MicroJIT: A Lightweight Just-in-Time Compiler to Improve Startup Times

by

Federico Sogaro

Master of Information Engineering
Padua University, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Masters in Computer Science

In the Graduate Academic Unit of Computer Science

Supervisors:	Kenneth Kent, PhD, Computer Science
	Eric Aubanel, PhD, Computer Science
Examining Board:	Suprio Ray, PhD, Computer Science, Chair
	Gerhard Dueck, PhD, Computer Science
	Brent Petersen, PhD, Electrical and Computer Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September, 2016

©Federico Sogaro, 2016

Abstract

The startup phase of an application represents a tiny fraction of the total runtime, but it is considered, nevertheless, a critical phase for both client and server environments. In the JVM, the Just-in-Time compiler (JIT) translates Java bytecodes to native machine code to improve the performances of an application. This thesis investigates whether using two different JIT compilers in the same JVM can improve startup time. A lightweight JIT system (i.e. MicroJIT) has been integrated into the J9 JVM and it has been set up to perform an initial, low-optimized, but fast compilation while, at a latter time, the standard JIT recompiles Java bytecodes with better, but more expensive, optimizations. This solution has been evaluated using different benchmarks and metrics and has been compared to the JVM in different configurations: default configuration, standard JIT tuned for optimal startup time (option *Xquickstart*) and executing the Java application with Ahead-of-Time code available from the Shared Class Cache.

Experimental results show that enabling MicroJIT, compared to the default configuration, significantly reduces the startup time of the considered bench-

marks but with a cost in memory usage and, in some cases, a reduction in throughput performance.

The solution presented in this thesis does not improve the startup time as much as the other solution already implemented in the JVM, however in some conditions it could be the best option since, the penalty on throughput performance is less than the one caused by using the option `Xquickstart` to tune the main JIT system for optimal startup time, and it does not require the presence of the Shared Class Cache.

Acknowledgements

I would like to thanks those who made this project possible: my supervisors Kenneth B. Kent and Eric Aubanel for the support during the whole Master's program, the CASA project manager, Stephen MacKay, for proofreading my thesis and for providing valuable suggestions on how to improve my English, the colleagues of the CASA project for their help when first introduced to the project, all who made the lab a positive and knowledgeable environment. Special thanks to my contacts at IBM, Peter Shipton, Vijay Sundaresan and Marius Pirvu, who helped during the full duration of the project with technical assistance and feedback, and for coming up with the original idea for the project.

I would also like to thank my parents, for the moral and financial support, and my wife and daughter for being so patient when I could not dedicate them the rightful amount of time.

This project was possible thanks to the financial support provided by the Atlantic Canada Opportunity Agency (ACOA) through the Atlantic Innovation Fund (AIF) program.

Table of Contents

Abstract **ii**

Acknowledgments **iv**

Table of Contents **v**

List of Tables **ix**

List of Figures **x**

List of Acronyms **xii**

1 Introduction **1**

2 Background **5**

2.1 Java Language 6

2.1.1 Java Compilation 7

2.2 Java Virtual Machine 8

2.2.1 Java Stack 10

2.2.1.1 Operand Stack 11

2.2.2 Initialization 12
2.2.3 Synchronization 13
2.2.4 Garbage Collection 15
2.2.4.1 GC Policies in IBM J9 VM 18
2.2.4.2 Generational Concurrent 19
2.2.4.3 Object Allocation 20
2.2.5 Port and Thread Libraries 21
2.2.6 Shared Class Cache 21
2.2.7 Just-in-Time Compiler (JIT) 22
2.2.8 MicroJIT 25
2.2.8.1 MicroJIT-JVM Interface: Initialization 27
2.2.8.2 MicroJIT-JVM Interface: Compilation 28
2.2.8.3 MicroJIT-JVM Interface: Invocation 29
2.2.8.4 MicroJIT-JVM Interface: Returning to the Interpreter 29
2.2.8.5 MicroJIT-JVM Interface: Execution 32
2.3 Benchmarks 34
2.4 Related work 37
2.4.0.1 Multi-level Compilation 37
2.4.0.2 Off-line Profiling 37
2.4.0.3 Ahead-of-Time Compilation 38

3 Design and Implementation 39

3.1 Requirements 41
3.2 Strategy 42
3.2.1 Phase 1 - Port MicroJIT to Java 8 43
3.2.1.1 MicroJIT Compilation Times 43
3.2.1.2 Quality of the Native Code Generated by MicroJIT 44
3.2.1.3 Optimal *count* and *bcount* Values for MicroJIT 44
3.2.2 Phase 2 - Enable MicroJIT with TRJIT 45
3.2.2.1 Optimal *count* and *bcount* Values for TRJIT . 46
3.2.2.2 Throughput of the JVM with MicroJIT . . . 46
3.2.3 Phase 3 - Improving MicroJIT Compilation 46
3.3 Implementation 46
3.3.1 Phase 1 - Port MicroJIT to Java 8 47
3.3.2 Phase 2 - Enable MicroJIT with TRJIT 49
3.3.3 Phase 3 - Improve MicroJIT 52
3.3.3.1 Bytecodes that MicroJIT cannot compile . . . 53
3.3.4 Write Barrier for GenCon GC Policy 56
3.4 Development Platform 57
3.4.1 Benchmarking 58

4 Evaluation **60**
4.1 Evaluation of MicroJIT Implementation 61
4.1.1 Temporarily Returning to the Interpreter 62

4.1.2 Implementation of `invokeVirtual` 63
4.1.3 Return of a Method Compiled by MicroJIT 66
4.1.4 Write Barrier Check 67
4.1.5 Optimal *count* and *bcount* Values for MicroJIT 70
4.2 Evaluation of MicroJIT with TRJIT 73
4.2.1 Existing Solution for Startup Time 74
4.2.1.1 Compilation Time and Compilation Delay . . 75
4.2.2 Threshold Counter for TRJIT 80
4.2.3 Throughput Analysis 82
4.2.4 Footprint Analysis 86

5 Conclusions and Future Work **88**

Bibliography **98**

Vita

List of Tables

2.1 Optimization Implemented in Testarossa JIT 26

4.1 Bytecodes that MicroJIT does not Translate (DaCapo-Batik) . 64

4.2 Bytecodes that MicroJIT does not Translate (Eclipse Startup) 65

4.3 Ratio of Write Barriers with an Old Destination Object 69

4.4 Compilation Timing and Delay 79

4.5 Number of Invocations of Methods Compiled with MicroJIT . 81

4.6 Throughput Results (SPECjbb2005) 86

List of Figures

2.1 Java Program Flowchart 7

2.2 High-level Architecture of the IBM J9 Java Virtual Machine . 9

2.3 Operand Stack 12

2.4 Example of Lost Object during Concurrent Marking 20

2.5 Testarossa JIT (TRJIT) 24

2.6 Simple Method that does not need a Java Stack Frame 30

2.7 Temporary Jump to the Interpreter 31

3.1 Flowchart for Triggering Compilation with MicroJIT and TRJIT 51

3.2 Flowchart for invokeVirtual Implementation in MicroJIT . . . 55

4.1 MicroJIT versus Interpreter 62

4.2 MicroJIT with and without Implementation of `invokeVirtual` 66

4.3 MicroJIT with and without "fast path" on Method Return . . 67

4.4 MicroJIT with and without Write Barrier Check 69

4.5 Effect of Varying *count* and *bcount* in MicroJIT 72

4.6 TRJIT with and without MicroJIT 73

4.7 TRJIT (Xquickstart) with and without MicroJIT 76

4.8 Contribution to Startup Speed from AOT Code in the SCC . 76
4.9 TRJIT (AOT) with and without MicroJIT 77
4.10 Effect of Varying *count* and *bcount* for TRJIT (DaCapo-Batik) 83
4.11 Effect of Varying *count* and *bcount* for TRJIT (Eclipse Startup) 84
4.12 Throughput Results for Liberty-Tradelite 85
4.13 Memory Footprint 87

List of Acronyms

AOT	Ahead-of-time compilation, related to Just-in-Time Compilation
CAS	Centre Advanced Studies or Compare-and-Swap
CASA	Centre Advanced Studies—Atlantic
GC	Garbage collection
IBM	International Business Machines
JavaME	Java Mobile Edition
JavaSE	Java Standard Edition
JIT	Just-in-Time Compiler
JNI	Java Native Interface
JVM	Java Virtual Machine
SCC	Shared Class Cache
STW	Stop-the-world, related to Garbage Collection
TLH	Thread Local Heap, related to Object Allocation
TRJIT	Testarossa Just-in-Time Compiler

Chapter 1

Introduction

The Java language, created in 1995 by James Gosling at Sun Microsystems, is one of the most popular languages currently available. IEEE Spectrum ranked Java in the first position in the 2015 Top Ten Programming Languages, based on multiple metrics including searches on Google, open source projects on GitHub, job offers (e.g. Dice) and mentions in journal articles (i.e. IEEE Xplore) [9].

The Java runtime, responsible for loading and executing a Java program, is called the Java Virtual Machine (JVM). Oracle, which bought Sun Microsystems in 2010, owns the Java trademark and maintains OpenJDK, an open-source implementation of the Java Virtual Machine which has been selected as the official reference implementation since Java Standard Edition 7 [15].

Other implementations exist and the Technology Compatibility Kit (TCK) is the test suite used to verify that a specific implementation is compliant

with the Java Virtual Machine Specification [17].

International Business Machine (IBM) has a strong interest in the Java language. It develops the J9 JVM, a high-performance JVM implementation, which plays an important role in IBM business and on which IBM invests significant effort to improve its performance.

A Java Virtual Machine virtualizes the underlying operating system and hardware and provides the software necessary to load and execute a Java program. The Java compiler translates Java source files into an intermediate format in which the instruction set consists of Java bytecodes, the class files. These files cannot be executed directly on the hardware and it is the task of the JVM to load and interpret the bytecodes that represent the Java program.

Just-in-Time compilation is a known approach in computer science, to improve the performance of interpreter languages, or, in the case of Java, of an interpreter intermediate format. A Just-in-Time compiler (JIT) translates at runtime the bytecode instructions that must be interpreted by the software (JVM), into native machine code, which can be executed directly on the hardware.

JIT systems are very effective in improving performance in the long term, after the application has run for sufficient time for the JIT to compile and optimize the methods in the most frequent execution paths. Before then, during startup, the JIT system is not as effective, especially at the very beginning of the execution when the application runs exclusively in interpreted

mode. Different solutions have been researched to make JIT more effective earlier in the execution of the application [31], [30], [6].

This thesis investigates whether using two different JIT compilers in the same JVM can improve the startup time of a Java application. The first JIT system would have the task of performing an initial, low-optimized, but fast compilation. The main JIT system would compile the same method with better optimization, if necessary, later during the execution of the application.

The main contributions of this thesis are:

- The considered lightweight JIT system (MicroJIT) can improve, in some conditions, the startup time of a Java application.
- MicroJIT compilation has been analyzed and a number of areas have been identified to improve the quality of the generated code. Some optimizations has been implemented and evaluated.

In the first part of the project MicroJIT, a lightweight JIT compiler designed and implemented in the J9 JVM for Java 6 Mobile Edition, is ported to the J9 JVM for the Java 8 Standard Edition and is modified to work with the standard JIT system. The JVM with two active JIT systems is evaluated to determine how enabling MicroJIT affects the startup time of the application. The background information, necessary to understand the concepts used in the rest of the project, is presented in Chapter 2. This includes an introduction to the Java language and a detailed description of the Java Virtual

Machine and of its relevant components. The standard JIT system and the MicroJIT system will be explained in detail, focusing on the interface between MicroJIT and the other components of the JVM. Chapter 2 concludes showing the existing solution related to the problem of startup time. With Chapter 3 the design of this project is explained and the implementation is described in detail. Chapter 4 presents and evaluates the results. Finally, in Chapter 5, the conclusions are drawn and an overview of the future work is given.

Chapter 2

Background

This chapter presents the necessary concepts to understand the rest of the thesis. The first section (2.1) introduces the Java language and describes how Java source code is compiled to Java bytecode, usually referred to simply as “bytecode”[33]. The Java Virtual Machine (JVM) is then described in detail (Section 2.2), starting from the basic concept on how it works, to the description of some of the critical parts of the JVM which the Just-in-Time (JIT) compiler interacts with: synchronization, garbage collection, object allocation and the shared class cache.

Section 2.2.7 describes in detail the existing JIT system called Testarossa JIT (it will be referred as TRJIT) and the JIT system on which the work in this thesis will focus: MicroJIT. Section 2.3 is dedicated to the benchmarks that will be used and finally, Section 2.4 presents related work.

2.1 Java Language

The Java programming language is a high-level, object-oriented programming language. Java was designed with some very useful features that make the language easy to learn and to use. The Java syntax is very similar to the syntax of C and C++, so programmers familiar with those languages could start writing Java programs without much effort[24]. The Java language, instead of being compiled to native code like C/C++, is compiled to a machine-independent format, called "Java bytecode" or simply "bytecode" [24]. Splitting compilation and execution into different phases is an important feature that allows a programmer to develop on a platform and compile the program in a format that can be executed on any other platform for which a Java runtime exists, without any modification of the program itself [33].

Java was also designed to remove from the developer the burden of managing memory [33]. This was a major achievement in simplicity compared to lower level languages like C and C++, where memory must be managed by the programmer and where it is relatively easy to make mistakes that could cause memory leaks or memory corruption. These problems are particularly important nowadays with security being a major concern. According to CERT advisories, memory corruption is responsible for about half of the reported attacks [32].

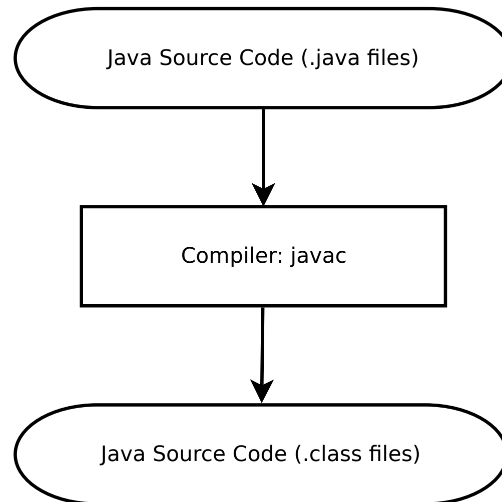

Figure 2.1: Java Program Flowchart.

2.1.1 Java Compilation

Java source code is defined by the Java Language Specification [24]. The Java compiler (i.e. `javac`), a program written in Java itself, compiles Java source code to Java bytecode that is a set of instructions defined in the Java Virtual Machine Specification [33] (Figure 2.1). The same specifications describe in detail how the bytecode and all information required at runtime are stored in the generated class files. Java bytecode can be seen as the assembly language (in respect to C/C++) of the Java language, while a programmer does not need to know it, understanding Java bytecode can help writing Java programs [12].

Each Java Virtual Machine instruction is represented by a single byte (op-code) that can be followed by additional parameters as required, therefore up

to 255 different bytecodes can exist [33]. The Java Virtual Machine Specification defines all valid bytecodes, i.e. all bytecodes that can be generated by the Java compiler. With fewer than 255 bytecodes currently defined, some undefined bytecodes can be used internally by the Java Virtual Machine to simplify and optimize execution.

2.2 Java Virtual Machine

The Java Virtual Machine (JVM) is a software abstraction of the operating system and the machine on which the JVM is installed. The JVM must provide all the functionalities necessary to execute a Java program (i.e. bytecode) [33] (Figure 2.2):

- Manage the threads spawned by the application and synchronize them among themselves and with the native threads of the JVM itself
- Load and verify the classes that make up the Java program
- Allocate and manage the memory
- Interpret and execute the bytecode instruction set

The listed features do not depend on the platform on which the JVM is running, therefore the same Java program can be executed on any JVM without any modifications.

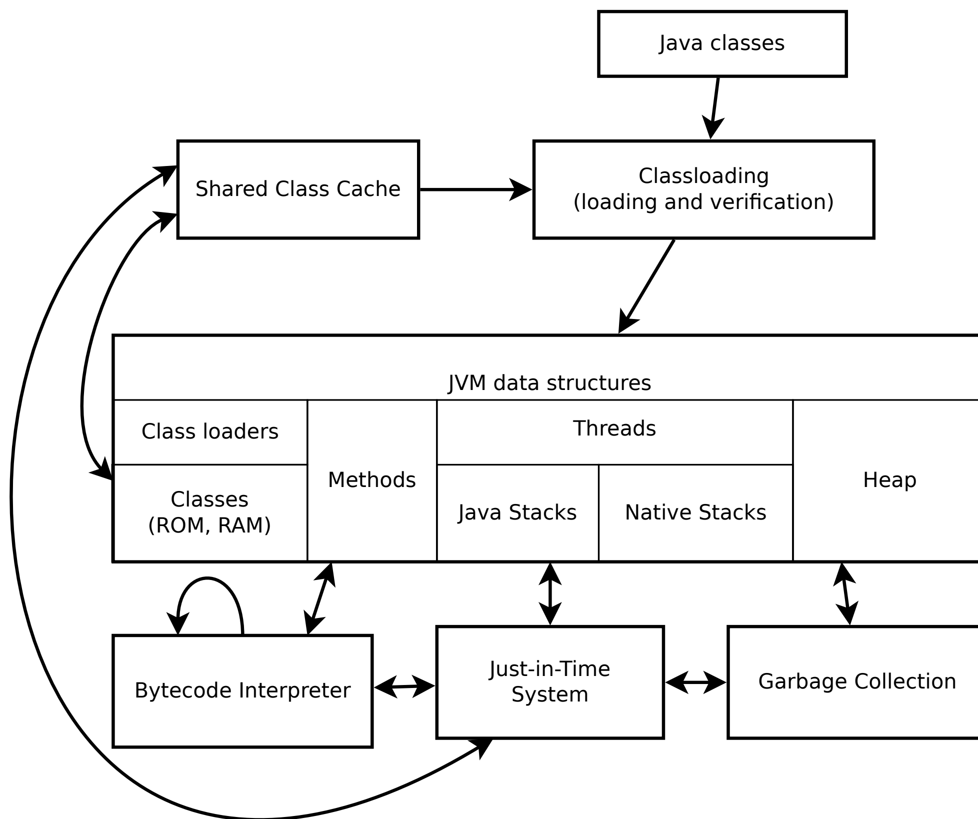

Figure 2.2: High-level Architecture of the IBM J9 Java Virtual Machine.

2.2.1 Java Stack

The JVM is a stack-based state machine that continuously fetches and decodes bytecode instructions until the Java program terminates. The Java stack is the most important structure storing the state of the Java program after each bytecode is interpreted. Each thread is associated with a Java stack that no other thread can access during normal operations. When a method is invoked, a new Java stack frame is created and pushed on top of the Java stack belonging to the specific thread. When a method returns, the corresponding Java stack frame is popped and the previous frame becomes the active frame.

Each frame in the Java stack includes information specific to the executed method and some additional information necessary to jump to the previous frame when the method returns. This approach is similar to how a native C/C++ program is executed:

- Program counter - points to the bytecode currently interpreted
- Method data:
  - Pointer to the method structure itself
  - Parameters passed to the method
  - Local variables
  - Operand stack

- Special values - required to pop the current frame and make the previous frame current when the method returns

The Java stack starts small and can increase and shrink in size during execution. If the application needs a stack larger than the specified maximum size, the application terminates with a StackOverflowException.
Each thread is also associated with a native stack that is used to execute the functions of the JVM itself (C/C++), and the native methods through the Java Native Interface (JNI).

2.2.1.1 Operand Stack

The Operand stack is a section of the Java stack used to store the data loaded from memory or produced by other bytecode instructions. This approach is different from the native register-based one used, for example, in x86-based and POWER processors, in which operand position is encoded in the instructions.
Figure 2.3 shows a simple example where values are pushed and popped from the stack depending on the operation:

- `iconst_1` - pushes the constant integer 1 on the stack. Some popular values, such as integers between -1 and +5, have a dedicated bytecode;
- `iload` - pushes the content of a local variable on the stack. Some bytecodes, such as `iload`, require additional parameters stored in the bytes after the bytecode itself;

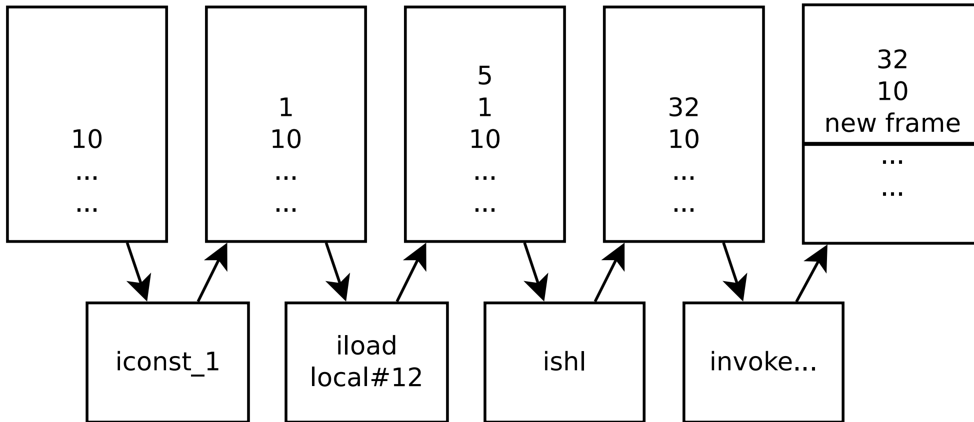

Figure 2.3: Operand stack.

- `ishl` - shift left operation for integers, it pops two operands and pushes the result back on the stack.

- `invoke...` - The stack-based approach applies also when a new method is called with an `invoke` bytecode. The arguments for the newly called method have already been pushed on top of the operand stack in the current Java stack frame and they become part of the new frame.

2.2.2 Initialization

Modules of the JVM are compiled as shared libraries. The libraries are loaded during initialization. Some libraries are always loaded (e.g. garbage collection), while others are loaded only if specified in the options passed to the Java executable. The module on which this project focuses, MicroJIT, is organized to generate two different shared libraries when the JVM is compiled.

One library contains all the functions needed during normal execution, while the second library includes functions used for debugging purposes. This approach was used to reduce the total footprint of the JVM during execution when debugging information was not required.

2.2.3 Synchronization

Multithreading is a technique used by an application to execute different parts of the code in parallel. High performance languages like Java, support multithreaded programming as a standard feature. When a multithreaded application is executed, threads act simultaneously and can access the same data since they share the same memory address. Synchronization is the mechanism that protects and coordinates access to the shared data contended among threads. Locks are the building blocks to implement synchronization; they are thread-safe structures that store the state of the application and on which threads relay to determine if they can access a critical region. A critical region is the section of the code where resources are shared between threads [23].

A mutual exclusion lock (mutex), the simplest implementation of a lock, can be held only by a single thread. Before a thread enters a critical region, it changes the state of the mutex to stop any other thread from entering the same critical region. This operation is referred to as *acquiring the lock*. When the thread exits the region it must *release the lock* to allow other threads to proceed with execution. The intrinsic lock in Java (i.e. the `synchronized`

keyword in the Java language) acts as a mutex.

In the IBM J9 JVM locks are implemented as Tasuki locks [35] [21]. A Tasuki lock can exist in two states: flat lock and inflated lock.

A flat lock consists of a single variable. To acquire the lock a thread saves its ID in the variable with a compare-and-swap (CAS) instruction. If the CAS instruction fails the thread will try to inflate the monitor, if not inflated already. In the inflated lock, the variable stores a pointer to a monitor object instead, which contains all the information required to manage a group of threads waiting to acquire the lock. When an inflated lock is not required anymore, it can be deflated to return to the flat lock state.

A Tasuki lock, in the absence of contention, is extremely fast, requiring only a CAS instruction to acquire the lock and a test and assignment to release the lock while, in the presence of contention, the threads will manage the lock in an elegant way with little or no busy waiting [35].

In Java, objects and methods can be used for synchronization. In the case of an object, the lock variable is stored in the object itself. In the case of a method, the lock variable is stored in the Java stack. Since Java 6, not all objects include the lock variable. This decision was taken to reduce the memory footprint of Java applications, based on the assumption that some types of objects (e.g. String) are almost never used for synchronization. To comply with the Java Virtual Machine Specification these objects can still have a lock variable stored in a separate structure (i.e. hashtable).

2.2.4 Garbage Collection

In Java, memory management is hidden from the developer [33]. Memory management is a necessary part of the JVM: it manages allocation of new objects in the heap region of the memory and frees the memory when objects are no longer used. The part of memory management that deals with reclaiming dead objects is called Garbage Collection (GC). GC is a complex and time consuming activity. A typical program continuously creates new objects during its execution and abandons them as soon as they are not needed. Since in Java memory is automatically managed by the runtime, the programmer does not explicitly free the memory occupied by dead (unreachable) objects and the heap fills quickly. Periodically, the JVM must intervene and reclaim the space used by dead objects by performing a GC.

There are two main approaches to garbage collection: reference counting and marking [28]. The former focuses on identifying dead objects, keeping a counter for each object, which stores the number of references to the object. When the counter reaches zero the object is unreachable and the space can safely be reclaimed. Reference counting has the important feature of distributing the GC workload during execution, however it also has two major disadvantages: it cannot detect unreachable objects that are part of a cycle and it leads to heap fragmentation.

The latter approach, marking, focuses on identifying the live objects. At any given time, in the life of an application, the state of all the Java objects can be represented by a directed graph. Nodes and edges correspond respectively to

Java objects and to references between them. The JVM is only directly aware of the objects to which it stored a reference on the Java stack. This set of objects is referred to as the root set. Following the references between objects (i.e. walking the graph), starting from the root set, it is possible to identify all the objects that the application will be able to access. The marking phase consists of walking the graph and flagging the reached objects as *alive*. The GC, knowing which objects are alive, can then safely reclaim the space occupied by the dead objects. Marking algorithms require the application to pause, to avoid any change in the object graph. This requirement, called stop-the-world (STW), is an important limitation of this approach but, there are solutions.

Marking algorithms can be categorized further. The simplest approach is, after the marking phase, to set the space used by dead objects as free (sweeping phase). This approach is fast but it leads to fragmentation [28]. The second approach consists of moving live objects next to each other in the same space (compacting collector) or in a different space in the heap (copying collector). This approach however is more complex and requires updating the pointers to the moved objects [28].

Another popular optimization to manage memory allocation is to divide the heap into regions, this is based on the assumption that objects tend to die young [28]. New objects are created in a specific region, often called the nursery, and when they survive for a specific time, measured in number of GCs they have survived, they are moved to other regions. Regions can be

garbage collected separately, specifically a partial GC will collect only the nursery region and a full, more expensive GC, will collect all the regions periodically. This approach is called Generational GC [28].

The root set, in the context of a generational GC, assumes a different meaning compared to the one previously described. When performing a partial GC, the references stored on the Java Stack cannot be directly used because it would lead to the need to perform a GC in all the regions. For each region of the heap, on which a partial GC can be performed, a special set is necessary. This set, called the remembered set, is used to obtain all references from objects in another region to objects in the region that needs to be garbage collected. A remembered set is used in exactly the same manner as a root set in a partial GC.

Parallelization of the GC algorithm is another popular approach to speed up GC. Modern computer architectures can execute many threads in parallel and when a GC operates in the STW phase, a considerable amount of computing power is available. Parallelization of the GC algorithm makes better use of the available CPU time, dividing the workload among multiple threads and therefore reducing the total GC time.

A related concept to parallelism is concurrency. While parallelism is about executing possibly related tasks simultaneously, concurrency is about organizing the execution of independent tasks [2]. A concurrent GC is a further optimization that reorganizes the GC activity to be concurrent with the execution of the application. The STW phase in a concurrent GC can be reduced

considerably leading to fewer wasted resources and allowing the application to continue execution during GC itself. Concurrency however, introduces complexity, and in particular, it is necessary to keep track of the activity of the application related to references.

A write barrier is a technique used during execution to track all changes of references between objects. Every time a reference is modified (write operation), the memory management system is made aware of the change so it can perform the necessary operations to maintain its state consistency.

2.2.4.1 GC Policies in IBM J9 VM

J9 implements multiple GC policies. A policy can be selected when launching the Java application, with the `gencon` policy being the default. The choice on the policy depends on the application [8] and while some policies are more versatile than others, there is no perfect policy:

- `gencon` (default) - *Generational Concurrent* is a generational GC with a concurrent mark phase. This policy minimizes GC pause times and works well for applications with a large number of short-lived objects.

- `balanced` - *Balanced GC* is a region-based GC that uses multiple techniques (mark, sweep, compact, copy) aiming at reducing the maximum pause time. Balanced GC can be enabled only on 64-bit architectures and is designed specifically for large heaps.

- `optavgpause` - *Optimize for pause time* is a mark and sweep GC in

which both the marking and the sweeping phases are concurrent. This policy offers a lower pause time compared to `optthruput`.

- `optthruput` - *Optimize for throughput* is a mark and sweep GC with occasional compaction. The policy is optimal for applications that prioritize throughput and that can tolerate long GC pauses.

- `metronome` - *Metronome GC* is the only policy that allows fine control on the pause time and is the policy of choice for real-time applications.

2.2.4.2 Generational Concurrent

Generational Concurrent is the GC policy used in this project. Memory is divided into two main areas depending on the age of the objects: the *tenure* space, for old objects, and the *nursery* space for newly allocated objects and objects not old enough to be copied to the tenure space. The age of the objects depends on the number of GCs they have survived.

In the `gencon` GC policy, write barriers are particularly important and they are necessary in two situations:

- To maintain the remembered sets up-to-date. The remembered set is necessary to perform partial GCs on the nursery space.

- To keep track of any change in references between objects if the concurrent mark phase is active, and therefore to avoid losing track of live objects (see Figure 2.4).

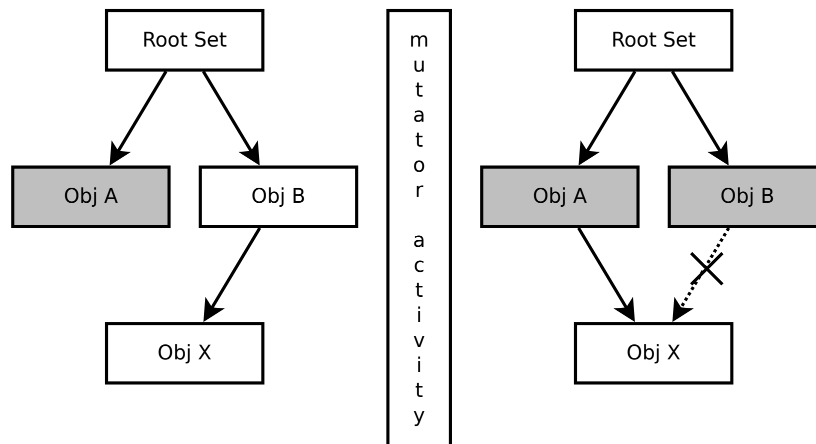

Figure 2.4: Example of lost object during a concurrent marking phase without write barrier. *Before* the reference of X is moved from B to A: X is not marked as alive because only A has been examined. *After*: X is not marked as alive because B does not point to X anymore and A was already examined.

2.2.4.3 Object Allocation

The GC module manages object allocations in the J9 VM. Normally the procedure to allocate new objects is protected by a lock so only one thread is allowed to modify the heap at any given time. The J9 JVM implements a technique, called Thread Local Heap (TLH), to preallocate a section of the heap and to assign it to a specific thread. When an allocation is requested, the thread first tries to allocate the object in its TLH (*fast path*), which does not require a lock since the area is reserved for the specific thread. If the *fast path* allocation fails (i.e. the object does not fit in the TLH) the allocation proceeds through the usual *slow* path. TLH areas are seen by the GC as a non-collectable large object. When a TLH is full, the owner thread returns it to the GC and obtains a new TLH area.

2.2.5 Port and Thread Libraries

The J9 JVM is designed to run on a vast number of platforms. An interface has been implemented to abstract the underlying operating system. The JVM implementation makes use of this interface to avoid, as much as possible, platform dependent code. Some components are, however, too low level to make use of the port and thread libraries. In particular the Just-in-Time compiler, which generates native instructions for the platform on which the JVM is executing, remains platform dependent.

2.2.6 Shared Class Cache

Class sharing is a feature introduced in the IBM JVM in Java 5, to share memory between JVMs. The Shared Class Cache (SCC) is designed to reduce startup time of an application by storing information that does not change between different runs of the JVM, such as loaded classes and Just-in-Time compilation data [10]. The SCC can be assigned a name so a specific application can be associated to a dedicated SCC. The memory available to each SCC is in fact limited. The SCC is kept in memory for optimal performance, but in the case of a persistent SCC (default), it can be saved to the filesystem to be reused in the future, even after the system has been turned off. Another advantage of storing the SCC in a file, is that it can be moved to a different machine, as long as the platform matches the machine where the SCC was created.

2.2.7 Just-in-Time Compiler (JIT)

Interpreting bytecode instructions is slow compared to natively compiled code where the CPU executes the code directly [22]. This is a common problem, shared with interpreted languages, such as Python [16], Lua [14], JavaScript [13]. Just-in-Time compilation (JIT), also called dynamic compilation, is a popular strategy that partially solves this problem and consists of translating, at runtime, the interpreted instructions to native code. The first published works about JIT dates back to the '60s. McCarthy in a paper from 1960 mentioned a system in which the compilation was fast enough that its output did not need to be stored and reused [34]. In 1966 the University of Michigan Executive System for the IBM 7090 described a system in which the assembler and loader could translate code during execution [19].

Java is not considered an interpreted language since it is compiled to bytecode, however the bytecode instructions are interpreted and the abstraction imposed by the JVM, on top of the OS and the hardware, causes a significant overhead [22]. Testarossa JIT (TRJIT) is the JIT system used in the IBM J9 JVM [38]. At runtime TRJIT translates methods from bytecode to native code, which the JVM can directly execute.

An important limitation of dynamic compilation, compared to static compilation, is the necessity to manage CPU and memory resources. A static compiler operates before the execution of the application and the time it spends performing the compilation is not important in a production environment. TRJIT instead compiles code during runtime and it must compromise

between using all available resources to quickly compile a large number of methods with the highest optimization possible, and keeping the usage of resources low to allow the application to perform its work.

TRJIT minimizes the amount of time spent compiling code, while maximizing the overall gain in performance, by compiling only the methods used often (i.e. selective compilation). The JVM keeps track of the number of times methods are invoked by keeping a counter for each method; when a method is called more than a specified threshold, the JVM requests TRJIT to compile it. TRJIT compilation is asynchronous and multithreaded: requests to compile are added to a priority queue to be later retrieved by compilation threads and the interpreter does not have to wait until compilation is completed (Figure 2.5). Methods with loops and without loops are treated differently. In the rest of the thesis *count* will refer to the threshold value for methods without loops and *bcount* will refer to the threshold value for methods with loops. The keywords *count* and *bcount* are the names of the options for the threshold values that can be set when launching a Java application (e.g. `java -Xjit:count=1000,bcount=500 ...`). The advantage of selective compilation is not only in CPU time but also in a more efficient memory usage since methods rarely used or never used are never compiled [37].

To further optimize how resources are used, TRJIT applies different optimizations when compiling methods, depending on how often they are executed. Methods invoked often are recompiled with more effective, but also more time consuming, optimizations. This approach is referred to as multi-level

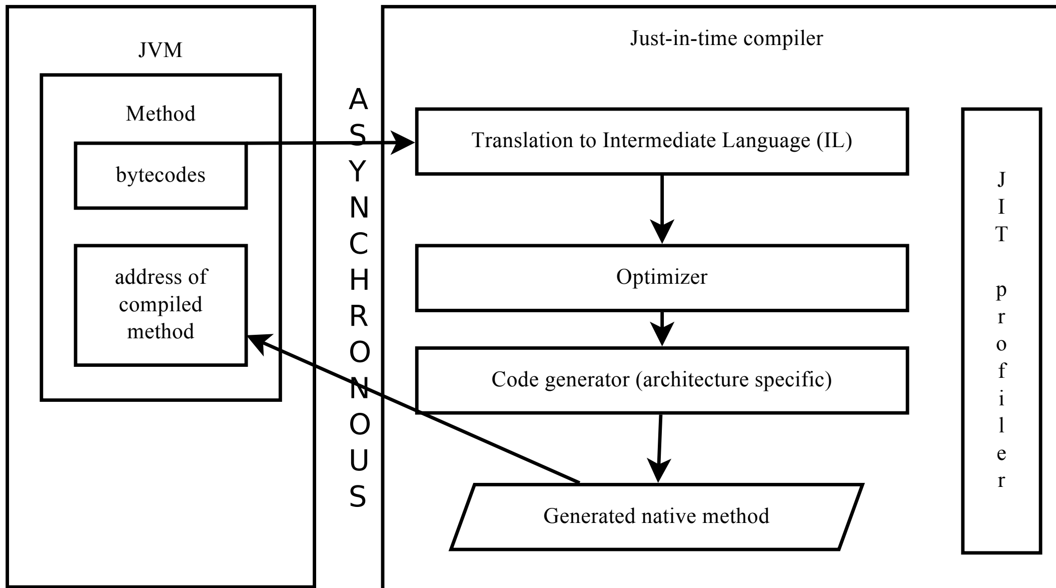

Figure 2.5: Testarossa JIT (TRJIT).

compilation. The levels at which methods can be compiled in TRJIT are, from the lowest level of optimization to the highest: *cold*, *warm*, *hot*, *very hot*, *scorching* [7]. The highest levels require a profiling phase, performed by the JIT profiler. JIT profiling, performed by the code generated by TRJIT, collects more information than the profiling performed by the interpreter, however, because of the overhead, it is applied sparingly only to the hottest methods in the JVM.

While static compilation is not possible in Java, because of the dynamic nature of the language, the advantage of compiling before execution can be partially achieved using the Shared Class Cache feature. TRJIT can save compiled methods in a relocatable format in the SCC when an application is executed for the first time. When the same application is executed again,

the TRJIT can retrieve and prepare the stored compiled methods for execution with considerably less work compared to a full compilation. The JVM performs linking at runtime and that information cannot be saved in the SCC. Therefore the code included in the SCC have all references to methods, classes and fields unresolved, and additional information are included so the TRJIT can patch the code when the AOT code is loaded.

TRJIT is a very advanced JIT system and performs a large number of optimizations to generate high performing native code. Some of the implemented optimizations are listed in Table 2.1.

2.2.8 MicroJIT

MicroJIT is a lightweight JIT system with a single pass compiler, originally designed for the Java Mobile Edition (J2ME) [20]. Mobile platforms, when MicroJIT was implemented (i.e. 2003), had extremely limited hardware capabilities. Therefore it was necessary to design a JIT system that could run efficiently with little resources available. CPU time was scarce, with often only a single hardware thread available, so compilation time had to be fast. MicroJIT uses a template based approach: each bytecode and each accessory operation (e.g. check of Java stack size), is associated with a block of predefined native code. MicroJIT, to compile a method, copies the templates one after the other.

This approach is very rigid and makes optimizations hard to implement. MicroJIT performs only a few optimizations compared to the more advanced

optimization	*description*
Inlining	Replace the call of a method with the method's body.
Cold block outlining	Keep cold code in a separate block to avoid jumps and to use the CPU cache more efficiently.
Value propagation	Track variable values and replace them with constant values where appropriate.
Block hoisting	Move code outside a loop if possible.
Loop unroller	Reduce the number of iterations by duplicating the code in the loop and removing the loop itself.
Asynchronous check removal	Remove unnecessary checks.
Copy propagation	Replace occurrences of direct assignment targets with their values.
Loop versioning	Optimize loops selecting a path that does not perform bounds checking if appropriate.
Common subexpression elimination	Remove duplicate expressions that would result in the same value.
Partial redundant elimination	Remove expressions that are redundant on some, but not all, execution paths.
Optimal store placement	Determine how to store a variable in memory (main memory, stack, registers).
Simplifier	Replace complex expressions with equivalent simpler expressions.
Escape analysis	Determine how to store objects depending on how they are accessed.
Dead tree removal	Remove code that does not affect result.
Switch analysis	Reorganize branch conditions.
Redundant monitor elimination	Remove unnecessary monitor entry (i.e. reentrancy).
Devirtualization	Replace virtual calls with nonvirtual or static calls.
Partial inlining	Inline only the hot part of a method.
Lock coarsening	Merge multiple locks into a single lock.
Live range reduction	Use the information available on the range variable to simplify a loop.
Idiom recognition	Replace recurrent bytecode sequences to reduce stack usage.

Table 2.1: Optimization implemented in Testarossa JIT [36].

TRJIT: method inlining, common expression detection and optimized register assignment.

Memory footprint was also a very scarce resource. MicroJIT uses a limited amount of memory for the compiled code (512KB, 524288 bytes) and can delete already compiled code to make space for new compilation. This feature has little value on modern machines where 512KB represent a negligible fraction of the available memory and where applications normally use orders of magnitude more memory.

MicroJIT was designed with another useful feature to keep the resources it uses at a minimum: native methods operate directly on the Java stack. Transitions from interpreted to generated code, and vice versa, are almost immediate.

MicroJIT was not further developed after Java 6 and supports only 32-bit architectures (x86, ARM, PowerPC, MIPS).

MicroJIT is a separate module with many points of interaction with the rest of the JVM. The interface can be divided into the following parts: initialization, compilation, invocation, returning, execution.

2.2.8.1 MicroJIT-JVM Interface: Initialization

The MicroJIT module is compiled into two libraries, a *normal* library and a larger *debug* library. The choice of which library must be loaded depends on the options specified when launching the Java program: if debug options are specified, the larger *debug* version is loaded.

During initialization, MicroJIT code, is also responsible for:

- Initialization of the method structures and the counters used by the interpreter to determine when methods should be compiled.
- Starting a thread that monitors execution and anticipate the compilation of the method currently active.

2.2.8.2 MicroJIT-JVM Interface: Compilation

The interpreter is responsible for counting the number of times a method has been invoked and to trigger the compilation with MicroJIT. MicroJIT, during compilation, interacts with the JVM to retrieve the following information necessary to compile a method:

- The body of the method (i.e. bytecodes)
- Arguments and local variables
- The type of the method (e.g. static, final, native, synchronized)
- Class details (e.g. lock variable offset)

At the end of a successful compilation, the MicroJIT updates the method structure with the address of the compiled method. If the compilation failed, the method is flagged as such and the method will continue to be interpreted. MicroJIT compilation is synchronous, the compilation is performed by the thread that tried to invoke the method in the first place. The compilation is

also sequential, with a single lock that allows only a single thread to compile a method at any given time.

2.2.8.3 MicroJIT-JVM Interface: Invocation

After a method has been compiled successfully by the MicroJIT, the interpreter will directly invoke it instead of interpreting its bytecode instructions. To pass execution to the native method, a context switch to the Java stack is necessary. The interpreter executes on the native stack while the code compiled by MicroJIT expects to be executed directly on the Java stack. MicroJIT generates native code that manages the Java stack from the initial invocation to when the method returns. While the Java stack is always in a consistent state when returning execution to the interpreter, native code simplifies execution and does not keep the Java stack consistent after every block of instructions representing a bytecode. With this approach MicroJIT can significantly optimize the body of a method to the point of never creating the Java stack frame if a method is very small (example: Figure 2.6).

2.2.8.4 MicroJIT-JVM Interface: Returning to the Interpreter

The execution of methods compiled with MicroJIT return to the interpreter in the following cases:

1. The native method has completed successfully. The result is pushed onto the Java stack frame of the calling method.

```
static public int increment8(int i) {
        i += 8
        return i
}

; native instruction
add EAX, 8
```

Figure 2.6: The body of this trivial method is compiled by MicroJIT to a single instruction. There is no need to create a Java frame or to check the size of the Java stack because the variable i is stored in a register and eventually is directly pushed onto the Java stack frame of the calling method.

2. The native method has failed and throws a Java exception. The execution jumps from the native method to a specific function in the JVM that manages exceptions.

3. MicroJIT could not generate native code to execute a complex operation and the execution must jump to:

    - The interpreter, which will interpret the problematic bytecode.
    - A function in the JVM. This case is similar to the previous one since the JVM is not aware of the MicroJIT and the execution from helper functions will eventually jump to the interpreter.

The latter case is implemented by the MicroJIT using a specific technique that *tricks* the interpreter into returning execution to the native code. The technique, illustrated in Figure 2.7, shows the case where execution returns to the interpreter because MicroJIT could not generate the native code for a

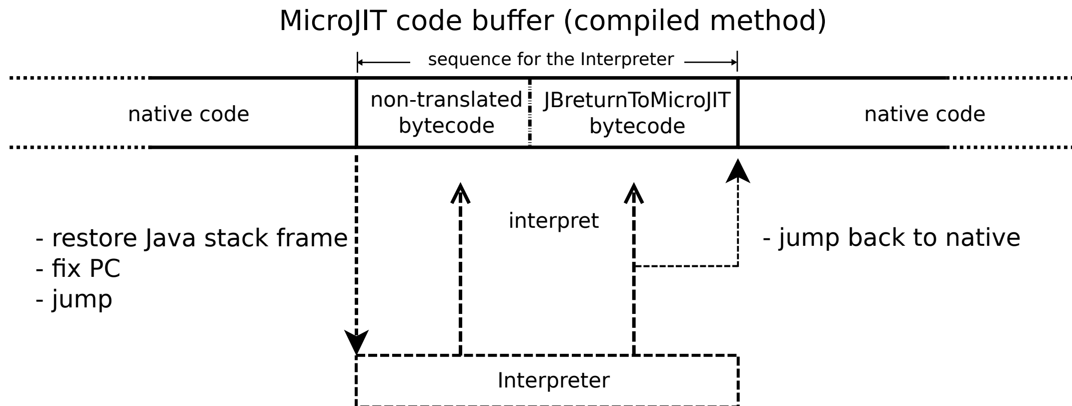

Figure 2.7: Procedure that allows native code, generated by MicroJIT, to *jump* to the interpreter to perform a complex operation and to have the interpreter *jump* back to the native code when finished.

specific bytecode. The case in which the execution must jump to a function in the JVM is very similar and in that case only the JBreturnToMicroJIT bytecode is required.

During compilation, MicroJIT copies the single bytecode that cannot translate to a section of the code buffer. The JBreturnToMicroJIT bytecode is added immediately after the problematic bytecode.

During the execution of the compiled method, just before reaching the section with the bytecode in the code cache:

1. The Java stack is restored. This operation is necessary every time execution is passed to the interpreter.

2. The program counter (PC) is modified to point to the copy of the bytecode inside the code cache.

3. Execution jumps to the interpreter.

4. The interpreter executes the first bytecode, the one that MicroJIT could not translate.

5. The interpreter executes the second bytecode, JBreturnToMicroJIT, and execution jumps back the native method.

The interpreter needs only two modifications to make the technique work:

- JBreturnToMicroJIT must be implemented and, when encountered by the interpreter, execution must jump to the address following the bytecode itself, which contains native instructions.

- The procedure that walks the Java stack must be modified to use the address of the original bytecode. The address itself is stored in the JBreturnToMicroJIT bytecode during compilation. This is necessary when a GC is triggered because the Java stack does not include information on its fields and the GC examines the bytecodes of the method to find all the references to the objects and to generate the root set. If the program counter points to a copy of a single bytecode in the code cache of MicroJIT, GC will start reading native machine instructions treating them as bytecodes and the JVM will fail.

2.2.8.5 MicroJIT-JVM Interface: Execution

The native code generated by the MicroJIT, during its execution, interacts with the JVM, either temporarily jumping to the interpreter with the technique seen in the previous section, or, it can call a JVM function that will

directly return without the involvement of the interpreter. The reasons why a compiled method needs to interact with the other parts of the JVM are listed below:

- **Stack Overflow** - When a method is invoked, it is necessary to check the size of the Java stack. If it is not large enough to contain the new frame for the method, the Java stack must be re-sized. This is a complex operation for which the MicroJIT does not generate code, however MicroJIT generates a check for the Java stack size to avoid calling the function in the JVM if avoidable.

- **Asynchronous check** - Java threads yield execution on a voluntary basis. Each time a method is entered the thread checks if it has to suspend its execution to allow an important task to proceed (e.g. GC).

- **Write Barrier** - When a reference in an object is modified, the memory manager system must be notified.

- **Object Allocation** - MicroJIT generates code to try TLH allocation (*fast path*) of objects and arrays. If the allocation fails, a function in the memory management (*slow path*) is called.

- **Synchronization** - MicroJIT generates code to try to acquire a flat lock. If it fails the JVM function that can manage the more complex cases (e.g. inflating the lock) is called instead.

2.3 Benchmarks

Benchmarks are applications used to evaluate the performance of a system based on one or more metrics. Using the same benchmark to measure the performance of different systems it is possible to determine if one is better than others, according to the selected metrics.

In this thesis, the system to evaluate is the J9 JVM with and without the changes implemented. Benchmarks are used to determine if the modified JVM behaves correctly, other than the obvious case when the JVM crashes. Some of the benchmarks verify that the results returned are correct before returning the measured values (e.g. timing).

There are a number of metrics useful to evaluate the JVM performance. Depending on the field of application some metrics can be more relevant than others. Changes in the JVM can result in an improvement according to a metric but a regression according to another metric. In these cases the evaluation must be based on a tradeoff.

The metrics considered in this project are execution time, throughput and memory usage.

- Execution time - the time spent by the JVM to execute the Java program.
- Throughput - the rate at which a program works. For web applications it is often expressed in requests per second.
- Memory footprint - the memory allocated to a process. Only the mem-

ory allocated in the RAM. In the Linux memory management terms, this value is called the Resident Set Size (RSS).

The following list describes the benchmarks used. The metrics returned by each benchmark are listed in between parenthesis after the name.

- **DaCapo-Batik** (execution time) - DaCapo-Batik is part of the DaCapo suite [3]. Apache Batik is an open source application which converts images from the Scalable Vector Graphics (SVG) to the Portable Network Graphics (PNG) format.

- **Eclipse Startup** (execution time) - Eclipse C/C++ Development Tooling (Eclipse CDT [4]) can be set to run in headless mode, to be executed in the server environment from the terminal, and in debug mode, to print the time required to load Eclipse CDT itself and the default plug-ins. Eclipse is also used as a core block to build other applications (e.g. IBM Notes) [11].

- **WebSphere Liberty** (execution time, memory footprint) - IBM WebSphere Liberty, which is part of WebSphere Application Server, is a Java EE application server. It focuses on providing a high-performance and easy-to-use framework to deploy and manage Java-based web applications. It is designed to easily integrate with other frameworks such as Docker and Chef Puppet [18]. The benchmark measures the time required to load the Liberty framework and the server application,

printing the maximum memory footprint required. Two benchmark applications are considered:

  - **DayTrader** - DayTrader is a benchmark application which simulates an online stock trading system. It was originally developed by IBM and donated to the Apache Geronimo community in 2005. DayTrader is a complex application that uses a large set of Java EE technologies such as Java Servlets, JavaServer Pages (JSP), Java Database Connectivity (JDBC), Java Message Service (JMS), Enterprise JavaBeans (EJBs) and Message-Driven Beans (MDBs) [1].
  - **TradeLite** - TradeLite is, like DayTrader, a simulated share trading Java EE web application. However it is lighter, using only of a subset of the Java technologies used in DayTrader (Java Servlets, JSP and JDBC). TradeLite is often used as a benchmark in WebSphere Application Server performance studies [5].

- **SPECjbb2005** (throughput) - SPECjbb2005 is a benchmark designed to measure the throughput of a Java Virtual Machine. It emulates a 3-tier system in which most of the work is in the 2nd tier (i.e. business logic). Clients are simulated by user threads and the benchmark simulates the database tier with in-memory Java Collection objects. The results are given in *bops* (business operation per seconds).

2.4 Related work

Different solutions have been proposed to make JIT more effective earlier in the execution of the application [27]. Testarossa JIT already implements two of the following solutions: multi-level compilation and Ahead-of-Time compilation.

2.4.0.1 Multi-level Compilation

The multi-level compilation feature of the JIT consists of compiling the same method multiple times with different levels of optimization. The first time, the method is compiled with minimal optimization, but compilation is fast. If the method is executed very often (i.e., the method call counter reaches further thresholds) it is compiled again with a higher level of optimization and so on until the method is compiled at the highest level of optimization. JIT systems that implement this technique can start compiling with lower thresholds and consequently to run compiled code earlier [31].

2.4.0.2 Off-line Profiling

Online profiling consists of collecting execution information at runtime, a technique that can add overhead. In contrast, offline profiling consists of collecting execution data before the program is started and making it available during subsequent executions. Using a combination of off-line and on-line profiling, the JIT system collects profiling data at runtime, when it first

analyzes an application, and it will share it with JVMs running the same application all subsequent runs. The JIT system will use the shared data, if it is available from a previous execution, to quickly determine which methods the JIT should compile. This approach however, depends on the condition that the application is executed in the initial JVM and it only optimizes the collection of profiling information while the compilation must be performed anyway [30].

2.4.0.3 Ahead-of-Time Compilation

Ahead-of-Time compilation (AOT) is another approach to speedup the startup of the application. It consists of compiling the Java bytecode to native machine code before the Java application is executed [25]. The main disadvantage of this technique is the lack of portability, however, in the J9 JVM, this approach has been implemented taking advantage of the compilations performed by the JIT system. Some of the compiled code is shared between JVMs using the class sharing feature. When a new JVM is launched and the code for a method is already found in the Class Sharing cache it is used directly [6].

Chapter 3

Design and Implementation

The main phases of execution of a Java application are startup, warm-up and throughput. These are *loose* definitions, for instance JIT can be active during the startup phase.

- Startup phase - the JVM starts and main classes are loaded.
- Warm-up phase - the JIT is triggered and it improves the performance of the application as more and more methods are compiled.
- Throughput phase - the application has reached the optimal performance level.

Startup, while much shorter than the rest of the application execution, is a critical phase. Users want applications to respond quickly with minimal delay, while on the server side a low startup time allows an application to be ready to reply to requests sooner, providing a better service. Startup of Java

applications, which is inherently slow compared to statically compiled code, is a topic of interest to find new ways to reduce the startup time and reach the throughput phase sooner.

TRJIT is successfully used to improve performance of Java applications, however the main focus is on throughput performance and the choices made in that direction are not always ideal to make the applications start faster. The default settings cause methods to be queued for compilation after they have been interpreted 3000 times. Methods could be interpreted considerably more times than the default threshold, since the compilations are not guaranteed to start immediately after they are queued and compilations themselves are not instantaneous.

The main assumption of this project is that the delay and the duration of the compilations performed by TRJIT are high enough that including a lightweight JIT system in the JVM, to compile methods before they are compiled by the default TRJIT, can improve startup time. MicroJIT is a good candidate since it is a small JIT system originally designed to run on mobile devices with limited resources.

The project also investigates how the proposed solution compares to two alternative settings of the JVM, which already improve startup times: `-Xquickstart` and `-Xshareclasses`.

The option `Xquickstart` was introduced specifically to tune TRJIT for better startup time. It disables profiling in the interpreter, reduces count thresholds to 250 (*count*) and 1000 (*bcount*), and forces TRJIT to compile with

fewer optimizations. `Xquickstart` is very effective in reducing the startup time however, it may have a negative effect on the throughput of an application. Therefore it is not an ideal solution when the application is supposed to start quickly but also to run for a long period of time (e.g. server applications).

The Shared Class Cache (SCC) can be enabled with the option `Xshareclasses`. SCC significantly improves the startup time when enabled, however it is not enabled by default and in some environments, where using SCC is not trivial, is rarely enabled (e.g. virtualization in cloud environments).

3.1 Requirements

The project has been planned considering the following requirements:

1. Any change to the JVM must not violate the Java Virtual Machine Specification. This is a hard requirement when working on the J9 JVM and it guarantees that a developer will not be required to apply any change to the application depending on which JVM is used. The only differences when using different JVMs are performance and memory footprint.

2. MicroJIT and TRJIT must be enabled at the same time.

3. The interface used by MicroJIT to interact with the existing JVM should reuse what is already available in the interface already existing

for TRJIT, keeping changes at a minimum.

4. Any modification to the JVM outside the MicroJIT module should be kept at a minimum to reduce the risk of introducing new bugs. If a change is necessary it must be tested in all cases. For example the JVM with MicroJIT implemented must run correctly if the MicroJIT is not enabled.

5. Overhead must be kept at a minimum. Ideally the JVM with MicroJIT disabled should perform like the unmodified JVM. Performance should be measured using multiple metrics (i.e. execution time and memory footprint).

In addition of the listed requirement, the project is constrained by the availability of MicroJIT only on 32-bit platforms. While the 32-bit architecture is obsolete in server environments and is being phased out in desktop environments, the results restricted to a 32-bit architecture will still be valid and, if positive, MicroJIT can be ported to x86-64 and ppc64 architectures at a later date. IBM J9 JVM does support the 32-bit architecture.

3.2 Strategy

In this section the approach to the project is described from a high level point of view, with particular emphasis on the motivations behind the tests that are performed during evaluation.

The project is divided into 3 phases:

1. Porting the MicroJIT to the J9 JVM for Java 8.
2. The JVM is modified to support enabling both MicroJIT and TRJIT.
3. Different approaches are investigated to improve the code generated by MicroJIT

3.2.1 Phase 1 - Port MicroJIT to Java 8

The first task consists in porting MicroJIT from the J9 JVM for Java 6 Mobile Edition (Java6) to the J9 JVM for Java 8 Standard Edition (Java8). The implementation of the JVM was changed between Java6 and Java8. In Java6 the interpreter was implemented in Builder, a proprietary programming language derived from Smalltalk. The interpreter in Java8 is written mainly in C/C++ with only some parts in Builder. Therefore the interface used by MicroJIT must be reimplemented for the most part.
The first phase is concluded with a series of tests to benchmark MicroJIT and the code that it generates. In particular we are interested in the compilation time and how code generated by MicroJIT compares to the interpreter and to the code generated by TRJIT.

3.2.1.1 MicroJIT Compilation Times

The hypothesis that MicroJIT can improve startup time is based on the assumption that MicroJIT compilations are much faster than TRJIT compi-

lations.

Therefore the JVM is instrumented to collect this information. MicroJIT compilation is synchronous and measuring the time of the function called to compile a method can be performed from the interpreter where the compilation is triggered. The result is printed directly to the terminal to be further analyzed and compared to compilation times from TRJIT.

3.2.1.2 Quality of the Native Code Generated by MicroJIT

MicroJIT is expected to generate native code that performs significantly better than the interpreter but slower than the native code generated by TRJIT. The difference in performance is important to determine if MicroJIT is a feasible solution to improve startup time. The available benchmarks are used to measure the execution time of the JVM in different configurations.

3.2.1.3 Optimal *count* and *bcount* Values for MicroJIT

To maximize the usefulness of MicroJIT we aim to trigger compilation as soon as possible. However compiling too soon is not desirable for the following reasons:

- Java relies on dynamic loading, therefore a method could include unresolved references. For example, a method could require access to a static field of a different class that has not been loaded yet and that will be loaded only at the first execution of the method itself. When MicroJIT encounters an unresolved reference it generates less optimized

native code (i.e. additional branches to the interpreter, no inlining of methods with unresolved references).

- Using a low count increases the number of methods to compile. The extreme case of using a count of zero would effectively cause all the methods to be compiled even if used only once, resulting in wasted resources (both CPU time and memory footprint).

The JVM is benchmarked with different values of *count* and *bcount* to determine their optimal values.

3.2.2 Phase 2 - Enable MicroJIT with TRJIT

In the second phase the JVM is modified to enable MicroJIT to work with TRJIT enabled at the same time. The J9 JVM currently does not support multiple JIT systems, therefore it must be modified in the initialization of the JVM and the interpreter must be changed to implement this feature. The end of this phase consists of running the available benchmarks in the final configuration. The results of benchmarks and profiling are helpful to determine if using MicroJIT is a good proposition. The JVM is also analyzed to identify bottlenecks in the MicroJIT and the code it generates and, depending on the difficulty of the implementation, optimization are proposed and implemented. Changes that are more interesting from a research point of view will be prioritized while leaving straightforward implementations that will improve performance in a predictable way with lower priority.

3.2.2.1 Optimal *count* and *bcount* Values for TRJIT

The default values of *count* and *bcount* for TRJIT are used when benchmarking the JVM. However, if MicroJIT generates code which performs better than the interpreter, the equilibrium with TRJIT will change as well (i.e. methods will be queued for TRJIT compilation earlier). Therefore the final JVM is evaluated also varying the *count* and *bcount* values for TRJIT.

3.2.2.2 Throughput of the JVM with MicroJIT

The option `Xquickstart` is effective to reduce the startup time of a Java application, however it is known to penalize its throughput. Enabling MicroJIT in the JVM reduces the throughput because the code generated by MicroJIT does not profile the execution like the interpreter does. It is important to quantify the penalty in throughput for both cases (i.e. the existing approach with `Xquickstart` and the solution with MicroJIT) and to compare them to the optimal throughput obtained by the JVM with default settings.

3.2.3 Phase 3 - Improving MicroJIT Compilation

In the third and final phase MicroJIT is modified to improve the compilation time and the quality (i.e. execution time) of the code it generates.

3.3 Implementation

This section focuses on some of the implementation details.

3.3.1 Phase 1 - Port MicroJIT to Java 8

MicroJIT is already a working JIT system in Java6 and most of the work consisted of adapting the interface to Java8 without modifying its logic. Reimplementing the interface used by MicroJIT requires a considerable amount of time and investigation on how the JVM is structured and how it works internally. While this work is essential to prepare the JVM to be evaluated and for the second phase of the project, the changes are not significant from a research point of view and they are listed here with only a brief description.

- A new project for MicroJIT was created using Eclipse IDE. The project contains the source code that is then built into the MicroJIT libraries. MicroJIT is divided into a library for standard execution and a library that enables debugging during execution to expose details about compilation during execution.

- MicroJIT supports x86, ARM, PowerPC and MIPS in JavaME 6. MIPS was disabled when ported to Java8.

- The initialization of the JVM was modified to load MicroJIT libraries if the option `-Xmjit` is provided to the `java` command.

- The internal representation of classes and Java objects, in the JVM, has been modified since Java6. In particular the structure for objects has been optimized to reduce footprint by removing from the structure itself information that could be obtained in different ways (e.g.

the age of the object) or could be made optional (e.g. lock variable). MicroJIT compilation had to be modified to take into account the new structure layout in synchronization and in TLH object allocations (*fast allocation*).

- The signature of some functions in the Port Library and Thread Library have been modified. For example when memory is allocated in Java8 the type of memory must be specified (e.g. memory for JIT code cache).
- The default GC policy in Java8 (GenCon) is not present in Java6 where a mark-sweep-compactor was used instead (i.e. optthruput). The code specific for the default GC in Java6 was disabled.
- J9 JVM for Java 8 modifies the body of methods (bytecodes) during class loading to simplify and optimize common bytecode patterns and merge them to bytecodes used internally. One of these optimizations consists in replacing the bytecode for new object (`new`) followed by the bytecode to duplicate a reference (`dup`) to the special bytecode `newdup`. MicroJIT did not support the new bytecode `newdup`, which had to be implemented from scratch in MicroJIT to generate correct native code.
- The internal bytecode JBreturnToMicroJIT, described in the section 2.2.8.4, had to be reimplemented from scratch in the interpreter since this bytecode is specific for MicroJIT.
- The logic to invoke MicroJIT code was implemented in the interpreter,

duplicating, with minimal changes, the interface already used for TR-JIT.

- MicroJIT starts an additional thread during initialization to sample the application during execution and anticipate the compilation of methods in which the interpreter spends a significant amount of time. The sampler thread was disabled in MicroJIT, for two reasons:
  - MicroJIT is expected to compile methods early so the sampler thread would not help significantly.
  - The JVM is going to be modified to enable TRJIT with MicroJIT. TRJIT also has a sampler thread with the same purpose, making the sampler thread from the MicroJIT redundant.

3.3.2 Phase 2 - Enable MicroJIT with TRJIT

The internal structure for the methods (J9Method) in the JVM was modified to support MicroJIT. The information relevant for the JIT systems in the J9Method structures are stored in the following three fields:

- `Flags` - One of the flags in the J9Method structure is used to determine if a method has been compiled by TRJIT. If this is the case there is no need to proceed with updating the counter of the number of invocations and the native version of the method can be called directly.
- `Extra field 1` - The field is initialized, during class loading, with the

initial threshold value for the method. The counter is decremented at each method invocation and when the counter reaches zero the method is compiled. Once the counter is not needed anymore (i.e. TRJIT has compiled or has tried to compile the method) the field will contain the address of the compiled method or a unique value that indicates the compilation failure. It is possible to distinguish the special value from an address because alignment guarantees that the two rightmost bits in an address are zero.

- `Extra field 2` - This field has been added specifically for the MicroJIT and stores the difference between the threshold value for TRJIT and the threshold value for MicroJIT. This value is calculated and set when the method in question is loaded. The choice of adding an additional field is to minimize the changes in the code and to simplify the implementation.

The flowchart in Figure 3.1 shows how the interpreter was modified to take into consideration both MicroJIT and TRJIT when determining if a method should be compiled and when determining which compiled method should be executed.

In the existing implementation, the first extra field is decremented using the Compare-And-Swap (CAS) atomic instruction. The field could be updated concurrently by different threads during execution. A CAS instruction was used because protecting the field with a lock would be too expensive. Adding

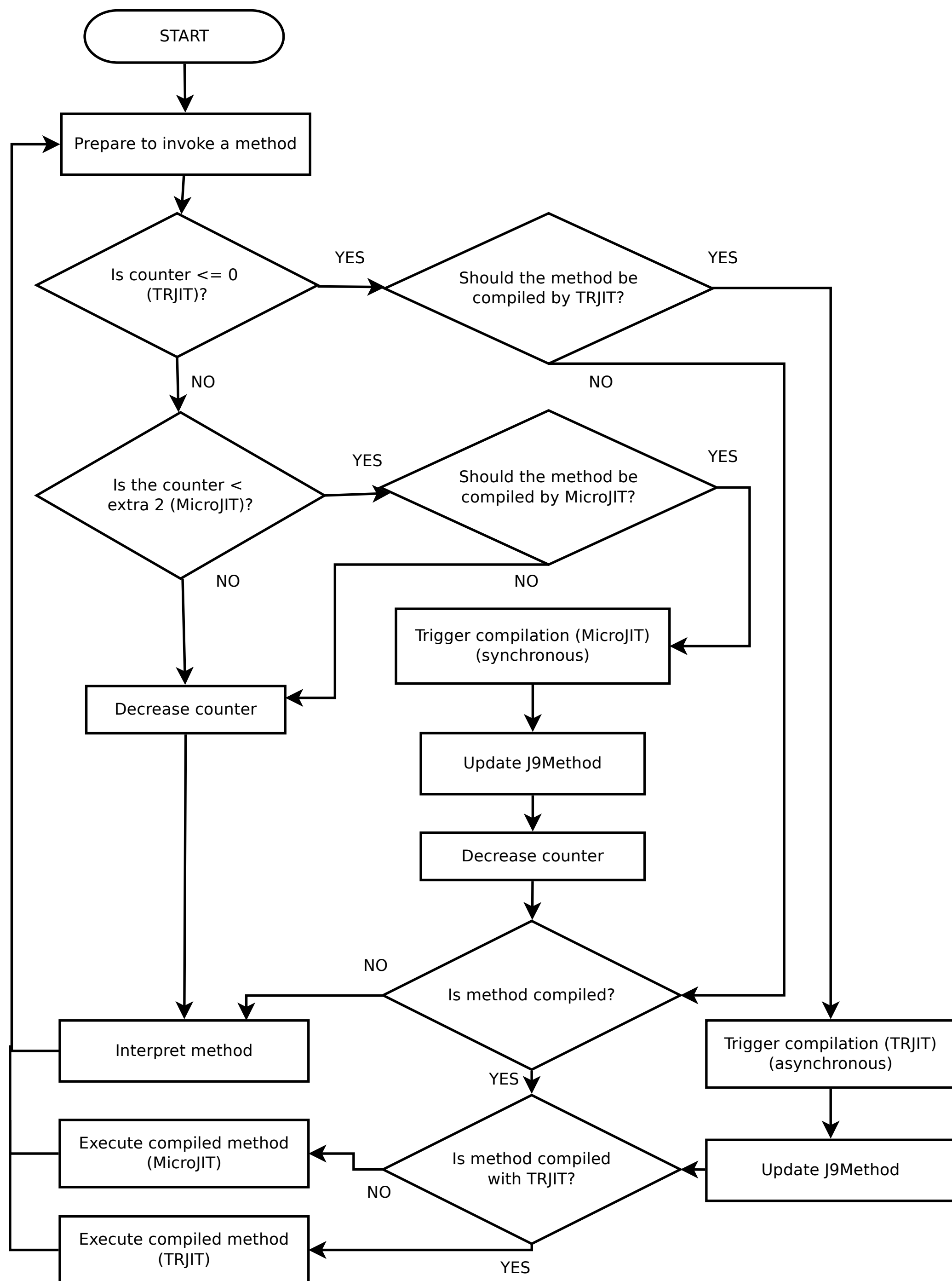

Figure 3.1: Flowchart for Triggering Compilation with MicroJIT and TRJIT.

MicroJIT did not require changing this approach: the counter is decremented in the same way in the new implementation, with the only difference being that the counter is now compared to two values: one for MicroJIT (`extra field 2`) and one for TRJIT (`extra field 1`). MicroJIT compilation occurs before TRJIT compilation for each given method. After a method is compiled with MicroJIT the counter keeps being decremented until `extra field 1` reaches zero, at which point the compilation with TRJIT is triggered and the counter is not updated anymore.

The address of the methods compiled with MicroJIT is stored in the second extra field, while the address of the methods compiled with TRJIT is stored in the first extra field (replacing the counter). Keeping the two fields separated was chosen to keep the implementation simple. Using the same field to store methods compiled with either JIT systems would have required to modify TRJIT to make it aware that the extra field could be written by the MicroJIT. This solution is not optimal from the point of view of the memory footprint since it adds 4 bytes for each loaded method, regardless if it is ever used. However since the main interest of the project is in the startup time and the memory overhead introduced was considered tolerable for this prototype.

3.3.3 Phase 3 - Improve MicroJIT

In the last phase of the project, the work has focused on identifying the areas in the MicroJIT code and in the code generated by MicroJIT that could benefit from optimizations.

3.3.3.1 Bytecodes that MicroJIT cannot compile

When MicroJIT encounters, during compilation, a bytecode that cannot be translated, it generates native code to temporarily return execution to the interpreter, as described in Section 2.2.8.4. Returning to the interpreter is an expensive operation that requires restoring the Java stack frame and two context switches between the Java stack and the native stack [29].

Therefore there is potential of improving the performance of code generated by the MicroJIT. Since implementing all the missing bytecode requires a considerable effort, the most expensive bytecodes when executing DaCapo-batik (i.e. `invokeVirtual`), has been selected to be implemented and the results can be used to estimate the effect of implementing other bytecodes.

The optimization implements a *fast path* for `invokeVirtual` that can be selected only if both the caller method and the called methods have been compiled by the MicroJIT and if the caller method (native version) is being executed. The implementation is divided into two parts:

1. The caller method invokes a virtual method without passing through the interpreter.

2. When the called method returns, the execution will continue in the caller method without passing through the interpreter.

The latter part is trivial to implement and very effective since the *fast path* is selected also when the called method is not virtual (i.e. static). When a method compiled by MicroJIT returns, if the program counter points to a

JBreturnToMicroJIT bytecode, the execution jumps directly to the address following the bytecode, which, by design, points to the native instruction where the execution must continue in the caller method. Checking if the method called is a method compiled with MicroJIT is not necessary since the bytecode JBreturnToMicroJIT is used only by MicroJIT so it is sufficient to guarantee that the caller is indeed a method compiled by MicroJIT.

The former part is more complex. The flowchart in Figure 3.2 shows the operations, implemented in assembly code, required to invoke a virtual method. A series of checks is necessary to determine if the execution should return to the interpreter:

- The called method is not compiled by the MicroJIT.
- The method has been already compiled by TRJIT. TRJIT generates faster native code compared to MicroJIT so, if available, it should be executed instead.
- The value of the counter indicates that the interpreter should trigger the compilation with TRJIT.

If the execution proceeds without returning to the interpreter, the counter for the method must be increased. The Compare-And-Swap (CAS) instruction has been used to avoid locking, in the same way it is used by the interpreter.

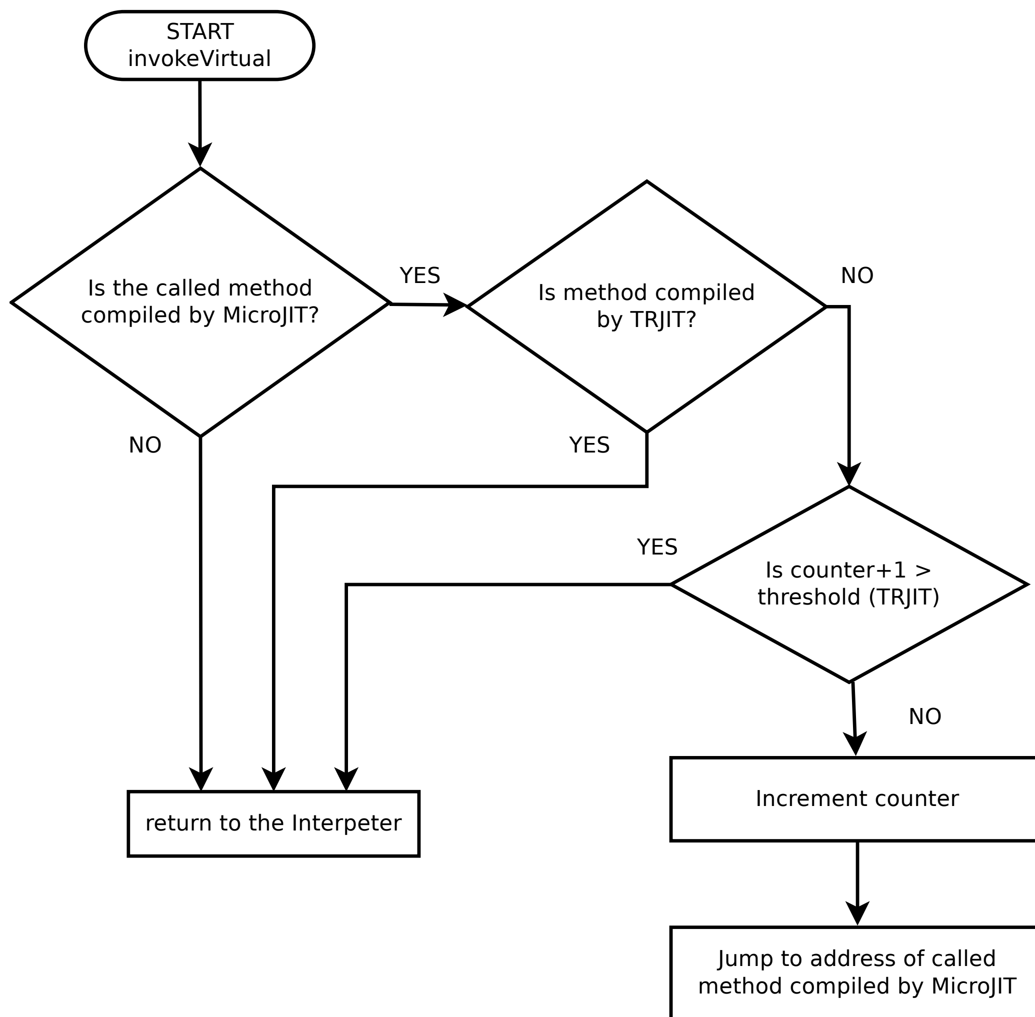

Figure 3.2: Flowchart for invokeVirtual Implementation in MicroJIT.

3.3.4 Write Barrier for GenCon GC Policy

GenCon is a generational GC with a concurrent mark phase. Write barriers (2.2.4.2) are required when a reference is modified in an object. MicroJIT generates code for the write barrier to call the corresponding function in the GC module of the JVM. This operation does not require a return to the interpreter because it is implemented with a call instruction instead of a jump instruction. However, even if the cost of the call is cheaper when compared to the cost of temporarily returning to the interpreter, completely avoiding it could result in better performance.

In the context of a modified reference in an object:

- The *destination* is the object that contains the reference that has been updated.
- The *source* is the object to which the up-to-date reference points to.

In `gencon` calling the write barrier is necessary only if all the conditions regarding the concurrent mark phase are met, or all conditions regarding the remembered set are met:

- Concurrent mark phase - the write barrier is necessary only if:
    1. The concurrent mark phase is active.
    2. The destination object is old. In `gencon` only the tenure space is marked concurrently.

- Remembered set - the write barrier is necessary only if:

    1. The destination object is old. The partial GC collects only the nursery space. There is no partial GC that collects only the tenure space so there is no remembered set for the tenure space.
    2. The source object is not old. The remembered set is not affected if both the destination and source objects are in the tenure space.
    3. The destination object is not already in the remembered set.

MicroJIT compilation was modified to generate native code to check if the destination object is old (condition common for the two cases) and, only in this case, to call the function to notify the GC of the change of reference.

3.4 Development Platform

Given the requirements specified in Section 3.1 and the hardware available, it was decided to port MicroJIT to the Intel x86 platform. Two servers with Intel processors, with equivalent hardware configurations, were available for the project. GMC is the server dedicated to compilation and testing and Dodge is the server dedicated to benchmarking:

- CPUs: 4 Intel Xeon E7520, 1.8 GHz
- RAM: 64 GB DDR3 (16 GB per CPU socket)
- OS: CentOS 6

Eclipse IDE, that support C and C++ programming languages, was used for development. IBM provides to the CASA project plug-ins for Eclipse necessary to build the J9 JVM, including specific software to execute Builder code. The Builder language has only Windows support, therefore the choice to have Windows 7 installed on the main workstation even if the target platform for the project is Linux.

Other software used in the project:

- Compilation: GNU compiler (`gcc`), Java compiler (`javac`), Java disassembler (`javap`)
- Revision control: Mercurial (`hg`)
- Debugging: GNU Project Debugger (`gdb`), J9 JVM dump viewer (`jdmpview`)
- Scripting, testing and benchmark automation: Bash, Go
- Benchmarks: DaCapo benchmark suite, Eclipse CDT, WebSphere: Liberty and SPECjbb2005, as described in Section 2.3.

3.4.1 Benchmarking

Unless specified differently, the results are the arithmetic mean of 30 repetitions of the same test. The error bars in the graphs represent the standard deviation. The Student's t-test is a statistical methods that can help to compare two groups of results when benchmarking the JVM [26]. This test is particularly useful when the difference in the results is small and the standard

deviation overlaps significantly. With the t-test it is possible to determine, with a specific confidence interval, if the tested version of the JVM is statistically better than the baseline.

Chapter 4

Evaluation

The implementation must be tested for correctness. First, all of the applications must terminate correctly. A segmentation fault causes the sudden termination of the JVM and is immediate to detect. A bug in the JVM can however cause the JVM to exit without errors and to return an incorrect result. Many benchmarks validate the output of the application before providing information about the performance, at the end of the execution. DaCapo benchmarks returns the status `PASSED` if the application result is validated successfully and the status `FAILED` if the result is not correct. DaCapo-batik is the only application used as a benchmark from the DaCapo suite, however other applications in the same suite were used to verify the correctness of the JVM during development: `DaCapo-avrora`, `DaCapo-luindex`, `DaCapo-lusearch`, `DaCapo-pmd` and `DaCapo-xalan`.

In the Section 4.1 the JVM with MicroJIT enabled was compared to the

JVM running only with the interpreter. Each implemented optimization was evaluated independently to determine how effective it is and if similar optimization should be implemented in the future to further improve the quality of the code generated by MicroJIT. The Section concludes with an analysis of the optimal values of *count* and *bcount* for the MicroJIT, useful to determine how early MicroJIT should start performing compilations. In the Section 4.2 the JVM with both TRJIT and MicroJIT enabled was evaluated to determine if MicroJIT can improve performances when TRJIT is also enabled. The JVM was tested with the TRJIT in different configuration to determine how MicroJIT compares to the existing solution already implemented in the JVM to improve startup time. The Section concludes with the evaluation of the implementation when considering metrics other than execution time: throughput and memory footprint.

4.1 Evaluation of MicroJIT Implementation

MicroJIT was ported to Java8 in the first phase of the project, and was improved in the third and last phase of the project with two approaches: the number of times the execution from the native code returns to the interpreter was reduced and the number of times the function in the GC module responsible for the write barrier is called was decreased.
In this section the JVM, without any JIT system enabled (interpreter only), is compared to the JVM with MicroJIT active, with and without the im-

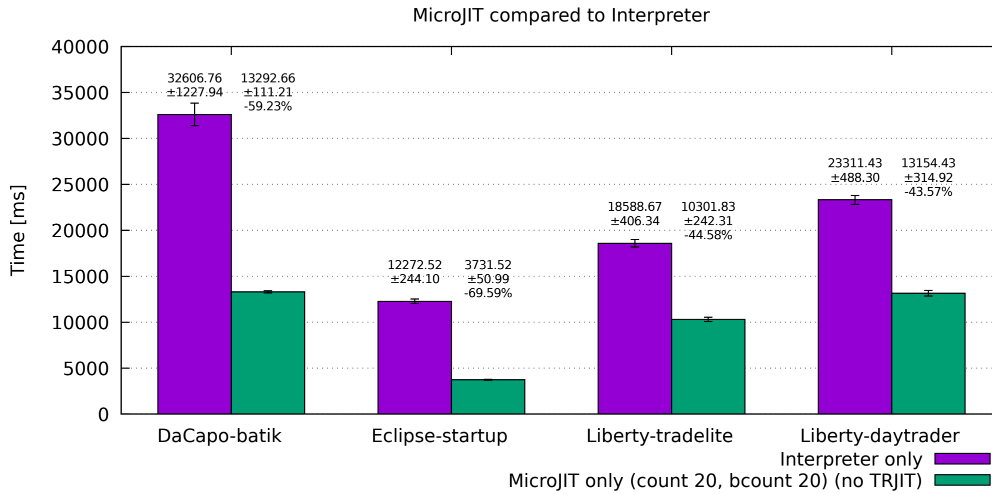

Figure 4.1: Comparison of MicroJIT (with all the optimizations enabled) with the Interpreter.

provements enabled.

Figure 4.1 shows that the MicroJIT, with the improvements previously described, can generate native code that executes much faster than the interpreter. Not having to keep the Java stack up to date and directly using the hardware registers available in the processor, saves a large amount of CPU time, even with the poorly optimized code generated by the MicroJIT.

4.1.1 Temporarily Returning to the Interpreter

The MicroJIT, if unable to generate the native code for a specific operation, generates code to temporarily return to the interpreter. Returning execution to the interpreter is an expensive operation, as described in Subsection 3.3.3.1. So reducing the number of times this happens was considered a

high-priority optimization in the project.

The code generated by the MicroJIT returns to the interpreter because either it requires a call to a specific function in the JVM for a complex operation (e.g. expanding the Java stack), or because the MicroJIT is not able to translate a specific bytecode. The latter case is the most frequent and the amount of work performed by the interpreter is small enough to be easily implemented in the MicroJIT.

4.1.2 Implementation of `invokeVirtual`

In Table 4.1, the bytecodes that are not translated by MicroJIT when executing DaCapo-batik, are listed in order, starting from the bytecode that requires a temporary jump to the interpreter more often. The bytecode `invokeVirtual` required to be interpreted about 25 million times, 5 times more than the second problematic bytecode (i.e. `invokeinterface`) at 5 million. The bytecode `invokevirtual` has therefore been selected to be implemented in the MicroJIT.

This optimization implements the *fast path* to invoke a virtual method as described in Subsection 3.3.3.1. Figure 4.2 shows how the implementation of `invokeVirtual` results in a significant improvement, between 9% and 10% for DaCapo-Batik, Liberty-Tradelite and Liberty-DayTrader. Eclipse Startup gains are not as important (-3.58%) because in Eclipse Startup static methods are invoked more often than virtual methods (Table 4.2).

From the measured improvements it is easy to suggest that more bytecodes

Bytecode	Slow path	Fast path	ratio
JBinvokevirtual	25278938	4911942	-80.57
JBinvokeinterface	5036767	5037118	0.01
JBinvokespecial	3901533	3899517	-0.05
JBinvokestatic	2984004	2982653	-0.05
JBcheckcast	1320657	1319952	-0.05
JBlmul	891034	891034	0.00
JBlushr	855635	855635	0.00
JBfcmpl	751279	751279	0.00
JBfcmpg	682901	682901	0.00
JBdmul	448083	448083	0.00
JBaastore	394402	406165	2.98
JBinstanceof	315386	315386	0.00
JBdcmpl	251986	251986	0.00
JBdcmpg	204573	204573	0.00
JBmonitorenter	163300	163394	0.06
JBmonitorexit	163244	163338	0.06
JBlcmp	130963	130965	0.00
JBgetfield	53965	53973	0.01
JBlshr	45823	45823	0.00
JBputfield	24819	24819	0.00
JBlshl	21307	21469	0.76
JBgetstatic	20523	20505	-0.09
JBnewarray	18606	18789	0.98
JBf2i	12598	12598	0.00
JBldc	7181	7181	0.00
JBputstatic	5338	5340	0.04
JBd2i	5197	5197	0.00
JBlrem	3039	3039	0.00
JBldiv	1868	1868	0.00
JBanewarray	145	148	2.07
JBldcw	3	3	0.00
JBd2l	1	1	0.00

Table 4.1: List of Bytecodes that MicroJIT does not Translate (DaCapo-Batik). Fast-path refers to the JVM in which *invokeVirtual* translation is implemented in MicroJIT

.

Bytecode	Slow path	Fast path	ratio
JBinvokestatic	13078061	13078523	0.00
JBinvokevirtual	8199490	4657159	-43.20
JBinvokespecial	3851877	3852280	0.01
JBinvokeinterface	2392994	2392994	0.00
JBlmul	2161881	2161866	0.00
JBlushr	2021043	2021037	0.00
JBaastore	497662	497732	0.01
JBcheckcast	424887	424936	0.01
JBlcmp	339836	339946	0.03
JBmonitorenter	263494	263547	0.02
JBmonitorexit	263467	263520	0.02
JBgetstatic	187487	187493	0.00
JBfcmpg	115223	115223	0.00
JBlshl	82287	82288	0.00
JBlshr	79318	79318	0.00
JBf2i	77476	77476	0.00
JBinstanceof	67268	67268	0.00
JBfcmpl	40453	40453	0.00
JBdmul	24306	24306	0.00
JBd2i	23041	23041	0.00
JBgetfield	12804	13174	2.89
JBldc	5993	5993	0.00
JBlrem	4652	4652	0.00
JBputfield	3326	3468	4.27
JBldiv	2992	2992	0.00
JBdcmpl	1265	1265	0.00
JBanewarray	929	930	0.11
JBputstatic	567	567	0.00
JBnewarray	525	533	1.52
JBldcw	164	164	0.00
JBdcmpg	9	9	0.00

Table 4.2: List of Bytecodes that MicroJIT does not Translate (Eclipse Startup). Fast-path refers to the JVM in which *invokeVirtual* translation is implemented in MicroJIT

.

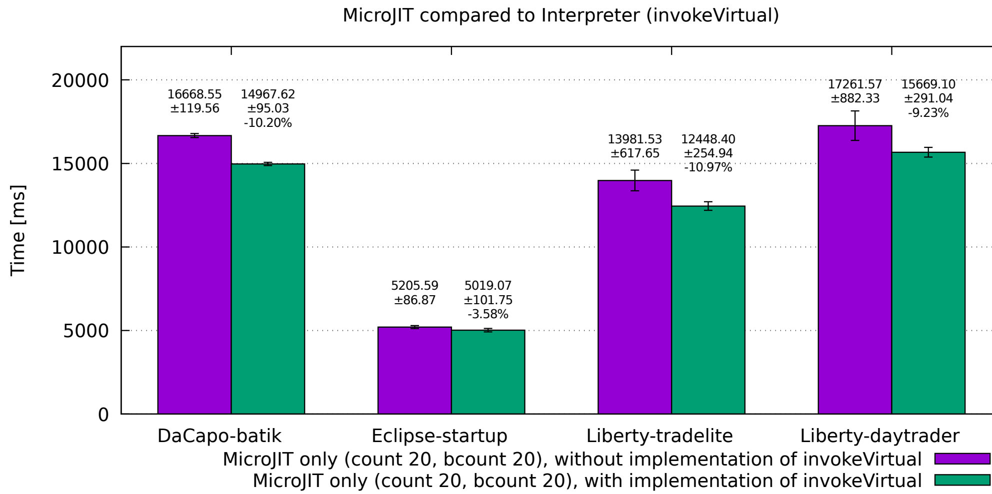

Figure 4.2: Comparison of MicroJIT with and without the Implementation of the Bytecode `invokeVirtual`.

should be implemented if the MicroJIT will be considered for implementation in a production environment. Kent et al previously showed that context switching back and forth between software interpreter and a hardware accelerator, because the bytecode instruction is not supported in hardware, results in a significant penalty. To deal with this issue, they proposed only offloading to the faster hardware when the predicted work to be completed outweighed the context switch penalty [29].

4.1.3 Return of a Method Compiled by MicroJIT

This optimization implements the *fast path* to return from a called method and jump to the correct native instruction in the caller method without requiring a pass through the interpreter, as described in Subsection 3.3.3.1.

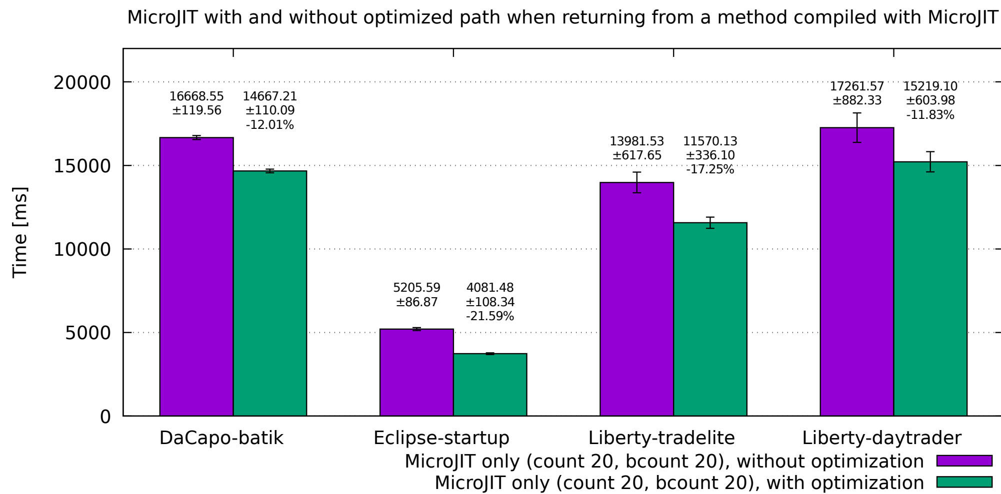

Figure 4.3: Comparison of MicroJIT with and without the optimization that avoids passing the execution to the interpreter when returning from a method compiled with MicroJIT to another method compiled with MicroJIT.

This optimization does not depend on the type of methods that is returning so it is effective also when the called method is not virtual (i.e. static).
The positive results (Figure 4.3) depend also on the fact that the *fast path* adds minimal complexity to the assembly code: it fetches the next bytecode in the program counter, verifies it is a `JBreturnToMicroJIT` and jumps to the native code of the caller method.

4.1.4 Write Barrier Check

This optimization implements the write barrier check as described in Subsection 3.3.4. The optimization is specific for the GC policy `gencon`.
The JVM was instrumented to determine if implementing a check only based

on the age of the destination object is sufficient. Two counters were used to track the number of times the GC function, that must be notified of changes in references, is called, and the age of the destination object. In Table 4.3 the total number of calls (*WriteBarriers*) is considerably higher than the number of calls in which the destination object is old, therefore the write barrier is often not necessary. Eclipse Startup is the best case for this optimization: checking the age of the destination object is enough to avoid 94% of the calls to the GC function.

While the GC activity depends on the application and is not deterministic for multithreaded applications, all new Java objects are created in the nursery space and only after a number of GCs they are copied to the tenure space. The tested applications are short lived and the importance of the age of the destination object to avoid the write barrier is expected, since most objects are young during early execution of the application.

MicroJIT was modified to generate the native code to check if the destination object is in the tenure space and, only in that case, to call the GC function responsible for the write barriers. The results, represented in Figure 4.4, show minimal improvements and in the case of Liberty-Daytrader, given the variability of the tests, the difference is not statistically significant (60% confidence interval).

The small gain is attributed to the relatively low number of avoided calls and to the fact that the called function performs the same check shortly after so each avoided call does not save a significant amount of CPU time.

Application	WriteBarriers	Destination object is old	Ratio
DaCapo Batik	236359	71892	0.30
Eclipse startup	1081297	69988	0.06
Liberty Tradelite	37426	17584	0.47
Liberty Daytrader	39071	17759	0.45

Table 4.3: Write barriers are necessary only if destination object is old. The column *Ratio* indicates, in percentage, how many calls to the GC function are necessary. The values are not averages and are based on a single run of the application.

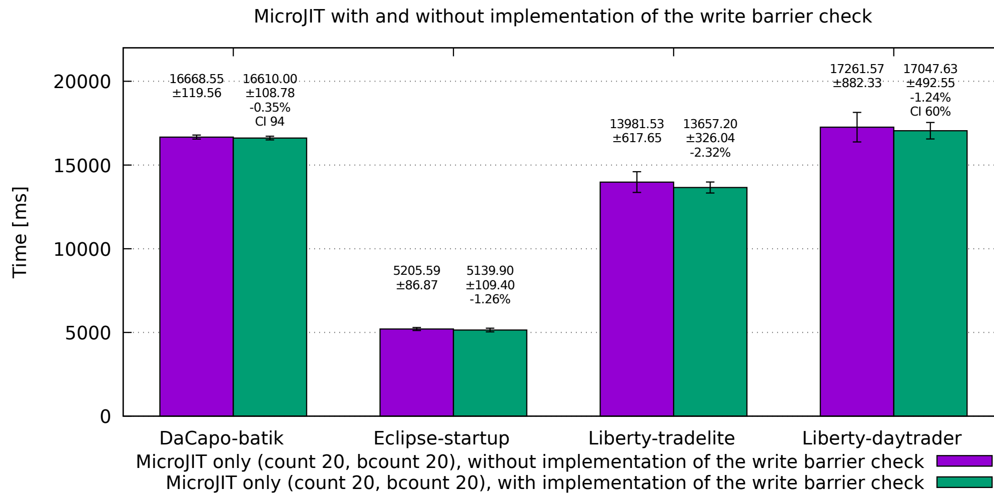

Figure 4.4: Comparison of MicroJIT with and without implementation of the write barrier check.

4.1.5 Optimal *count* and *bcount* Values for MicroJIT

The objective of the project is to use MicroJIT to compile methods as soon as possible, to reduce the number of times methods are interpreted before they are compiled by the main JIT system, TRJIT. The JVM with MicroJIT was tested with different values of the threshold *count* and *bcount* to determine their optimal values; *count* and *bcount* represent respectively the number of times a method, without loops and with loops, is invoked before its compilation is triggered with MicroJIT.

The process is time consuming but fully automated with a *naive* machine learning approach:

1. The JVM is executed 20 times for each entry in a matrix. Each entry corresponds to a combination of *count* and *bcount*. The starting entry is the middle of the matrix. The final value in the entry is the median of the execution times.

2. After each entry is calculated, up to 4 more entries are added to a priority queue. The additional entries are the midpoint between the current entry and the closest entry already computed, or the edge of the matrix, in one of the 4 directions: up, down, right, left.

3. Each entry, when added to the priority queue, is assigned a *forecast* execution time based on the execution time of the current entry. The priority queue is sorted by the *forecast* value.

4. The procedures loop until the program is stopped or until all entries are assigned the actual values.

The results have been visualized assigning a colour to each entry based on the execution time. This approach (Figure 4.5) gives an intuitive way to read the matrix and to identify the best values of *count* and *bcount*.
The threshold for the *bcount* value appears to be more important than the threshold for the *count* value, that is equivalent to say that methods with loops benefit more from the JIT compilation compared to methods without loops. This can be explained considering that for each invocation of a method without loops each bytecode is executed at most once while for each invocation of a method with loops each bytecode can be executed more than once, making compilation more effective.
The threshold value 20 for both *count* and *bcount* was selected as it shows as one of the optimal cases.

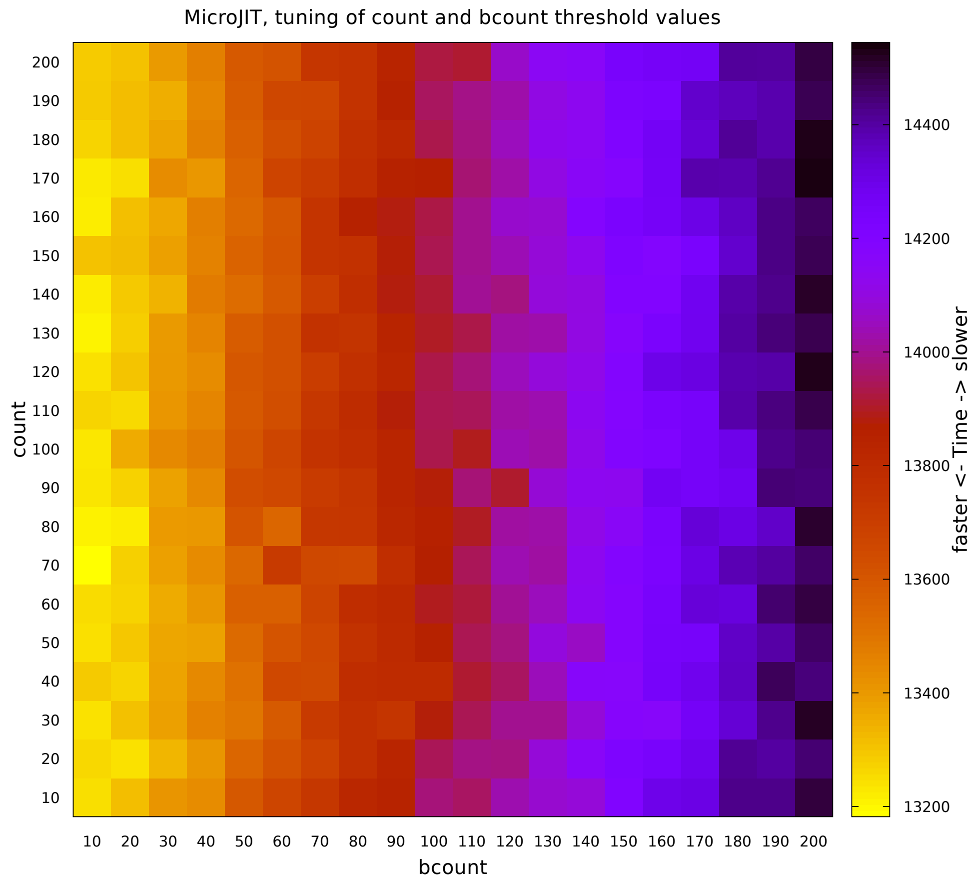

Figure 4.5: The *heatmap* gives a visual indication of the best values for *count* and *bcount*. Although not necessary, enough results were eventually collected to complete the image (DaCapo Batik).

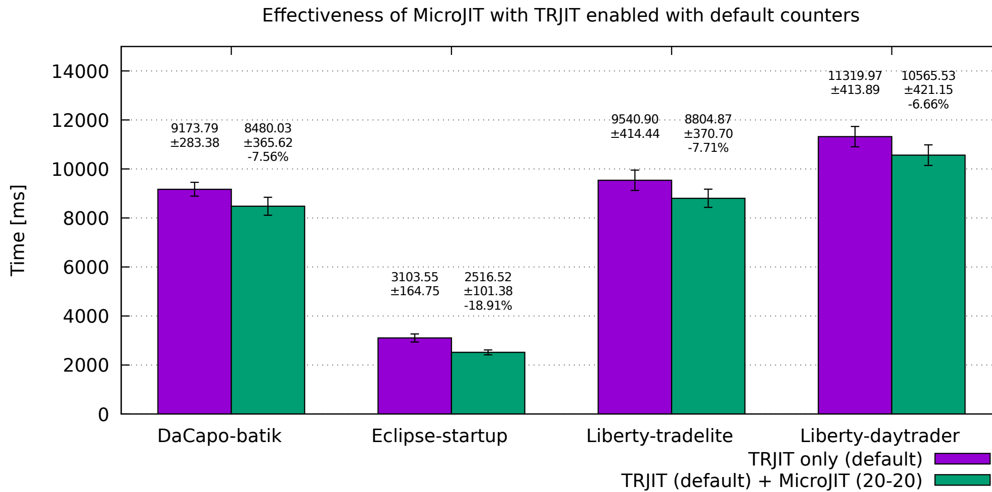

Figure 4.6: Comparison between JVM running with TRJIT only and default counters (3000-3000) and JVM running with TRJIT (3000-3000) and MicroJIT (20-20).

4.2 Evaluation of MicroJIT with TRJIT

After implementing the changes required to enable MicroJIT and TRJIT at the same time, as described in Subsection 3.3.2, the JVM was tested and benchmarked. The default counters for TRJIT are 3000 for both *count* and *bcount*. MicroJIT threshold values for *count/bcount* are both set at 20.

In Figure 4.6, the results show that the MicroJIT improved the performance of the JVM when it is used to compile methods before TRJIT. The improvement is significant and amount to about 7% for DaCapo Batik and Liberty Tradelite and Daytrader. Eclipse Startup improves by 18.91%.

4.2.1 Existing Solution for Startup Time

MicroJIT is effective when used to improve the JVM when TRJIT is used with default parameters. However, it is necessary to compare the solution proposed in this project with the other existing solutions:

- The option `Xquickstart` is equivalent to configuring TRJIT with the following settings.
  - *count* is set to 250 and *bcount* to 1000. These values, determined by IBM, are optimal to minimize startup time but not to maximize throughput.
  - Profiling is disabled in the interpreter.
  - TRJIT compiles at a lower level of optimizations.

  The results are presented in Figure 4.7. The option `Xquickstart` is more effective in reducing the startup time, compared to MicroJIT, by between 20% and 30% depending on the benchmark. MicroJIT, if enabled with `Xquickstart` set, does not improve startup time for DaCapo Batik and Eclipse Startup and have a negative effect for both Liberty benchmarks, increasing their startup time by between 1.3% and 2%.

- When Shared Class Cache (SCC) is enabled, TRJIT performs a special, relocatable, compilation (i.e. AOT compilation) and saves the translated methods in the cache during the first execution of an application.

When the application is executed again TRJIT can directly use the saved compilation to generate the native version of the method instead of compiling from scratch.

The results are presented in Figure 4.9. The JVM with TRJIT and MicroJIT, with SCC disabled, is between 58% and 96% slower than the JVM running with TRJIT enabled without MicroJIT and with AOT code available from the SCC. The difference in performance caused by enabling SCC depends also on the fact that ROM classes are already available. However, the availability of the AOT code is the most important factor as shown in Figure 4.8.

If MicroJIT is enabled when SCC is active, the benchmarks show a degradation in performance between 1% and 8%.

MicroJIT not only does not improve performance compared with the JVM executed with `Xquickstart` set or with AOT code available from the Shared Class Cache, but causes a slower startup time when used in addition to the other options. To explain this behaviour the JVM was instrumented to collect compilation timing for both the MicroJIT and TRJIT.

4.2.1.1 Compilation Time and Compilation Delay

Compilation time is the amount of time used by a thread to compile a method. In the case of MicroJIT the method is compiled by the application thread and the compilation time represents the time from when the

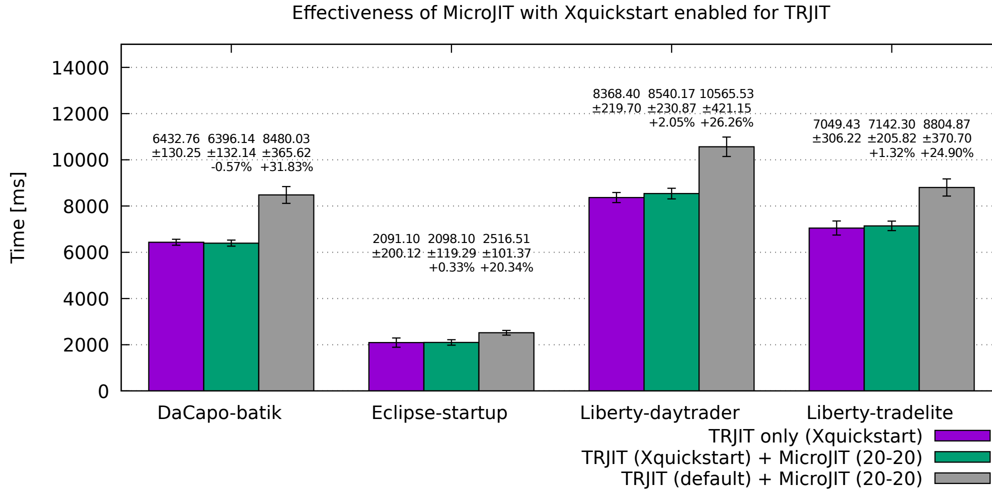

Figure 4.7: Comparison between JVM running with TRJIT Xquickstart (250-1000) and JVM running with TRJIT Xquickstart (250-1000) and MicroJIT (20-20).

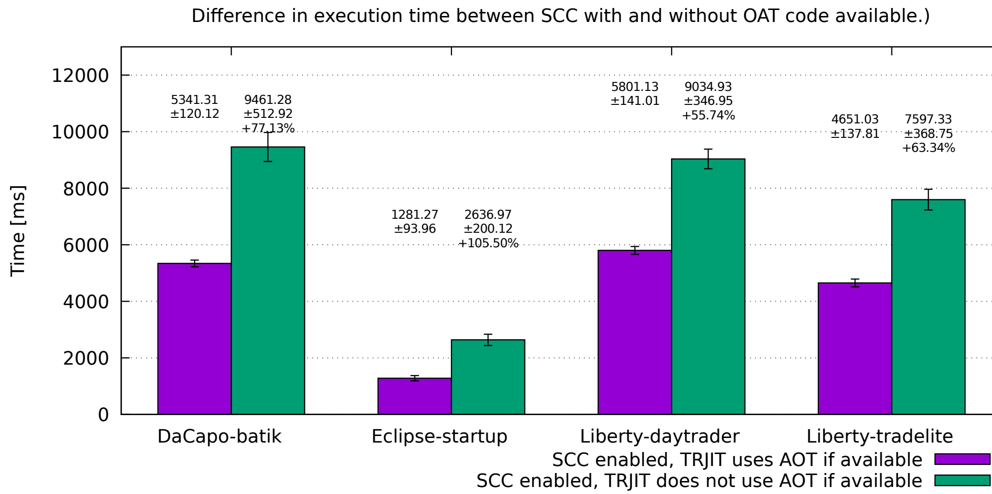

Figure 4.8: AOT code is responsible for most of the execution time gain when Shared Class Cache is enabled.

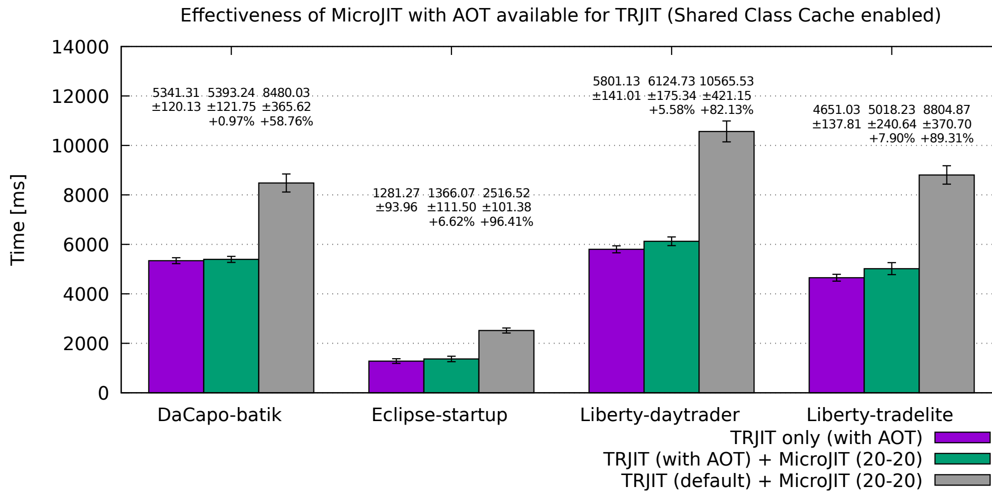

Figure 4.9: Comparison between JVM running with TRJIT with AOT code and JVM running with TRJIT with AOT code and MicroJIT (20-20).

interpreter passes the method to be compiled to the MicroJIT to when the address of the compiled method is returned to the interpreter.

TRJIT compilation is more complex: it is asynchronous and it is performed by dedicated threads (i.e. compilation threads). The function that the interpreter calls to trigger the compilation of a method with TRJIT, can return, not only before the compilation has finished, but also before the compilation has started. For TRJIT, it is therefore important to measure the compilation delay as well, defined as the time interval between when a method is queued for compilation and when its compilation starts.

The results from the measurements have been summarized in Table 4.4. For both compilation time and compilation delay a few methods require significantly more time compared to the majority of the other methods. The

1st, 2nd (i.e. median) and 3rd quartile are listed as better descriptors for the distribution.

When TRJIT is enabled with default settings the compilation delay can be significantly higher than the compilation time itself. When the option `Xquickstart` is set the compilation time is considerably faster and the compilation delay is also reduced. Finally, when AOT code is available, compilation time is on the same order or magnitude of MicroJIT compilation (method is already compiled and stored in the Shared Class Cache) and the delay is also further reduced compared to the case with `Xquickstart`.

To put the measured timings in a different context the JVM was instrumented to count the number of times a method is interpreted and executed using the native version generated with MicroJIT. The instrumentation adds a considerable overhead and to reduce contention a list of hashtables have been used to store the counters for each method. The collected data is merged and printed when the JVM terminates. The counter is not precise because it considers only invocations from the interpreter: all the invocations of methods inlined during compilation and invocations from the native code itself are not accounted for. The data collected gives sufficiently precise evidence of how the compilation delay measured in number of method invocations is considerably higher than the threshold at which the compilation is triggered. Table 4.5 lists the number of times a method is invoked using the version of the method compiled by the MicroJIT.

With the measured compilation times and compilation delays, it is possible to

TRJIT (default) with MicroJIT			
	Compilation time [μs]		Compilation delay [μs]
	MicroJIT	TRJIT	TRJIT
Max	3498	388868	824000
Min	3	112	0
Average	95	10070	224743
3rd quartile	95	9565	327000
Median	48	2950	155000
1st quartile	28	1223	0
TRJIT (Xquickstart) with MicroJIT			
	Compilation time [μs]		Compilation delay [μs]
	MicroJIT	TRJIT	TRJIT
Max	3821	253187	255000
Min	4	119	0
Average	98	1991	13000
3rd quartile	101	1891	21000
Median	50	979	2000
1st quartile	28	661	0
TRJIT (with AOT code) with MicroJIT			
	Compilation time [μs]		Compilation delay [μs]
	MicroJIT	TRJIT	TRJIT
Max	4354	134966	246000
Min	4	20	0
Average	99	1504	12000
3rd quartile	99	445	12000
Median	49	141	3000
1st quartile	29	94	0

Table 4.4: Compilation timing and compilation delay for MicroJIT and TR-JIT executing the benchmark DaCapo Batik. Note: compilation delay was measured in milliseconds.

explain why MicroJIT is improving startup time when enabled with TRJIT with default settings (i.e. *count* and *bcount* thresholds set at 3000), while, if the option `Xquickstart` or if AOT code is available enabling MicroJIT causes startup time to increase. In the former case, TRJIT spends a large amount of time compiling and there can be long delays. The methods compiled by MicroJIT are executed enough times to offset the cost of the compilations and to improve on the time the interpreter would have required to interpret the methods. In the latter case (`Xquickstart` or AOT code) methods compiled by MicroJIT are executed significantly fewer times, not enough to offset the compilation time itself.

4.2.2 Threshold Counter for TRJIT

The JVM was benchmarked with different *count* and *bcount* values of TRJIT, while maintaining *count* and *bcount* for MicroJIT, fixed. The purpose of this test is to determine if TRJIT threshold values could be adjusted considering that code generated by MicroJIT is faster than the interpreter.

The high variability of the results, represented in Figures 4.10 and 4.11 makes it hard to distinguish a pattern that would indicate that TRJIT behaves differently with and without MicroJIT over the default values of 3000 for both *count* and *bcount*.

When MicroJIT is disabled, it is possible to notice that the execution time, of both DaCapo-Batik and Eclipse startup, is lower when the threshold counter for TRJIT is lower than 3000. This behaviour is expected. The default values

TRJIT (default) with MicroJIT	
N. invocations	Method signature
368820	java/math/Multiplication.unsignedMultAddAdd(IIII)J
271800	org/apache/batik/.../png/PNGImageEncoder.clamp(II)I
96324	java/io/FilterInputStream.read()I
90449	org/apache/batik/.../png/PNGRed.paethPredictor(III)I
70200	java/awt/image/WritableRaster.setPixel(II[I)V
68400	java/awt/image/DataBufferInt.getElem(I)I
66459	java/lang/Math.pow(DD)D
48138	java/io/DataInputStream.readChar()C
45157	java/awt/image/DataBufferByte.getElem(II)I
45157	java/awt/image/Raster.getPixel(II[I)[I
...	...
5150958	Total
TRJIT (Xquickstart) with MicroJIT	
19064	java/io/FilterInputStream.read()I
11230	sun/java2d/pipe/Region.appendSpan([I)V
11205	sun/java2d/pipe/Region.needSpace(I)V
9511	java/io/DataInputStream.readChar()C
9430	sun/java2d/pipe/Region.endRow([I)V
5435	sun/java2d/loops/GraphicsPrimitiveMgr...
4457	java/io/BufferedInputStream.read()I
4024	java/awt/geom/Path2D$Float$TxIterator.currentSegment([F)I
3842	java/awt/geom/AffineTransform.transform([FI[FII)V
3735	sun/java2d/loops/GraphicsPrimitiveMgr...
...	...
1368070	TOTAL
TRJIT (with AOT code) with MicroJIT	
14926	java/util/ArrayList.ensureCapacityInternal(I)V
14926	java/util/ArrayList.ensureExplicitCapacity(I)V
5084	java/io/BufferedInputStream.read()I
4506	java/util/zip/Inflater.ensureOpen()V
3977	java/lang/StringBuffer.append(C)Ljava/lang/StringBuffer;
3196	java/util/zip/Inflater.inflate([BII)I
2670	java/lang/String.charAt(I)C
2661	java/lang/StringBuffer.¡init¿(I)V
2657	java/lang/StringBuilder.append(C)Ljava/lang/StringBuilder;
2255	java/lang/String.codePointAt(I)I
...	...
407803	TOTAL

Table 4.5: Most invoked native methods generated by MicroJIT. Application: DaCapo Batik.

(3000) for *count* and *bcount* are set by IBM to maximize the throughput of a Java application while the option `Xquickstart`, which changes the settings of TRJIT to minimize startup time, does in fact reduce threshold counters to 250 for methods with loops (*bcount*) and to 1000 for methods without loops (*count*).

4.2.3 Throughput Analysis

The option `Xquickstart` is very effective in reducing the startup time of a Java application, however it causes TRJIT to generate less optimized code. Therefore the performance of the JVM with `Xquickstart` set, is not optimal when the throughput of the application is considered.

The JVM, with and without the option `Xquickstart` was benchmarked and compared to the JVM running with TRJIT and MicroJIT, without `Xquickstart`, to determine how MicroJIT behaves in a throughput context. The first benchmark considered to measure throughput of the JVM is the application Tradelite running on Liberty server. Apache HTTP server benchmarking tool was used to send 100000 requests to the Liberty server and the response time for each request was measured. To minimize the effect of delays caused by the operating system and the network interface the 95% percentile of the requests have been considered. Since this is a throughput test, the application was *warmed up* with 100000 requests before starting measurements.

Liberty-Tradelite with TRJIT (default settings) has better throughput com-

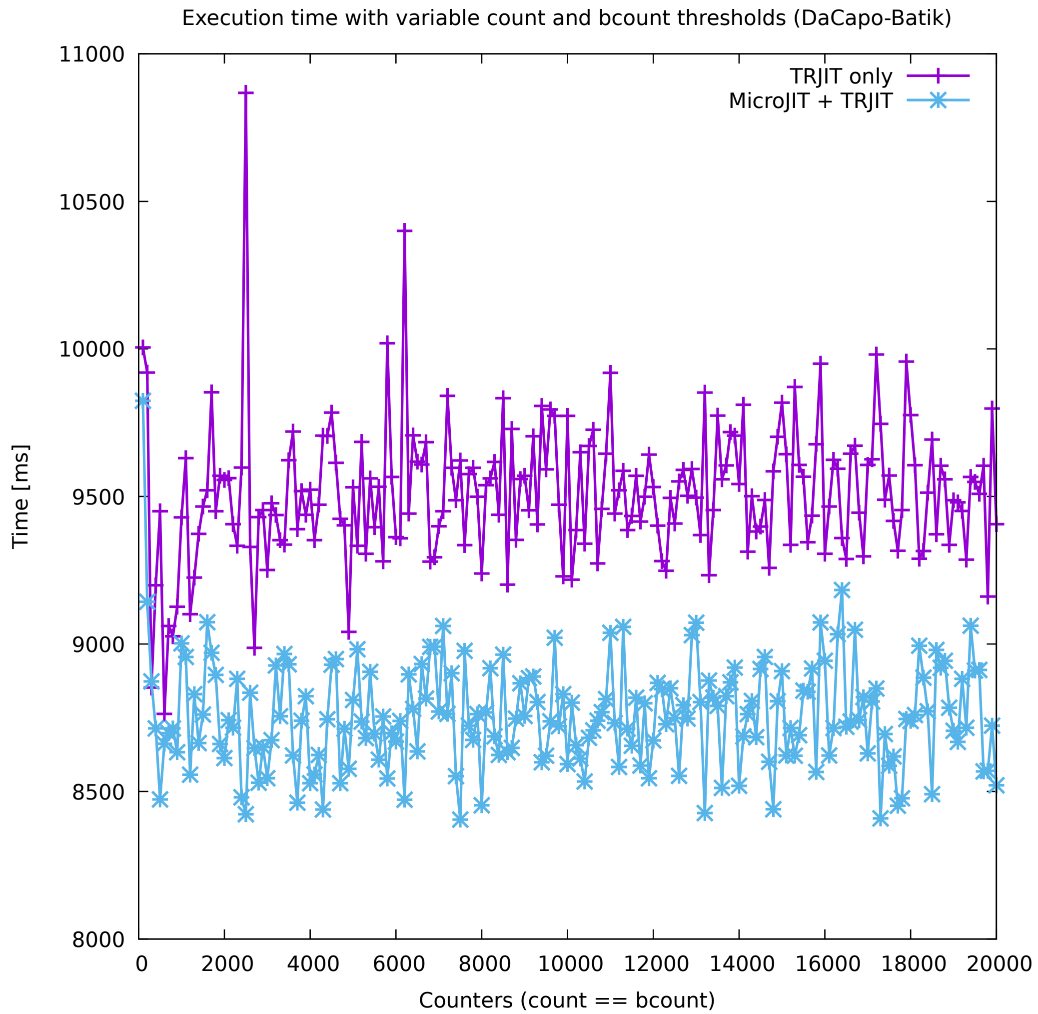

Figure 4.10: Each point is the median of 21 executions of DaCapo-Batik with MicroJIT threshold counters set at count=20, bcount=20, and with TRJIT threshold counters set both at the same value indicated on the X axis.

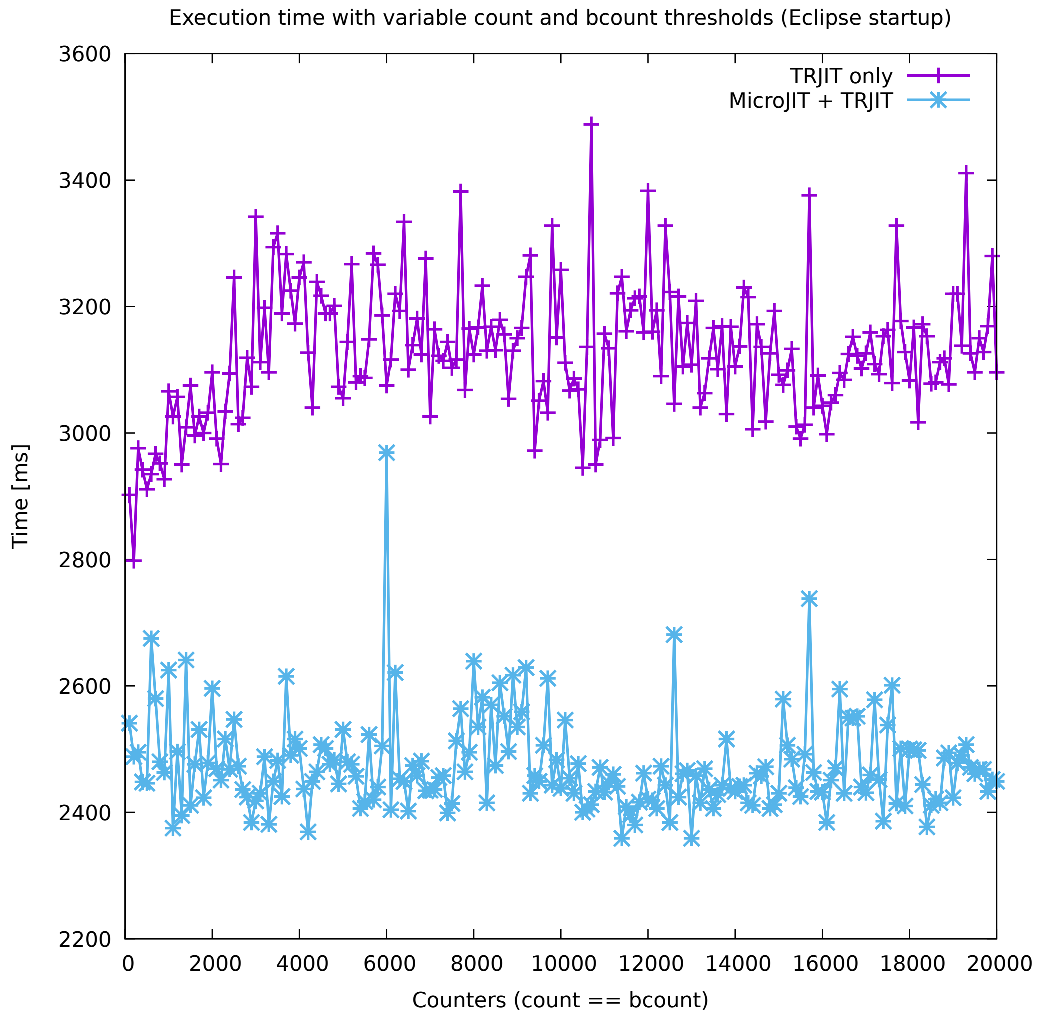

Figure 4.11: Each point is the median of 21 executions of Eclipse startup with MicroJIT threshold counters set at count=20, bcount=20, and with TRJIT threshold counters set both at the same value indicated on the X axis.

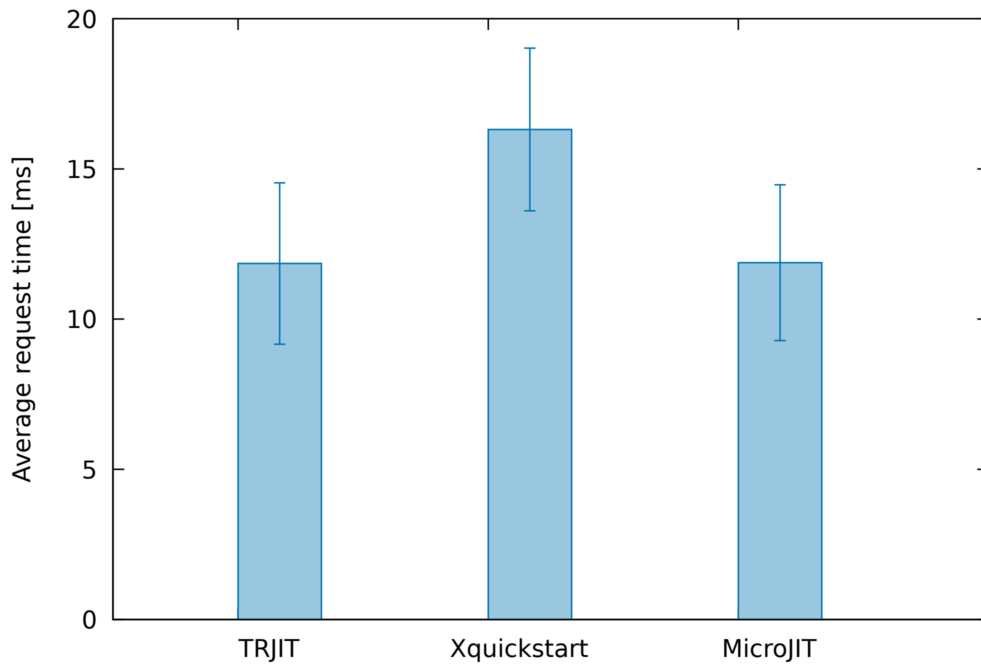

Figure 4.12: Throughput Results for Liberty-Tradelite

pared to the same application running with the option `Xquickstart` set, as expected. If MicroJIT is enabled while running TRJIT with default settings (without `Xquickstart`) Liberty-Tradelite does not show a significant slowdown in the response time. The code generated by the MicroJIT does not profile the execution of the methods like the interpreter does. The TRJIT optimization is therefore limited because it could not have enough information to determine the *hot execution path* of a method correctly. This specific benchmark does not seem to be penalized by this limitation of the MicroJIT. The throughput performance of the JVM was measured with another throughput-oriented benchmark, SPECjbb2005. The results show the JVM with the op-

Threads	TRJIT	Xquickstart	MicroJIT
8	175898	146228	160877
9	194340	156033	172983
10	213715	171767	195145
11	229564	187026	207326
12	234269	208679	226862
13	260650	219625	233363
14	276968	237648	254736
15	285056	252267	266550
16	293803	248608	276499
Average	240474	203098	221593
		-15.54%	-7.85%

Table 4.6: SPECjbb2005 results in business operations per second (bops). Percentages represent the reduction in throughput compared to the optimal JVM with TRJIT running with default settings.

tion `Xquickstart` enabled having 15.55% less throughput of the JVM with TRJIT using default settings. If the MicroJIT is enabled while running TRJIT with default settings, the throughput is reduced by 7.85%. MicroJIT in these cases penalizes the throughput of the benchmark compared to the optimal settings (TRJIT only with default settings), however it does perform better than the benchmark executed with the option `Xquickstart`.

4.2.4 Footprint Analysis

The memory footprint of the JVM with MicroJIT enabled was measured and compared to the memory footprint of the JVM with TRJIT enabled with default settings. The memory allocated to the `java` process in RAM, referred to as Resident Set Size or RSS in the Linux OS, was obtained using

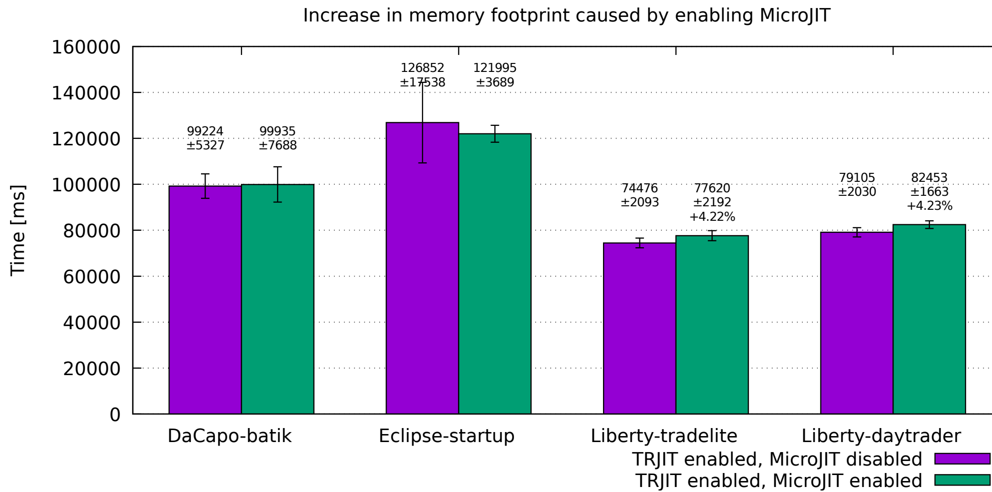

Figure 4.13: Memory Footprint.

the commands `/usr/bin/time` and `ps`.

The results for DaCapo-batik and Eclipse Startup are inconclusive. The variability of the multithreaded programs does not allow us to draw any conclusions from the measurement. Liberty, in this case, is a better benchmark: the RSS memory is measured after the server has completed startup and is ready to receive requests. The amount of work before the footprint is measured is constant and while the variability is still high (about 2 MB) the results show that the MicroJIT increases the memory usage by about 4%. Part of this footprint depends on the additional field in the internal structure for Java method (`J9Method`), which could be removed in future work, by integrating the MicroJIT with TRJIT.

Chapter 5

Conclusions and Future Work

The MicroJIT has been successfully ported to Java8 and integrated in the current JVM. MicroJIT compilation is triggered by the interpreter after only a few invocations of a method. The precise number of invocations (values for *count* and *bcount*) is defined from the command line when executing the `java` command and has been set to 20 in this project. TRJIT will then compile the same method again at a higher threshold to generate a better optimized, native, version of the method. By default TRJIT is requested to compile a method after up to 3000 invocations. The actual number could be lower due to the presence of a sampling thread that can anticipate compilation of methods frequently active.

Benchmarks show that the median of TRJIT compilations is about two orders of magnitude slower than the median of MicroJIT compilations, and that the delay of the compilations in TRJIT can be higher than the compilation

time itself. In these conditions, enabling MicroJIT to pre-compile methods early causes a reduction in startup time, or execution time for short lived applications. The improvement is significant among the considered benchmarks with a reduction between 6% (Liberty-Daytrader) and 18% (Eclipse Startup).

The IBM J9 JVM already implements two strategies to improve startup time. The first consists in using the Shared Class Cache feature, which enables the TRJIT to store the compiled version of the methods in the cache and to reuse them in future executions of the JVM. The second approach consists in tuning TRJIT to compile methods earlier and with less optimization (i.e. command line option `Xquickstart`).

Both approaches are more effective than using MicroJIT to compile methods at low invocation counts. The startup time for the JVM with MicroJIT enabled and TRJIT with default options, is between 58% and 96% slower than the JVM with Shared Class Cache enabled and MicroJIT disabled, and is between 20% and 31% slower compared to the JVM with `Xquickstart` set and MicroJIT disabled.

The effect of MicroJIT, when the Shared Class Cache is enabled and when `Xquickstart` is specified, has been examined as well. In both cases enabling MicroJIT does not improve the startup time.

In the case of the JVM with Shared Class Cache enabled, using MicroJIT causes a slow down that varies from 0.96% to 7.90%. The MicroJIT is not a feasible solution unless the Shared Class Cache is not available or is not

usually enabled, such as in a virtualized server in cloud environments.

When the option `Xquickstart` is set, the startup time does not vary significantly in the case of DaCapo-Batik (-0.56%) and Eclipse-Startup (0.33%). In the case of Liberty, the increase is between 1.32% and 2.05%. There is no limitation on using the option `Xquickstart`, since it is a shortcut to tune TRJIT to minimize startup time. However, the throughput of an application when running with the option `Xquickstart`, is lower compared to the same application running in the JVM with TRJIT with the default parameters. The difference has been examined using Liberty-Tradelite and with the SPECjbb2005 benchmark. In the case of the application Tradelite enabling MicroJIT does not causes a significant slow down in throughput, with an average response time (95% percentile) of about 12ms. The response time with the option `Xquickstart` is significantly higher, about 16%.

The results from the benchmark SPECjbb2005 show that using MicroJIT does lower the throughput (-7.85%) however MicroJIT is a better solution where compared to using the option `Xquickstart` that causes a reduction in throughput of 15.54%.

Using a lightweight JIT system to improve startup time is, in the current state of the MicroJIT, a solution that cannot be used universally. In some cases however it can be a feasible approach compared to the existing solutions in the J9 JVM.

Further research in this direction is advisable and some areas have already been defined to improve the performance of MicroJIT:

- **Asynchronous and multithreaded compilation** - MicroJIT performs compilation synchronously and the compilation itself is protected by a single lock. This was a valid solution for what MicroJIT was designed for: mobile devices with poor computational capabilities and single core CPUs. Modern computers, and even mobile devices, offer many CPU cores, therefore asynchronous and multithreaded compilation should be a good approach to improve performance of the MicroJIT. The implementation should use the framework already available for TRJIT, which already performs multithreaded, asynchronous compilation.

- **Minimize interpreter intervention** - The MicroJIT, in some cases, is not able to generate native code and requires the intervention of the interpreter to perform complex operations. The number of times the execution returns to the interpreter should be reduced further. The bytecode `invokevirtual` has already been improved in this project proving that there is still a significant margin of improvement available.

- **Interpreter profiling** - When methods are interpreted, information about the execution path is collected to help TRJIT generate better performing code. Methods generated by the MicroJIT should collect the same information so TRJIT would be able to generate the same highly optimized code and enabling the MicroJIT would not affect negatively the throughput performance.

- **Analysis of methods during class validation** - one of the fundamental phases of class loading is class validation: each method loaded in a class must be analyzed to determine if it can be interpreted safely. The MicroJIT, before compiling, analyzes the methods to determine how the Java stack is used and to identify all the branching operations. If it were possible to collect some information during class validation without additional cost MicroJIT compilation could be optimized further.

- **Compilation criteria** - The decision of triggering the first compilation of a method, by the interpreter, is currently based on the number of times the method has been invoked (compilation can be anticipated by a sampling thread). The amount of collected profiling information could be another approach to determine when a method should be compiled.

Bibliography

[1] *Apache Geronimo v2.2 Documentation - DayTrader*, "http://geronimo.apache.org/GMOxDOC22/daytrader-a-more-complex-application.html", Access date: 26/07/2016.

[2] *Concurrency is not parallelism - The Go Blog*, "https://blog.golang.org/concurrency-is-not-parallelism", Access date: 26/07/2016.

[3] *DaCapo Benchmarks Home Page*, "http://www.dacapobench.org/", Access date: 26/07/2016.

[4] *Eclipse CDT*, "https://eclipse.org/cdt/", Access date: 26/07/2016.

[5] *IBM CICS Performance Series: Comparing Type 2 and Type 4 JDBC Driver Performance with IBM CICS Transaction Server for z/OS V5.2 Liberty JVM server*, "http://www.redbooks.ibm.com/abstracts/redp5208.html", Access date: 26/07/2016.

[6] *IBM developerWorks - Enhance performance with class sharing*, `http://www.ibm.com/developerworks/library/j-sharedclasses/`, Accessed: 2015-09-24.

[7] *IBM Knowledge Center*, "`https://www.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/com.ibm.java.win.70.doc/diag/tools/jitpd_disable_some.html`", Access date: 19/07/2016.

[8] *IBM Knowledge Center - Frequently asked questions about the GC*, "`https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.lnx.80.doc/diag/understanding/mm_faq.html`", Access date: 07/07/2016.

[9] *IBM Knowledge Center - Selectively disabling the JIT or AOT compiler*, "`http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages`", Access date: 26/07/2016.

[10] *IBM Knowledge Center - Understanding class data sharing*, "`https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.aix.80.doc/diag/understanding/shared_classes.html`", Access date: 07/07/2016.

[11] *IBM Notes*, "`http://www-03.ibm.com/software/products/en/ibmnotes`", Access date: 26/07/2016.

[12] *Java bytecode*, "http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/", Access date: 07/07/2016.

[13] *JavaScript*, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference, Accessed: 2016-08-15.

[14] *Lua*, https://www.lua.org/, Accessed: 2016-08-15.

[15] *Moving to OpenJDK as the official Java SE 7 Reference Implementation*, "https://blogs.oracle.com/henrik/entry/moving_to_openjdk_as_the", Access date: 07/07/2016.

[16] *Python*, https://www.python.org/, Accessed: 2016-08-15.

[17] *TCK tools and documentation*, "https://jcp.org/en/resources/tdk", Access date: 07/07/2016.

[18] *WASdev - About WebSphere Liberty*, "https://developer.ibm.com/wasdev/websphere-liberty/", Access date: 26/07/2016.

[19] John Aycock, *A brief history of just-in-time*, ACM Comput. Surv. **35** (2003), no. 2, 97–113.

[20] D. Bacon, P. Cheng, D. Grove, and M. Vechev, *System and method for concurrent garbage collection*, January 25 2007, US Patent App. 11/188,024.

[21] P.W. Burka, N. Grcevski, C.B. Hall, and Z.L. Wang, *Lock reservation using cooperative multithreading and lightweight single reader reserved locks*, September 11 2012, US Patent 8,266,607.

[22] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko, *Compiling java just in time*, Micro, IEEE **17** (1997), no. 3, 36–43.

[23] Brian Goetz and Tim Peierls, *Java concurrency in practice*, Pearson Education, 2006.

[24] James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley, *The java language specification*, Pearson Education, 2014.

[25] Cheng-Hsueh A Hsieh, John C Gyllenhaal, and Wen-mei W Hwu, *Java bytecode to native code translation: The caffeine prototype and preliminary results*, Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, IEEE Computer Society, 1996, pp. 90–99.

[26] Charlie Hunt and Binu John, *Java performance*, Prentice Hall Press, 2011.

[27] Michael R Jantz and Prasad A Kulkarni, *Exploring single and multilevel jit compilation policy for modern machines 1*, ACM Transactions on Architecture and Code Optimization (TACO) **10** (2013), no. 4, 22.

[28] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: The art of automatic memory management*, Chapman & Hall/CRC Applied Algorithms and Data Structures series, Taylor & Francis, 2011.

[29] Kenneth B Kent and Micaela Serra, *Context switching in a hardware/-software co-design of the java virtual machine*, Designers Forum of Design Automation & Test in Europe (DATE), 2002, pp. 81–86.

[30] Chandra Krintz, *Coupling on-line and off-line profile information to improve program performance*, Code Generation and Optimization, 2003. CGO 2003. International Symposium on, IEEE, 2003, pp. 69–78.

[31] Prasad Kulkarni, Matthew Arnold, and Michael Hind, *Dynamic compilation: the benefits of early investing*, Proceedings of the 3rd international conference on Virtual execution environments, ACM, 2007, pp. 94–104.

[32] Jingmin Lei, *Cyber situational awareness and mission-centric resilient cyber defense*, 2015 4th International Conference on Computer Science and Network Technology (ICCSNT), vol. 1, IEEE, 2015, pp. 1218–1225.

[33] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley, *The Java Virtual Machine Specification: Java SE 8 Edition*, Pearson Education, 2014.

[34] John McCarthy, *Recursive functions of symbolic expressions and their computation by machine, part i*, Communications of the ACM **3** (1960), no. 4, 184–195.

[35] Tamiya Onodera and Kiyokuni Kawachiya, *A study of locking objects with bimodal fields*, SIGPLAN Not. **34** (1999), no. 10, 223–237.

[36] Colin Renouf, *Pro (IBM) WebSphere Application Server 7 Internals*, 1st ed., Apress, Berkely, CA, USA, 2009.

[37] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani, *Overview of the ibm java just-in-time compiler*, IBM systems Journal **39** (2000), no. 1, 175–193.

[38] Vijay Sundaresan, Daryl Maier, Pramod Ramarao, and Mark Stoodley, *Experiences with multi-threading and dynamic class loading in a java just-in-time compiler*, Proceedings of the International Symposium on Code Generation and Optimization (Washington, DC, USA), CGO ’06, IEEE Computer Society, 2006, pp. 87–97.

Vita

Candidate's full name: Federico Sogaro
University attended (with dates and degrees obtained): Degree in Information Engineering, University of Padua, 2007
Publications: N/A
Conference Presentations: N/A