Compressed Abstract Syntax Trees for Mobile Code

Christian H. Stork
cstork@ics.uci.edu
and
Vivek Haldar
vhaldar@ics.uci.edu
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

ABSTRACT

Abstract syntax trees (ASTs) have numerous advantages as a mobile code format over the more commonly used bytecode-based formats. Not only are ASTs portable, inherently safer, and more suitable for optimization, but we show that they also compress more densely. We have developed a prototype framework for AST compression, which we have used to compress Java programs. Our generic implementation reduces the size of regular Java archives by a factor of 3 to 8 and improves on the best published Java-specific compression scheme by 5–50%.

Keywords — compression, abstract syntax tree, mobile code, Java, prediction by partial match (PPM)

Introduction

We envision a mobile code infrastructure where code producers distribute programs as compressed abstract syntax trees (ASTs) and code consumers deploy these programs after decompression by compiling them for their specific platform. Compressed ASTs provide the following prerequisites for mobile code: platform independence, compactness, well-formedness and suitability for target-specific optimization.

We compress the AST of a program using a novel statistical approach. Since the AST conforms to a given abstract grammar (AG), we are using domain knowledge about the underlying language to achieve a more compact encoding than a general-purpose compressor could achieve.

The code producer distributes software as compressed ASTs, which constitutes a platform-independent format at a high level of abstraction. Compression of ASTs is allowed to be computationally expensive because it is only a one-time effort performed by the code producer. Thus we can imagine augmenting the encoding with hard-to-compute but easy-to-verify annotations, e.g., alias information for further optimizations or proofs of safety properties [Nec97], thereby further shifting the computational load from consumer to producer.

On the code consumer side, the code format has to meet several requirements: (1) short start-up time, (2) potentially pliable to more advanced optimizations, and (3) safe to execute. We meet the first requirement by providing a very dense encoding, which can be compiled directly into machine code on arrival. As shown by Franz [Fra94], the time saved for transmission (or file access) can pay for the additional decompression and compilation effort.[1]

Since our compression format contains all the machine-readable information provided by the programmer at source language level, the runtime system at the code consumer site can readily use this information to provide optimizations and services based on source language guarantees. Kistler [Kis99] uses the availability of the AST to make dynamic re-compilation at runtime feasible.

Furthermore, distributing code in source language-equivalent form provides the runtime system with the choice of a platform-tailored intermediate representation. For example, it is possible to use an existing target-specific backend. As proof of concept, we implemented a GCC-based backend.

High-level encoding of programs protects the code consumer against attacks based on low-level instructions, which are hard to control and verify. Even if tampered with, a file in our format guarantees adherence to the AG (and certain semantic constraints) or is invalidated, providing some degree of safety by construction. As an example, the Java language enforces restricted control flow, whereas Java bytecode allows arbitrary gotos, which necessitates verification upon class loading.

Compactness is an issue when code is transferred over networks limited in bandwidth, particularly wireless ones. It is also becoming increasingly important with respect to storage requirements, especially when code needs to be stored on embedded devices. Processor performance has increased exponentially over storage access time in the last decade. It is therefore reasonable to investigate compression as a means of using additional processor cycles to decrease the demand on storage access [Fra94], leading to a net gain in performance.

Our prototype implementation compresses Java ASTs, which are then compiled to native code, thereby circumventing compilation into bytecode and execution on the JVM. We chose Java as the language to compress because there have already been considerable efforts [Pug99] [ECM98] to compress Java programs. This gives us a viable yardstick to gauge our results against.

Compressing Abstract Syntax Trees

An *abstract syntax tree* (AST) is a tree representing a source program abstracting away irrelevant concrete details, e.g., which symbols are used to open or close a block of statements. Note also that properties like operator precedence and different forms of nesting are already implicit in the AST's tree structure.

[1] By now the consensus seems to be that on-the-fly compilation is preferable over bytecode interpretation. For example, in Microsoft's .NET architecture, code in intermediate language format is never interpreted but always compiled.

Every AST conforms with an *abstract grammar* (AG) just as every source program conforms with a concrete grammar. AGs give a succinct description of syntactically correct programs by eliminating superfluous details of the source program.

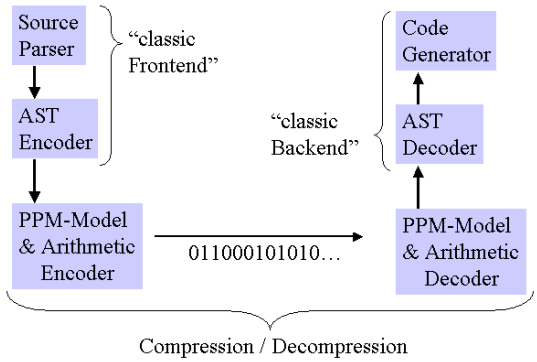
AGs consist of *rules* defining symbols much like concrete grammars define terminals and nonterminals. Whereas phrases of languages defined by concrete grammars are character strings, phrases of languages defined by AGs are ASTs. Each AST node corresponds to a rule, which we will refer to as the *kind* of node. It is sufficient to know about three types of rules in order to understand how our compression scheme works in principle: *aggregate rules* define AST nodes (*aggregate nodes*) with a fixed number of children; *choice rules* define AST nodes (*choice nodes*) with exactly one child, which has to be chosen from a fixed number of alternatives; and *string rules* define AST leaf nodes (*string nodes*), which contain an arbitrary string.

Overview

Our compression framework has the following conceptual stages:

- *Parsing*: produce an AST from the source code (i.e. source code's parse tree)
- *Serialization*: traverse the AST while generating a stream of symbols
- *Modeling*: use the AG to build a predictive statistical model
- *Arithmetic Coder*: compress the AST using the provided model

At the consumer's end the AST is incrementally rebuilt in the order it was traversed at the producer's end. The information being used by the modeler at the producer's end is limited to the symbols seen in the traversal upto the one currently being encoded. Thus the coder and modeler at the consumer's end operate conceptually in lockstep with those at the producer's end.



Encoding ASTs

In order to encode ASTs they need to be serialized. ASTs can be serialized by writing out well-defined traversals. We traverse ASTs depth-first, thereby generating a pre-order representation.

The actual tree representation can make effective use of the AG. Given the AG, much information in the pre-order encoding is redundant. In particular, the order and the kind of children of aggregate nodes are already known. Therefore the encoding boils down to noting the choices made at each choice node. Since the order of alternatives in choice nodes is fixed, it suffices to encode only the position of the chosen alternative. Of course, if only one alternative is given there is "no choice" and therefore nothing needs to be encoded.

Once the serialization of aggregate and choice rules is reduced to encoding the choices made at each choice node as an integer, an arithmetic coder [WNC87] is the preferred means to encode the choices among alternatives of varying probability. The probability distribution of the various alternatives is called the *model* for the arithmetic coder. When encoding, an arithmetic coder takes a sequence of choices along with their respective models as argument and outputs an (almost) optimal encoding of the sequence of choices as a sequence of bits.

Prediction by Partial Match

Prediction by Partial Match (PPM) [CW84] is a statistical, predictive text compression algorithm. PPM and its variations have consistently outperformed dictionary-based methods as well as other statistical methods for text compression.

PPM maintains a list of previously seen string prefixes, called *contexts*. For example, after processing the string *ababc*, the contexts are the empty context, *a, b, c, ab, ba, bc, aba, bab, abc, abab, babc,* and *ababc*. For each context PPM maintains a list of characters that appeared after the context. PPM also keeps track of how often the subsequent characters appeared. So in the given example the counts of subsequent characters for, say, *ab* are *a* and *c* both with a count of one. Normally, efficient implementations of PPM maintain contexts dynamically in a *context trie* [CT97]. A context trie is a tree with characters as nodes and where any path from the root to a node represents the context formed by concatenating the characters along this path. The root node does not contain any character and represents the empty context (i.e., no prefix). In a context trie, children of a node constitute all characters that have been seen after its context. Based on this information PPM can assign probabilities to subsequent characters in a predictive manner.

The length of a context is also called its *order*. Note that contexts of different order might yield different counts leading to varying predictions. Different strategies have been devised to blend the information given by contexts of different orders.

Our experience shows that PPM adapts so fast to each program's peculiarities that efforts to improve compression by using statistically predetermined fixed probabilities for the models did not yield any significant gains in compression.

Adapting PPM for ASTs

When applying PPM to trees the first problem to solve is the definition of contexts for ASTs. We chose a simple definition: The *context* of an AST node N is defined as the concatenation of nodes from the root to N, exclusively. This means our modified PPM algorithm treats AST nodes like the original PPM algorithm treats characters. Our alphabet corresponds therefore to the symbols/rules of the AG.[2] The PPM algorithm is applied to the nodes as they appear while traversing the AST in depth-first order.

However, above changes by itself are not sufficient to adapt PPM for ASTs. The maintenance procedure of the

[2]Note that if an aggregate node has several children of the same kind then their position is relevant for the context. Since this does not happen that often, we have not implemented this refinement yet.

context trie needs to be augmented, since the input seen by the modified PPM does not consist of contiguous characters anymore. No change is needed when the tree traversal descents to a child node. This corresponds to the familiar addition to the current context. But when the traversal proceeds from a leaf node up to an internal node (as in depth-first traversal), thereby invalidating the current contexts, the following modification is needed.

PPM maintains a set of nodes in the context trie called *active nodes*. Active nodes mark the positions, where the current contexts end. The root of the trie, representing the empty context, is always active. New nodes in the context trie are created as children of active nodes. However, in our adaptation of PPM, unlike regular PPM, whenever we reach a leaf node of the AST, we *pop* the context, i.e., all nodes marked as active (except the root) in the context trie are moved up one node to their parents. This ensures that all children of a node N in the AST appear as children of N in the context trie too. This works because we traverse the AST in depth-first order while building up contexts.

We adapted the unbounded variant of the PPM algorithm (PPM*) [CT97] for our implementation. Given our definition of context, the depth of our context trie is bounded by the AST's depth, thereby circumventing the unlimited growth of the context trie faced by PPM*.

In order to generate the model for the next encoding/decoding step, we look up the counts of symbols seen after the current context in the context trie. Since the active nodes, to which we have direct pointers, correspond to the last seen symbol, this is a fast lookup and does not involve traversing the trie. We have to decide how to weigh these predictions of various orders to construct a suitable model. There is a trade-off here: shorter contexts occur more often, but fail to capture the specificity and sureness of longer contexts (if the same symbol occurs many times after a very long context, then the chance of it occurring again after that same long context is very high), and longer contexts do not occur often enough for all symbols to give good predictions. Note that the characteristics of AST contexts differ from text contexts—AST contexts are bound by the depth of the AST and tend towards more repetitions since the prefixes of nodes for a given subtree are always the same

Among the various weighing strategies we tried were: weighing all contexts of all orders (including 0) equally, considering only the longest deterministic context (a deterministic context is one which gives only one prediction), weighing contexts proportionally to their lengths, and the fully blended PPMA, PPMB and PPMC weighing methods [BCW90]. Our experiments indicate that ignoring predictions made by order 0 contexts (which are simply relative frequencies of symbols, and form the first level of the context trie) and weighing all other predictions equally yields the best compression.

Compressing Constants

A sizable part of an average program consists of constants like integers, floating-point numbers, and, most of all, string constants and identifier names. In our simplified definition of AGs, we used the predefined STRING symbol to represent constants within ASTs. An efficient way of encoding constants is to specify an index into a table which contains all constants. The number of bits of this index can be further reduced by maintaining different tables for different kinds of constants, such as strings, type names, identifiers and so on. Our prototype stores constants as references into compressed tables of different kind of constants.

Implementation and Results

Our current implementation is a prototype written in Python. Our Java frontend is written in Java and uses the Barat framework [BS98] for parsing. After decompressing the compressed AST, the prototype interfaces directly to the backend of the GNU C compiler (GCC) for code-generation.

All information necessary to specify the AST's compression/decompression is condensed into one configuration file. Given the availability of our framework at the code producer and consumer sites, the only requirement for supporting the compression/decompression of an additional language is that identical copies of the configuration file are present at both sites.

The AG currently used in our framework fulfills most requirements for Java's binary compatibility. It is based on Barat's representation of Java programs, which among other advantages removes ambiguities like the ones caused by the type-import-on-demand declaration (e.g., `import some.package.*;`) by performing a static name analysis and always referring to fully-qualified type names.

In the following we compare the compression results of our prototype against other general-purpose and special-purpose compression algorithms. We split the comparison in two parts: First we measure compression of single classes and, second, we measure compression of collections of classes as they appear in Java archives. These collections of classes share the same pools (lists of strings, etc.) thereby reducing redundancy caused by entries, which appear in several classes.

The Java code chosen for compression is the Java compiler package for Linux (Linux Blackdown, version 1.1.2) and Jess, a rule engine and scripting environment (version 5.1). In case of Jess, we compressed all classes that are part of the distribution, i.e., including sub-packages and example classes. The class files were compiled under javac (Linux Blackdown Version 1.1.2) without debugging information. We compare only the compression of Java classes proper by eliminating all other resource files including the manifest.

We chose Pugh's compression scheme [Pug99] for comparison because, to our knowledge, it provides the best compression ratio for Java archives and class files. We use the evaluation version 0.8.0 of Pugh's Java Packing tool and feed it with `jar`-files generated with the `-M` option (no manifest).

We furthermore compare our results with two widely available general purpose compression algorithms, gzip and bzip2. Collections of classes have been `tar`'ed before applying `gzip` or `bzip2`.

The comparison of compressing Java classes is presented in Table I and the comparison for collections of classes is presented in Table II. Our choice of single classes tries to be representative of the sizes of classes in the SPEC JVM98 Benchmark suite. The resulting numbers show that our compression scheme is an improvement by 5–50% over Pugh's results, which translates to a compression of regular `jar`-files by a factor of 3 to 8, roughly. The results indicate that we compress very well for either very

Class Name	Size in Bytes					CAST/Pugh
	Class File	Gzip	Bzip2	Pugh	CAST	
`ErrorMessage`	305	256	270	209	105	50%
`CompilerMember`	1192	637	641	396	230	58%
`BatchParser`	4939	2037	2130	1226	1069	87%
`Main`	11363	5482	5607	3452	3295	95%
`SourceMember`	13809	5805	5705	3601	2988	83%
`SourceClass`	32157	13663	13157	8863	7849	89%

TABLE I

File size comparison of compressed AST files (CAST) with class files from `sun.tools.javac` compressed using alternative techniques.

Package Name	Size in Bytes					CAST/Pugh
	Jar	Gzip	Bzip2	Pugh	CAST	
`sun.tools.javac`	36787	32615	30403	18021	14070	78%
`jess`	232041	133146	97852	48331	31083	64%

TABLE II

File size comparison of compressed collections of classes from two Java packages.

small classes or larger collections of classes. Some more statistical investigation is needed to precisely analyze and, ultimately, enhance our compression results.

Related Work

The initial research relevant to AST compression was conducted in the 1980's and focused on the reduction of storage requirements for Pascal source files. Cameron [Cam88] was the first to combine tree encoding and arithmetic coding. He assigns fixed probabilities to alternatives appearing in the grammar. Tarhio [Tar95] uses PPM to drive the arithmetic coder in a fashion similar to ours and Cheney [Che00] suggests similar ideas for term compression. Franz [Fra94] was the first to use a tree encoding for mobile code.

In the realm of Java, Pugh [Pug99] achieves the best compression. His tool compresses Java archives by a factor of 2 to 5. In contrast to Pugh, who essentially compresses class files including their bytecode, Eck, Changsong, and Matzner [ECM98] employ a compression similar to ours. They report compression results similar to Pugh's, although more detailed information is needed to assess their approach.

Future Work

We are currently working on extending our work in several directions, mostly focusing on improving compression and extending our framework to include languages other than Java. To improve compression, we intend to specialize the compression of constants and to further explore the spectrum of models (different blending schemes, PPMA, PPMB with exclusion, etc.) to drive the arithmetic coder.

This work is part of our ongoing transPROse project [ADF+01] investigating novel mobile code representations.

Conclusion

We showed that ASTs are highly compressible using our generic framework. Our technique compresses 5–50% better than Pugh's Java-specific compression scheme and 3 to 8 times smaller than regular Java archives, without any substantial tuning of our framework. This supports our claim for compressed ASTs as a better alternative to bytecode-based mobile code formats.

Acknowledgments The authors would like to thank Peter Fröhlich, Peter Housel, Naomi Carpenter, Matthew Beers, and Michael Franz for their comments. This effort is partially supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536 and by the National Science Foundation, Program in Operating Systems and Compilers, under grant CCR-9901689.

References

[ADF+01] W. Amme, N. Dalton, P. Fröhlich, V. Haldar, P. S. Housel, J. v. Ronne, Ch. H. Stork, S. Zhenochin, , and M. Franz. Project transprose: Reconciling mobile-code security with execution efficiency. In *DARPA Information Survivability Conference and Exposition*, June 2001.

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, 1990.

[BS98] Boris Bokowski and André Spiegel. Barat – A front-end for Java. Technical Report B-98-09, Freie Universität Berlin, December 1998.

[Cam88] Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, July 1988.

[Che00] James Cheney. Statistical models for term compression. In *Data Compression Conference*, page 550, 2000.

[CT97] John G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *Computer Journal*, 40(2/3):67–75, 1997.

[CW84] John G. Cleary and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[ECM98] Peter Eck, Xie Changsong, and Rolf 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.

[Fra94] M. Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, ETH Zurich, March 1994.

[Kis99] Thomas Kistler. *Continuous Program Optimization*. PhD thesis, University of California, Irvine, November 1999.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[Pug99] William Pugh. Compressing java classfiles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 247–258, 1999.

[Tar95] Jorma Tarhio. Context coding of parse trees. In *Data Compression Conference*, page 442, 1995.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.