Improving the Java Virtual Machine Using Type-Separated Bytecode

Philipp Adler and Wolfram Amme

Friedrich-Schiller-Universität, Ernst-Abbe-Platz 1-4, 07743 Jena, Germany,
[phadler|amme]@informatik.uni-jena.de

Abstract. Java Bytecode is currently the most used mobile code representation, although it contains some well-known major flaws. In the paper we introduce the principle operation of type-separated bytecode. Type-separated bytecode is a new intermediate representation that compensates some of the drawbacks introduced by Java Bytecode. In particular the use of type-separated bytecode can considerably speed-up the verification process and supports optimizations performed on the producer side of a mobile code generation system. This is especially important when using bytecode on constrained devices like JavaCards or in embedded systems.

1 Introduction

No other mobile code format reached such a popularity like Java bytecode. It is used in many different domains although it has some serious flaws. The most two important drawbacks are the space and time consuming verification process and an intermediate representation which is difficult to optimize.

On modern computers the consumption of time and space during the verification process is mostly irrelevant. There is plenty of memory available and computing power increases more and more. Even though the possibility of exploiting the time behavior of the verification algorithm to mount an attack on virtual machines exists [6], this danger is mostly ignored. Also, the increasing capabilities of personal computers lead to many improvements to execute bytecode efficiently. Beside the formerly important interpretation there exist just-in-time (JIT) and adaptive compilers which produce native machine code on the fly. The generated code is considerably faster than interpreted execution, but these compilers do have an overhead.

On constrained devices the situation is not as simple as on modern computers. Their computing power and memory increased, too, but they have far less ressources than their larger counterparts. The verification algorithm must not excess available memory which is just a fraction of what can be used on desktop systems. Because of the low computing power the time spend during verification is also a multiple of the normal time used. These problems are not completely solved, leading frequently to compromises where one ressource is exchanged with the other. To lower the verification time more space is needed [19] and vice versa [4, 13]. Another difficult problem is the actual program execution. Because modern JIT and adaptive compilers can often not be employed on constrained devices classical interpretation is inevitable. Every single instruction takes up precious time, so it would be helpful if optimizations could eliminate unnecessary operations. This can be achieved through different frameworks, for example SOOT [21], but the more important optimizations like null- and bound-checks can not be applied on Java bytecode because of its granularity.

In this paper we introduce a new intermediate representation based on existing bytecodes. Using type separation it is possible to reduce the amount of verification work and memory space significantly. In addition, it is possible to optimize the mobile code and support an efficient (interpretable) execution using annotations. Because the annotations are embedded into the intermediate representation their correct use can be verified, thus they become secure. As a side effect the type separation can be used for simple program parallelization on processors with multiple instruction pipelines.

In the rest of the paper we try to sketch the main ideas of type-separated bytecode. In section 2 the related work regarding verification process, annotations and optimizations is presented. Section 3 shows the main concepts of type-separated bytecode and section 4 points to future work and research regarding our project. The final section 5 gives a short summary.

2 Related Work

Since Java bytecode has some serious flaws new intermediate representations were developed. Beside the stackbased techniques used in bytecodes there are mainly two other representations: syntaxoriented (like Architecture Neutral Distribution Format (ANDF) [17], Slim-Binaries [11], SafeTSA [1]) and proof carrying (for example Proof-Carrying-Code (PCC) [16], typed assembler language (TAL) [15]). Although these intermediate representations do have advantages their use is not widespread, at least not commercial.

Java bytecode is still widely-used and thus a lot of research went into improving this intermediate representation. This section summarizes the two main problems with bytecode and pictures some research activities.

2.1 Verification Process

The verification process for Java bytecode occurs directly before program execution and is used to guarantee security aspects of the mobile code [14]. Besides structural inspections every instruction is checked for receiving the right type and number of operands and that the stacksize is between zero and its given maximum size. These examinations take time and consume a lot of space.

Reducing the time factor of the verification process is an important task. The complexity class is $O(n^2)$ with n being the number of code instructions. It is possible to construct programs which do have exactly this worst-case time behavior and which can be used to mount denial of service (DoS) attacks [6].

To reduce the time factor there are mainly two approaches. The first one deals with simplifying the bytecode, for example on the problem of subroutines [8, 20], while the second one tries to speed up the verification process, e.g. with modified algorithms or by adding special information to the bytecode [3, 19]. These techniques can reduce the amount of verification time, but as a compromise the program size increases due to restructuring or additional information.

Memory space consumed during verification is another important issue with Java bytecode, especially on devices with constrained ressources. Large and medium sized programs can easily have an overhead of multiple kilobytes [12], being too much for small systems. To reduce memory usage one can use requirements for the program code, leading to simpler code with less control points [13]. Another technique modifies the algorithm and uses multiple verification passes, each solving subproblems [4]. While conserving main memory these techniques often lead to increases in code size or verification time.

2.2 Annotations and Optimizations

Annotations in our sense are code informations added to the mobile code during its generation. They can be used by the runtime environment to improve execution speed or to increase program security. In general, no verification of the annotations is possible, so blind trust into them could lead to semantical errors and security exploits. Currently there are just two techniques to guarantee the right usage: either by using proof-carrying techniques or with secure annotations like in SafeTSA. Yet, neither of these are directly convertible to bytecode so the problem still arises. On the other hand code annotations on Java bytecode are not directly possible. One can use attributes, but there is no way to detect manipulations. In the last Java version, program annotations made by the programmer are introduced to the language and to bytecode, however for code annotations this is not of interest.

Often, annotations are used for optimizations on the runtime environment. They are applied dynamically during runtime in JIT and adaptive compilers, whereas for interpreters we do not know of any system utilizing annotations. Common optimizations which can be annotated are removal of bound checks [22], escape information [9] and virtual register placement [10]. Reig [18] suggested some other high level language information which can be used as annotation. In general, an intermediate representation which supports secure annotations could be very beneficial.

3 Type-separated Bytecode

In this section we try to picture the main ideas of our new intermediate representation called type-separated bytecode. The first part deals with basic concepts and functionality. During the second part ideas for two more advanced topics are presented.

3.1 Type-Separation

The type-separated bytecode is based on existing representations, e.g. Java bytecode [7] and the Common Intermediate Language (CIL) (as part of the Common Language Infrastructure (CLI) [5]). Like these formats type-separated bytecode uses a stack-based machine model. But instead of using just a single stack and register set the method of type-separation is applied.

The basic idea behind type-separation is to separate every single type from all different types. Thus, instead of having one untyped stack we now have as many stacks as types do exist, each associated with its appropriate type. To work with these stacks we must also have typed register sets, again one for each type. In figure 1 this concept is visualized. On the top one can see the untyped stack and register set. After type-separation we now have four stacks, three for the values on the stack and one corresponding to the register. This empty *float* stack is needed since in a stack machine every register access happens via a stack. The register set is split up into its three values. An extra register set for *double*-values can be omitted because they are just temporally on the stack. As can be seen only the stacks and register sets whose type is used in the program are present in the representation.

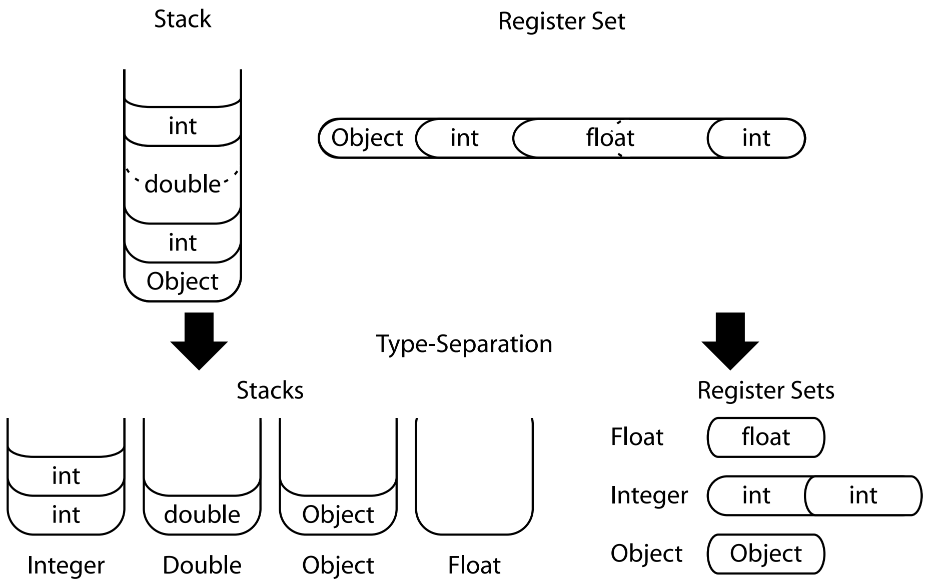

Fig. 1. Type-Separation

Given that every stack and register set must only contain values of its dedicated type, the memory usage of the verification process is reduced. It is no longer necessary to save a stack and register map on each control point as it is done in the traditional verification algorithm. It is sufficient to remember the heights of the stacks to prevent over- and underflows. To achieve type consistent behavior the used instructions have a special property. Each instruction knows per se where the operands of an operation must come from and the result should go to. Because of this property it is not possible to undermine the type system, thus the type-separated bytecode becomes type secure. For instance, an *integer* compare instruction would take its two operands from the *integer* stack and put the resulting *boolean* value onto the *boolean* stack. In contrast, an *integer* subtraction operation would place its resulting *integer* back onto the *integer* stack and also took its operands from there. Figure 2 shows an example of a simple program in type-separated format. Its execution and the effect on stacks and registers are displayed.

Java Sourcecode	Type-Separated Bytecode	Stacks			Register Sets	
		int	double	bool	int_0	double_0
int i = 1;	10: iconst_1	1				
	11: istore_0				1	
double d = 2.4;	12: ldc 2.4		2.4		1	
	15: dstore_0				1	2.4
if (d - i >= 1) {	16: dload_0		2.4		1	2.4
	17: iload_0	1	2.4		1	2.4
	18: i2d		2.4; 1.0		1	2.4
	19: dsub		1.4		1	2.4
	20: dconst_1		1.4; 1.0		1	2.4
	21: dcmp_ge			true	1	2.4
	22: if_false 27				1	2.4
d = 1;	25: dconst_1		1.0		1	2.4
	26: dstore_0				1	1.0
}	27: . . .					

Fig. 2. Example of the Execution of Type-Separated Bytecode

A further important point in creating type-separated bytecode is using constraints. For example we can demand that every register is initialized with a neutral value prior code execution, for example 0 for number types and *null* for reference types. This constraint saves space during verification because one does not need to save the state of register sets. There are other constraints like keeping the stack empty on jumps [13], but we have to decide which ones to incorporate into the intermediate representation yet.

Because of the type separation a great amount of different stacks and register sets are present. We expect that a direct execution could pose some serious penalties on runtime behavior. Therefore, it could be beneficial to transform the program after verification into a one stack machine model prior to execution. First examinations show that such a transformation could be achieved with just one program pass, resulting in a linear time algorithm (in the number of instructions). Together with our linear time verification algorithm this shows that type-separated bytecode can be used for linear time verification of mobile code at least.

3.2 Advanced Techniques

There are mainly two ideas for other features of type-separated bytecode. The first one deals with program annotations whereas the second can be used for simple program parallelization.

In our intermediate representation we plan to add annotations in the form of types and corresponding instructions. This way they merge with the language and can be verified during the verification phase, leading to secure annotations. These annotations can be used for program optimizations or to test security requirements. For example one may use such code annotations to reduce the number of nullchecks. Everytime a local object is accessed its type changes from insecure to secure. Further uses of this reference can occur without nullchecks. Such reductions may be superfluous for desktop computers with JIT and adaptive compilers but for constrained devices using interpreters this can give another performance increase.

Another technique to be developed is simple program parallelization. In type-separated bytecode every type has its corresponding stack. Therefore, if multiple instruction pipelines are available it is possible to associate them with appropriate stacks. Every operation performed on this stack can now be handled by the underlying instruction pipeline. Though this method is just a basic one we expect some performance increases. In future work this process can be refined to achieve better results.

4 Future Work

The development of type-separated bytecode is currently at the beginning stage. The presented work is just a mere overview and will be refined further in the future. We first have to decide on a general stack machine model, were it is possible to transform a broad choice of programs into. After construction of a common model we have to do first evaluations on the type-separated bytecode to find problems and restrictions. Further research should lead thereafter to the final stack machine model.

In latter phases of the project we will incorporate annotations into the language. After developing a general method to accomplish this the next stage will be to find suitable code annotations. Mostly they come from the areas of code optimization and security properties. Perhaps it is possible to add aspect oriented programming styles [2] to type-separated bytecode. This will lead to a broader usage of annotations for program security.

For the time being the last thing to accomplish is the building of a runtime environment for constrained devices, for example cell phones and JavaCards. In this area we suppose the largest improvements due to type-separated bytecode.

Not only the verification algorithm is better adopted to this setting. Through the developed annotations the mobile code can be made more efficient and the runtime behavior is improved.

5 Summary

In this paper we presented our primary ideas concerning a new mobile intermediate representation called type-separated bytecode. We suppose that this representation can lead to major improvements concerning program verification and runtime efficiency. This is especially true for constrained devices like cell phones, JavaCards and embedded systems. Further research will lead to the final version of type-separated bytecode and special techniques like annotations and simple parallel program execution.

References

1. W. Amme, N. Dalton, M. Franz, and J. von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*, volume 36 of *ACM SIGPLAN Notices*, pages 137–147, Snowbird, Utah, USA, June 2001. ACM Press.
2. Aspect-Oriented Software Association (AOSA). Aspect-Oriented Software Development (AOSD). Annual conference, visit http://aosd.net/.
3. I. Bayley and S. Shiel. JVM bytecode verification without dataflow analysis. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'2005)*, pages 185–202, 2005.
4. C. Bernardeschi, G. Lettieri, L. Martini, and P. Masci. A space-aware bytecode verifier for Java cards. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'2005)*, pages 216–232, 2005.
5. ECMA (European Computer Manufacturers Association) International. Common Language Infrastructure (CLI), Standard ECMA-335, Dec. 2002. http://www.ecma-international.org/publications/standards/Ecma-335.htm.
6. A. Gal, C. W. Probst, and M. Franz. A denial of service attack on the Java bytecode verifier. Technical Report 03-23, School of Information and Computer Science, University of California, Irvine, Oct. 2003.
7. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Reading, MA, USA, second edition, 2000.
8. M. Hagiya and A. Tozawa. On a new method for dataflow analysis of Java Virtual Machine subroutines. *Lecture Notes in Computer Science*, 1503, 1998.
9. A. Hartmann, W. Amme, J. von Ronne, and M. Franz. Code annotation for safe and efficient dynamic object resolution. In *Proceedings of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV'2003)*, pages 18–32, Warsaw, Poland, Apr. 2003.
10. J. Jones and S. N. Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):389–406, May 2000.
11. T. Kistler and M. Franz. A Tree-Based alternative to Java byte-codes. *International Journal of Parallel Programming*, 27(1):21–34, Feb. 1999.

12. X. Leroy. Java bytecode verification: An overview. In *Proceedings of the International Conference on Computer Aided Verification (CAV'2001)*, pages 265–285, London, UK, June 2001. Springer-Verlag.
13. X. Leroy. Bytecode verification on Java smart cards. *Software – Practice and Experience*, 32:319–340, 2002.
14. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, second edition, Apr. 1999.
15. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.
16. G. C. Necula. Proof-carrying code. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'1997)*, ACM SIGPLAN Notices, pages 106–119, New York, NY, USA, Jan. 1997. ACM Press.
17. OpenGroup. Architecture Neutral Distribution Format (XANDF) Specification. *Open Group Specification P527*, page 206, Jan. 1996.
18. F. Reig. Annotations for portable intermediate languages. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.
19. E. Rose and K. H. Rose. Lightweight bytecode verification. In *Proceedings of the Workshop on Formal Underpinnings of the Java Paradigm (OOPSLA'1998)*, Oct. 1998.
20. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'1998)*, pages 149–160, New York, NY, Jan. 1998. ACM.
21. R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Soot: A Java Optimization Framework, 1999. Sable Research Group of McGill University, http://www.sable.mcgill.ca/soot/.
22. D. E. Yessick. Removal of bounds checks in an annotation aware JVM, May 17 2004.