version 2.1.0. In our experiments, we assume that the offline analysis of an application (benchmark)

program does not have access to the library code, and that our offline analysis of the library does not have access to the application. That is, we separately analyze the benchmark and library code.

In our implementation, we turn off Soot’s bytecode-to-bytecode optimizations so that our experiments will measure the effects of our analysis. As shown in Table 2, our annotations increase the class file size from 12.9% to 90.6% and by 68.5% on average. As we will see in the following section, these annotations have a negligible impact on runtime performance.

	Classfile Size [kb]	Classfile Size with Annot. [kb]	Increases Ratio
mtrt	60.7	115.7	90.6%
moldyn	13.9	15.7	12.9%
montecarlo	36.85	49.1	33.2%
raytracer	21.1	28.9	37.0%
boundedbuf	10.1	12.3	21.8%
disksched	11.2	14.4	28.6%
geneticalgo	40.1	48.8	21.7%
hashmap	20.6	27.7	34.5%
seive	6.3	7.8	23.8%
library	4707.3	6869	45.9%
(classpath v0.0.8)			
avg			68.5%

Table 2: Class File Size Increase by Annotations

### 3.2 Online Analysis Result

The platform we used for the experiments is a Dell PowerEdge 6600 SMP with 4 Intel 1.5Ghz Xeon processors, each with 1MB cache each. The system has a total of 6GB memory and runs Red Hat Linux Release 9. Our online system is implemented in the Jikes RVM [5] version 2.3.1. We ran our experiments using a FastAdaptiveSemiSpace configuration. In this configuration, adaptive optimization is enabled and a copying garbage collector is used.

The Jikes RVM is itself written in Java, and we assume that the VM code is well-synchronized and does not need fences inserted to maintain sequential consistency. Note that if the Jikes RVM were written in C or C++, as other VMs are, our dynamic analyses and optimizations would not be needed for the VM code. Therefore, we do not analyze VM classes whose names start with ‘com.ibm.JikesRVM’, instead we make optimistic assumptions when these classes appear in our call graph, and we neither insert additional fences nor perform fence insertion optimization on those classes.

We evaluate our online escape analysis in two ways. First, we present the details of our online analysis cost in the adaptive compilation system. Second, we evaluate our online escape analysis in terms of compilation time overhead by comparing it to the fastest (on average) known online escape analysis, that of [32] (`connect3`), and the field-sensitive type-based escape analysis of [29] (`field-type`). As the worst case, a naive escape analysis (`naive`) that conservatively assumes all objects to thread-escape, is also compared with others. Wong [32] compared `connect3` with traditional well-known escape analyses (Ruf [25] and Bogda et al. [7]) by applying them to the two problems (fence insertion for sequential consistency and unnecessary synchronization removal), and showed that `connect3` is the fastest, on average, while providing better precision. The analysis precision for both problems are correlated in his study, and thus improvements in sequential consistency performance in with

our two-phase analysis will be indicative of improvements in synchronization removal.

#### 3.2.1 Online Escape Analysis Cost

Like the original Jikes RVM configuration, programs are initially compiled using the baseline compiler, which performs no optimizations. We insert fences between every pair of memory operations to enforce sequential consistency, i.e. we make the worst case assumption for baseline compiled codes. As the program executes, hot methods are detected by the adaptive system, our online analysis is performed, and optimized fence insertion based on the results of our escape analysis is applied to these hot methods.

Table 3 shows our online escape analysis overhead in seconds. The second column shows the overhead of class loading, which includes parsing the annotations and reconstructing the CG. The third, fourth and fifth columns indicate the IR independent interprocedural analysis time. The sixth column is the IR dependent intraprocedural analysis time, and the last column is the total analysis time. The last row shows most of the analysis time (93% of the total) is spent doing interprocedural analyses (i.e. call graph building and escape information propagation). The time to load annotations is small compared to the time to perform the other components of the analysis: approximately 100 ms in absolute time and 2.5% of the total analysis cost, on average.

	Annot. Load	Call Graph Const.	Thread Info. Analy.	Escape Info. Prop.	Escape Info. Conv.	Total Cost
mtrt	0.099	0.998	0.018	1.278	0.014	2.407
moldyn	0.034	0.7	0.018	0.587	0.03	1.369
monte.	0.043	0.818	0.03	1.019	0.048	1.958
raytr.	0.019	0.719	0.017	0.591	0.055	1.401
bound.	0.031	0.697	0.015	0.587	0.001	1.331
disk.	0.034	0.81	0.017	0.591	0.011	1.463
genet.	0.039	0.845	0.038	0.624	0.044	1.59
hash.	0.037	0.819	0.018	0.635	0.294	1.803
sieve	0.031	0.728	0.031	0.596	0.008	1.394
[%]	2.5	48.5	1.4	44.2	3.4	100

Table 3: Escape Analysis Cost in Seconds

#### 3.2.2 Evaluation and Comparison With Prior Work

In this section, we compare our two-phase escape analysis with that of the previous fastest escape analysis known to the authors [32], and with the field-sensitive type-based escape analysis of [29]. These two analyses do not run under the adaptive optimizing system, but under the Jikes RVM optimization system that optimizes all methods – not just hot ones. The numbers in this section have all analyses, including delay set analysis and thread structure analysis, enabled.

The analysis of [29] is field-sensitive for all objects, and to reduce overhead it merges the escape properties of all object fields of the same type. It iterates over SCCs in the call graph until the escape information converges. The analysis of [32] is field-insensitive for object types that do not implement `Runnable`, and field-sensitive for object types that do. It tracks objects reachable by multiple threads similar to [25]. The analysis is unification-based, and merges two alias sets in-place via a union-find data structure [4], allowing it to not iterate over SCCs. Both analyses identify exclusively accessed thread objects as described in Section 2.2.3.

Figure 4 compares the total dynamic compilation time

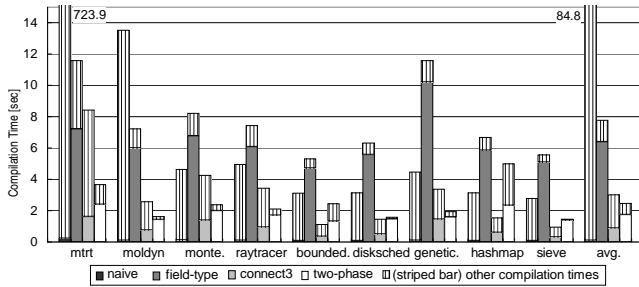


Figure 4: Comparison of Compilation Times in Seconds.

cost, and the escape analysis time cost, of the **naive**, **field-type**, **connect3**, and **two-phase** analyses. **naive** has a negligible escape analysis time, 0.1 second, on average since it simply assumes all objects and their accesses thread-escape. The total compilation time of the **naive** analysis, however, changes depending on the benchmark, since the total compilation time (i.e. escape analysis time plus the other compilation time) is affected by the escape analysis precision. As described in Section 1, “other compilation time” is the time to perform thread structure analysis and delay set analysis. The less precise escape analysis produces the more escaping references, and apparently imposes a greater burden on these other analyses. The average total compilation time is 84.8 second and its standard deviation is 239.7 second, which is mainly caused by the other compilation time. The escape analysis times for the **field-type**, **connect3**, and our **two-phase** analyses are 6.4, 0.9, and 1.7 seconds respectively. **connect3** has the fastest average analysis time because of its lack of iteration over SCCs and its mostly field-insensitive analysis. Total compilation times, however, are 7.7, 3.1, and 2.4 seconds respectively on average. Thus, our analysis has the lowest total compilation time. Moreover, if we compare the standard deviations of the total compilation time, they are 2.34, 2.33, and 1.16 seconds, respectively: our analysis has the smallest standard deviation of analysis cost, indicating its performance is more predictable.

Delay set analysis and thread structure analysis are performed to get delays (memory access orders) that must be enforced. This delay information is used to prevent compiler transformations under sequential consistency, and is used by the optimizing fence insertion algorithm of [13] to insert a fence along every delay to ensure that the hardware maintains the order of the memory accesses at each end of the delay. Figure 5 presents dynamic fence counts being executed, which directly affect the time needed for the application to run.

Figure 6 compares the application execution slowdown relative to the default Jikes RVM relaxed consistency memory model. Here, the execution time is the steady state execution time, i.e. it does not include the compilation or classloading time, and only the analysis precision affects this slowdown. **naive** has an average slowdown of 5.13. **field-type** has the least slowdown, 1.21 on average, and **two-phase** is comparable with an average slowdown of 1.27, but at a significantly lower analysis time cost. Among the three realistic escape analyses, **connect3** has the largest slowdown – 1.92 on average, but at a lower average analysis time cost. Therefore, the average application execution

time of **two-phase** is 1.5 (1.92/1.27) times faster than that of **connect3**, with only 80% (2.4/3.1) of the total compilation time of **connect3**.

As part of an on-going limit study of escape analysis, we have implemented an address trace-based perfect escape analysis (**perfect**) that determines which objects are *truly* accessed by more than one thread for a particular input set. In the subset of benchmarks we have analyzed, we obtained preliminary results showing the average slowdown of **two-phase** and **perfect** are 1.29 and 1.19 respectively. Our **two-phase** escape analysis is already precise enough that performance is only degraded by 8.4% (0.1/1.19) compared to the **perfect** case, which forms the upper limit of escape analysis precision.

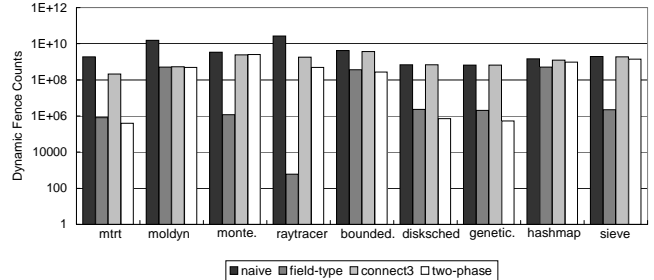


Figure 5: Comparison of Dynamic Fence Counts

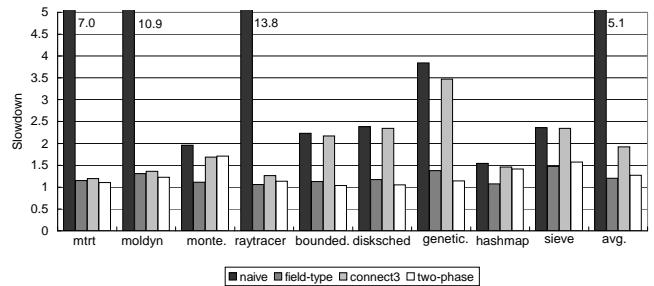


Figure 6: Comparison of Slowdown

## 4. RELATED WORK

Many static whole program escape analyses for Java have been developed [8, 7, 25, 31, 30]. Most of them target synchronization removal and stack allocation. Bogda et al. [7] and Rub [25] apply their analyses to synchronization elimination. Because they are unification-based analyses, they are simple and relatively fast (compared to most points-to graph-based analyses) while providing sufficient precision for synchronization removal. Points-to and Connection graph-based analyses [8, 31, 30] are generally both more precise and more expensive than the unification-based analyses. Choir et al. [8] present a graph-based compositional data flow analysis that we use as the basis of our offline analysis. It targets synchronization elimination and stack allocation of non-escaping objects. It uses a simpler lattice, and is less precise than ours in its ability to detect *thread* escaping objects. Whaley et al. [31] present another points-to algorithm which achieves higher precision than the previous mentioned algorithm, and like that algorithm also targets synchronization removal and stack allocation. Vivien et al. [30] presents an

incremental flow sensitive analysis that allows it to analyze arbitrary regions of complete or incomplete programs. However, due to the high-order complexity of [8, 31, 30], they are slow enough to raise concerns about their use in a dynamic compilation environment.

Wong [32] implements a fast, runtime escape analysis. By supplying a partially field sensitive analysis that exploits exclusive thread object accesses, it effectively augments its less precise field-insensitive analysis. Like ours, it primarily targets fence insertion to provide sequential consistency for Java programs. It also differs from ours in that it is implemented in a non-adaptive compilation system and has a higher overhead during program startup. Kotzmann et al. [18] present a runtime escape analysis with dynamic compilation and deoptimization framework. Like traditional offline escape analyses, it is used for scalar replacement, synchronization removal, and stack allocation. Their approach to identify thread-escaping information for objects is too conservative for our purpose.

Pominville et al. [23] and Qian et al. [24] apply runtime optimizations (e.g. array bounds check, null check) by conveying the results of offline analysis via user-attributes in the class file. Le et al. [19] use offline interprocedural side-effect analysis results to enable efficient runtime optimization in dynamic compilation. Their usage of the annotated results is fairly straightforward, whereas in our two-phase analysis, we extensively use the results of the offline analysis to improve analysis precision during dynamic compilation.

Sreedhar et al. [28] propose extant analysis, which identifies the program parts where static analysis and optimization can safely be applied, and runtime safety test for correct execution behavior even with additional classes being loaded. Pechtchanski et al. [21] describe a general approach for interprocedural whole-program analysis and optimistic optimization in the context of dynamic classloading and compilation with an example using online interprocedural type analysis. Hirzel et al. [17] describe a more complicated pointer analysis in the presence of dynamic classloading and evaluates its correctness. We differ in that we use offline analysis to create data structures that can be exploited at runtime, both to enable a fast interprocedural propagation, and to remove the need to analyze method bodies during the local propagation part of the interprocedural phase.

Our compilation model is closely related to staged compilation, e.g. DyC [15] in that we break our compilation into a static and dynamic phase. Our primary difference from this work is that we are not targeting optimizations based on partial evaluation, we do not focus on dynamic regions but consider all of the program that is available, and our decision of when to perform a particular analysis is guided purely by reducing runtime analysis overheads, not by when information is available to the analysis. We also differ in our target language, i.e. Java instead of C, and in our targeting an interprocedural analysis rather than intraprocedural analyses and transformations. Similarly, we analyze all of the program that is available, rather than focusing on dynamic regions as in DyC, which is motivated by a desire in DyC and other staged compilation systems to exploit partial evaluation. Given separately analyzed portions of the program, we perform an iterative dataflow analysis over the summaries for the methods in the different portions. DyC, in contrast, focuses on intraprocedural optimizations in dynamic regions where important variables have constant

values. Other staged compilation systems, e.g. Tempo [10] focus on partial evaluation for non-Java languages, or are motivated by a desire to enable users to write their own optimizations [22].

## 5. CONCLUSIONS

We have presented a two phase offline/online escape analysis for detecting thread-escaping objects that provides the precision of much slower analyses, and speed competitive with, and precision superior to, the fastest known escape analysis. Our analysis is suitable for use in a dynamic compilation framework for longer running programs. Because it uses an IR independent representation, and can therefore utilize the adaptive framework, the overall compilation time of our analysis is only 80% of that of the previous fastest escape analysis while providing better precision, which is reflected in an application execution time that is 1.5 times faster than programs compiled with the previous fastest known analysis. Moreover, it has a much lower standard deviation than the algorithms it was compared against, making its overhead more predictable, and increasing its usability in a production environment. It accomplishes this by doing much of the expensive computation offline, and exploiting a fast run-time level summary propagation algorithm while improving precision with thread information analysis. Our technique shows the feasibility of using two-phase analyses to safely enable the use of otherwise too expensive interprocedural analyses in a dynamic compilation environment.

## 6. REFERENCES

- [1] *The Java Grande Forum Benchmark Suite*. URL: <http://www.epcc.ed.ac.uk/javagrande/javag.html>.
- [2] *Soot - a Java Optimization Framework*. URL: <http://www.sable.mcgill.ca/soot/>.
- [3] *SPEC JVM Client98 Suite*. URL: <http://www.specbench.org/jvm98/jvm98>.
- [4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM System Journal*, 39(1), February 2000.
- [6] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Proceedings of the ACM SIGPLAN 2000 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2000.
- [7] J. Bogda and U. Holzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th ACM Conference on Object-Oriented Programming, Systems, and Applications*, pages 35–46, Denver, CO, USA, 1999.
- [8] 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 Transactions on Programming Language Systems*, 25(6):876–910, 2003.

- [9] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 258–269, Berlin, Germany, 2002.
- [10] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noye. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys*, 30(3es):19, 1998.
- [11] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference for Object-Oriented Programming*, pages 77–101, Aug 1995.
- [12] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *Proceedings of the 2003 ACM International Symposium on Memory Management*.
- [13] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 285–294, San Francisco, CA, USA, June 2003.
- [14] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization*, pages 241–252, 2003.
- [15] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. The benefits and costs of DyC’s run-time optimizations. *ACM Transactions on Programming Languages and Systems*, 22(5):932–972, 2000.
- [16] S. Hartley. *Concurrent Programming: the Java Programming Language*. Oxford University Press, 1998.
- [17] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *Proceedings of the 18th European Conference for Object-Oriented Programming*, pages 96–122, 2004.
- [18] T. Kotzmann and H. Mossenbock. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 111–120, 2005.
- [19] A. Le, O. Lhotak, and L. Hendren. Using inter-procedural side-effect information in JIT optimizations. In *Proceedings of 14th International Conference on Compiler Construction*, pages 287–304, 2005.
- [20] D. Lea. *Concurrent Programming in Java*. Addison Wesley, 1999. URL: <http://gee.cs.oswego.edu/dl/cpj/>.
- [21] I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 195–210, Tampa Bay, FL, USA, Nov. 2001.
- [22] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, 1999.
- [23] P. Pominville, F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proceedings of the 10th International Conference on Compiler Construction*, pages 334–354, London, UK, 2001.
- [24] F. Qian, L. J. Hendren, and C. Verbrugge. Comprehensive approach to array bounds check elimination for Java. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 325–342, London, UK, June 2002.
- [25] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, pages 208–218, British Columbia, Canada, 2000.
- [26] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 12–23, Snowbird, Utah, 2001.
- [27] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, 1988.
- [28] V. C. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 196–207, 2000.
- [29] 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 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 2–13, Chicago, IL, USA, 2005.
- [30] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 35–46, 2001.
- [31] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on object-oriented Programming, Systems, Languages, and Applications*, pages 187–206, Denver, CO, USA, 1999.
- [32] C.-L. Wong. *Thread Escape Analysis for a Memory Consistency Model Aware Compiler*. PhD thesis, University of Illinois at Urbana-Champaign, 2004.