Online Verification of Offline Escape Analysis

Michael Franz[+*] Vivek Haldar[+] Chandra Krintz[!] Christian Stork[+]

[+]Dept. of Information and Computer Science
Univ. of California, Irvine

[!]Dept. of Computer Science
Univ. of California, Santa Barbara

Abstract

Dynamic compilation often comes at the price of reduced code quality since there is not enough time available to perform expensive optimizations. One solution to this problem has been the addition of annotations by the code producer that enable an annotation-aware dynamic code generator on the code consumer's side to shortcut certain analysis and optimization steps. However, code annotation often creates a new problem in that most annotations are unsafe—if they become corrupted during transit, the safety of the target system is jeopardized.

In this paper, we consider annotations that transport escape-analysis information for Java programs. This information can be highly beneficial, not only for enabling stack allocation instead of costly heap allocation, but also for reducing synchronization overhead. The traditional solution of annotating allocation sites, however, is unsafe and cannot be verified without repeating the complete analysis.

Our solution consists of annotating variables rather than allocation sites, partitioning them into two classes. This annotation can be verified in linear time. It is a conservative annotation and less precise than annotating allocation sites, i.e., some

[*]Authors in alphabetical order.

optimization opportunities get lost in the process. However, our benchmarks suggest that the optimization potential that is actually lost in practice is close to zero.

An interesting consequence of our annotation method is that it can be integrated with an inherently safe program encoding that makes it impossible to represent illegal programs in the first place. Adding our safe annotations actually increases encoding density since it reduces the number of valid choices that need to be represented, so that the addition of the annotations comes at almost no space cost.

1 Introduction

Many common compilation techniques are time consuming, due to the complexity of program analysis and optimization algorithms. This becomes a problem when code is generated on-the-fly while a user is waiting for execution to commence. Because the available time is limited, just-in-time compilation systems often don't make use of the best possible optimization algorithms.

Annotation-guided optimization [19, 3, 18, 22, 14, 24] offers a solution to this problem: analysis is performed off-line and communicated to the compilation system as program annotations. Annotations reduce just-in-time compilation overhead and enable optimizations that are too time consuming to perform on-line. An example of such an analysis is *escape analysis* [27, 8, 4], 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 accessed by a single thread. This information can then be used to eliminate unnecessary synchronization overhead.

Escape analysis is both time- and space-consuming since it often requires interprocedural analysis, fixed-point convergence, and graph representations for each method in the program. However, existing escape analysis implementations indicate that its use offers substantial performance gains [27, 8, 4]. Ideally, we 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: they introduce transfer overhead (extra bytes now need to be transferred with the program) and their use is *unsafe* in the case of escape analysis. That is, if someone accidentally or maliciously changed the escape-analysis result that is 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 either needs to trust such annotations or needs to verify them, similar to the way that the Java bytecode itself is verified. Verification, however, potentially introduces an overhead that is as complex as performing the analysis itself. In the case of traditional escape analysis annotations of allocation sites, one would essentially have to repeat the complete analysis, negating the original objective of reducing the workload on the target computer.

In this paper, we describe a new approach to escape analysis annotation that makes it possible to verify the results in linear time and without the need to maintain any additional data structures beyond the annotations proper. Rather than annotating allocation sites, we partition the variables in the program into those that are guaranteed not to escape, and those that might escape. As a result of this change in focus, some optimization potential will be lost, but our benchmarks show that in practice this is negligible.

On the other hand, the benefits are considerable, because with the small loss in precision, we can now verify this information efficiently. This makes it possible to integrate escape-analysis information into a mobile-code format that can provably represent only valid programs and annotations and hence provide *safety by construction*. We know of no other technique for communicating escape-analysis results safely. All published annotation-based solutions [3, 18, 22] are unsafe, i.e., they are never verified at the target machine.

In the following, we briefly describe escape analysis and then introduce our variable partitioning scheme and its impact on the precision of the analysis (Section 2). The issue of precision is quantified in Section 3. The next two sections then describe the basic concepts behind grammar-based encoding (Section 4), and how this leads to safety by construction (Section 5). In Section 6, we articulate how we integrate our novel escape annotations with our encoding technique to enable a very dense encoding of both program code and annotations. We also explain how this encoding guarantees that the annotations are valid when decoded at the target site. Empirical results on this part of our work are presented in Section 7. Finally, we present related work (Section 8) and future work (Section 9) and then conclude the paper.

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 allocated. Captured object identification enables two optimizations: Firstly, captured objects can be allocated on the stack avoiding the overhead of garbage collection. Secondly, 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 [27, 8, 4].

Commonly, escape analysis is realized by constructing a graph (called *points-to graph*), which models object lifetimes and assignments at runtime. Based on this model, the analysis indicates which objects are captured by the method in which they are allocated. Whaley and Rinard's escape analysis described in [27], follows this approach. We initially implemented their algorithm for an investigation of safe encoding of potentially dangerous annotations. In doing so, we discovered that most captured objects can be identified by a much simpler approach.

Often it is unnecessary to model objects as separate entities. It suffices to consider the variable that the newly created object is assigned to. If this variable is not returned from the method nor passed into some other method (where it might escape) and it is not assigned to an escaping variable, then the object is captured. This simple intuition translates into the following invariants.

I0 Captured variables are not returned as results.

I1 Captured variables are only assigned to other captured variables.

I2 Captured variables are only passed into methods and constructors as arguments of captured parameters. When calling an instance method this applies also to the instance's *this* reference.

With these invariants in mind, we annotate variable declarations as either captured or escaping. We write this annotation as a type modifier, `captured`. Our notion of capturedness only refers to variable references and not to objects. A variable can be `captured` whereas the object it refers to escapes—the type modifier merely restricts the operations that are possible on the variable. Hence, a variable that is marked as `captured` may at times refer to objects that escape.

More specifically, we annotate the following language constructs as `captured` or escaping (the default).

Local Variables We annotate all locally declared variables of reference types, i.e. references to objects or arrays. For variables of primitive types (like *int* or *double*) this annotation doesn't make sense. For the purpose of our analysis we assume that primitive variables escape.

Method Parameters We annotate each declared parameter, including the self-reference *this* of instance methods. Note that we don't annotate the method's return parameter.

Constructor Parameters We annotate each declared parameter, including the implicit return parameter *this*.

Note that we don't annotate class or instance variables, array components, or exception-handler parameters. We assume that they escape.[1]

Our invariants **I0** through **I2** guarantee that no object escapes via a captured variable because captured variables can only be assigned to other captured variables and none of the permitted actions let the referenced object of a captured variable escape directly.

In order to guarantee **I2**, we need to show that, for each method call site where captured variables are passed, the corresponding formal parameters of all method implementations that are accessible due to dynamic method lookup are captured. This is trivial in case of static or final methods or if the creation of the dispatching object is within the scope of our analysis so that we know its exact runtime type. In general, this requires a global type analysis, which considers all the relevant *implements* and *extends* relationships. Intuitively, each

[1] This is a simple and crude approximation of a real escape analysis.

overriding method needs to have the same or more captured parameters as the overridden method.

Note that none of the rules rely on the fact that captured variables actually refer to captured objects in order to ensure the capturedness of other variables. It is perfectly legal to pass escaped references into captured parameters or to assign an escaping variable to a captured one. In short:

```
cap = esc;  /* allowed */
esc = cap;  /* not allowed */
```

Capturedness of actual *objects* is identified at allocation sites. If the constructor's self-reference is captured and the newly created object reference is assigned to a captured variable or passed as a captured parameter then this object is captured.[2]

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".

Online verification of our annotations proceeds as follows. Firstly, the code consumer receives the type hierarchy including method and constructor signatures. It verifies that overriding methods capture at least the same parameters as the methods they override.[3] Secondly, the receiver traverses the classes verifying the invariants **I0**, **I1**, and **I2**. Any assumptions about annotations of classes outside the set of transmitted classes is matched against the installed libraries.

[2]For the purpose of this analysis, if a constructor body does not begin with an explicit constructor invocation we consider the implicit constructor call as part of the constructor body. This ensures that no escaping *this* goes unnoticed in one of the implicitly invoked constructors.

[3]This step is actually overly conservative since we might be able to show that the relevant method calls cannot be overridden and therefore overriding methods do not matter.

Annotations for a set of classes can be constructed as follows: Mark every target variable/parameter of a constructor assignment as captured. This is the minimal set of profitable annotations. Minimal in the sense that no additional annotation would allow to find more captured objects. From now on the analysis only undoes captured annotations. First mark every returned variable as escaping. Next perform the validation of the classes until it either finishes successfully or until it breaks either invariant **I1** or **I2**. In the latter case, if **I1** was broken, mark the offending variable as escaping and redo the last step; if **I2** was broken, undo the parameter annotation and ensure that the overriding methods are still compatible, then redo the validation step.

3 Quantifying Annotation Precision

To evaluate the efficacy of our encoding of escape annotations, we performed the analysis for a number of benchmarks. The benchmarks we selected are a subset of the applications developed by the JavaGrande Forum [11]. The following is a list of these benchmarks and a brief description of their fuctionality:

- Euler: Computational fluid dynamics
- MolDyn: Molecular dynamics simulation
- Montecarlo: Monte Carlo simulation
- Raytracer: 3-dimensional ray tracer
- Search: Alpha-beta pruned search

Table 1 shows the number of static allocation sites that allocate captured objects (over the total number of allocation sites). Objects allocated at these sites can be stack-allocated, and synchronization performed on these objects can be removed. The type declarations of each stack-allocatable object as well as occurrences in method signatures of method parameters that do not escape inside are annotated with `captured`. On average, 34% of the static allocation sites are such sites. These results are in line with those presented in prior work [27].

Surprisingly, our variable-partitioning based escape analysis was able to cover all captured allocation sites identified by the much more complex analysis.

Program	Captured / Total	Percent Captured
Euler	16 / 48	33%
MolDyn	5 / 9	55%
Montecarlo	34 / 105	32%
Raytracer	17 / 57	29%
Search	13 / 29	44%
Avg	17 / 50	34%

Table 1: Escape Analysis Results for Whaley and Rinard's escape analysis; our method can cover every single allocation site.

4 Grammar-Based Compression

Grammar-based compressors encode sentences that are known to fully conform to the given grammar (i.e., there are no syntax errors). It is fairly easy to construct such an encoder simply by numbering the available choices whenever there is an alternative in a production of the grammar, and transmitting the language sentence as a sequence of choice designators. Since the decoder has the same grammar specification available to it, it is able to reconstruct the sentence based only on these choices; in particular, sequences with no choices require no extra communication at all.

For example, consider the following excerpt from a very simple grammar given in extended Backus-Naur form:

```
statement ::=
  | ``LET'' identifier ``:='' expression ``;''
  | ``WHILE'' expression ``DO'' { statement } ``END''
  | ``IF'' expression ``THEN'' { statement } ``END''
  | ``REPEAT'' { statement } ``UNTIL'' expression ``;'' .
```

To encode a *statement* in this grammar, we need to communicate which of the four choices (assignment, while-statement, if-statement, repeat-statement) we are dealing with. However, once we have selected a choice, we do not need to send additional information until we arrive at another choice. Take the program fragment

```
IF ex1 THEN
   REPEAT S1 UNTIL ex2;
   LET i := j;
 END
```

Assume that the encoding of *ex1* yields the choice sequence *choices-ex1*, that the encoding of *S1* yields the choice sequence *choices-S1*, etc., the choices in *statement* are numbered (1, 2, 3, 4), and that the *END* and *UNTIL* choices are numbered zero, then the above sentence can be encoded as:

```
3  choices-ex1
  4  choices-S1  0  choices-ex2
  1  choices-i  choices-j
  0
```

Hence, in particular, we do not need to explicitly encode the facts that there is a *THEN* in the if-statement, nor that there is an ":=" in the assignment. Note how the above encoding corresponds to the depth-first traversal of the program's abstract syntaxt tree.

5 Safety By Construction

Intuitively, one senses that compression and safety must be complementary to each other—if we design an encoding that can represent only a subset of all possible programs (the "legal" ones, according to some statically decidable analysis) then there are fewer alternatives to encode and hence the encoding should be denser. This merely takes the idea of grammar-based encoding one step further: A grammar-based compressor encodes only those character sequences that are valid sentences of its grammar; now we are further limiting ourselves to those sentences of the grammar that conform to some additional static semantic constraints.

As an example of such semantics, we could design an encoding that implicitly enforces some of the typing rules of a programming language. Take the following Java program fragment:

```
class Basic {...};
class Extended extends Basic {...};
...
    static void sample() {
        Basic b1, b2; Extended x1 ,x2;
        ...
    }
...
```

Now consider which assignments can be written down inside method *sample*: some of these assignments are illegal under Java's type system, while certain others, although legal, are pointless because they assign a variable to itself. In this particular example, only half of all possible assignments are actually simultaneously legal and useful.

In a type-unaware encoding, each assignment between two variables represents one choice out of 16 (four possible left sides and four possible right sides), which might be encoded by using two bits each for each variable, for a total of 4 bits. Conversely, an encoding that

Assignments		
useful	illegal	pointless
b1 := b2	x1 := b1	b1 := b1
b1 := x1	x1 := b2	b2 := b2
b1 := x2	x2 := b1	x1 := x1
b2 := b1	x2 := b2	x2 := x2
b2 := x1		
b2 := x2		
x1 := x2		
x2 := x1		

incorporates static semantics might enumerate all eight useful assignments and simply use the index of the appropriate assignment in this enumeration to communicate the choice—this would require only 3 bits. Hence, incorporating the type semantics into the encoding results in a greater encoding density since there are only half as many assignments to choose from[4].

More importantly, the type-aware encoding is *inherently immune to malicious modifications that would undermine type safety*, since programs that violate the assignment compatibility rule cannot be represented in the first place. Hence, unlike programs expressed in bytecode, which need to be checked in a separate verification pass upon arrival, a type-aware decoder performs this verification implicitly by maintaining the assignment compatibility via the construction of alternatives to choose from.

In practice, of course, the set of valid choices in an assignment might be infinite, since it may contain expressions other than simple variables. For example, when working with dynamically linked data structures we might need to encode the statement

```
l.next = l.next.next
```

[4]Effectively, adding the type rules "squeezes some entropy out of the encoding domain". Hence, the resulting coding density should always be greater or equal, even if one uses a more intelligent encoding for communicating the actual choice than simply using *log(#choices)* bits.

However, as we explained in Section 4 above, our type-aware encoding works in conjunction with a grammar-based encoding that already possesses the capability of encoding these choices.

A further requirement is that the decompressor needs to be able to reconstruct the valid choices using the information available to it; i.e., based only on some static rules and the information already transmitted. In the example above, this means that the variables used in the program, as well as their types, need to be transmitted before the actual statements.

6 Encoding Escape Analysis Safely—By Construction

In the previous section, we explained how one can design an encoding that provides *safety by construction*, by restricting the domain of "what can be encoded" to apply only to "legal" programs in the first place. The interesting point is that the same idea can extended to the specific variant of escape analysis annotations that we developed in Section 2.

In particular, we have designed an inherently safe encoding that integrates the transport of the results of escape-analysis. By this we mean that if a program can be encoded in our format at all, then its escape-analysis annotations are guaranteed to be correct. Stated the other way around, it is impossible to even hand-craft a program in our representation that contains references marked as "non-escaping" that do escape.

The main idea is surprisingly similar to the encoding described in Section 5: we extend the underlying type system by the additional dimension representing "capturedness". The task of the encoding then becomes to disallow all assignments between variables that could possibly allow a captured reference to escape. For example, consider the following Java program fragment:

```
static void sample2 () {
   captured Object cap1, cap2; Object o1, o2;
   ...
}
```

The annotation “captured” in the declaration of variables *cap1* and *cap2* indicates that an analysis in the compiler front-end has determined that these variables will never be involved in an assignment that would let the referenced objects escape. As a consequence, assignments from captured to other variables must not be representable in the encoding. The following table summarizes the assignments that would be allowed or prohibited in method *sample2* under the capturedness type rules:

Assignments		
legal	illegal	pointless
cap1 := cap2	o1 := cap1	cap1 := cap1
cap1 := o1	o1 := cap2	cap2 := cap2
cap1 := o2	o2 := cap1	o1 := o1
cap2 := cap1	o2 := cap2	o2 := o2
cap2 := o1		
cap2 := o2		
o1:= o2		
o2 := o1		

Using the method described in Section 5, the capturedness property can therefore be transported in a fully safe manner. If an adversary were to change the annotation of an escaping variable to erroneously claim that it was captured, then our encoding would not be able to encode any assignment that would let the variable escape. Conversely, if one were to change the annotation of a captured variable to escaping, then that would simply mean that a potential for optimization had been lost, without making the program any less safe[5].

[5]Note that the additional “capturedness” type dimension needs to be considered during linking. Hence,

Hence, our method overcomes a major drawback of existing approaches to using annotations with mobile code, namely that corrupted annotation information could undermine the safety of the system. In previous approaches [3, 18, 22], annotations were generally unsafe because there would have been no way of verifying their correctness at the code consumer's site other than by repeating the analysis they were targeting to avoid. Using our method, any object that at its creation time is marked "captured" is guaranteed to be stack-allocatable. No verification is required at the destination to ensure that the annotations we encode are safe to use.

7 Implementation and Measurements

Interestingly enough, because the escape-analysis annotations limit legal choices in the inherently-safe encoding itself, fewer bits are required to transport this information "on the inside" of the encoding in this manner than would any other method of transporting it "on the outside".

In the following, we present our encoding relative to four different compression techniques: Jar, Pack, Gzip, and Bzip. The Java archive (jar) format is the most common tool for collecting (archiving) and compressing Java application files [17]. The format is based on the standardized PKWare zip format [21] and enables archiving of various components of Java applications (class, image, and sound files).

Pack [23] is a jar file compression tool from the University of Maryland. This utility

if an adversary modifies a class in transit, changing the annotation of an *imported* reference in a method signature from captured to escaping, then that would be detected during link-time signature matching. In our current solution, if one changes the implementation of a library method in such a manner that it effects the capturedness of any parameter in its signature, then all clients of the library should be recompiled.

defines a compact representation of class file information and substantially reduces redundancy by exploiting the Java class file representation, and by sharing information between class files. The compression ratios achieved by this tool are far greater than any other compression utility for Java applications.

Gzip and bzip are both standard compression utilities, commonly used on UNIX operating system platforms. Gzip does not consider domain specific information and uses a simple, byte-oriented algorithm to compress files. As such, gzip has very fast decompression times but does not achieve the compression ratios of pack. Bzip is a freely available, high-quality data compression utility [6] that makes use of the Burrows-Wheeler method for compression.

Table 2 shows the size in bytes of bytecode programs encoded using jar, pack, gzip, and bzip compression (columns 2-5, respectively). The last two columns show the sizes (in bytes) of the programs when we use our encoding, which we call Compressed Abstract Syntax Trees (CAST), on the Java source files. The encoding results for our format are given for the cases where escape analysis annotations are included (ACAST) and omitted (CAST). No escape analysis annotations are included in the results presented for the non-CAST encodings.

Table 3 quantifies the amount of annotation information that is actually transported in our encoding. All numbers are given in bytes. The first column shows the size of the actual Java class file, the second shows the amount of annotation information one would need to add in order to transport the results of escape analysis using the annotation format of the annotation-aware Open Runtime Platform (ORP) [9] described in [19]. The third column gives the size of our CAST format *without* annotation information encoded, and the last column gives the delta between our ACAST format (including annotations) and CAST.

The size overhead for incorporating annotations into ASTs is smaller than that for doing

	Source			ByteCode				Compressed AST	
Program	Text	Gzip	Bzip2	Jar	Pack	Gzip	Bzip	ACAST	CAST
euler	31689	5251	5015	10250	4108	9978	10460	4171	4111
moldyn	10732	2920	2910	6533	2346	6305	6804	2290	2260
montecarlo	37210	6465	5992	20496	5340	19191	19718	5241	5065
raytracer	15690	4141	3922	12038	3079	11008	11493	2881	2773
search	11011	3247	3234	7146	2833	6797	7296	2754	2647
section3	216707	27786	23997	62243	15561	57436	58008	15944	15504

Table 2: CAST compared to compressed source and bytecode formats. For each benchmark, we show the size in bytes. The source columns show the sizes of the Java source files without compression, and when compressed with gzip or bzip2. The bytecode columns show the sizes when compressed using jar, pack, gzip, and bzip compression. The final two columns show the sizes of the programs when we use our encoding, both including annotations communicating the results of escape analysis (ACAST) as well as without (CAST). The section3 benchmark is the combination of all of the preceding benchmarks.

the same for Java class files in all but one case. Evidently, beyond the advantage of being safe, our method is also quite space effective. This is important: annotation transport size should be small so as not to negate the benefit in execution time with transfer delay—especially in a wireless networking environment. Past annotation frameworks have reported file size increases ranging from 7% to 97% [22, 18, 16].

8 Related Work

The initial research on syntax-directed compression was conducted in the 1980s primarily to reduce the storage requirements for source text files. Cameron [7] introduced a combination of arithmetic coding with an encoding scheme similar to ours. And more recently Tarhio [26] suggests the application of PPM variants to the compression of parse trees. For a more detailed bibliography on compressing ASTs see [25]. None of these techniques attempt to represent the program's semantic content in a way that is well-suited for further processing

	Java With Annotations		**Compressed AST**	
Program	bytecode	Annotation	ACAST	ACAST–CAST
euler	10250	85	4171	60
moldyn	6533	51	2290	30
montecarlo	20496	193	5241	176
raytracer	12038	125	2881	108
search	7146	71	2754	80
section3	62243	**525**	15944	**454**

Table 3: Amount of annotation information implicit in ACAST. The bytecode column shows the size of Java class file and the annotation column shows the size of additional annotation information had it been encoded as annotations in the Java classfile format. The third column shows the sizes of the programs when we use our encoding without annotation, and the final column the delta that is added (in bytes) when annotations are incorporated into our encoding.

such as dynamic code generation or interpretation. Franz [12, 13] was the first to use a tree encoding for transporting (executable) mobile code. He uses a dictionary-based encoding to compress the abstract syntax tree of Oberon programs.

Necula [20] uses a technique very similar to tree compression in order to compress PCC proofs. Rather than transmitting the entire proof, only those points in the proof are transmitted where a choice must be made among alternative paths.

Java, currently the most prominent mobile code platform, has attracted much attention with respect to compression. Horspool and Corless [15] compress Java class files to roughly 36% of their original size using a compression scheme specifically tailored towards Java class files. In a follow-up paper Bradley, Horspool, and Vitek [5] further improve the compression ratio of their scheme and extend its applicability to Java archives (*jar*-files). An even better compression scheme for *jar*-files was proposed by Pugh [23]. His format is typically 1/2 to 1/5 of the size of the corresponding compressed *jar*-file (1/4 to 1/10 the size of the original class files). Pugh offers his tool for free evaluation.

All of the above Java compression schemes start out with the bytecode of Java class files. Eck, Changsong, and Matzner [10] employ a compression scheme similar to Cameron's and apply it to Java source programs. They report compression down to around 15% of the original source file, although more detailed information is needed to assess their approach.

9 Future Work

We are currently working on encoding the full Java type semantics in using the technique described in Section 5, which would give us a safe mechanism for transporting Java programs that would reduce the need for separate *code verification* passes. (Of course, link-time matching of method signatures across class boundaries is still necessary.) Unfortunately, this turned out to be far trickier than we expected, and we weren't able to finish debugging by the paper submission deadline.

In order to extend the reach of our escape analysis, we are exploring ways to integrate mulidimensional 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.[6]

We are currently also investigating other annotation-based optimizations, including in-

[6]A more comprehensive approach would be to change the representation of Java programs to use single static assignment (SSA) form, i.e. to replace variables by values.

lining, optimization filtering [19], and register allocation. For the latter, complex algorithms are currently required to effectively allocate registers for Java programs. The use of such techniques in a dynamic compilation setting is infeasible due to the compilation delay imposed. However, register allocation, like escape analysis and other compilation techniques become viable if performed off-line and communicated via annotation. Since our encoding methodology eliminates major drawbacks previously associated with mobile code annotation (transfer size and safety) we hope to enable highly optimized as well as highly-compact encodings and safe execution of mobile programs.

10 Conclusion

Previously, escape analysis in mobile code contexts had to be performed at the code consumer side, or be transported as unsafe annotations. We have solved the problem of how such annotations can be transported in a safe manner. Our method is independent of any intermediate representation used, and could for example be used for annotating JVM bytecode.

However, in conjunction with grammar-based compression, our approach can be taken one step further, arriving at an intermediate representation in which all escape-analysis annotations are *inherently correct by construction.* Interestingly enough, because the annotations are used to limit choices in the encoding itself, adding the annotations "on the inside" in this manner requires fewer bits than adding them "on the outside".

References

[1] *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, British Columbia, 18–21 June 2000. *SIGPLAN Notices* 35(5), May 2000.

[2] *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices,* 36(5), May 2001.

[3] 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.

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

[5] Q. Bradley, R. N. Horspool, and J. Vitek. JAZZ: An efficient compressed format for Java archive files. In *Proceedings of CASCON'98*, pages 294–302, Toronto, Ontario, Nov. 1998.

[6] Bzip2 compression utility.

[7] R. D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, July 1988.

[8] 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)*, Nov. 1999.

[9] M. Cierniak, G.-Y. Lueh, and J. N. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI)* [1], pages 13–26. *SIGPLAN Notices* 35(5), May 2000.

[10] P. Eck, X. Changsong, and R. Matzner. A new compression scheme for syntactically structured messages (programs) and its applications to Java and the Internet. In *Data Compression Conference*, page 542, 1998.

[11] J. G. Forum. The Java Grande Forum benchmark suite.

[12] M. Franz. *Code-Generation On-the-Fly: A Key to Portable Software.* PhD thesis, ETH Zurich, Mar. 1994.

[13] M. Franz and T. Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[14] 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.

[15] R. N. Horspool and J. Corless. Tailored compression of Java class files. *Software–Practice and Experience*, 28(12):1253–1268, Oct. 1998.

[16] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the Java Bytecodes in Support of Optimization. In *Journal Concurrency:Practice and Experience, Vol. 9(11)*, Nov. 1997.

[17] S. M. Inc. The Java ARchive utility. http://java.sun.com/products/jdk/1.1/docs/tooldocs/solaris/jar.html.

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

[19] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation* [2], pages 156–167. *SIGPLAN Notices,* 36(5), May 2001.

[20] G. C. Necula. A scalable architecture for proof-carrying code. In *Fifth International Symposium on Functional and Logic Programming*, Waseda University, Tokyo, Japan, 7–9 Mar. 2001.

[21] PKWare Inc. `http://www.pkware.com/`. PKZip format discription: `ftp://ftp.pkware.com/appnote.zip`.

[22] 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.

[23] W. Pugh. Compressing Java class files. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)*, pages 247–258, Atlanta, Georgia, 1–4 May 1999. *SIGPLAN Notices* 34(5), May 1999.

[24] F. Reig. Annotations for portable intermediate languages. In N. Benton and A. Kennedy, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[25] C. H. Stork, V. Haldar, and M. Franz. Generic adaptive syntax-directed compression for mobile code. Technical Report 00-42, Department of Information and Computer Science, University of California, Irvine, Nov. 2000. revised April 2001.

[26] J. Tarhio. On compression of parse trees. In *Proc. of Eigth Symposium on String Processing and Information Retrieval (SPIRE 2001)*. IEEE, 2001.

[27] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Nov. 1999.