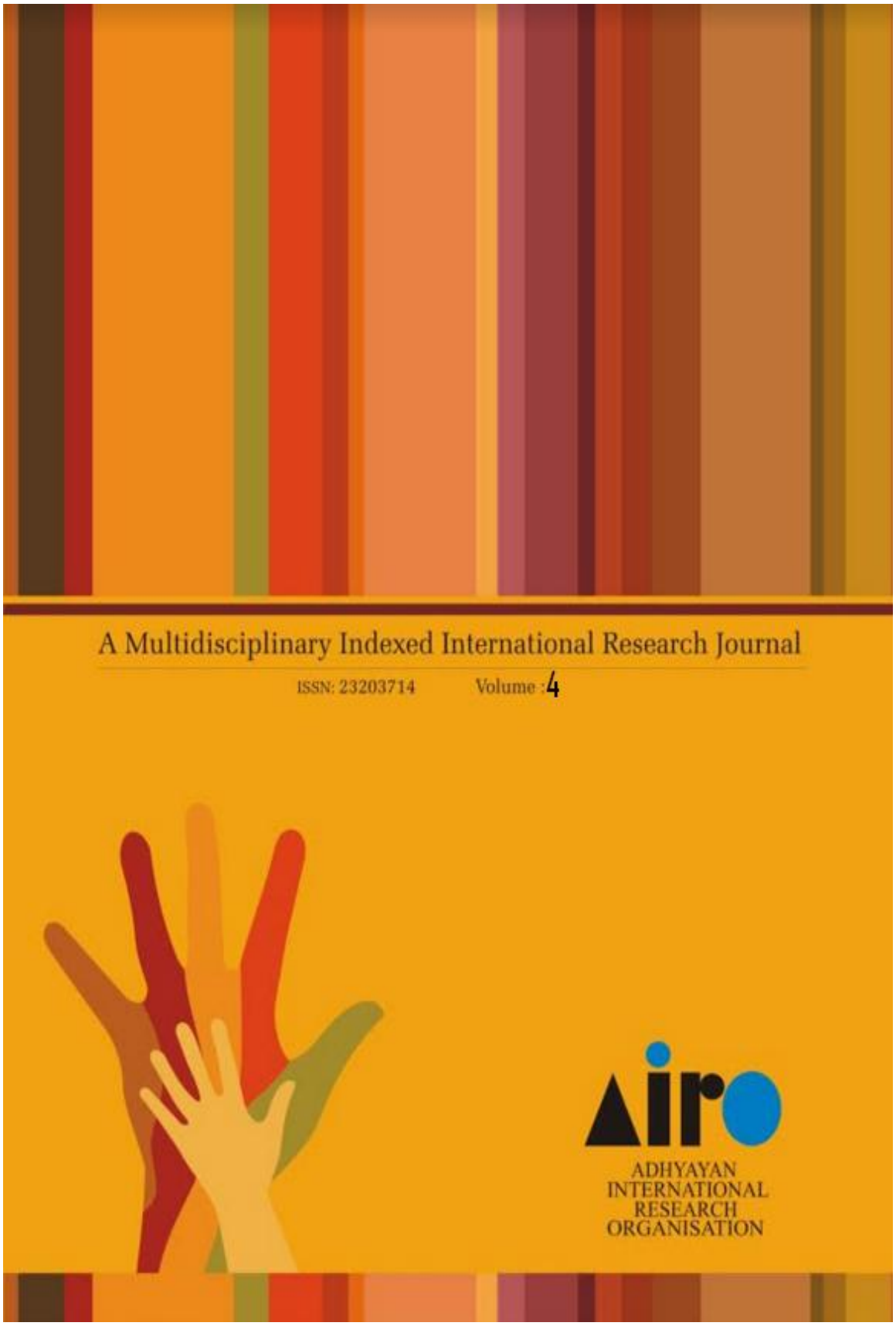
A Multidisciplinary Indexed International Research Journal
ISSN: 23203714
Volume : 4
ADHYAYAN
INTERNATIONAL
RESEARCH
ORGANISATION

OPTIMISING JAVA PROGRAMS THROUGH BASIC BLOCK DYNAMIC COMPILATION

Akhilesh Kumar (MCA, MPhil(CS))

Rajebdra Chandra Pandey (MCA)

Abstract

Executing Java bytecodes in hardware is typically faster than interpreting Java bytecodes. Many bytecodes are complex and may need to be interpreted by an additional controlling processor. In particular, method calls to the underlying system cannot be dealt with by Java bytecode alone. Dynamic compilers are executed in parallel to the bytecode they have translated. They are required to produce high quality code quickly.

The majority of existing dynamic compilers work by translating a method the first time it is called. This slows the Java environment down initially but, hopefully, in the long run will prove to have made the program execute faster. State-of-the-art JVMs [Sun99b] are using multiple levels of compilation; they aim to translate quickly at first and then target expensive compiler optimizations.

Keywords*:*

Java, Program, Block, Dynamic, Compilation

Introduction

Java programs are object-oriented. An object is viewed in object-oriented design as a black box that contains state which has defined behaviours for receiving and sending messages. The message passing mechanism is typically a method call, however, in Java information can also be passed by reading and writing to publicly accessible variables (state). Objects are instances of data-types called classes. Classes are able to inherit attributes from other classes, this is done in the Java programming language using the extends keyword. All classes are subclasses of the base class called java.lang.Object.

Although basic block translation of instructions happens in existing dynamic binary translation environments, dynamic Java compilers are method oriented. Method compilation reflects conventional compilation, with the addition of execution information. Basic block compilation is different in that it focuses on a smaller unit of execution and groups these to form larger blocks that are then optimised. There is a perceived cost in creating this information about larger blocks, and although trace scheduling is a natural consequence of a basic block compilation model, all existing JVMs are method oriented.

A new class definition can redefine or, as it is commonly referred to, override methods from a superclass. A method that takes an object of a particular class as an argument can have a subclass of it passed in also. If a method is to be called on an object then the exact method called depends on the class type of the object.

In Cardelli and Wegner's model, this is called inclusion polymorphism. Java also supports parametric polymorphism (more commonly referred to as overloading) whereby a method is uniquely identified by not only its name but also its parameter types. The JVM calls a parameter list a method descriptor.

Every time an object is created an initialiser is run which optionally takes parameters. Java initialises primitive types (see later) to default values, an object initialiser goes on to overwrite these values. Multiple inheritance in Java is a compromise. The Java programming language only allows a class to have one superclass unlike other object-oriented programming languages, such as C++, that allow a class to have multiple super classes. Having multiple super classes complicates method calls (message dispatch).

Programming languages that compile to a static binary file can calculate what methods can be called and optimize the dispatch mechanism appropriately. As Java is a dynamic language it was decided to keep the message dispatch mechanism quick and simple by only allowing single inheritance. The compromise in Java is that it supports interfaces. An interface is a class that defines methods but provides no

implementation, they are implicitly abstract and all methods are both public and abstract. An interface can have no variables but it can define constants.

As an interface provides no implementation or variables it cannot be instantiated. As with class inheritance, interfaces do provide a way of describing an abstract feature that several classes can implement. A class is permitted to implement as many interfaces as it requires. As with classes, interfaces can inherit features from another interface. As interfaces do not require any state they do not alter the object layout for a class. They only complicate the execution of a Java program by requiring a method dispatch mechanism that first has to select the appropriate interface for the method call and then the appropriate method.

As well as methods and state which are associated with objects, Java allows methods and state to be associated with a class. The keyword static is used to declare these in the programming language. The method main is a static method that is first called by the JVM.

A typical main method creates the programs objects and then passes control to them. In the Java environment there are certain global objects for performing primitive tasks such as string handling and file IO. These global objects are declared as static. They are initialized by a set of rules defining when a class is to be loaded. A special method called a class initializer is called the first time a class is loaded, this method sets up these global variables. The phase of execution where the JVM is setting up global variables is called the bootstrapping phase.

Linking and loading

A key part of a JVM is the class loader. The class loader is responsible for loading the class files and integrating them into the JVM. The primordial class loader is the class loader provided by the JVM, however, Java has an extensibility feature that allows it to be replaced by class loaders with extra facilities. For example, the primordial class loader may only be able to load uncompressed class files and not be able to load class files from over the Internet. A library can replace the primordial class loader with a

class loader of its own with these extra features incorporated.

As part of the security features of Java, this separate class loader is given its own name space so none of its classes can conflict with system classes. By writing the class loader in Java and having the JVM load it, the class loader can take advantage of Java's rich library code. Class resolution is the process of loading a class when it is required by a JVM. The JVM specification requires classes to be resolved in a lazy manner so that a class is only resolved the first time it is referenced.

References are made to other classes in the information contained in the class file about what classes this class file extends and implements. To enable a class to be resolved, these other classes must be loaded and resolved. Other classes are also referenced in bytecodes, these should not be resolved until the bytecode is executed for the first time. This places an unfortunate requirement on the JVM's interpreter or dynamic compiler; they cannot resolve and optimise certain bytecodes ahead of time.

Interpreter JVMs

Java's portability as a language stems from the fact it is compiled into a machine independent bytecode format that is intended to be interpreted within a JVM. To speed interpretation the operation represented by each bytecode is encoded into a single byte at the beginning of the instruction. Following the operation are zero or more operand bytes describing the operation. Common operations and operands are encoded into one bytecode to reduce the size of the compiled code.

A key feature to the bytecodes machine independent format is that it uses a stack to carry out operations instead of registers. By using a stack, processors with few addressable registers, such as the Intel IA32 instruction set architecture [Int87], are able to use their indirect addressing modes to emulate the stack. Instruction set architectures with more registers, such as the SPARC architecture can map the stack into registers. As the bytecodes use the top of the stack implicitly, bytecodes do not have to encode which register or memory address to use for an operation.

As well as a stack, Java bytecodes can access a pool of local variables and swap them in and out of the stack to have general purpose computations performed on them. One exception to this is the iinc bytecode that directly increments a specified local variable. Bytecodes are addressed by a virtual machine's program counter (PC) register. All PC values within a method are relative to the beginning of the method. The PC is used to calculate branch targets but its value can never be read. This allows a target machine to replace the PC with a value to speed up hardware, translation or interpretation.

Just-In-Time and dynamic compilation JVMs

Just-In-Time (JIT) compilation and dynamic compilation both refer to the method of executing Java bytecodes by translating them into native code and then executing the translated code instead of the bytecode. This causes a performance improvement as the translated code is cached and executed each time the bytecode should be executed. A JIT compiler compiles the bytecode into native code either when a class is loaded or when a method is executed for the first time.

A dynamic compiler is more selective over when compilation is performed and in certain cases a dynamic compiler may choose to interpret instead of compile the bytecode. This can hopefully avoid any unnecessary compilation expense. A dynamic compiler may also choose to optimise a hot region of code. The optimisation takes the form of conventional static compiler optimisations which, due to slowing the compilation process down, were not executed when the compiler was run the first time.

JIT compilation is a form of dynamic compilation where the heuristic of when to compile is very simple, once, when the class or method is first loaded or executed. The heuristic of when to compile can make all the difference with a JVM: compile too late and the benefit of faster code is never realised, compile too early and the dynamic compiler could compile code that will infrequently be executed.

The dynamic compiler, like computer hardware, has to choose what will

happen in the future from its past experience (for example, temporal locality properties exploited by caches). Java gives clues with methods such as class initialisers that they can only ever be run once. However, because a method is run once does not mean a lot of time can not be spent in it, for example, if there was a large loop in the body of the method. The optimisations to increase program speed performed by dynamic compilers are largely the same as in conventional static compilers.

Dynamic compilers have a benefit in that execution statistics of the code to compile are available as optimisations are performed. As with Java hardware the overhead of a stack is eliminated by the dynamic compiler. Java bytecode allows a direct mapping of the stack to registers by making it a requirement of the bytecode that each time it is executed the stack depth be the same.

Dynamic Binary Translation

Dynamic binary translation is a technique used to run programs that are not natively compatible with a computer platform. Binary programs have little structural information contained within them and are sometimes impossible to reverse engineer, thereby inhibiting migration to a new computer platform. For example, code compiled for an Intel IA-32 machine, running Microsoft Windows, has information about functions and subroutines in the source code at the time of compilation.

The compiler generates binary code that has data structures placed in with the code, the data typically being used for jump tables and other pre computed values. The compiler or the programmer may even create code that is self modifying (for example, value specific optimisation). A static binary translator can only translate the effect of this data and code if it knows the format in which it was generated. This may alter as computer languages and compilers change; hand-crafted assembler code has no style to conform to. Dynamic Binary Translators (DBTs) do not suffer from problems of code discovery, as they translate and execute code in a lazy manner.

Only code used within a program's execution needs to be translated. If a previously untranslated path is taken by a branch then the DBT is called to

translate the path. The new path taken is calculated by the executing dynamically generated code. The term subject machine is used to refer to the computer architecture a program was intended to run on. The subject environment is the environment the DBT produces. This may be little more than the subject machine, but it may include loaders, linkers, and emulated software calls or hardware devices. The subject program is the program the subject environment executes. The instructions contained in the subject program are subject code. Likewise, the code produced by the DBT is called target code and is executed in a target environment.

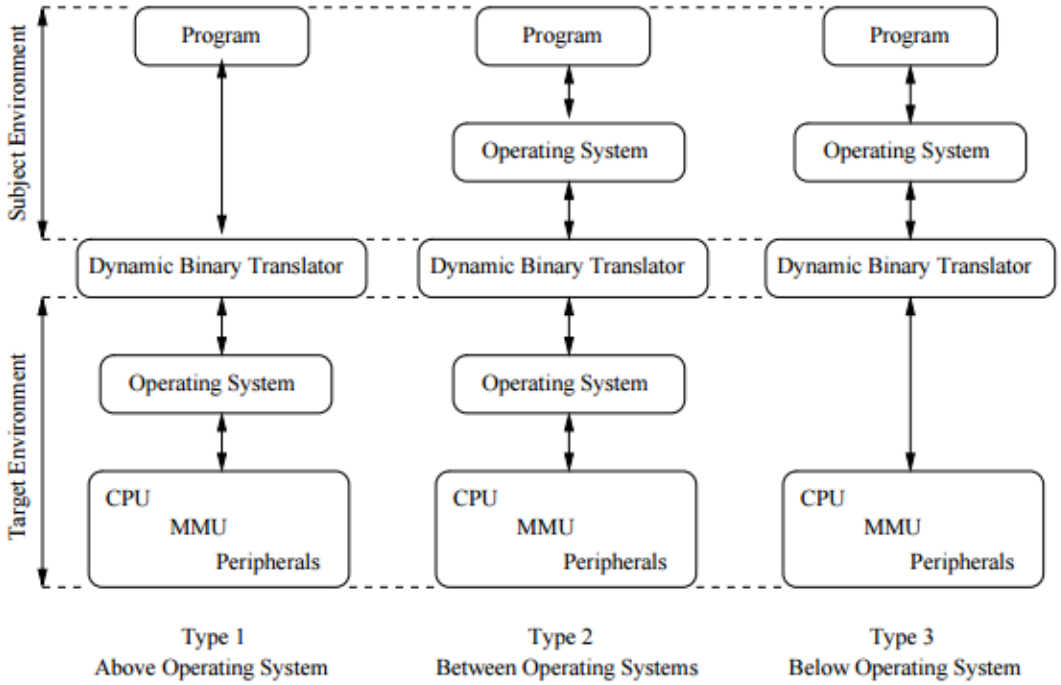

Figure 1: Dynamic binary translation environments

Conclusions

Existing dynamic optimization techniques have been applied from a dynamic binary translator and used to create a JVM environment. This has not been at the expense of throwing away the rich amount of extra information Java gives its dynamic execution environment. Although the techniques described have been demonstrated to have low overhead, the thesis hasn't shown they result in a JVM faster than the state-of-the-art. Consideration on

how this performance can be improved has been presented throughout the thesis. Improving the translation of the IR to target machine code, and optimizing the class library are likely to improve performance the most.

The JVM shares similarities with a number of virtual machines and computer architectures. By recreating high-level procedure call and return semantics within the dynamic binary translator, fixed register windows and lazy recursion detection can be used with other architectures and virtual machines.

References

1. B. Alpern, C. Attanasio, and et al. The Jalapeno virtual machine. The IBM Systems Journal, 39(1), 2010.

2. Ole Agesen and David Detlefs. Mixed-mode bytecode execution. Technical Report SMLI TR-2000-87, Sun Microsystems Inc., June 2010.

3. Erik Altman, David Kaeli, and Yaron Sheffer. Welcome to the opportunities of binary translation. IEEE Computer, 33(3):40–45, March 2010.

4. Denis N. Antonioli and Markus Pilz. Analysis of the Java class file format. ifi-98.04, Department of Computer Science, University of Zurich, April 28 2008.

5. Macintosh Application Environment 2.0. Technical report, Apple Computer Inc., Cupertino, California, USA, 2014.

6. Apple enhances Macintosh Application Environment for HP and Sun workstations. http://product.info.apple.com/pr/press. releases/1997/q1/961118.pr.rel.mae.html, November 2006.

7. Kristy Andrews and Duane Sand. Migrating a CISC computer family onto RISC via object code translation. In Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS), volume 27, pages 213– 222, New York, NY, 2012. ACM Press.

8. Alfred V. Aho, R. Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 2006.

9. V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical report, HP Labratories Cambridge, 2009.

10. Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In SIGPLAN Conference on Programming Language Design and Implementation, pages 1–12, 2010.

11. R. Bedichek. Some efficient architecture simulation techniques. In Proceedings of the USENIX Winter 2010 Technical Conference, pages 53–64, Berkeley, CA, 2010. USENIX Association.

12. Arndt B. Bergh, Keith Keilman, Daniel J. Magenheimer, and James A. Miller. HP 3000 emulation on HP precision architecture computers. Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company, 38(11):87–89, 2007.

13. Zoran Budimlic, Ken Kennedy, and Jeff Piper. The cost of being Object-Oriented: A preliminary study. Scientific Computing, 7(2):87–95, 2009.

14. Bruno Blanchet. Escape analysis for Object-Oriented languages: application to Java. ACM SIGPLAN Notices, 34(10):20–34, 2009.

15. Alex Brown, Geraint North, Frank Thomas Weigel, and Gareth Anthony Knight. Method and apparatus for performing native binding. GB Patent Number GB2404042, September 2013.

16. Bochs: the cross platform IA-32 emulator. http://bochs. sourceforge.net/, 2011.

17. Aart J. C. Bik, Juan E. Villacis, and Dennis B. Gannon. javar: A prototype Java restructuring

compiler. Concurrency: Practice and Experience, 9(11):1181–1191, 2007.