ICS

TECHNICAL REPORT

Efficiently Verifiable Escape Analysis

Matthew Q. Beers
mbeers@uci.edu

Christian H. Stork
cstork@ics.uci.edu

Michael Franz
franz@uci.edu

Technical Report 03-29
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

December 2003

Information and Computer Science
University of California, Irvine

Abstract

Escape analysis is highly beneficial for optimizing object-oriented programming languages such as Java, significantly reducing memory management and synchronization overhead. However, existing escape analysis algorithms are generally too heavyweight to be applicable in just-in-time compilation contexts.

We present an alternative analysis that is less precise than traditional escape analysis, but that can be verified extremely efficiently. Hence, it becomes feasible to perform the analysis ahead of time and ship its result as an annotation with the bytecode, without sacrificing safety at the code consumer. In contrast, existing escape-analysis annotation approaches are unsafe. Moreover, unlike any other escape analysis that we know of, our method optionally provides for dynamic class loading, which is necessary for full Java compatibility.

Benchmarks indicate that our verifiable analysis can pinpoint on average 66% of all non-escaping allocation sites (56% when dynamic loading is supported), with negligible space overhead for the transport of annotations and negligible time overhead for the verification.

Contents

1 Introduction 2

2 Escape Analysis 3

3 Minimal Annotations and Speedy Verification 6

4 Evaluation 6

5 Related Work 10

6 Future Work 11

7 Summary of Contributions and Conclusion 11

1 Introduction

Just-in-time compilation systems for mobile code don't always use the best available optimization algorithms. Many of the analyses and optimizations that are commonplace in off-line compilers are simply too time-consuming to perform while an interactive user is waiting for program execution to commence. As a result, most just-in-time compilers are skewed towards compilation speed, rather than code quality.

Annotation-guided optimization systems [KC01, ANH99, JK00, PQVR+00, GMP+00, Rei01] try to bridge this conflict between compilation speed and code quality. In these systems, analyses are performed off-line and appended to the mobile code as program annotations. This reduces the just-in-time compilation overhead at the code consumer and enables optimizations that would otherwise be too time consuming to perform online.

An example of such a complex program analysis is *escape analysis* [WR99, CGS+99, BH99], a technique that identifies objects that can be allocated on the stack as opposed to on the heap. Escape analysis can also reveal when objects are accessible only to a single thread. This information can then be used to eliminate unnecessary synchronization overhead.

Escape analysis is not just time consuming, but also requires lots of memory for the internal graph representations of each method in a program [WR99, CGS+99, BH99]. On the other hand, benchmarks indicate that its use can result in substantial performance gains, even in case of more simplified linear-time analyses [GS00]. Ideally, we would wish to annotate programs with escape analysis information that can then be transported with the program and exploited by an annotation-aware just-in-time compilation system at the target site.

However, there are two primary drawbacks to the use of such annotations for escape analysis: first, they introduce transfer overhead (for the extra annotation information) and second and more seriously, their use is *unsafe*. That is, if someone accidentally or maliciously changed the escape-analysis result recorded for an allocation site from "heap allocated" to "stack allocated", then the memory safety of the whole target system would be in jeopardy.

Hence, one would need to *verify* such annotations, similar to the way that Java bytecode itself is verified. Verification of traditional escape analysis annotations of allocation sites, however, would essentially be as complicated as performing the original analysis in the first place, negating the original objective of reducing the workload on the code consumer. We are not aware of any prior work on *safe* annotations of escape analysis that would be applicable to Java bytecode, i.e., annotations that could actually be verified at the target. All published annotation-based solutions in this domain [ANH99, JK00, PQVR+00] are unsafe.

In this paper, we evaluate a simple escape analysis annotation scheme

- that can be performed by the code producer in linear time.
- that has a very small space overhead for the annotations, and
- that can be verified with negligible time overhead by the code consumer.

In the following, we briefly describe escape analysis and then introduce our variable partitioning scheme and its impact on the precision of the analysis. We then present benchmarks to quantify this impact (Section 4). Following this are sections on related work (Section 5) and future work (Section 6). A concluding section summarizes our contributions.

2 Escape Analysis

Escape analysis identifies *captured objects*, i.e., objects with lifetimes that do not exceed that of the method in which they are created. Captured object identification enables several optimizations. Most importantly, captured objects can be allocated on the stack avoiding the overheads of memory allocation and garbage collection. Furthermore, all synchronization of captured objects can be eliminated since only a single thread can ever access a captured object. Both optimizations have been shown to significantly improve program performance [WR99, CGS+99, BH99]. Capturedness enables further minor optimizations, for example, dead store removal and object inlining, i.e., replacing objects by local variables representing their fields [GS00, HAvRF03, LH02].

Commonly, escape analysis is achieved by constructing a variant of a *points-to-graph* that models object lifetimes and object aliasing. Based on this model, the analysis indicates which objects are captured by the method in which they are created. Whaley and Rinard's escape analysis [WR99] follows this approach.

Instead of considering individual *objects*, our method instead looks at the (pointer) *variables* that objects are attached to. Intuitively, if an object during its lifetime is only ever pointed to by variables that don't escape, then the object won't escape. A variable is considered captured if it is never returned from its defining method, isn't passed as an escaping parameter to some other method, and is never assigned to another variable that isn't also guaranteed to be captured.

Generally speaking, our analysis traverses the code to be annotated and produces a list of constraints on captured variables. This constraint equation is then solved so that the number of captured variables is maximal.

For the purpose of presentation we will look at slightly transformed source code.[1] Table 1 illustrates the relevant conceptual transformations. First of all, we make the otherwise implicit declaration of the `this` parameter explicit in order to be able to uniformly treat all parameters including the `this` reference. See the transformations for instance method and constructor declarations with their invocations changed correspondingly. Constructors are dealt with in a special way by splitting the creation of new objects into two consecutive statements. The first of which returns a reference to the allocated and zeroed object. The second statement calls the corresponding initializer with the otherwise implicit `this` reference as argument for the first parameter. Formally, initializers are treated like regular method calls.

Our analysis produces a boolean predicate $esc(v)$ and a runtime type property $rtt(v)$ for every local variable v and parameter p. The value of $rtt(v)$ is either a class C_i, uninitialized ($\bot$), or unknown ($\top$). These elements form a flat lattice with partial

[1] Our implementation actually operates on a canonicalized abstract syntax tree.

	Before	After

Instance Method Declaration	`class` C {...
	  $C_{\texttt{ret}}$ m(C_{p_1} p_1,...){
	  ...}
	...}
		`class` C {...
		  $C_{\texttt{ret}}$ m(C `this`, C_{p_1} p_1,...){
		  ...}
		...}

Instance Method Call	v_0.m(v_1,...)	m(v_0,v_1,...)

Constructor Declaration[2]	`class` C {...
	  C(C_{p_1} p_1,...){
	  ...}
	...}
		`class` C {...
		  `void` $init_C$(C `this`, C_{p_1} p_1,...){
		  ...}
		...}

Constructor Call	v_0 = `new` C(v_1,...v_n)	v_0 = `new` C; $init_C$(v_0,v_1,...v_n)

Table 1: Conceptual source code transformations

order $\leq$ and $\top$ being the least upper bound of any two distinct elements. The meaning of $rtt(v) = \top$ is "v's runtime type is its declared type or any subtype thereof".

The runtime type property has to obey certain constraints that are generated for code fragments as shown in Table 2. Assigning a new object of class C to a variable v lifts $rtt(v)$ to at least C. Note that if our analysis encounters another assignment of a new instance of class D to v then $rtt(v)$ becomes $\top$ unless $D = C$. We conservatively assume the runtime type of static and non-static fields and method results to be unknown. The runtime type constraint corresponding to an assignment v_0=v_1 among variables enforces that "v_0 is at least initialized with v_1's runtime type".

Our analysis specifies the boolean predicate $esc(v)$ for local variables or parameters v. If $esc(v)$ is false we say that v is *captured*. Note the difference to most other escape analyses, which model capturedness of individual objects. The meaning of the $esc(v)$ predicate is roughly "an object referenced through v might escape". Therefore, if an object o is only referenced by captured variables, then o is captured. This does not mean that a captured variable always references a captured object! For example, it is perfectly in accordance with our analysis to assign an escaping variable to a captured variable.

The constraints for a series of representative source code statements are given in Table 2. The first set of $esc(v)$ constraints define our notion of "directly escaping": References escape if they are returned, thrown, or assigned to static variables, fields, or array elements. We only have two dependent $esc(v)$ constraints (listed under combined constraints). First, we exclude the assignment of a captured variable to an escaped one—i.e., such an assignment by definition might cause the variable to escape.

Second, we assume all method results to escape. We also ensure that passing a variable v_i as an argument to a method call m lets this variable escape if the corresponding parameter p_i could escape. For an object-oriented language like Java, the former necessitates knowledge of the type hierarchy in order to determine all method declarations m' that could be invoked when calling m. Again, we chose a very conservative approximation of what we mean by "invokable". If m is a static, private, or final

Escape Constraints	
`return` v	$esc(v)$
`throw` v	$esc(v)$
s `=` v	$esc(v)$
v_0`.`f `=` v_1	$esc(v_1)$
v_0`[...]` `=` v_1	$esc(v_1)$
Runtime Type Constraints	
v `=` `new` C	$C \leq rtt(v)$
v `=` s	$\top \leq rtt(v)$
v_0 `=` v_1`.`f	$\top \leq rtt(v_0)$
Combined Constraints	
v_0 `=` v_1	$rtt(v_1) \leq rtt(v_0) \quad \wedge \quad esc(v_0) \Rightarrow esc(v_1)$
v_0 `=` m`(`$v_1, v_2, \ldots v_n$`)`	$\top \leq rtt(v_0) \quad \wedge \quad esc(v_0) \quad \wedge$ $\forall$ parameters $p_i^{m'}$ of method declarations m' invokable as m: $esc(p_i^{m'}) \Rightarrow esc(v_i)$

Table 2: Representative statements and their corresponding constraints where v, v_i stand for local variables or formal parameters, s for static fields, f for instant field names, m for method names, and C for classes.

method then there is exactly one invokable implementation; otherwise the invokable methods depend on the runtime type property of the self reference. $rtt(v_1)$ is either a specific class C, in which case only C's implementation of m is invokable, or $rtt(v_1)$ is unknown, i.e., $\top$, in which case all implementations of m for v_1's declared type or any of v_1's subclasses are invokable.

Here is the only place where supporting dynamic loading produces different results. In a "closed world" without dynamic loading, a whole program analysis can inspect all subclasses of v_1's declared type. In an "open world" in which additional classes can be added dynamically at any time, this is not possible. We have to assume the worst case, so all arguments of m escape. Therefore the escape constraints produced by a method invocation under the open world assumption shrinks to $esc(p_i)$ for all parameters.

Given an allocation site

v `=` `new` C

the allocated *object* o is captured if the *variable* v is captured. This holds because by our definition "no object can escape through the v" and v is the only reference to o; therefore o cannot escape. Arrays are handled like objects and array elements are treated as object fields. Therefore a one-dimensional array can be captured but its elements escape and multi-dimensional arrays can only be captured in their first dimension due to the fact that they are modelled as nested one-dimensional arrays in Java.

Newly constructed arrays that are assigned to captured variables or passed as captured parameters are captured. As mentioned before, the array's components are not captured. Therefore multidimensional arrays are only "captured in their first dimension".

3 Minimal Annotations and Speedy Verification

In order to transport the analysis results we mark local and formal variables v at their declaration site with their $esc(v)$ predicate and their runtime type property $rtt(v)$. Actually, the property $rtt(v)$ can be turned into a boolean predicate assuming we allow for some simple code transformations. Note that $rtt(v) = \perp$ means that v is not the target of a non-`null` assignment in the program and all its occurrences in the program could be replaced by `null`. We can therefore ignore the case of $rtt(v) = \perp$. The boolean version of $rtt(v)$ would mean "v's runtime type is equal to its declared type D". This definition leaves only the cases uncovered where $rtt(v) = C \neq D$. But since C has to be a subclass of D and all assignments to v are—by definition of our constraints—of declared and runtime type C, we can change v's declared type to C.

The decision of whether to allocate an object on the stack or on the heap is made at each object (and array) allocation site as described in the previous section.

Verifying the annotations is as easy as verifying the constraints from Table 2 while traversing the code (either while loading the code or while compiling it). In particular, during this verification, coming across an assignment of a captured variable to an escaping variable implies that the program or the annotations have been tampered with after the analysis was performed.

Note that in both the open and the closed world scenarios we analyze the library methods that are called by the subject program. Even in the open world case, our analysis depends on these annotations to coincide between code producer and receiver. Therefore, this has to be checked at link-time.

Even though our implementation augments canonicalized abstract syntax trees, the annotations are easily adaptable to Java classfiles. This would provide a low-impact addition of escape analysis optimizations for existing JVMs.

4 Evaluation

To evaluate the efficacy of our escape analysis, we performed the analysis for a number of benchmarks. The benchmarks we selected are a subset of the applications provided by the JavaGrande Forum [Jav] and a subset of the SPECjvm98 [SPE] benchmarks. The benchmarks we selected are listed in Table 3.

Our method is by its definition less accurate at denoting the capturedness of an object allocation than the traditional escape analysis. This loss of precision is acceptable because by annotating declaration sites we gain verifiability. Likewise, by introducing an open world assumption, we lose accuracy but gain dynamic class loading with escape analysis.

JavaGrande Benchmarks	
euler	Computational fluid dynamics
moldyn	Molecular dynamics simulation
montecarlo	Monte Carlo simulation
raytracer	3-dimensional ray tracer
search	Alpha-beta pruned search
SPECjvm Benchmarks	
jess	Java Expert Shell System
raytrace	3-dimensional ray tracer
db	SPEC Database benchmark
javac	Java compiler
jack	Java parser generator

Table 3: Description of benchmarks

To quantify this loss of precision, we counted the number of static allocations assigned to variables annotated as captured. This method was chosen to highlight the coverage we have against the currently most thorough pointer analysis described by Whaley and Rinard. While counting, an allocation site was considered to escape if it was

- assigned to variable marked escaped [3]
- returned from the method
- the exception in a `throw` statement

Each benchmark was analyzed under both open and closed world assumptions, and always with runtime type checking. All escape annotations were made against the canonicalized abstract syntax tree described above. Our compiler parses Java source, and implements the escape analysis by performing multiple passes over the intermediate representation.

The results of the Whaley and Rinard analysis were obtained by inserting a custom counting pass into the PointerAnalysis package of the FLEX research compiler. The counting pass utilizes the points-to-graph to determine the escapedness for each analyzed allocation site. It is important to note that the FLEX compiler infrastructure begins with a Java classfile, and only analyzes methods that are reachable from a `main` method. Therefore, a small variation in the calculated total number of allocation sites exists between our analysis and their results.

Table 4 shows the number of static allocation sites that allocate captured objects (over the total number of allocation sites). Objects allocated at these sites may be stack-allocated, and synchronization of these objects can be removed. The type declarations of each stack-allocatable object as well as occurrences in method signatures of method

[3]This includes assignements to array elements or a fields

parameters that do not escape inside are annotated with captured. On average, Whaley and Rinard mark 36% of the static allocation sites are such sites.

	Whaley and Rinard		Our Analysis		
Benchmark	Sites	Closed World Captured	Sites	Closed World Captured	Open World Captured
euler	39	11 (28%)	39	9 (23%)	9 (23%)
moldyn	7	2 (29%)	7	1 (14%)	1 (14%)
montecarlo	41	22 (54%)	40	17 (43%)	12 (30%)
raytracer	46	9 (20%)	44	4 (9%)	3 (7%)
search	18	8 (44%)	19	3 (16%)	3 (16%)
raytrace	129	55 (43%)	128	19 (15%)	4 (3%)
jess	433	164 (38%)	424	148 (35%)	138 (33%)
db	41	30 (73%)	41	24 (59%)	21 (51%)
jack	209	123 (59%)	212	112 (53%)	103 (49%)
javac	760	200 (26%)	793	150 (19%)	125 (16%)
Total	1723	624 (36%)	1747	487 (28%)	419 (24%)

Table 4: Comparison of captured allocation sites.

As expected, their comprehensive analysis marks a higher percentage of static allocation sites captured. Our escape analysis fares quite well, however, marking on average 28% allocation sites captured. Perhaps more interesting is how well the algorithm performed in the open world case, covering an average of 24% of the static allocation sites.

The unexpected performance of the analysis under an open world assumption seems to suggest that significant numbers of variables reference only a single class throughout their lifetimes. This pattern represents an ideal case for the runtime type analysis to detect and take advantage of, allowing for a pruning of the invokable methods at a given invocation site.

An advantage of Whaley and Rinard's algorithm is the explicit tracking of objects through method calls, potentially through many method calls. The completeness of the algorithm is part of what contributes to its large overhead. However, under both the closed and open world of our analysis, this completeness does not seem to provide a large advantage.

Object allocations inside loops warrant extra attention since they are potentially executed many more times then allocations outside of loops. To better understand how our analysis performs with respect to loops, we compare the relative number of captured allocation sites within and outside of loops. (We considere only whether an allocation site is located inside a loop or not, i.e, we ignore the nesting level of loops.)

The results in Tables 5 and 6 show what percentage of allocation sites located in loops are marked captured as opposed to the percentage of allocation sites marked captured outside of a loop. We found that the percentage of allocation sites captured inside of loops is significantly higher than those captured outside of loops. It appears that objects are normally more short-lived within loops and that our analysis benefits from this.

Benchmark	Allocations in loop (AIL)	Alloc. outside loop (AOL)	Captured AIL	Captured AOL	CAIL/AIL	CAOL/AOL
euler	7 (18%)	32 (82%)	0	9	0%	28%
moldyn	2 (29%)	5 (71%)	0	1	0%	20%
montecarlo	6 (15%)	34 (85%)	5	12	83%	35%
raytracer	2 (5%)	42 (95%)	0	4	0%	10%
search	3 (16%)	16 (84%)	3	0	100%	0%
raytrace	14 (11%)	114 (89%)	1	18	7%	16%
jess	71 (17%)	353 (83%)	26	122	37%	35%
db	8 (20%)	33 (80%)	5	19	63%	58%
jack	56 (26%)	156 (74%)	44	68	79%	44%
javac	147 (19%)	646 (81%)	29	121	20%	19%
Total	316 (18%)	1431 (82%)	113	374	39%	26%

Table 5: Distribution of captured allocation sites within loops under closed world assumption

Benchmark	Allocations in loop (AIL)	Alloc. outside loop (AOL)	Captured AIL	Captured AOL	CAIL/AIL	CAOL/AOL
euler	7 (18%)	32 (82%)	0	9	0%	28%
moldyn	2 (29%)	5 (71%)	0	1	0%	20%
montecarlo	6 (15%)	34 (85%)	4	8	67%	24%
raytracer	2 (5%)	42 (95%)	0	3	0%	7%
search	3 (16%)	16 (84%)	3	0	100%	0%
raytrace	14 (11%)	114 (89%)	0	4	0%	4%
jess	71 (17%)	353 (83%)	23	115	32%	33%
db	8 (20%)	33 (80%)	5	16	63%	48%
jack	56 (26%)	156 (74%)	44	59	79%	38%
javac	147 (19%)	646 (81%)	17	108	12%	17%
Total	316 (18%)	1431 (82%)	96	323	30%	23%

Table 6: Distribution of captured allocation sites within loops under open world assumption

A large percentage of the static allocations, especially within the SPEC benchmarks, is actually String concatenations. The standard method to deal with this operation is to translate the concatenation into a series of append method calls of a `java.lang.StringBuffer` object. The created string buffer is alive only long enough to output the final concatenated string and in these series of operations, the string buffer does not escape. The large number occurences of this pattern contribute significantly to the effectiveness of the escape analysis.

The variance between total allocation sites found in Whaley and Rinard versus our algorithm is primarily due to the aforementioned method the FLEX infrastructure uses to determine methods to analyze. We have attempted to limit our analysis to only these methods to present a more accurate comparison. Other deviations can be attributed to semantic differences between source and bytecode.

Table 7 shows the coverage our algorithm achieves when compared with Whaley and Rinard. For each benchmark, the percentage is number of allocation site our analysis marks as captured against the number of sites their analysis marks. On average, we cover 66% of the static allocation sites in the closed world, and 56% in the open world. This means that even with dynamic class loading, we can still find and potentially optimize approximately half of the allocation sites that Whaley and Rinard's analysis finds—and transmit this information to a just-in-time compiler in a verifiable manner.

Benchmark	Closed World	Open World
euler	82%	82%
moldyn	50%	50%
montecarlo	77%	55%
raytracer	44%	33%
search	38%	38%
raytrace	35%	7%
jess	90%	84%
db	80%	70%
jack	91%	84%
javac	75%	63%
Total	66%	56%

Table 7: Measure of percentage of captured allocation sites pinpointed by our algorithm as compared to Whaley and Rinard

5 Related Work

Lifetime analysis as dealt with in this paper was first described by Ruggieri and Murtagh [RM88]. The term *escape analysis* was coined by Park and Goldberg [PG92] in the context of functional languages. Their work spawned work on algorithms, which represent the escaping objects by integers [Deu97, Bla03]. In contrast, the most precise es-

cape analyses for Java use augmented points-to-graphs to model a program's behavior [CGS+99, WR99].

With the exception of Gay and Steensgard's analysis [GS00] our (closed world) analysis is the only one of which we know that can be performed in time $O(N \cdot T)$ where N is program size and T is the size of the class hierarchy. Their approach is also similar to the phase 1 analysis in Bogda and Hölzle [BH99] with the major difference that Bogda and Hölzle deal with alias sets instead of variables. In the case of the open world assumption our runtime complexity is even $O(N)$. Over all, our analysis can be viewed as flow-insensitive variant of Gay and Steensgard's algorithm with the added benefit of trivial verifiability.

Hartmann et al. [HAvRF03] use a similar analysis to ours (one of the authors of their paper is also a co-author of this paper). They augment an SSA-based intermediate representation with type modifiers corresponding to "captured" vs. "may-escape". Their method is predicated on SSA and not directly applicable to Java bytecode. The method presented in this paper is much more lightweight and could be incorporated into existing Java virtual machines with very little overhead.

6 Future Work

In order to extend the reach of our escape analysis, we are exploring ways to integrate multidimensional arrays, instance fields, and array components into our notion of capturedness while at the same time maintaining easy verifiability.

Currently, if a variable is used in different rôles — once holding an escaping reference and, at a different location in the code, holding a captured reference — then we have to mark it as escaping. Maybe it warrants the effort to look for such cases and split the variable in two with two different annotations.

7 Summary of Contributions and Conclusion

In this paper, we evaluated a verifiable annotation of escape analysis information for the purpose of transferring the cost of analysis from code consumer to code producer.

Our method is the only such analysis that we know of that can support dynamic loading.

Our benchmarks evaluated the various trade-offs of escape analysis along several different axes: with respect to complexity, with respect to open vs. closed world (with or without dynamic loading of classes), and with respect to the payoff of the run-time type analysis.

Our method is able to provide a safe performance boost at an almost negligible overhead, targeting low- to medium-performance just-in-time compilers. At the same time, it is completely complementary to any consumer-side escape analysis that a high-end just-in-time compiler might still wish to perform.

Acknowledgements. We are thankful to Alexandru Sălcianu for his help with the FLEX compiler and to Peter Fröhlich for his comments on earlier versions of this

paper.

References

[ANH99] A. Azevedo, A. Nicolau, and J. Hummel. Java annotation-aware just-in-time compilation system. In *ACM Java Grande Conference*, pages 142–151, June 1999.

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1999.

[Bla03] Bruno Blanchet. Escape Analysis for Java(TM). Theory and Practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, November 2003.

[CGS+99] J. Choi, M. Gupta, M. Serrano, V. Shreedhar, and S. Midkiff. Escape analysis for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, November 1999.

[Deu97] Alain Deutsch. On the complexity of escape analysis. In *Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 358–371. ACM SIGACT and SIGPLAN, ACM Press, 1997.

[GMP+00] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. Dyc: An expressive annotation–directed dynamic compiler for c. Technical Report Tech Report UW-CSE-97-03-03, University of Washington, 2000.

[GS00] D. Gay and B. Steensgard. Fast escape analysis and stack allocation for object-based programs. In *Compiler Construction 2000*, Berlin, Germany, March 2000.

[HAvRF03] Andreas Hartmann, Wolfram Amme, Jeffrey von Ronne, and Michael Franz. Code annotation for safe and efficient dynamic object resolution. *Electronic Notes in Theoretical Computer Science*, 82(2), 2003.

[Jav] Java Grande Forum. The Java Grande Forum benchmark suite.

[JK00] Joel Jones and Samuel Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):389–406, May 2000.

[KC01] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 156–167, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices*, 36(5), May 2001.

[LH02] Ondrej Lhoták and Laurie Hendren. Run-time evaluation of opportunities for object inlining in java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande (JGI-02)*, pages 175–184, New York, November 3–5 2002. ACM Press.

[PG92] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–127, 1992.

[PQVR+00] P. Pominville, F. Qian, R. Vallee-Rai, L. Hendren, and C. Verbrugge. A Framework for Optimizing Java Using Attributes. In *Sable Technical Report No. 2000-2*, 2000.

[Rei01] Fermín Reig. Annotations for portable intermediate languages. In Nick Benton and Andrew Kennedy, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[RM88] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the Conference on Principles of Programming Languages*, pages 285–293. ACM SIGACT and SIGPLAN, ACM Press, 1988.

[SPE] SPEC JVM98 benchmarks. See online at http://www.spec.org/osg/jvm98 for more information.

[WR99] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, CO, November 1999.