1-1-2009

Triggering of Just-In-Time Compilation in the Java Virtual Machine

Rouhollah Gougol
San Jose State University

TRIGGERING OF JUST-IN-TIME COMPILATION IN THE JAVA VIRTUAL MACHINE

A Research Paper

Presented to

The Faculty of the Computer Science Dept.

San José State University

In Partial Fulfillment

of CS298, Masters Project Requirements for the Degree

Master of Science in Computer Science

by

Rouhollah Gougol

December 2009

ABSTRACT

TRIGGERING OF JUST-IN-TIME COMPILATION IN THE JAVA VIRTUAL MACHINE

by Rouhollah Gougol

The Java Virtual Machine (Standard Edition) normally interprets Java byte code but also compiles Java methods that are frequently interpreted and runs them natively. The purpose is to take advantage of native execution without having too much overhead for Just-In-Time compilation. A former SJSU thesis tried to enhance the standard policy by predicting frequently called methods ahead of their actual frequent interpretation. The project also tried to increase the compilation throughput by prioritizing the method compilations, if there is more than one hot method to compile at the same time. The paper claimed significant speedup. In this project, we tried to re-implement the previous work on a different platform to see if we get the same results. Our re-evaluation showed some speedup for the prediction approach but with some adjustments and only for server applications. It also showed some speedup for the prioritizing approach for all the benchmarks. We also designed two other approaches to enhance the original policy. We tried to reduce the start-up delay that is due to overhead of Just-In-Time compilation by postponing some of Just-In-Time compilation. We also tried to increase the accuracy of detecting frequently interpreted methods that contain nested loops. We could not gain any speedup for our former postponing approach but we could improve the performance using our latter measuring approach.

CONTENTS

TABLES

FIGURES

CHAPTER 1

ANALYSIS

1.1 Introduction

1.1.1 Just-In-Time Compilation

Just-In-Time compilation is the process of compiling code just prior to and parallel to the code execution. This approach is very flexible and advantageous but its most well-know application is in JVM. Since a Java application is in the format of Java byte code, JVM needs to either interprets the byte code or dynamically translates it to the native code of the running platform and executes the native code simultaneously. Such Just-In-Time compilation can improve the performance without compromising portability, security, and other features of Java.

A Just-In-Time compiler can detect the frequently called methods and the frequently iterated loops to bias the optimizations for the most useful code. This approach is significantly effective since, in an application run, often less than 50% of the methods ever run [14] and only 3% to 4% of all the methods perform most of the functionality [8]. A dynamic compiler can adapt the optimizations using a small threshold for a fast compilation and using a larger execution limit for an aggressive optimization.

Historically, computer systems used to be low in memory and could not do the compilation of the whole source code in memory. Such compilers did the source code gradually, so they selected only a section of the source code, compiled it, saved it as files, and then went for the other parts of the source code. The name of the process was compilation ”on the fly” [28], the name which is nowadays used for modern Just-In-Time compilation too. This process was also analogous to today’s Just-In-Time compilation since Just-In-Time also compiles the source code gradually.

Adapting the Just-In-Time optimizations based on the execution frequency can also improve the start-up performance. Because the call frequency of a method is low during the software initialization, the Just-In-Time engine will perform only the light optimizations. The light compilation overhead, code growth, and memory footprint, which can prevent memory swapping, will let the code start up efficiently [8].

The Just-In-Time compiler may profile the code, then optimize the code based on an assumption, and later de-optimize the code if the future profiling indicates the assumption is not valid any more. For example, the dynamic optimizer can assume some variables never change, and consequently some code blocks are useless, so it can eliminate the blocks. If the variables change later, the Just-In-Time compiler can change the optimized code back to the original functionality [13] (refer to Section 1.1.7).

A Just-In-Time compiler can transparently make the code cross platform. The host architecture can use a dynamic compiler to translate native code of a guest architecture into its own specification and run the code parallel to the compilation. For instance, the new 64bit Architecture Intel that is called Itanium is fundamentally incompatible with the Intel traditional 32bit architecture. However, the Itanium-based operating systems, including Windows and Linux, can still run 32bit software. They dynamically convert the traditional 32bit code into the 64bit instructions during the code execution [12]. The Just-In-Time compiler can work without any 32-bit hardware support. Even though some Itanium processors have an on-chip emulator of the x86 architecture, this dynamic compiler achieves better performance and is a competitive to the equivalently clocked processors that are based on the x86 architecture, such as AMD64 [12].

Just-In-Time compilation can maintain the original security specification. For example, JVM performs byte code verification, and then generates the native code together with the security tests such as null pointers checks and array out of bound exceptions. The native code execution is as safe as the byte code interpretation in the Standard Edition [14].

1.1.2 Java HotSpot Compiler

HotSpot is the Just-In-Time compiler that is part of JVM, Standard Edition. The JVM, SE. starts a Java application by interpreting the byte code and later during the interpretation, the HotSpot engine compiles each method that is frequently called or whose inside loops are frequently iterated. So, the JVM, by default, runs a Java application using combination of Just-In-Time compilation and byte code interpretation. If there is more than one method to compile at the same time, the compilation order will be first-in-first-served (See Appendix A). The combination of the interpretation and dynamic compilation is because of the policy of adaptive optimizations and efficient start-up, as we already discussed.

Client Mode and Server Mode of HotSpot The JVM, the Standard Edition, by default, starts the HotSpot compiler in the client mode, which has a low compilation threshold (refer to Table 1.2) and some light optimizations to have an efficient startup and low memory consumption. The Java environment has an alternative server mode, which has a large amount of interpretation prior to the Just-In-Time compilation and aggressive optimizations such as in-linings in order to gain performance during a long run. The Java HotSpot compiler, in addition to counting the frequencies of the method calls and loop iterations, profiles the values in the methods for more advanced optimizations such as de-virtualization (refer to Section 1.1.7).

Tiered Mode Compilation in Java HotSpot HotSpot VM, JDK 6, already has a multi-tier mode that starts in the client mode, which has a good startup and then cruises in the server mode, which has a good steady-state execution [7]. The benchmark results are already positive for both the start-up and long run performance on all the supported platforms, but the future development is still under progress. The Java HotSpot compiler, as a result, adapts with both the client mode and the sever mode transparently.

Open JDK Open source software, as a novel development methodology, allows the users and third party companies contribute in removing the defects and enhancing the features. Such participation may be numerous and free of the work place restrictions. The administrators will not have to suffer from the overhead and delay of reporting to and requesting from the developers. It can also reduce the burden of the deadlines, budget shortage, and work stress from the main engineers as well [2].

HotSpot together with JDK 1.7.0 is already free and open-source software, which is mostly in C++. It, however, has an Assembly interpreter per each supported platform, such Windows x86-32bit and Solaris SPARC 64bit in addition to a C++ interpreter for some other platforms such as Itanium. Even the Assembly interpreter code often calls helper C++ methods.

1.1.3 Previous Work

Azeem Jiva, who was a former graduate student at the Computer Science department, San Jose State University and was an engineer at Sun Microsystem, Inc. claimed that the compilation of the methods that were already frequently called suffered from the overhead of interpretation for a fixed number of times in addition to the compilation overhead. He tries to predict the hot methods in order to trigger the Just-In-Time compilation on them sooner , so they suffer less interpretation overhead and run more natively[1].

Jiva claims that an "extremely large" number of methods may become hot at the same time, in the Java HotSpot Virtual Machine, and may need to wait for the Just-In-Time compilation. JVM still will interpret such a waiting method upon its method-calls until HotSpot completes Just-In-Time compilation of the other methods that are ahead. So the more methods wait in line of compilation, the more overhead of the method interpretation occur. Jiva claims scheduling the compilation of the concurrently hot methods should be based on a method size, since, he claims, the compilation duration of a larger method is longer [1].

1.1.4 Integrating Previous Work with Multi-Tier Compilation

Since the approaches of priority queue and the method grouping have some performance flaw during the startup and some optimization gain in the steady state execution, a multi-tier approach may adapt the Just-In-Time triggering. The JVM can, using multiple tiers of optimizations, apply the normal compilation queuing and interpretation frequency counting in the first tier and take advantage of sorting the compilation tasks and predicting the frequently executed blocks in the higher tiers.

1.1.5 Our Two New Approaches

We designed the following two approached for the original version of JVM. However, these approaches are still related to Jiva's works since these approaches indicate which methods to compile and when.

Relative Approach Many JVMs try to postpone Just-In-Time compilation from application start-up time to the steady state execution in order to reduce delay of application start-up that is due to overhead of Just-In-Time compilation. Most such JVMs, including up-coming version of HotSpot, use a multi-tier approach and recompile methods that get even significantly hotter than when they were first compiled. However, many smaller Just-In-Time environments do not have such complicated features and even the multi-tier compilation feature of HotSpot is not finished yet.

We decided to bring up an approach of postponing Just-In-Time compilation from Application initialization that is easier than approach of multi-tier compilation. We tried to detect the point of an Application execution, when the application switches from initialization to a steady state execution. We tried to detect this point based on the average interpretation frequency of all the running methods. So we tried to reduce amount of compilation that HotSpot does before that point and increase the amount after that point.

The approach computes the average of interpretation frequencies of running methods. The approach assumes that when the average is getting bigger than an amount, the application has already passed the initialization phase and has moved to continuous run. So it adjusts the amount of JIT compilation relative to the average of interpretation frequencies.

Blocksize Approach The Java HotSpot framework computes the sum of the number of times a method is called and the number of times backward branches (loop) take place in the method. Java HotSpot considers that sum as the hotness of the method. We assume it does not precisely measure interpretation frequency of a method since a huge method may be soldemly called but may contain a tiny code block that is looped more than JIT threshold. Java HotSpot policy detects such a method as hot even though the majority of instructions in the method are not frequently interpreted at all. We tried to enhance measuring the hotness by taking into account the size of a method and the sizes of the code blocks that are looped.

1.1.6 Our Evaluation Approaches

We used three variant, famous benchmark tools: SPECjvm2008, Ashes, and Volano Mark. SPECjvm2008 is a very standard and industry-level benchmark suite which used to be proprietary but became free in May 2008. SPECjvm2008, which replaces SPECjvm98, measures the performance of a Java Runtime Environment. The suite involves various general purpose applications and computations and evaluates the performance on both client and server systems [29].

Ashes is also a free collection of Java benchmarks that includes various Java benchmarks and includes bash scripts that can run each benchmark application ten times and can compute the average of the benchmark results. These benchmarks are like ordinary Java applications so the lower the benchmark time, the better the performance [30].

The VolanoMark 2.1.2 is a pure Java benchmark that can measure performance of a Java server and can also measure scalability. We just used the raw performance measurements since benchmarking scalability required sophisticated network environment which we did have access to. The benchmark creates numerous client and socket connections and at the end computes the average number of the messages that were transferred so the higher number means better performance [31].

1.1.7 Dynamic Optimizations

Just-In-Time compilation provides numerous dynamic optimizations, some of which are not practical in traditional compilations. Dynamic optimizations still drive from the traditional compiler designs, but they can apply current runtime statistics. Here we mention some of them as a way to introduce Just-In-Time compilation and its optimization techniques.

Specialization The compiler detects variables that have constant value for a significant amount of the time by profiling the values. Then it generates a version of the method based on such constant values of the variables in addition to the normal version of the method. A specialized version of a method may have many optimization potentials such as constant folding and unreachable code elimination. The compiler adds some guard tests that allows the special code run only if it has the criteria, otherwise, the unspecialized code will run.

Figure 1.1 is the control flow of an Assembly method that illustrates specialization. The value distribution of the register $18 comes in Figure 1.2 where the value 1 has about 70% probability. The method code is specialized based on $18 equals to 1 in Figure 1.3. First, a test and a branch guard the optimization. The specialization provides the following optimizations. The branch in the code blocks B1 does not happen, the block B6 is useless, and the blocks B1 and B2 combine. The branch in block B2 does not need the comparison anymore and directly happens based on the register $1. The branch in the block B7 never happens, and the block is useless. The load in block B8 can go into block B2, which makes the equivalent load in the block B3 redundant and useless.

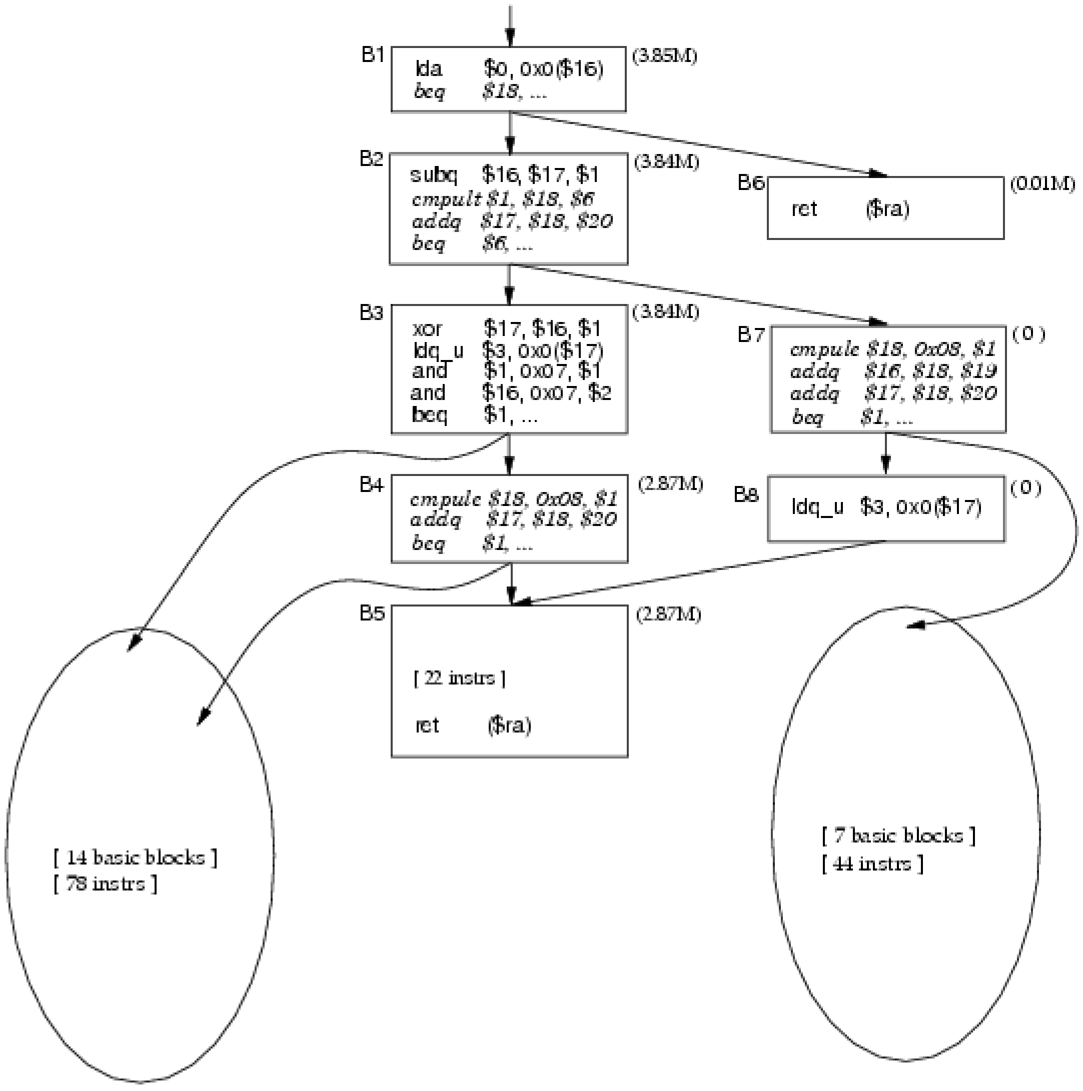


Figure 1.1: Control Flow of the Original Method Code that is to be Specialized based on Register $18 [32]

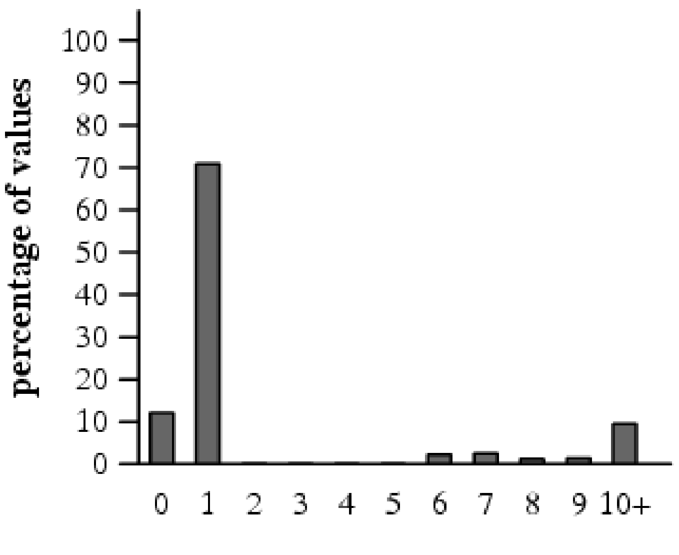


Figure 1.2: Distribution of Register $18 [32]

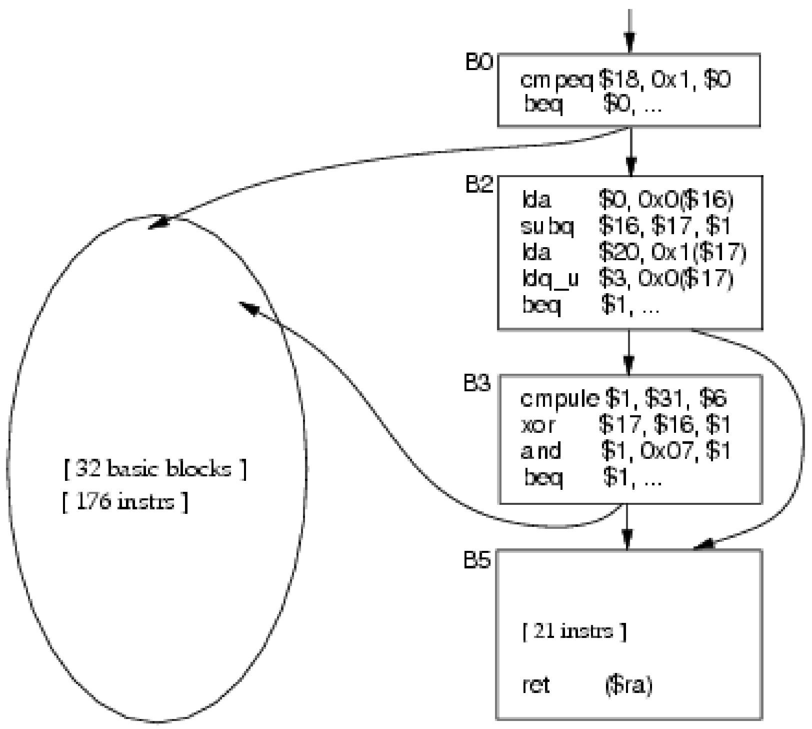


Figure 1.3: Control Flow of Specialized Method with Register $18 Equals 1 [32]

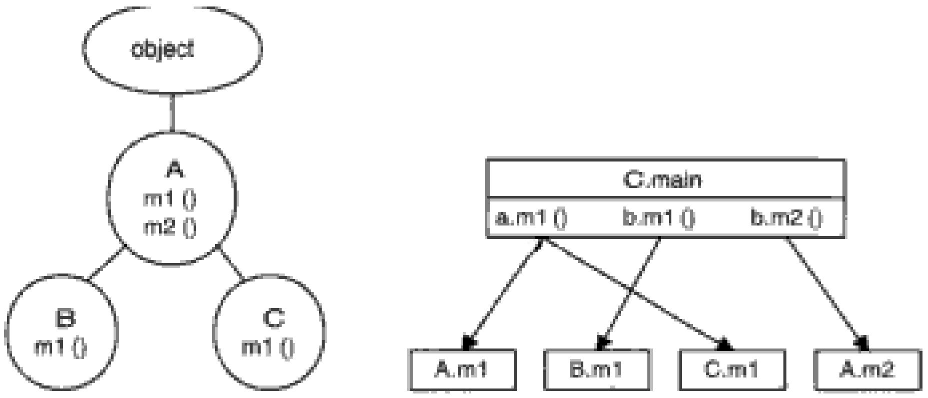


Figure 1.4: De-virtualization based on Class Hierarchy Analysis

De-virtualization A virtual call has the overhead of looking up the target method at runtime. The overhead is heavy in the dynamically typed languages but it is already low in statically typed programming including Java where the call resolution is no more than a few loads and an indirect jump.

Analyzing the class hierarchy provides the set of the possible target methods for the dynamic calls. Figure 1.4 and Listing 1.1 illustrates each method call in the main function belongs to which target method. The Java dynamic class loader, however, may change the class hierarchy of the objects. A Java class loader, which extends `ClassLoader`, can load a class byte code through a file system.

Caching the method target addresses can also indicate the set of the recent target methods for a virtual call in order to find the target addresses faster [18]. Type predication [16] [17] and method test [18] can enhance this dynamic call resolution by predicating the frequently called classes.

De-virtualization is the optimization of replacing a dynamic call with a static call, which removes the overhead of looking up the dynamic call. It also provides opportunity of the other optimizations particularly in-linings.

A dynamic compiler can de-virtualize the methods that have only one imple-

Listing 1.1: Object Methods with and without Overriding

```
class A {
        void m1() { }
        void m2() { }
}
class B extends A {
        void m1() { }
}
class C extends A {
        void m1() { }
        void main() {
                A a = new C();
                B b = new B();

                a.m1(); b.m1(); b.m2();
        }
}
```

mentation (no other method override them) and if the dynamic class loader overrides the de-virtualized method call, de-optimization can fix the problem [15]. The de-virtualization can keep the original instructions in the code as backup and put in some jumps that always skip the saved code. The de-optimization, if required, replaces the de-virtualized calls with some direct jumps to the backed-up original instructions, Figure 1.5. The optimization should be thread-safe, and only one instruction should atomically changes.

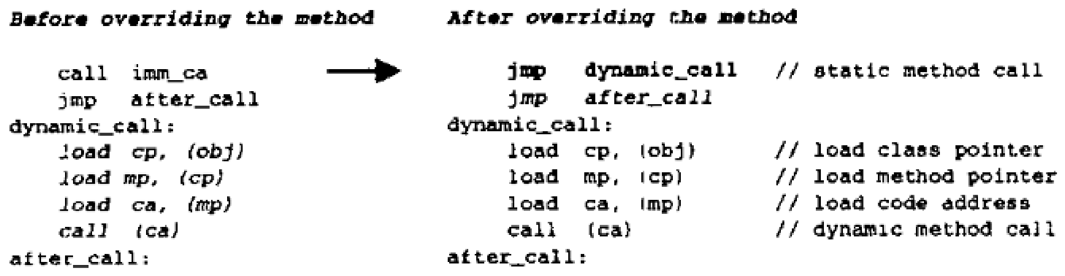
Another approach of de-virtualization is to put the static call with a guard test together with the original virtual call. If the test does not verify the de-virtualized call, the original dynamic method will take place. A common kind of a guard test is a class test, which compares the class of the called method with that of the de-virtualized function [18]. The algorithm is as follow:

(1) Add the address of the receiving object to the offset of the class in the object

(2) If the computed address equals the class address of the de-virtualized method, run the optimized method

(3) Otherwise, run the original dynamic method

If there is more than one class that is acceptable for the de-virtualized method, the test guard can simply test for only one of them and not validate the others. A



```
Before overriding the method        After overriding the method

    call  imm_ca            ──►        jmp   dynamic_call     // static method call
    jmp   after_call                   jmp   after_call
dynamic_call:                       dynamic_call:
    load  cp, (obj)                    load  cp, (obj)        // load class pointer
    load  mp, (cp)                     load  mp, (cp)         // load method pointer
    load  ca, (mp)                     load  ca, (mp)         // load code address
    call  (ca)                         call  (ca)             // dynamic method call
after_call:                         after_call:
```

Figure 1.5: Reverting a De-virtualized Method Call to the Original [19] page 122

more complex approach is to store all the acceptable address and make conjunction comparisons.

In-linings Object-oriented programming often calls small methods such as the object constructors, which have the overhead of branching and disrupting the CPU pipelines. In-lining a method body in the place of a method call removes the overhead and even allows the cross-functions optimizations.

Exception Tests Optimizations Compilers can safely eliminate test code of an exception that never happens. For example, in Listing 1.2, the former array bound exception tests can prove the array access of `a[0] a[1] a[2]` and some others are already valid.

An exception test can become a simple trap instruction, which is called light weight, on Power PC architecture. For each of the tests such as null-pointer, array index bound, and division by zero tests, the Just-In-Time compiler can generate a

Listing 1.2: Array Bound Exception Check Elimination [19]

```
a[i]=0;
a[i+2]=2;
a[i++]=0;
a[i+2]=a[i]+a[i+1];
a[i+1] =a[0] + a[1] +a[2];
```

Listing 1.3: Exception Checks with Light Weight Trap Instructions on a Power PC [19]

```
; r4 : array index
; r5 : array base
; r6 : array size
; r7 : divisor

twi EQ, r5, 0    ; Check null-pointer
tw GE, r4, r6    ; Check array-bounds
mulli r4, r4, 2
lwzx r3, r4(r5) ; Get array element
twi LLT, r7, 1   ; Check divisor
divi r3, r3, r7

// The handler
void TrapHandler (struct context * cp)
{
        int *iar = cp->IAR;      // Get the address at which
                                 // the exception occurs
        if IS_TRAPI_EQ(iar) {    // is inst. twi EQ
                process_NULLPOINTER_EXCEPTION()
        } else if IS_TRAP_GE(iar) { //is inst. tw GE
                process_ARRAYOUTBOUND_EXCEPTION()
        } else if IS_TRAP_LLT(iar) { is inst twi LLT
                process_ARITHMATIC_EXCEPTION()
        }
}
```

trap instruction without any register allocation that will take only one cycle. The trap tests whether the exception happens, if so, the trap handler will handle the exception. The trap handler should then indicate which kind of exception has happened since there is only one trap handler for all the trap checks. Listing 1.3 illustrates that the trap instructions `twi` and `tw` do not require any register allocation for the test results and the handler indicates the type of exception from the address of the instruction that invoked the trap.

Type Inclusion Test Just-In-Time compiler may optimize the exception test of casting an object into a class type, which is expensive and requires the traversal and analysis of the class hierarchy. Encoding the class hierarchy in a data structure can optimize the test so that verifying the type inclusion takes a constant time. The Just-In-Time compiler should update the data structure of a class hierarchy each time the dynamic loader modifies the hierarchy. Making this structure consequently requires space and time during the dynamic compilation.

Another approach is to test the exception cases from the simplest to the hardest.

Peephole	Optimization
a^2	$a * a$
$a * 2$	$a + a$
$a * 2^n$	lshift(a, n)
$a/2^n$	rshift(a, n)

Table 1.1: Peephole Optimizations

Testing some simple cases will not require more than a few clock cycles, Listing 1.4. The first case is when the operand object is null. The second case is when the original class type of the operand object is the same as the class cast type. The third case is when the environment has already cast the operand class type to the target class type, which requires caching the previous successful class cast. Experiments show the first three simple cases account for 91% on average of the type cast exception tests [19].

Listing 1.4: Optimized Type Inclusion Test

```
//Java code
        Type to = (Type) from;

// Type cast
        if (from == NULL) to = from;
        else if (from.type == Type ) to = from;
        else if (from.type.lastcast == Type) to = from;
        else if (run C run-time class cast test , if succeeded) to = from; from.type.lastcast ==
            Type;
        else throw exception;
```

Peephole Peephole optimizations mean replacing an operation with an equivalent but faster one, such as replacing a division by a power of two with a right shift. The optimization specially may make the computations inside a loop less expensive. Table 1.1 shows more ways of reducing the instructions strength.

Common Sub-expression Elimination The Just-In-Time optimizer can move an invariant computation of an instance access outside of the loop. The Just-In-Time compiler can eliminate a repeated instance variable access on an execution path by mapping the instance variable to a local variable or even a physical register. Listing 1.5 uses the C notation of arrays and pointers to illustrate replacing the instance variable of the class with a local variable.

A Just-In-Time optimizer can move an invariant computation of an array access whose object and index do not change out of the loop. The dynamic compiler can replace a similar array access with a local variable. In Listing 1.5, the local variables `v1` and `v2` obtain the value of the array access. In Listing 1.5 the interior pointers index the array elements.

We cannot eliminate a pointer to an object while there are interior pointers to the object. The reason is that garbage collector destroys an object without any pointer pointing to it even though there are pointers pointing to the middle of the object. For example, an pointer to top of an array should remain as long as there are interior pointer to the arrays since garbage collector do not check these middle object pointers. Reducing array access can reduce array-bound checks. In Listing 1.5 the array-bound checks are reduced from 6 to 2.

Listing 1.5: Common Sub-expression Elimination for Instance Vriables and Arrays

```
// Original class
class Foo {
        int [] a;
        public void foo () {
                a = new int [10];
                for (int i =0; i < 8; i++) {
                        if (a[i] < a[i +1]) {
                                int t = a[i];
                                a[i] = a[i + 1]; a[i + 1] = t;
                        }
                }
        }
}

// Instance variable CSE
```

```
class Foo {
        int a [];
        public void foo () {
        a = new int [10];
                int [] la = a;
                for (int i =0; i < 8; i++) {
                        if (la[i] < la[i +1]) {
                                int t = la[i];
                                la[i] = la[i + 1]; la[i + 1] = t;
                        }
                }
        }
}

// Array access CSE
class Foo {
        int a [];
        public void foo () {
        a = new int [10];
                int [] la = a;
                for (int i =0; i < 8; i++) {
                        int * v = la[i]; v1 = * v; v2 = * (v1 + 1);
                        if (v1  < v2) {
                                int t = v1;
                                * v = v2; * (v + 1) = t;
                        }
                }
        }
}
```

1.2 HotSpot Policies

To understand Jiva's approaches and also our approaches, the following information regarding Java HotSpot is useful.

1.2.1 Profiling in HotSpot

The algorithm is outlined below, in which native execution of a method takes place upon the next method call after compilation (i.e. method re-activation). The Java HotSpot compiler profiles a method only during a limited number of the method calls prior to compilation using the thresholds of the Just-In-Time profiling and compiling, Equation (1.1) on page 19, and Equation (1.2) on page 19, and the constants in Table 1.2. The profile threshold should be less than the compile threshold and non-negative.

The algorithm is as follows:

(1) Increment `invocation_counter`, the interpretation counter of the method

(2) $sum \leftarrow$ `backedge_counter` + `invocation_counter`

(3) If $sum <$ `InterpreterProfileLimit`, profile the method

(4) else if $sum \geq$ `InterpreterInvocationLimit`, replace the method bytecode with native code (refer to Listing 1.6).

Listing 1.6: HotSpot Source Code of Triggering Profiler

```

if (ProfileInterpreter) { // %%% Merge this into methodDataOop
      __ incrementl(Address(rbx,methodOopDesc::interpreter_invocation_counter_offset()));
        }
  // Update standard invocation counters
  __ movl(rax, backedge_counter);               // load backedge counter

    __ incrementl(rcx, InvocationCounter::count_increment);
      __ andl(rax, InvocationCounter::count_mask_value);  // mask out the status bits

        __ movl(invocation_counter, rcx);             // save invocation count
          __ addl(rcx, rax);                            // add both counters

            // profile_method is non-null only for interpreted method so
            // profile_method != NULL == !native_call
            // BytecodeInterpreter only calls for native so code is elided.

            if (ProfileInterpreter && profile_method != NULL) {
                    // Test to see if we should create a method data oop
                    __ cmp32(rcx,
                                     ExternalAddress((address)&InvocationCounter::
                                          InterpreterProfileLimit));
                        __ jcc(Assembler::less, *profile_method_continue);

                            // if no method data exists, go to profile_method
                            __ test_method_data_pointer(rax, *profile_method);
                              }

  __ cmp32(rcx,
                    ExternalAddress((address)&InvocationCounter::InterpreterInvocationLimit));
  __ jcc(Assembler::aboveEqual, *overflow);

}
```

$$InterpreterProfileLimit = \frac{CompileThreshold * InterpreterProfilePercentage}{100} \tag{1.1}$$

$$InterpreterInvocationLimit = CompileThreshold \tag{1.2}$$

1.2.2 Triggering during Loop Iterations

The Just-In-Time compilation and native execution of a method that contains a frequently iterated loop takes place in the middle of the method interpretation (i.e. without method re-entrance) using the mechanism of On-Stack-Replacement, Figure 1.6. This strategy is particularly beneficial for methods that contain long loops and never gets re-entered. Equation (1.3) on page 19 indicates the relation between OSR threshold and the ordinary compile threshold. The Just-In-Time triggering algorithm is:

(1) Increment `backedge_counter` of a method as an inside loop iterates

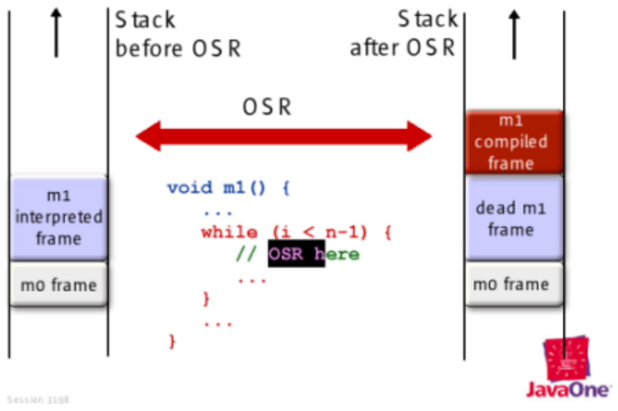
(2) $sum \leftarrow$ `backedge_counter` $+$ `invocation_counter`

(3) if $sum \geq$ `InterprterBackwardBranchLimit`, compile the method and start the native execution in the middle of the loop.

$$BranchLimit = Threshold * \frac{OSRPercentage - ProfilePercentage}{100} \tag{1.3}$$

Just-In-Time Triggering Constants	Note
`CompileThreshold`	The main threshold that triggers JIT compilation of a method upon method calls It is 1,500 in the client mode for the x86 platforms It is 10,000 in the server mode (x86)
`InterpreterProfileLimit`	Threshold to stop profiling the method
`InterpreterBackwardBranchLimit`	Threshold to compile a method during a loop in middle of method execution
`InterpreterInvocationLimt`	Threshold to compile a frequently called method
`InterpreterProfilePercentage`	33% for the x86 platforms
`OnStackReplacePercentage`	933% in the client mode and 140% in the server mode (x86). These percentages make a threshold during a loop bigger than during a method call

Table 1.2: Constant Values in the Formula of the Just-In-Time Compiling and Profiling Thresholds



```
void m1() {
   ...
   while (i < n-1) {
      // OSR here
      ...
   }
   ...
}
```

Figure 1.6: Just-In-Time Compilation and Native Code Execution with On Stack Replacement within the Loop Iterations[7].

1.3 Method Grouping

We called Jiva's approach of predicting methods (before they become really hot) me*Method Grouping* approach. Since this approach groups each method with the other methods that are called just prior to it, as described below.

1.3.1 Jiva's Analysis

In Jiva's Just-In-Time triggering policy, the interpreter groups each method with a given number of the preceding methods that run just prior to it. If a method becomes hot, the Just-In-Time compiler predicts the group members (i.e. the preceding methods) will become hot later and compiles the predicted methods upon their next calls regardless of their interpretation frequencies [1]. Jiva uses a method's signature, which is the method name, the method class name, and the method argument types, to keep track of methods. In his Future Works section, Jiva suggests that such computation with strings is too costly. To incorporate his suggestion, we assigned a numerical id to each method and used those numbers instead of the string signatures.

The purpose of method grouping is to reduce the overhead of interpretation prior to compilation. Jiva brings up a sample Java class to illustrate the benefits of his policy, and mentions the "locality of references" principle to justify his prediction approach. He may have meant spatial locality [22] which states that data references in a nearby location tend to be used together. The following is the prediction algorithm.

Upon a method call:

(1) If the method does not yet have an id, generate and assign a new, unique id to the method.

(2) Associate the id of the method that just preceded the current method to the current method as its previous one.

(3) Store the id of the current method as a preceding method id for an upcoming method.

(4) If the current method is hot, trigger compilation on it, store each of its previous ones as a predicted hot method.

(5) else if the id of the current method is already predicted to be hot, which means it has been a previous one of a formerly hot method, JIT compile the method.

1.3.2 Evaluation of Jiva's Approaches

(1) The Just-In-Time profiling threshold is a function of the method's Just-In-Time compile threshold (Equation (1.1) on page 19), but Jiva's approach does not update a profiling threshold as he changes its corresponding compile threshold. In our modification, we update all the thresholds while predicting a hot method and while reducing its compile threshold. (Refer to Section 1.3.3)

(2) The method grouping approach may delay the application's startup since it increases the Just-In-Time compilation and overhead during the software initialization. To prevent such overhead, we, besides implementing the ordinary approach of method grouping, implemented another version of method grouping in which the prediction process is postponed to the steady state execution.

1.3.3 Updating Method Threshold

HotSpot profiles methods before compiling them. The number of times a method gets profiled is proportional to compile threshold (Equation 1.1 on page

19). For example, HotSpot profiles each method 33% of the number times it interprets a method before compiling it in x86 architecture.

As already mentioned, we may compile a method sooner than the ordinary `CompileThreshold`, in the grouping approach, for the methods that we predict to be hot. That could violate the relation between compile threshold and profiling amount (Equation 1.1 on page 19). In fact, Jiva, in his version, schedule a predicted method for immediate compilation regardless of how much the method is already profiled.

In our version of the approach, we do not necessarily schedule predicted methods for immediate compilation. We instead reduce the compile threshold of such methods and also adjust their profiling limit so that we still keep the relation mentioned above.

So when we predict a method to be hot, we set the method profiling limit to the current method invocation count, meaning that the method should not be profiled any more. Then we compute the new `CompileThrehsold` which is supposed to be bigger than `InterpreterProfileLimit`, based on the new `InterpreterProfileLimit`. This is only in case that the original `InterpreterProfileLimit` is already above the `CompileThrehold` otherwise we could not reduce `InterpreterProfileLimit` and `CompileThreshold` any more, Equation 1.4 on page 23.

$$\begin{aligned} if InterpreterProfileLimit &> CurrentMethodInterpretationFrequency \\ InterpreterProfileLimit &\leftarrow CurrentMethodInterpretationFrequency \\ CompileThreshold &\leftarrow \frac{CurrentMethodInterpretationFrequency * 100}{InterpreterProfilePercentage} \end{aligned} \tag{1.4}$$

sectionPriority Approach We call Jiva's approach of prioritizing Just-In-Time compilation the *priority approach*, as described below.

1.3.4 Shortest First

If the Java HotSpot framework has more than one hot method to compile at the same time, it compiles them in the order of first-in-first-served. The interpretation of a method continues until its compilation fully finishes. The delay of compilation, while a method is being compiled, in the server mode, is significant since JVM uses heavy optimizations such as graph coloring for register allocation and deep method inlinings [1]. Also, Jiva claims the number of such waiting methods is high in a Java application run, for instance "over 30 on `_213_javac`", which is one of SPEC jvm98 benchmark suites, and "over 60 on `SpecJBB2000`," which is another SPEC benchmark. But Jiva does not cite any reference and we could not verify this claim. Such SPEC benchmarks are standard industry benchmarks to evaluate JVM and servers.

Jiva's Just-In-Time triggering compiles the methods that are already hot in the order of smallest method first. Since the prioritizing increases the rate of the native method production, Jiva maintains, it significantly reduces the extra interpretation overhead.

1.4 Our Two New Approaches

1.4.1 Relative Approach

This approach is to shift some of JIT compilation from application startup and application initialization phase to later execution and to steady state execution. To reduce the amount JIT of compilation at the beginning, the approach doubles the `CompileThreshold` value at the beginning. Since `CompileThreshold` is higher than normal, JVM interprets methods more before it compiles the methods so interpretation increases and compilation decreases.

JVM computes the average of a method's interpretation frequencies each time it is called. This approach does not consider backward branches inside methods. We decided not to develop this feature until we see the performance results in this basic version. When the average reaches a certain amount, JVM divides the `CompileThreshold` by two. This increases the amount of JIT compilation since methods become hot sooner. In this way there will be less JIT compilation at stratup and more JIT compilation furthur on.

1.4.2 Blocksize Approach

Figure 1.7 illustrates how we measure the hotness of a method contained inside loops. Assume method1 in the figure is called 5 times and the loop repeates 2000 times (average 400 times per call). We computes the hotness of the methods as $\frac{1000*5+3*2000}{1000} = 11$, based on Equation (1.5) on page 27, which means each instruction is interpreted in average 11 times, and we consider this average as the interpretation frequency of this method. On the other hand, the Java HotSpot original policy computes the hotness of the method in Figure 1.7 as $5 + 2000 = 2005$! Since the compile threshold (in Intel x86 architecture client mode) is 1500, this computation detects this method as hot and compiles it but this method has 997 instructions that are interpreted for only 5 times and it has only 3 instructions that are interpreted for 2005 times. So, HotSpot compiles the whole method while only 3 instructions out of 1000 instructions are hot and the others are not hot at all.

To make this implementation easier, we use the approximation that the number of instructions in a loop is proportional to the size of the loop. Equation (1.5) on page 27 estimates the average interpretation frequency of instructions in a method, with respect to the assumption mentioned. Since this approach causes the hotness of methods with loops, particularly methods with frequently tiny loops, to hit compile threshold later, it can reduce the amount of JIT computation. So we reduce the

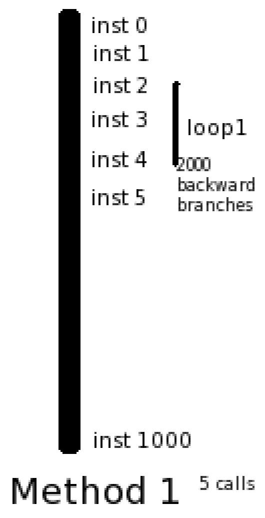


Figure 1.7: a Method with a inside Loop

compile threshold (divided by 2) so that JVM still does about the same amount of JIT computation.

$$hotness = invocation_count \sum_{loops} BranchCount * \frac{loopSize}{methodSize} \tag{1.5}$$

This approach makes the JIT compilation more fair but it may miss some frequent code blocks. For example, in Figure 1.7 this approach misses 3 instructions that are really hot and are interpreted for 2005 times but it prevents JIT compilation of 997 instructions that are not hot at all.

CHAPTER 2

DESIGN

2.1 HotSpot Architecture

2.1.1 Integration with the Interpreter

JVM uses an interpreter that is in assembly language so that both the interpreted, compiled, and native method frames can use the same stack [4]. The HotSpot VM generates and relocates the native code to the positions that execute the byte code. JVM implements object references as direct pointers, thus providing C language speed for instance accesses.

2.1.2 Just-In-Time Queuing

The Just-In-Time compiler makes a data structure, `CompileTask`, for each hot method and puts the data structure in a queue, `CompileQueue` that has a First-In-First-Served policy. Queue is a single linked list of the tasks with a pointer to the head, `_first`, and a pointer to the tail, `_last`.

2.1.3 The Bit Mask of the Counter

JVM, for space reasons, encodes an interpretation frequency counter and its states together in one word, as below. The state is in the least significant bits, and the counter is in the more significant bits. Because of the three non-count bits, JVM shifts the word of a threshold three bits to left before comparing it with the interpretation frequency counter, which will involve the state bits in the comparison as well.

```
Bit no: |31...3|  2  | 1 0 |
                  |
Format: [count |carry|state]
```

The state of a counter indicates the action when either the counter is initialized or it hits its threshold. Even though each counter may have four states, in the current version of HotSpot, there are two states defined. First state of `wait_for_nothing` means do nothing when `count()` $>$ `limit()` and second,state of `wait_for_compile`, which means introduce nmethod (compile the method) when `count()` $>$ `limit()`.

2.2 Method Grouping

This approach uses a counter for the method id generator, a variable for the previous method id, and a fixed length hash structure for the list of the predicted hot methods. We call the hash structure a method group, since it contains the list of the methods that run just prior to the given method. The counter, the variable, and the hash are static data members of a thread data structure. Since the interpreter runs as a thread, it has access to the thread data structure, and to the counter, etc.

The Just-In-Time triggering could not keep track of the methods using their method data structure addresses since the HotSpot environment moves the methods data structures and changes the addresses as it performs garbage collection. The interpreter might also keep track of a method by its handler, which the HotSpot engine updates each time the garbage collection runs. However, using numerical method ids is easier for hashing and retrieving.

2.3 Evaluation of Jiva's Design

Jiva's thesis manipulates the method invocation counter of a predicted hot method in order to trigger the Just-In-Time compilation of the method. It "flips the fourth bit beyond the state and carry" in the invocation counter of a predicted hot method, because he assumes the Just-In-Time compiler considers the setting of the bit as an overflow of the counter and will compile the method [1]. Jiva uses this approach in order to compile a predicted hot method sooner than when the method's invocation counter hits the Just-In-Time compilation threshold.

The fourth bit is nothing more than the least significant bit of the counter, which can change the counter just by one (refer to Section 2.1.3). Setting this bit can only increment the method call count by one, so the HotSpot compiler will continue the interpretation of the method and will not compile the method any sooner than that the method becomes actually hot. We believe Jiva's bit manipulation cannot change the Just-In-Time compilation policy of the HotSpot VM.

CHAPTER 3

IMPLEMENTATION

3.1 HotSpot Source Code

The Assembly interpreter uses a macro assembler, one for each platform, `assembler_i486.hpp` for Intel x86. The macro is in C++ and has a method for each assembly function, which emits the corresponding native code. One method is `movl` whose first argument represents the destination and the second argument, the source. Another example, for the i486 instructions, is `idiv`, which is the signed division operation with one operand, a divisor. It divides `EAX` by the 32bit operand register, stores the quotient in `EAX` (refer to Table 3.1), and puts the remainder in `EDX` [6].

3.2 Just-In-Time Compilation Triggering

We re-implemented Jiva's design, together with our own adjustments on JDK 1.7.0 for Linux Intel x86-32bit-platform using the Assembly interpreter, while Jiva had done with JDK 1.4.2 for the Itanium using the C++ interpreter. In the Assembly interpreter, `interpreter_i486.cpp`, which triggered the dynamic compiling and profiling, we used the `Address` class in the file `assembler_i486.hpp`, which is an abstraction for memory locations in any mode, in order to access the counter variables, threshold variables, and the method group array of a method. We used the base address of the C++ class that represents a method and the offset of the elements

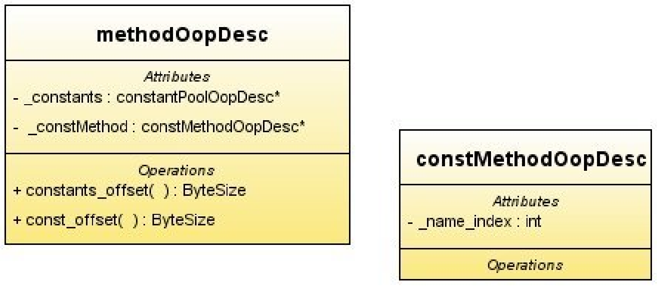

methodOopDesc
Attributes
- _constants : constantPoolOopDesc*
- _constMethod : constMethodOopDesc*
Operations
+ constants_offset( ) : ByteSize
+ const_offset( ) : ByteSize
constMethodOopDesc
Attributes
- _name_index : int
Operations

Figure 3.1: Class Diagram of a Method in C Interpreter of Java HotSpot.

inside the C++ class in the addressing. The C++ class that represents a method is `methodOopDec` and the class pointer is `methodOop`, see Figure 3.1.

We modified the assembly interpreter to generate a method id using a static counter inside the thread structure `JavaThread` in the file `thread.hpp`. Our implementation assigns the id, manipulats the counters, etc. in the interpreter function `InterpreterGenerator::generate_counter_incr` using the macro assembler. We sort the compilation tasks inside the `add` function of the Just-In-Time compilation queue, `CompileQueue`, which is in the file `compileBroker.cpp`,

3.3 Method Size

The size of a method already exists in the HotSpot data structures. Class `methodOopDesc`, which is a derivation of class `oopDesc`, contains a two byte field called `_method_size`.

3.4 Method Counters

The HotSpot source code contains a class `InvocationCounter`, which is used for counting method calls and loops. A method data structure has two fields: One is `_invocation_counter`, which is incremented before a method call, and the other

EAX	Accumulator Register
EBX	Base Register
ECX	Counter Register
EDX	Data Register
ESI	Source Index
EDI	Destination Index
EBP	Base Pointer
ESP	Stack Pointer

Table 3.1: Intel x86 registers

`_backedge_counter`, which is incremented before loop iteration inside the method. However, the method structure has another field called `_compiled_invocation_count` which is the number of invocations of a native method so far. This field is of type `int` and is only for debugging purposes.

To compute the product of a method backward branch count by the quotient loop size to the size of the method, in Blocksize approach, Equation 1.5 on page 27, we added a variable `_backward_branch_length` to the data strucute of a method. In each loop iteration, we add the loop size to this variable so this variable computes the loop size multiplied by its repetition count. We increment the backward branch counter of the loop when the variable is greater than the method size. Then we reduce the size of the method from the variable. In this way, we compute the loop repetition count multiplied by the loop size divided by the method size.

3.5 Tiered Compilation

When we implemented Jiva's Method Group in tiered mode we needed to detect when HotSpot is working in the first level (which is equivalent to client mode) and when it is working in the second level (server mode). It takes the ordinary policy in the first level and the method grouping policy in the second level. Java HotSpot source code already has a method `comp_level()` which gives the

Listing 3.1: C++ Call in Assembly Interpreter

```

#define CAST_FROM_FN_PTR(new_type, func_ptr) ((new_type)((address_word)(func_ptr
)))

typedef u_char*       address;
typedef uintptr_t     address_word; // unsigned integer which will hold a pointe
r
                                    // except for some implementations of a C++
                                    // linkage pointer to function. Should never
                                    // need one of those to be placed in this
_masm->call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::increase_hot
ness_average), rbx);
```

level at which Java HotSpot is running currently. It also has another method of `is_highest_tier_compile()` which takes a number as input and indicates if it is the number of highest level. Since in our approach the higher level is the same as the second level, in which we use the method grouping policy, this works only when `is_highest_tier_compile(comp_level())` is true.

3.6 C++ Code within Assembly

As already mentioned, the HotSpot VM generates an assembly interpreter and then threads the interpreter to run a Java application. Since many programming operations are very inconvenient in assembly, and the source code of HotSpot VM is already in C++, the assembly interpreter frequently calls C++ code.

For instance, in Listing 3.1, the `_masm` object is the C++ macro assembly that generates the assembly interpreter. The `_masm` object is generating the assembly code that calls a C++ method of `increase_hotness_average`. The object is calling its member function `call_VM` and is casting the function pointer of the C++ method `increase_hotness_average` into an unsigned char pointer. So, the `call_VM` is the C++ method that generates the assembly code that, in turn, calls the other C++ method of `increase_hotness_average`.

To switch from Assembly to C++, the assembly interpreter needs to save some of the required registers, since the C++ methods may manipulate the registers ran-

Listing 3.2: Accessing Thread in Assembly

```
void MacroAssembler::get_thread(Register thread) {
  movl(thread, rsp);
  shrl(thread, PAGE_SHIFT);

  ExternalAddress tls_base((address)ThreadLocalStorage::sp_map_addr());
  Address index(noreg, thread, Address::times_4);
  ArrayAddress tls(tls_base, index);

  movptr(thread, tls);
```

domly. The interpreter stores the registers within the structure of the running thread. In Listing 3.2, the first two lines of the method generate corresponding Assembly `mov` and `shr` instructions. JVM has stack page to thread mapping table, which can determine the thread address of the running thread from the page number of the current stack. It gets the stack page number from the `rsp` stack pointer register. Then it computes the address of the thread in to a thread register.

CHAPTER 4

EVALUATION

We performed the benchmarks, mentioned in the introduction, for each approach separately using the benchmarks scripts. For convenience, we also wrote some additional bash scripts that automated the executions of the benchmarks. All the benchmarking was done on a Linux Fedora 9 platform with an Intel Celeron M processor running at 1.5 GHz.

4.1 HotSpot VM Options

HotSpot VM has some specific instance options, starting with -X or -XX, for evaluation and customization purposes. These options are either Boolean, numeric, or string. Boolean options are set off and on with `-XX:-<option>` and `-XX:+<option>`. Numeric options are set with `-XX:<option>=<number>` where the number may include `k` or `K` for kilobytes, `m` or `M` for megabytes, and `g` or `G` for gigabytes. String options are set as `-XX:<option>=<string>`.

The SPECjvm2008 benchmarks also measure performance of hardware processors and memory systems but have little reflection on I/O and no reflection on networking subsystems.

We used the options that printed and logged the compilation tasks and timestamps, Table 4.1. We also modified JVM to print the method size, backward branch

Option	Description
`-XX:-CITime`	Prints Just-In-Time compilation time
`-XX:-PrintCompilation`	Prints the compiled methods
`-XX:+UnlockDiagnosticVMOptions`	
`-XX:+LogCompilation`	Shows the Just-In-Time queuing time and the method time stamps. It shows the method compilation and installation time.
`-XX:CompileThreshold`	Sets invocation threshold.

Table 4.1: Non-standard Options of JVM used for Testing.

count, and invocation count as well, so we could closely trace and verify the Just-In-Time triggering policy. We also added some other options in order to evaluate the implementation with some variant array sizes for the method group hashes and the predication hash.

4.2 IBM Ashes Benchmark

IBM ashes, as mentioned in the paper introduction, is one of benchmark suites that we used in measuring the performance. The benchmark itself includes different suites and each suite includes numerious benchmarks. These benchmarks are developed by differing people as open-source software. Since the number of benchmarks was too many we selected some of the benchmarks from variant suites of Ashes. The explanation of our selection comes in Table 4.2.

We also made some changes to the source code of some of Ashes Benchmarks, mentioned in Table 4.2. We wanted all the benchmarks to have the same number of digits in the elapsed time result. Even though Ashes benchmarks are so many, most of them are tiny benchmarks that last even less than a second and have two digits accuracy, for example 0.23 seconds. So we increased the amount of computation in

Benchmark	Explanation of Benchmark Functionality
JavaSrc	Javasrc creates a set of HTML pages out of some Java source code. The format looks like javadoc.
Kawa C	Kawa compiles some scheme code into bytecode.
factorial	It computes factorial 15324 using Java BigDecimal class
sablecc-j	This is a frozen version of the Sable Compiler. The given run produces the sablecc files (parser, lexer, etc.) for an preliminary version of the jimple grammar.
jpat-p	This is a copy of the Java Protein Analysis Tools.
schroeder-m	This is a copy of Schroeder version 0.2, a sampled audio editing application for the Java platform. It has been equipped with a benchmark harness provided by the author. This run edits a medium-length sound file.
testVirtualCall	It contains lots of virtual method calls.
javazoom	mp3 to wav converter.
probe	It contains filing operations such as searching a file
FFT	It computes fast Fourier transform of complex double precision data.

Table 4.2: The Set of IBM Ashes Benchmarks used in Tests

.

such benchmarks as factorial, testvirtual, and fft. On the other hand, the jpat-p benchmark was too long, above 10 seconds and it had a four-digit value, like 13.23 second. So we reduced the amount of computation in that.

4.3 Evaluation on Jiva's Benchmark Results

(1) Jiva benchmarks the combination of his modifications, the method grouping and priority queue. He does not compare the method grouping process alone with the original version.

(2) Jiva tests the JVM– in the server mode – using SPECJVM98 benchmarks, which include variant software packages such as a java compiler, an MP3 decompress, and a Java parser generator, and using SWINGMARK, which is a benchmark for drawing the Swing components. These applications, particularly the GUI one, are mostly for the personal computing, and proper for JVM in the client mode.

(3) Jiva does benchmark the sever mode using SPEJBB2000, which is the simulator of a three tier commercial system, and he achieves no performance gain, but he still insists on the significance of the speed-up.

4.4 Benchmark Results

4.4.1 Jiva's Approaches

4.4.2 Priority Approach

Jiva's approach of using a priority queue to order the hot methods that are waiting for their compilation gained better performance in all the three benchmark suites of IBM Ashes, Volano Mark, and SPECjvm2008. It improved the average

	Unmodified	Priority	Blocksize	Relative	NoFuncRelative	Grouping
factorial	6.35	5.29	7.32	5.28	5.4	6.03
sablecc-j	2.73	2.72	2.72	3.05	3.07	5.45
jpat-p	5.48	5.46	5.17	5.03	5.02	5.78
schroeder-m	5.83	5.85	5.88	8.28	8.21	6.86
soot-c	4.69	4.66	4.68	5.29	5.27	6.74
testVirtual	3.47	3.54	3.41	3.55	3.52	4.1
probe	3.48	3.52	3.58	3.49	3.42	4.04
fft	5.26	4.61	5.13	5.18	5.2	6.2
Javasrc-p	6.5	6.64	6.64	6.65	6.65	7.8
Kawa-c	8.43	8.57	8.92	9.93	8.89	9.65
Sum	52.22	50.86	53.45	55.73	54.65	62.65
Speedup		0.037	-0.02	-0.07	-0.05	-0.2

Table 4.3: Time of Ashes Benchmark for the Approaches (in seconds, lower better).

performance of Ashes benchmark by 4% (Table 4.3), performance of Volano Mark by 5% (Table 4.5), and performance of SPECjvm2008 by 4% (Table 4.4).

4.4.3 Blocksize

The Blocksize approach is again that JVM computes the average of interpretation frequency of the method instructions including looped and un-looped instructions for measuring method hotness. It gave 3% speedup in Volano Mark and 3% speedup in the composite result of SPECjvm2008 benchmarks. However, it had in average performance reduction in the Ashes benchmarks. Since the SPEC benchmark is the industry standard benchmark particularly for client applications and also Volan Mark is a popular benchmark for server applications, we can announce that this approach is an improvement to JVM. In fact, Ashes benchmarks are not standard benchmarks they were mostly used when SPECjvm98 benchmark was not free and there were no SPECjvm2008.

In fact, even the Ashes benchmarks produce average 1% speedup using Blocksize

	Unmodified	Priority	Relative	NoFuncRelative	Blocksize	Grouping
compiler	13.67	13.79	12.43	12.98	13.73	13.87
compress	11.73	13.06	11.64	12.83	13.25	12.44
crypto	6.97	6.94	6.72	6.65	6.95	7.03
derby	5.39	5.39	4.92	5.19	5.52	5.6
mpegaudio	4.75	4.95	4.87	4.85	4.88	4.95
scimark.large	2.25	2.41	2.39	2.41	2.3	2.39
scimark.small	8.8	10.84	9.08	9.09	10.24	10.39
serial	6.74	6.93	5.67	5.65	7.02	6.93
startup	8.26	7.52	6.98	7.62	7.65	4.93
sunflow	4.98	5.05	4.6	5.02	5.08	5.15
xml	15.86	16.18	14.77	14.69	16.65	5.3
composite result	7.15	7.41	6.77	6.99	7.4	6.43
speed-up		0.04	-0.05	-0.02	0.03	-0.1

Table 4.4: Throughput of SPECjvm2008 Benchmark for the Approaches (operations per minutes, higher better)

	Unmodified	Relative	NoFuncRelative	Blocksize	Priority	Grouping
Average	10480.58	10371	10533.92	10745.33	10856.42	10322
Speedup		-0.01	0.01	0.03	0.05	-0.02

Table 4.5: Throughput of Volano Mark Benchmark for the Approaches (messages per second, higher better)

approach, if we use the same set of benchmarks excluding factorial benchmark. Factorial benchmark is the only one that runs significantly slowly using the approach. We can still announce this approach successful since it works well for most of the benchmarks. This approach is not intended to be perfect as explained in the Analysis section but the idea is to improve performance in average.

In this approach, we set $InvocationLimit = 750$ so threshold of method call count was half the original. We left the threshold of loops to the original amount of $BackwardBranchLimit = 933$. In this way, JVM can compile the methods sooner and run them natively. However, reducing threshold of method call in the original approach would cause too much early compilation since JVM adds both method call count and inside loop counts and compare it with even $InvocationLimit$, and this addition would reach a small method call threshold too soon.

4.4.4 Grouping Approach

For the grouping approach we did not gain any speedup in any benchmark. In fact, we also had some speed reduction in the benchmarks. Since this approach is not our suggestion and the purpose of these evaluations are just to see what results we get by implementing the previous thesis on a different platform, we do not want to comment these results too much.

To even further investigate this approach of grouping, we also set the approach

	Tiered	Tiered-Grouping
Ashes	0.04	0.02
Volano Mark	-0.03	0.05
SPEC	0.04	0.02

Table 4.6: Speedup of Unmodified Tiered Mode and Tiered Grouping Relative to Unmodified Ordinary Mode (not Tiered)

works in Machine Tiered mode. Again, the compiler uses different set of optimizations when it is working in either client mode or server mode. In the tiered mode, it compiles methods for the first time using client mode and then it recompiles the methods that get even hotter using server mode. So our implementation was that the JVM use the ordinary approach in the client phase of the tiered mode and use the grouping approach in the server phase.

Tiered Grouping approach gained some performance improvement compared to ordinary tiered mode in Volano Mark benchmarks. It improved the performance of original tiered mode by 8%, Table 4.6. This was in the condition that the tiered mode was even 3% slower than ordinary mode so the tiered-grouping approach improved the performance 5% better than original ordinary mode and in fact solved the problem of performance flaw in tiered mode for Volano Mark benchmark.

Tiered grouping mode though could not improve the performance for the two other SPECjvm2008 and Ashes benchmarks, Table 4.6. However, the performance reduction was though lower than the grouping approach in ordinary modes. This shows by postponing the approach from initiation phase of an application to its steady state execution, the approach gives better results. Also the performance reduction was not heavy either. So tiered-grouping approach can be useful since it significantly improves sever applications and do not harm client applications too much.

4.5 Relative Approach

The Relative approach did not have any speedup using any of benchmark suites. It even reduced the performance of all the benchmarks. We modified the approach so it runs with the same amount of computation but the bottom line is the same as the unmodified approach. This modification even reduced the amount of performance penalty in the benchmarks of Ashes and SPECjvm2008. Refer to Table 4.3 and Table 4.4, in which the columns named NoFuncRelative means the neutralized Relative approach that still has the same amount of computation.

The reason of unsuccessfulness of this approach is that the JIT compilation should spread-out throughout the whole execution so the overhead should not delay any particular execution point too much. When the approach postpones the JIT compilation, particularly at the point of transition, too much compilation may happen together and that will interrupt with normal execution of the application.

Figure 4.1 summarizes all the evaluation results in the previous sections in a graph. The graph shows speedup percentage of each approach compared to the ordinary approach.

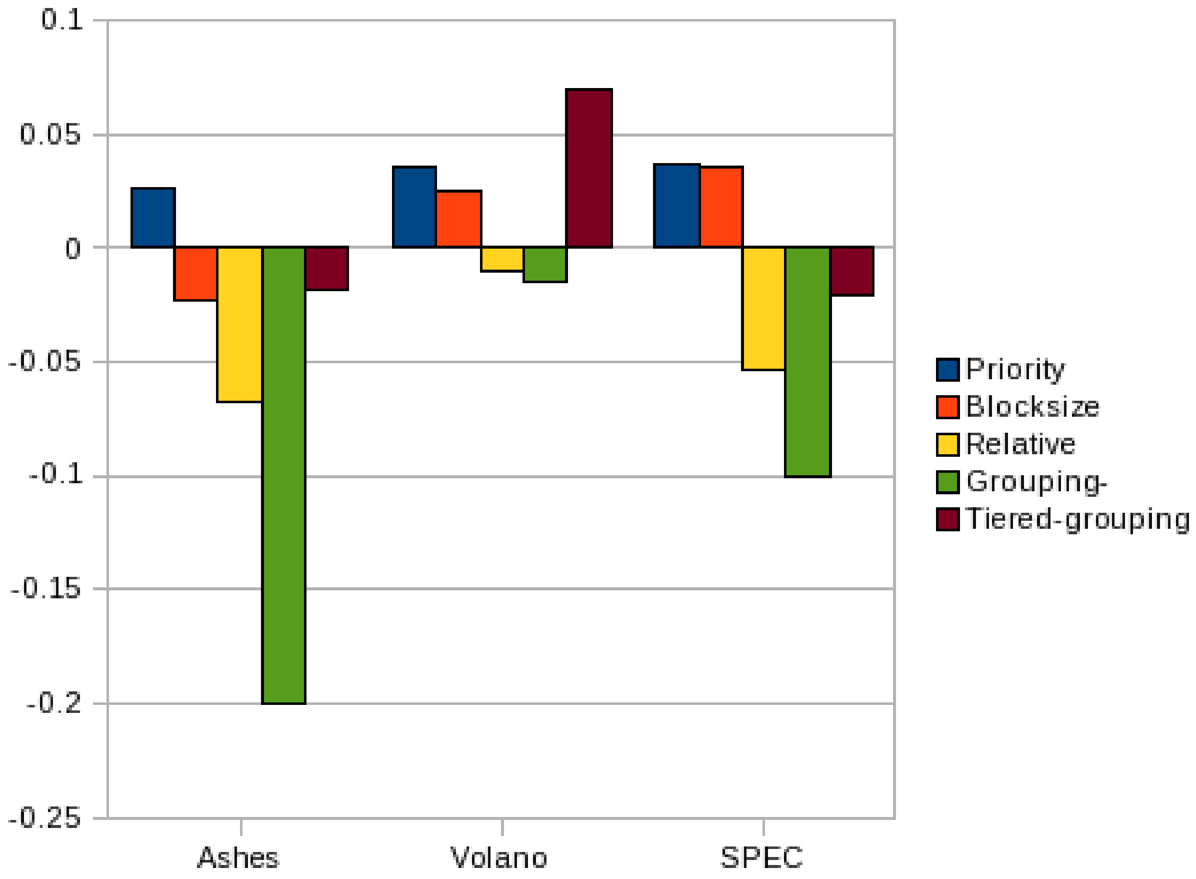


Figure 4.1: Average Speed of the Approaches using the Benchmarks Suites.

CHAPTER 5

RELATED WORK

5.1 Mixed Mode Interpretation (MMI) in the IBM JVM

The IBM VM employs a combination of Java interpreter and Just-In-Time compiler with three levels of optimizations [8]. The interpreter starts running Java software. The first level of the Just-In-Time compiler detects and compiles the frequently interpreted methods with limited optimizations. The second level re-compiles the again frequently executed methods with all the optimizations, and the third level performs optimization of specialization (refer to Appendix 1.1.7) on the most frequently executed methods.

JVM counts the method calls and the loop iterations, but the interpreter, during the first level compilation, may predict a loop iteration count by analyzing the byte code. If the loop count is large enough, the Just-In-Time compilation of the code block will happen immediately. After a method is compiled, its call count is reset to zero [26]. So the call count will re-increment in the next calls until it reaches the second level threshold and then it gets re-compiled.

The Just-In-Time engine constructs the call graphs of the compiled methods that become hot again in order to prevent redundant compilation. It re-compiles only the methods at the graph roots and inline the methods in the sub graphs in the root methods. This approach prevents duplicate compilation of the subgraph methods

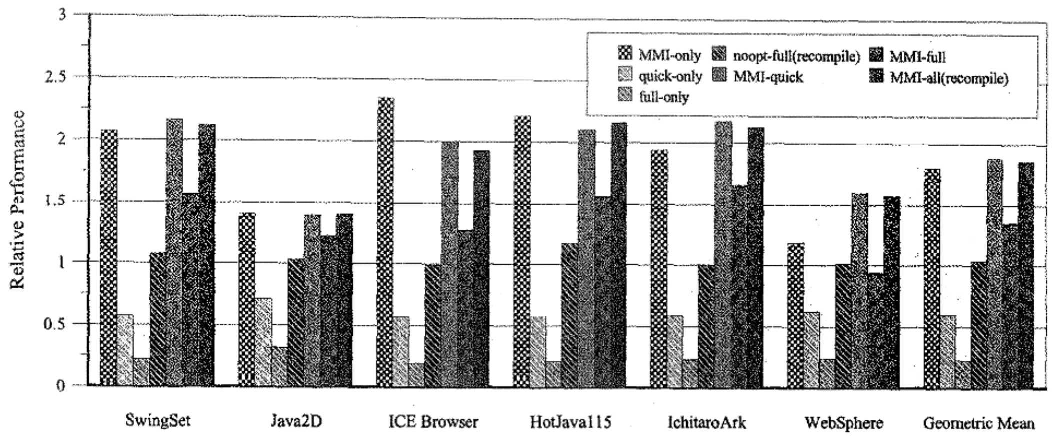


Figure 5.1: Startup Performance of GUI-Based Applications for IBM Just-In-Time in Variant Modes. The Bars Indicate the Execution Speed Relative to the Compile-Only Mode without any Optimization (the higher bars, the better). The Initialization Speed of the Multi-Level Just-In-Time Compilation is as fast as the Startup of the Interpretation-Only [8]. The approaches in this graph are indicated in Table 5.1

Alias	Approach
MMI-only	Interpretation-Only
noopt-only	No optimization compilation with no MMI
quick-only	Quick optimization compilation with no MMI
full-only	Full optimization compilation with no MMI
noopt-full	No optimization compilation with no MMI and recompilation using full optimizatio
MMI-quick	Quick optimization compilation with MMI
MMI-full	Full optimization compilation with MMI
MMI-all	All levels of compilation with MMI for adaptive recompilation

Table 5.1: Alias Names of the Approaches that are Evaluated in Figure 5.1.

that are already inlined in the root method and are already compiled within the root method.

During the second-level re-compilation, the Just-In-Time compiler inserts, in the beginning of an old method code, a branch to a compensation code that updates the direct call to the deprecated code. The compensation code finds the address of the direct call from the return address on the stack and then patches the call.

The chart in Figure 5.1 indicates the startup performance of the Just-In-Time environment in the interpretation-only mode, interpretation-compilation-mixed mode, and compile-only mode. The bars are the time from the execution command until the first window of the application pops up relative to the compile-only mode with no optimization. Table 5.1 lists and describes all the modes. The results indicate the interpretation-only and the mixed mode approaches have significantly better initialization than the compile only approaches. The chart in Figure 5.2 indicates that the Mixed Mode Interpretation (MMI) policies also generate code that is magnitudes smaller than the compile-only approaches.

Other benchmarks, including SPECjvm98 and SPECjbb2000, show that both the compile-only mode and the mixed mode behave similarly in steady-state execution, but interpretation-only mode performs far poorer, as expected. The code growth results, in the long-run execution, are parallel to the start-up evaluation, n which the compile-only approach has a large memory footprint.

5.2 Smart Just-In-Time

Plezbert et al. [21] introduces a Just-In-Time implementation that is based on the C language and either only interprets the code or just compiles and executes it natively. The framework takes a file of C methods and, based on the file size,

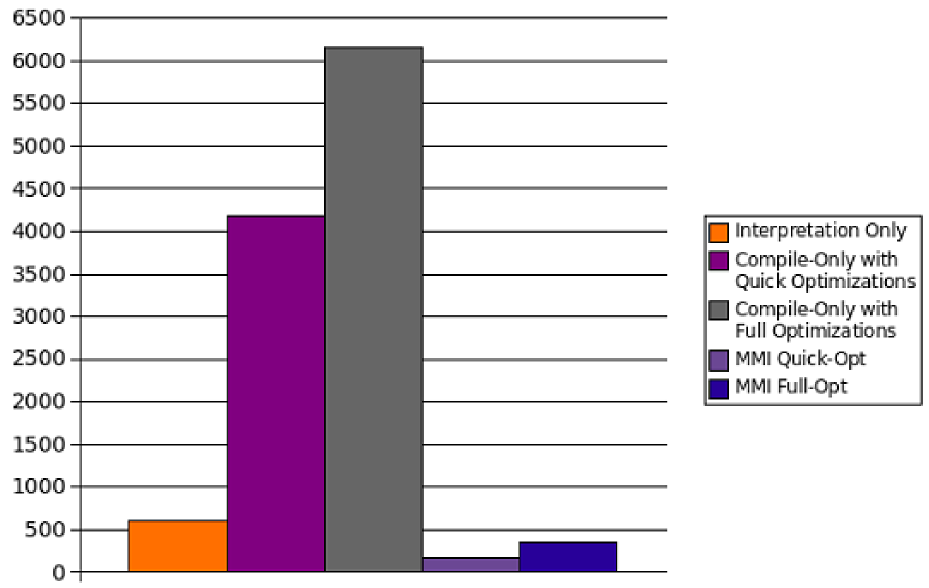


Figure 5.2: Size of the Just-In-Time Code Generation at the Startup of Application Websphere (in KB). Interpretation-Only Indicates Bytecode Size of the Interpreted Methods [8].

estimates whether the compilation of the whole file is worth it. The Just-In-Time engine predicts the duration of the file compilation from the file size using the formula (5.1), which is the closest-fit quadratic of some application sampling data. The Just-In-Time engine considers a file worth compilation if the compile time plus the native execution time is less than the interpretation time, Formula (5.2), where I is the interpretation time, T the execution time, and C the compilation time. Assuming the ratio of the interpretation to the execution is constant P, Equation (5.3) on page 50, is the simplification of Equation (5.2) on page 50.

$$y = 0.000066x^2 + 1531 \tag{5.1}$$

$$I = P * T > C + T \tag{5.2}$$

$$I > \frac{P * C}{P - 1} \tag{5.3}$$

Plezbert et al. implements two other Just-In-Time compilation approaches to compare them with the above approach: first, a compile-only approach that translates each method into native code before running it, second, a combination of interpretation and continuous Just-In-Time compilation, in which one processor just runs the code and the other only compiles. The compiler processor continuously selects the file of methods that has collectively taken the longest interpretation time so far. It takes the statistics for a file as a whole, instead of considering a single routine, and then translates all the methods in the file. So this approach compiles the most time consumptive files, not necessarily the most time consumptive individual routines. The executor processor does not get to the native code until the whole file compilation finishes, and it starts the native execution of a method by jumping to the native code

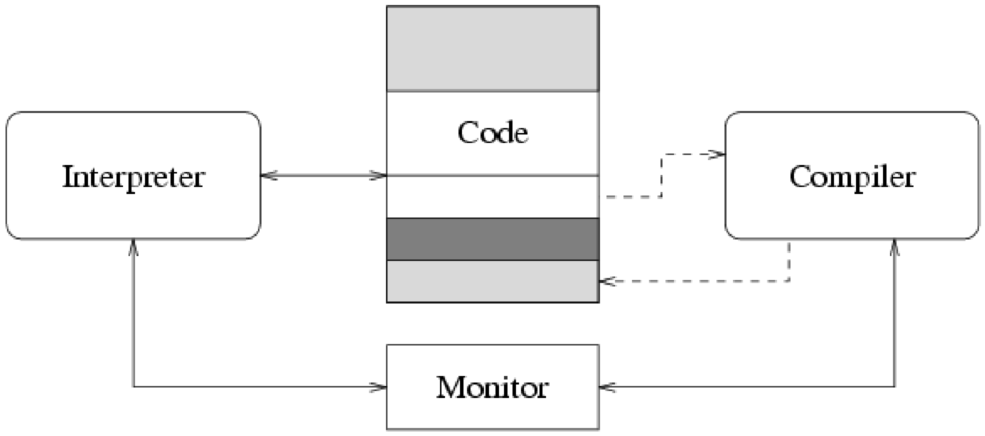


Figure 5.3: Mixed Mode Interpretation and Just-In-Time Environment with Two Processors of Interpreter and Compiler [21].

at the method call. Figure 5.3 illustrates the continuous Just-In-Time compilation, in which both the executor and the compiler processors communicate with each other through the Monitor data structure.

The compilation time prediction approach outperformed the compile-only mode, as expected, but could not gain better performance than the continuous compilation policy. The advantage of the mixed mode policy may have been due to its multi-processing architecture although the prediction approach had a more sophisticated heuristic.

5.3 Compile-only Approaches

Just-In-Time compilation happens immediately, without any interpretation, in the Intel JUDO system [20]. The compilation has two phases, first, fast code generation for the running bytecode, and second, optimizations for the frequently executed code. Jalapeno is another research Java dynamic compiler, which is itself in Java, with a compile-only approach [23] [24]. It has a baseline as the first compilation level and three more level of optimizations.

5.4 Compilation of Code Blocks

Toshio et. al. implements a Just-In-Time triggering that compiles only the code blocks of the frequently iterated loops and not the whole methods that contain the loops [11]. This approach may more precisely adapt the optimizations for the frequently interpreted bytecode. Code block compilation allows more aggressive inlinings and achieves average 5% speed-up [11].

5.5 JRuby

JRuby 1.1.2 has a Just-In-Time max that is the maximum number of methods that it may compile. The current default of the Just-In-Time max is 2048. JRuby also has a threshold for Just-In-Time compilation. The current default of the threshold is 20. There is debate to change these respectively to 4096 and 50 [25] .

5.6 Different Thresholds

Hickson [27] claims that in a "transactional environment" where the JVM responds to transactions from the client systems, a couple of methods and modules run repeatedly, causing many methods to hit the compilation threshold together. Hickson claims that this unduly increases the delay for software execution due to excessive compilation.

The proposed system uses the Euqation (5.4) on page 53, to initialize the invocation count of each method. Each method-call decreases the invocation count of the method and the Just-In-Time compilation of the method starts when the counter reaches zero Figure 5.4. The system initializes the invocation counter upon the first call of the method and increments the offset using modular addition, which resets the offset when it gets above the maximum offset.

$$JustInTimeCount = Threhsold + Offset \tag{5.4}$$

The system adjusts the offsets so that large methods do not have the same thresholds. The system calculates the average size of the methods as they are called and stores it. An array keeps track of each offset and size of the last method that used that offset Figure 5.5. If the size of a new method is above the average, the system uses a given offset only if its previous corresponding method size, which is already in the array, is less than the stored average. If the method size in front of the current offset in the array is above the average, the system increments the offset, using modular addition, and checks the next offset until it finds an offset whose size in the array is below or equal to the average.

5.7 Future Work

The priority method we used did not completely sort the compilation queue but only partially ordered it to avoid extra computation. HotSpot queues the hot methods for Just-In-Time Compilation using a linked list. Instead of a linked list, an array can be used to have the priority queue completely sort the list and have it still work fast.

The blocksize approach finds the methods that are on average the most frequent, but it still may miss some of the frequent inner loops. HotSpot can instead compile only these loops and integrate the native loop within the byte code of the method.

In order to predict the most frequent methods, another approach is to cache the analysis of Just-In-Time compilation in a file for later executions of the program. So the next time the application runs, HotSpot can take advantage of its former analysis and predict the methods that were already hot in the former execution to be hot

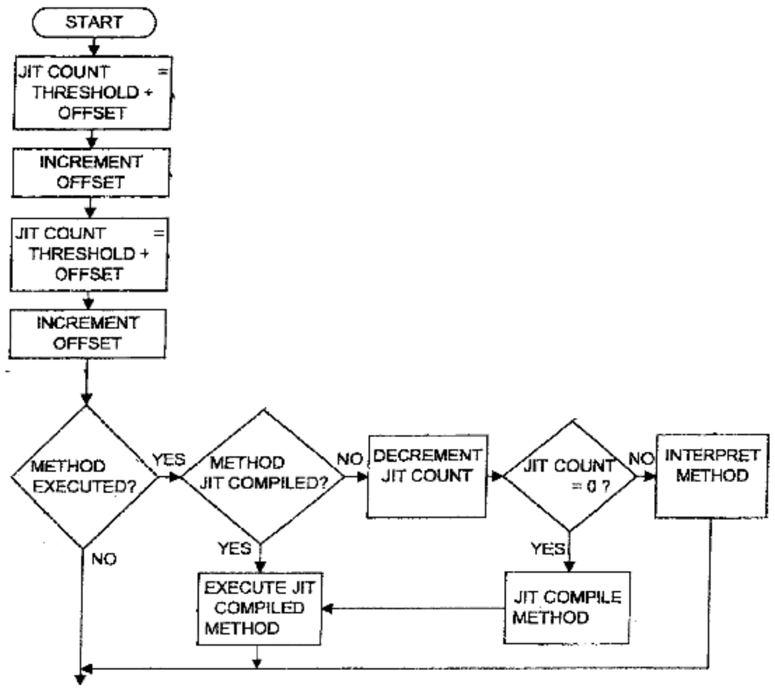
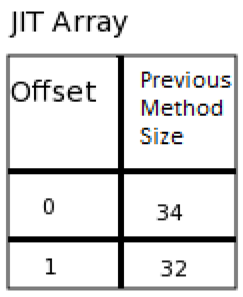


Figure 5.4: Flowchart of Initializing and Adjusting Invocation Counts of the Methods [27].


JIT Array

Offset	Previous Method Size
0	34
1	32

Figure 5.5: Offsets and the Sizes of the Corresponding Methods [27].

methods in the current execution.

5.8 Conclusion

We recommend the two approaches Priority (of Jiva's) and Blocksize (of ours) and do not recommend the relative approach. We also recommend the Tiered Grouping approach for server applications. Definitely the policy of detecting the hot methods and scheduling their compilation affects the overall performance. But what is the most important is that the policy be fast and light weight.

BIBLIOGRAPHY

[1] Jiva, A. S. Compilation Scheduling for the Java Virtual Machine. Computer Science Department, San Jose Sate University, Masters Degree Thesis. December 2004.

[2] Raymond, E.S. (1999). The Cathedral & the Bazaar. O'Reilly Retrieved July 22, 2009 from http://www.catb.org/ẽsr/writings/cathedralbazaar/

[3] Kenneth Russell Sun Developer forums http://forum.java.sun.com/

[4] Sun Microsystems, Inc. The Java HotSpot Performance Engine Architecture. White paper available at `http://java.sun.com/products/hotspot/whitepaper.html`, June 2007.

[5] Tim Lindholm, Frank Yellin. The JavaTM Virtual Machine Specification, Second Edition

[6] R. Hyde. The art of assembly language. Published 2005. NO STARCH PRESS. ISBN 1886411972

[7] Dever, S., Goldman, S. & Russel, K. New Compiler Optimizations in Java HotSpot Virtual Machine. Sun Microsystem, Inc. 2006 JavaOne conference. TS-3412.

[8] Suganumi, T., Yasue, T. Kawahito, M., Kornatsu, H., Nakatani, T. (2001). A Dynamic Optimization Framework for a Java Just-in-Time Compiler. In L. Northrop (Ed.), Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (p. 180 -195). New York: ACM Press.

[9] Todd Smith, Suresh Srinivas. Experiencies with Retargeting the Java HotSpot TM Virtual Machine. High Performance Programming Environments, Silicon Graphics, Inc. MS 178, 1600 Amphitheatre Parkway, Mountain View, CA 94043

[10] Tom Rodriguez, Ken Russell. Client Compiler for Java HotSpot Virtual Machine Technology. Sun Microsyst ,Inc. 2002.

[11] Toshio, S., Toshiaki, Y., & Toshio, K. A region-based compilation technique for a Java just-in-time compiler.IBM Tokyo Research Laboratory, 1623-14 Shimotsuruma, Yamato-shi, 242-8502, Japon.

[12] L Baraz, T Devor, O Etzion, S Goldenberg.IA-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications. 36th International Symposium on Microarchitecture, 2003 - ptlsim.oR

[13] Dustin Clingman, Shawn Kendall, Syrus Mesdagh.Practical Java Game Programming. Published 2004 Charles River Media.

[14] T Cramer, R Friedman, T Miller, D Seberger. Compiling Java just in time. Micro, IEEE, 1997.

[15] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, Toshio Nakatani. A Study of Devirtualization Techniques for a Java TM Just-In-Time Compiler. IBM Research, Tokyo Research Laboratory, Japan.

[16] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction, In Proceedings of the Conference on Object Oriented Programming Systems, Languages & Applications, OOPSLA 1995, pp. 108-123, 1995.

[17] Aigner, G. Holzle, U. Eliminating Virtual Function Calls in C++ Programs, In Proceedings of the 10th European Conference on Object-Oriented Programming " ECOOP 1996, volume 1098 of Lecture Notes in Computer Science, Springer-Verlag, pp. 142-166, 1996.

[18] Detlefs, D. and Agesen, O. 1999. In-lining of Virtual Methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming* (June 14 - 18, 1999). R. Guerraoui, Ed. Lecture Notes In Computer Science, vol. 1628. Springer-Verlag, London, 258-278.

[19] Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., et al. (1999). Design, implementation, and evaluation of optimizations in a just-in-time compiler. In G. Fox, K. Schauser & M. Snir (Eds.), Proceedings of the ACM 1999 conference on Java Grande (p. 119-128). New York: ACM Press.

[20] M. Ceirniak, G.Y.Lueh, J.M.Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Desing and Implementation*, pp. 13-26, Jun. 2000.

[21] Plezbert, M. P. and Cytron, R. K. 1997. Does j̈ust in time≐b̈etter late than never?̈ In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France, January 15 - 17, 1997). POPL '97. ACM, New York, NY, 120-131. DOI= http://doi.acm.org/10.1145/263699.263713

[22] Patterson, D. A.,& Hennessy, J. L. (2005). *Computer Organization and Design: The Hardware/Software Interface*. San Francisco, CA: Elsevier.

[23] Arnold, M., Fink, S., Grive, D., Hind, M., & Sweeney, P.F. (2000). Adpative Optimizations in the Jalapeno JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications* (pp.47 - 65). New York, NY: ACM.

[24] rnold, M., Fink, S., Grive, D., Hind, M., & Sweeney, P.F. (2000). Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices, 35*(10), 47-56.

[25] Just-In-Time max and Just-In-Time threshold should be adjusted for improvements in JRuby over the past months. [jruby-dev] [jira]. Retrieved December 28, 2008 from http://osdir.com/ml/lang.jruby.devel/2008-05/msg00078.html.

[26] IBM Developer Kit and Runtime Environment, Java 2 Technology Edition, Version 1.4.2, Java Diagnostics Guide 1.4.2 Win64AMD, Twelfth Edition (May 2008).

[27] Hickson, P. M. (2006). Just in Time Compilation of Java Software Methods. Patent No.: US 7,089,544 B2.

[28] Louden, K. C. (1997). *Compiler Construction Principles and Practice*. Boston, MA: PWS Publishing Company.

[29] Standard Performance Evaluation Corporation. (2008). *SPECjvm2008 User's Guide*. Copyright 2008 Standard Performance Evaluation Corporation. Section 1.1

[30] McGill. (2009). Ashes: a Java Compiler Infrastructure. Avalable at `http://www.sable.mcgill.ca/ashes/#support`. Accessed April 25, 2009.

[31] Plamondon, S. (1999). The need for speed, stability. from JavaWorld.com, retrieved 10/01/99.

[32] Brad Calder, Peter Feller, Alan Eustace, "Value Profiling," Microarchitecture, IEEE/ACM International Symposium on, pp. 259, 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97), 1997.

APPENDIX A

SOURCE CODE

The followings are the diff files of our implementation on Java(TM) SE Run-time Environment (build 1.7.0-ea-b38) for the Linux Fedora 9, Intel x86 32bit platform.

A.1 Priority

```
Only in /home/rgougol/cs298/build/priority-openjdk/hotspot/src: priority-.diff
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp /home/rgougol/cs298/build/priority-openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp     2009-06-29 14:17:02.000000000 +0430
+++ /home/rgougol/cs298/build/priority-openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp 2009-07-13 21:39:41.000000000 +0430
@@ -181,6 +181,7 @@

   _compile_id = compile_id;
   _method = JNIHandles::make_global(method);
+  _code_size = method->code_size();
   _osr_bci = osr_bci;
   _is_blocking = is_blocking;
   _comp_level = comp_level;
@@ -435,8 +436,23 @@
   } else {
     // Append the task to the queue.
     assert(_last->next() == NULL, "not_last");
-    _last->set_next(task);
-    _last = task;
+    if (task->code_size() <= _first->code_size())
+    {
+     task->set_next(_first);
+     _first = task;
+    }
+    else if (task->code_size() <= _first->code_size() << 1)
+    {
+      task->set_next(_first->next());
+      _first->set_next(task);
+      if (_last == _first)
+        _last = task;
+    }
+    else
+    {
+      _last->set_next(task);
+      _last = task;
+    }
   }

   // Mark the method as being in the compile queue.
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/compiler/compileBroker.hpp /home/rgougol/cs298/build/priority-openjdk/hotspot/src/share/vm/compiler/compileBroker.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/compiler/compileBroker.hpp     2009-06-29 14:17:02.000000000 +0430
```

```
+++ /home/rgougol/cs298/build/priority-openjdk/hotspot/src/share/vm/compiler/compileBroker.hpp
      2009-07-13 16:59:30.000000000 +0430
@@ -34,6 +34,7 @@
   Monitor*      _lock;
   uint          _compile_id;
   jobject       _method;
+  int           _code_size;
   int           _osr_bci;
   bool          _is_complete;
   bool          _is_success;
@@ -61,6 +62,7 @@
   void free();

   int           compile_id() const                 { return _compile_id; }
+  int           code_size() const                  { return _code_size; }
   jobject       method_handle() const              { return _method; }
   int           osr_bci() const                    { return _osr_bci; }
   bool          is_complete() const                { return _is_complete; }
```

A.2 Blocksize

```
 diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/c1_globals_x86.hpp /home/
      rgougol/cs298/build/blocksize-openjdk/hotspot/src/cpu/x86/vm/c1_globals_x86.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/c1_globals_x86.hpp 2009-06-29
      14:16:58.000000000 +0430
+++ /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/cpu/x86/vm/c1_globals_x86.hpp
      2009-06-29 14:42:35.000000000 +0430
@@ -36,7 +36,7 @@
 define_pd_global(bool, ProfileTraps,                  false);
 define_pd_global(bool, UseOnStackReplacement,         true );
 define_pd_global(bool, TieredCompilation,             false);
-define_pd_global(intx, CompileThreshold,              1500 );
+define_pd_global(intx, CompileThreshold,              750);
 define_pd_global(intx, Tier2CompileThreshold,         1500 );
 define_pd_global(intx, Tier3CompileThreshold,         2500 );
 define_pd_global(intx, Tier4CompileThreshold,         4500 );
 diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/interp_masm_x86_32.cpp /home/
      rgougol/cs298/build/blocksize-openjdk/hotspot/src/cpu/x86/vm/interp_masm_x86_32.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/interp_masm_x86_32.cpp      2009-06-29
      14:16:58.000000000 +0430
+++ /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/cpu/x86/vm/interp_masm_x86_32.cpp
      2009-06-29 14:42:35.000000000 +0430
@@ -1175,6 +1175,19 @@
     //increment_mdp_data_at(mdp, in_bytes(JumpData::taken_offset()));
     Address data(mdp, in_bytes(JumpData::taken_offset()));

+    Label skip_count;
+    push(rdi);
+    testl(rdx, rdx);                              // forward branch or backward branch?
+    jcc(Assembler::positive, skip_count);         // Count only if backward branch
+    //  increment counter
+    subl(Address(rcx, methodOopDesc::backward_branch_length_offset()), rdx);
+    incrementl(Address(rcx, methodOopDesc::backward_branch_length_offset()));
+    movl(rdi, Address(rcx, methodOopDesc::const_offset()));
+    movw(rdi, Address(rdi, constMethodOopDesc::code_size_offset()));    // load method code size
+    andl(rdi, 0x0000FFFF);
+    cmp32(rdi, Address(rcx, methodOopDesc:: backward_branch_length_offset()));
+    jcc(Assembler::above, skip_count);
+
     // %%% 64bit treats these cells as 64 bit but they seem to be 32 bit
     movl(bumped_count,data);
     assert( DataLayout::counter_increment==1, "flow-free idiom only works with 1" );
@@ -1182,8 +1195,10 @@
     sbbl(bumped_count, 0);
     movl(data,bumped_count);     // Store back out

+    bind(skip_count);
     // The method data pointer needs to be updated to reflect the new target.
     update_mdp_by_offset(mdp, in_bytes(JumpData::displacement_offset()));
+    pop(rdi);
     bind (profile_continue);
   }
 }
@@ -1193,7 +1208,7 @@
   if (ProfileInterpreter) {
     Label profile_continue;

-    // If no method data exists, go to profile_continue.
+    // share/vm/opto/parseHelper.cppIf no method data exists, go to profile_continue.
     test_method_data_pointer(mdp, profile_continue);

     // We are taking a branch.  Increment the not taken count.
 diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_32.cpp /home
      /rgougol/cs298/build/blocksize-openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_32.cpp
```

```
--- /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_32.cpp   2009-06-29 14:16:58.000000000 +0430
+++ /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_32.cpp 2009-06-29 14:42:35.000000000 +0430
@@ -1491,19 +1491,18 @@


 void TemplateTable::branch(bool is_jsr, bool is_wide) {
-  __ get_method(rcx);            // ECX holds method
-  __ profile_taken_branch(rax,rbx); // EAX holds updated MDP, EBX holds bumped taken count
-
-  const ByteSize be_offset = methodOopDesc::backedge_counter_offset() + InvocationCounter::counter_offset();
-  const ByteSize inv_offset = methodOopDesc::invocation_counter_offset() + InvocationCounter::counter_offset();
-  const int method_offset = frame::interpreter_frame_method_offset * wordSize;
-
   // Load up EDX with the branch displacement
   __ movl(rdx, at_bcp(1));
   __ bswapl(rdx);
   if (!is_wide) __ sarl(rdx, 16);
   LP64_ONLY(__ movslq(rdx, rdx));

+  __ get_method(rcx);            // ECX holds method
+  __ profile_taken_branch(rax,rbx); // EAX holds updated MDP, EBX holds bumped taken count
+
+  const ByteSize be_offset = methodOopDesc::backedge_counter_offset() + InvocationCounter::counter_offset();
+  const ByteSize inv_offset = methodOopDesc::invocation_counter_offset() + InvocationCounter::counter_offset();
+  const int method_offset = frame::interpreter_frame_method_offset * wordSize;

   // Handle all the JSR stuff here, then exit.
   // It's much shorter and cleaner than intermingling with the
@@ -1544,11 +1543,21 @@
     __ testl(rdx, rdx);             // check if forward or backward branch
     __ jcc(Assembler::positive, dispatch); // count only if backward branch

-    // increment counter
+    //load code size
+    __ movl (rax, Address(rcx, methodOopDesc::const_offset()));
+    __ movw (rax, Address(rax, constMethodOopDesc::code_size_offset()));
+    __ andl (rax, 0x0000FFFF);
+    __ cmp32(rax, Address(rcx, methodOopDesc::backward_branch_length_offset()));
+    Label not_yet_increment_backedge;
+    __ jcc(Assembler::above, not_yet_increment_backedge);
+    //__ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::trace_method));
+    __ subl(Address(rcx, methodOopDesc::backward_branch_length_offset()), rax);
+
     __ movl(rax, Address(rcx, be_offset));        // load backedge counter
     __ incrementl(rax, InvocationCounter::count_increment); // increment counter
     __ movl(Address(rcx, be_offset), rax);        // store counter

+    __ bind(not_yet_increment_backedge);
     __ movl(rax, Address(rcx, inv_offset));    // load invocation counter
     __ andl(rax, InvocationCounter::count_mask_value);     // and the status bits
     __ addl(rax, Address(rcx, be_offset));        // add both counters
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp   2009-06-29 14:17:02.000000000 +0430
+++ /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp 2009-06-29 14:42:35.000000000 +0430
@@ -281,8 +281,8 @@
   // print osr_bci if any
   if (is_osr) st->print("_@_%d", osr_bci());

-  // print method size
-  st->print_cr("_(%d_bytes)", method->code_size());
+  // print method size, backward branch length
+  st->print_cr("_(%d_bytes_+_%d_bytes)", method->code_size(), method->backward_branch_length());
 }

 // ------------------------------------------------------------------
@@ -319,7 +319,7 @@
   if (is_osr) tty->print("_@_%d", osr_bci());

   // print method size
-  tty->print_cr("_(%d_bytes)", method->code_size());
+  tty->print_cr("_(%d_bytes_+_%d_bytes)", method->code_size(), method->backward_branch_length());
 }


 diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp
```

```
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp 2009-06-29 14:17:06.000000000 +0430
+++ /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp 2009-06-29 14:42:35.000000000 +0430
@@ -505,6 +505,11 @@
   ObjectSynchronizer::trace_locking(h_locking_obj, false, true, is_locking);
 }

+IRT_ENTRY(void, InterpreterRuntime::trace_method(JavaThread* thread))
+  ResourceMark rm;
+  methodOop method = thread->last_frame().interpreter_frame_method();
+  tty->print_cr("%s size: %d looped: %d invocation#: %d backedge#: %d", method->name_and_sig_as_C_string(), method->code_size(), method->backward_branch_length(), method->invocation_counter()->count(), method->backedge_counter()->count());
+IRT_END

 //%note monitor_1
 IRT_ENTRY_NO_ASYNC(void, InterpreterRuntime::monitorenter(JavaThread* thread, BasicObjectLock* elem))
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp 2009-06-29 14:17:06.000000000 +0430
+++ /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp 2009-06-29 14:42:35.000000000 +0430
@@ -102,6 +102,9 @@
   static void post_method_exit (JavaThread *thread);
   static int  interpreter_contains(address pc);

+  //Blocksize-
+  static void trace_method(JavaThread*);
+
   // Native signature handlers
   static void prepare_native_call(JavaThread* thread, methodOopDesc* method);
   static address slow_signature_handler(JavaThread* thread,
Only in /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter: .invocationCounter.hpp.swp
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/constMethodOop.hpp /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/oops/constMethodOop.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/constMethodOop.hpp      2009-06-29 14:17:06.000000000 +0430
+++ /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/oops/constMethodOop.hpp 2009-06-29 14:42:35.000000000 +0430
@@ -264,6 +264,7 @@
   // Offset to bytecodes
   static ByteSize codes_offset()
                             { return in_ByteSize(sizeof(constMethodOopDesc)); }
+  static ByteSize code_size_offset() { return byte_offset_of(constMethodOopDesc, _code_size); }

   // interpreter support
   static ByteSize exception_table_offset()
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/methodKlass.cpp /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/oops/methodKlass.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/methodKlass.cpp 2009-06-29 14:17:06.000000000 +0430
+++ /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/oops/methodKlass.cpp 2009-06-29 14:42:35.000000000 +0430
@@ -60,6 +60,7 @@
   m->set_constMethod(xconst());
   m->set_access_flags(access_flags);
   m->set_method_size(size);
+  m->init_backward_branch_length();
   m->set_name_index(0);
   m->set_signature_index(0);
 #ifdef CC_INTERP
@@ -78,7 +79,6 @@
   m->set_highest_tier_compile(CompLevel_none);
   m->set_adapter_entry(NULL);
   m->clear_code(); // from_c/from_i get set to c2i/i2i
-
   if (access_flags.is_native()) {
     m->clear_native_function();
     m->set_signature_handler(NULL);
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/methodOop.hpp /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/oops/methodOop.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/methodOop.hpp   2009-06-29 14:17:07.000000000 +0430
+++ /home/rgougol/cs298/build/blocksize-openjdk/hotspot/src/share/vm/oops/methodOop.hpp 2009-06-29 14:42:35.000000000 +0430
@@ -110,6 +110,7 @@
   u2                _number_of_breakpoints;      // fullspeed debugging support
   InvocationCounter _invocation_counter;         // Incremented before each activation of the method - used to trigger frequency-based optimizations
   InvocationCounter _backedge_counter;           // Incremented before each backedge taken - used to trigger frequencey-based optimizations
+  u4                _backward_branch_length;  // Length of code iteration in backward branch
```

```
 #ifndef PRODUCT
   int                _compiled_invocation_count;  // Number of nmethod invocations so far (for perf. debugging)
 #endif
@@ -278,6 +279,8 @@
   InvocationCounter* backedge_counter()          { return &_backedge_counter; }
   int invocation_count() const                   { return _invocation_counter.count(); }
   int backedge_count() const                     { return _backedge_counter.count(); }
+  u4 backward_branch_length() const              { return _backward_branch_length; }
+  void init_backward_branch_length()             { _backward_branch_length = 0;}
   bool was_executed_more_than(int n) const;
   bool was_never_executed() const                { return !was_executed_more_than(0); }

@@ -485,6 +488,7 @@
   static ByteSize code_offset()                  { return byte_offset_of(methodOopDesc, _code); }
   static ByteSize invocation_counter_offset()    { return byte_offset_of(methodOopDesc, _invocation_counter); }
   static ByteSize backedge_counter_offset()      { return byte_offset_of(methodOopDesc, _backedge_counter); }
+  static ByteSize backward_branch_length_offset() { return byte_offset_of(methodOopDesc, _backward_branch_length); }
   static ByteSize method_data_offset()           {
     return byte_offset_of(methodOopDesc, _method_data);
   }
```

A.3 Relative

```
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp /home/rgougol/cs298/build/relative-openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp 2009-06-29 14:16:58.000000000 +0430
+++ /home/rgougol/cs298/build/relative-openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp 2009-07-11 23:01:53.000000000 +0430
@@ -327,6 +327,18 @@
   // Update standard invocation counters
   __ movl(rax, backedge_counter);               // load backedge counter

+  Label AlreadyCounted;
+  Label AverageEnd;
+  __ cmpl(rcx, 1);
+  __ jcc(Assembler::notEqual, AlreadyCounted);
+  __ incrementl(ExternalAddress((address)&InterpreterRuntime::method_count));
+  //__ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::count_new_method), rbx);
+  __ jmp(AverageEnd);
+  __ bind(AlreadyCounted);
+  __ incrementl(ExternalAddress((address)&InvocationCounter:: average));
+  __ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::increase_hotness_average), rbx);
+  __ bind(AverageEnd);
+
   __ incrementl(rcx, InvocationCounter::count_increment);
   __ andl(rax, InvocationCounter::count_mask_value);  // mask out the status bits

Only in /home/rgougol/cs298/build/relative-openjdk/hotspot/src/cpu/x86/vm: .templateInterpreter_x86_32.cpp.swp
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp 2009-06-29 14:17:06.000000000 +0430
+++ /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp 2009-07-11 22:58:36.000000000 +0430
@@ -56,6 +56,7 @@
   }
 }

+
 //----------------------------------------------------------------------------------------------------------------------
 // Constants

@@ -703,6 +704,24 @@
   }
 }
 #endif // !PRODUCT
+int InterpreterRuntime:: method_count = 0;
+static int hotness_average_counter = 0;
+IRT_ENTRY(void, InterpreterRuntime::count_new_method(JavaThread* thread, methodOopDesc * method))
+  method_count ++;
+IRT_END
+
+IRT_ENTRY(void, InterpreterRuntime::increase_hotness_average(JavaThread* thread, methodOopDesc* method))
```

```
+     hotness_average_counter ++;
+     if (hotness_average_counter >=method_count) {
+       hotness_average_counter -= method_count;
+       InvocationCounter::decrease_threshold();
+   //    ResourceMark rm;
+       InvocationCounter::average ++;
+       //if ( average % 100 == 0)
+//      tty->print_cr("Average %d %s I %d B %d II %d ", average, method->name_and_sig_as_C_string(), method->invocation_counter()->count() >>3, method->backedge_counter()->count() >>3, method->interpreter_invocation_count() >>3);
+     }
+     //tty->print_cr("");
+IRT_END

 IRT_ENTRY(nmethod*,
           InterpreterRuntime::frequency_counter_overflow(JavaThread* thread, address branch_bcp))
Only in /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/interpreter: .interpreterRuntime.cpp.swp
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp 2009-06-29 14:17:06.000000000 +0430
+++ /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp  2009-07-11 22:58:55.000000000 +0430
@@ -47,6 +47,7 @@
   static void      note_trap(JavaThread *thread, int reason, TRAPS);

  public:
+  static int method_count ;
   // Constants
   static void    ldc           (JavaThread* thread, bool wide);

@@ -126,6 +127,8 @@
 #ifdef ASSERT
   static void    verify_mdp(methodOopDesc* method, address bcp, address mdp);
 #endif // ASSERT
+  static void    count_new_method(JavaThread* thread, methodOopDesc*);
+  static void    increase_hotness_average(JavaThread* thread, methodOopDesc*);
 };


Only in /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/interpreter: .interpreterRuntime.hpp.swp
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/invocationCounter.cpp /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/interpreter/invocationCounter.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/invocationCounter.cpp 2009-07-11 21:36:11.000000000 +0430
+++ /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/interpreter/invocationCounter.cpp   2009-07-02 17:30:46.000000000 +0430
@@ -132,6 +132,8 @@
   return NULL;
 }

+static int unit = 1 << RelativeTresholdFactorPower;
+
 void InvocationCounter::reinitialize(bool delay_overflow) {
   // define states
   guarantee((int)number_of_states <= (int)state_limit, "adjust_number_of_state_bits");
@@ -141,9 +143,12 @@
   } else {
     def(wait_for_compile, 0, dummy_invocation_counter_overflow);
   }
+  initialize(CompileThreshold << RelativeTresholdFactorPower);
+}

-  InterpreterInvocationLimit = CompileThreshold << number_of_noncount_bits;
-  InterpreterProfileLimit = ((CompileThreshold * InterpreterProfilePercentage) / 100)<< number_of_noncount_bits;
+void InvocationCounter::initialize(int threshold) {
+  InterpreterInvocationLimit = threshold << number_of_noncount_bits;
+  InterpreterProfileLimit = ((threshold * InterpreterProfilePercentage) / 100)<< number_of_noncount_bits;
   Tier1InvocationLimit = Tier2CompileThreshold << number_of_noncount_bits;
   Tier1BackEdgeLimit   = Tier2BackEdgeThreshold << number_of_noncount_bits;

@@ -152,17 +157,63 @@
   // don't need the shift by number_of_noncount_bits, but we do need to adjust
   // the factor by which we scale the threshold.
   if (ProfileInterpreter) {
-    InterpreterBackwardBranchLimit = (CompileThreshold * (OnStackReplacePercentage - InterpreterProfilePercentage)) / 100;
+    InterpreterBackwardBranchLimit = (threshold * (OnStackReplacePercentage - InterpreterProfilePercentage)) / 100;
   } else {
```

```
-    InterpreterBackwardBranchLimit = ((CompileThreshold * OnStackReplacePercentage) / 100) << number_of_noncount_bits;
+    InterpreterBackwardBranchLimit = ((threshold * OnStackReplacePercentage) / 100) << number_of_noncount_bits;
   }
 
   assert(0 <= InterpreterBackwardBranchLimit,
-         "OSR threshold should be non-negative");
+        "OSR threshold should be non-negative");
   assert(0 <= InterpreterProfileLimit &&
-         InterpreterProfileLimit <= InterpreterInvocationLimit,
-         "profile threshold should be less than the compilation threshold "
-         "and non-negative");
+        InterpreterProfileLimit <= InterpreterInvocationLimit,
+        "profile threshold should be less than the compilation threshold "
+        "and non-negative");
+}
+
+int InvocationCounter:: average = 1;
+void InvocationCounter::increment_threshold() {
+  InterpreterInvocationLimit += (unit << number_of_noncount_bits);
+  static int InterpreterProfileLimitCounter = InterpreterProfilePercentage -1;
+  InterpreterProfileLimitCounter ++;
+  if (InterpreterProfileLimitCounter >= 100) {
+    InterpreterProfileLimitCounter -= (100 -InterpreterProfilePercentage +1) ;
+    InterpreterProfileLimit += (unit << number_of_noncount_bits);
+  }
+  //Tier1InvocationLimit = Tier2CompileThreshold << number_of_noncount_bits;
+  //Tier1BackEdgeLimit   = Tier2BackEdgeThreshold << number_of_noncount_bits;
+
+  // When methodData is collected, the backward branch limit is compared against a
+  // methodData counter, rather than an InvocationCounter.  In the former case, we
+  // don't need the shift by number_of_noncount_bits, but we do need to adjust
+  // the factor by which we scale the threshold.
+  static int BackwardBranchPercentage = ProfileInterpreter ?
+    OnStackReplacePercentage - InterpreterProfilePercentage :
+    OnStackReplacePercentage;
+
+  static int InterpreterBackwardBranchLimitCounter = BackwardBranchPercentage -1;
+  InterpreterBackwardBranchLimitCounter ++;
+  if (InterpreterBackwardBranchLimitCounter >= 100) {
+    InterpreterBackwardBranchLimitCounter -= (100 -BackwardBranchPercentage +1);
+    if (ProfileInterpreter) {
+      InterpreterBackwardBranchLimit += unit;
+    } else {
+      InterpreterBackwardBranchLimit += (unit<< number_of_noncount_bits);
+    }
+  }
+}
+
+void InvocationCounter::decrease_threshold() {
+
+  int shift;
+  shift = (average & AverageHit) ;
+  if (shift) {
+    InterpreterInvocationLimit >>= ThresholdShift;
+    InterpreterProfileLimit >>= ThresholdShift;
+    //Tier1InvocationLimit = Tier2CompileThreshold << number_of_noncount_bits;
+    //Tier1BackEdgeLimit   = Tier2BackEdgeThreshold << number_of_noncount_bits;
+    InterpreterBackwardBranchLimit >>= ThresholdShift;
+    AverageHit <<= MaskShift;
+  }
 }
 
 void invocationCounter_init() {
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/invocationCounter.hpp /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/interpreter/invocationCounter.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/invocationCounter.hpp  2009-07-11 21:34:31.000000000 +0430
+++ /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/interpreter/invocationCounter.hpp   2009-07-02 17:30:46.000000000 +0430
@@ -112,6 +112,12 @@
   // Miscellaneous
   static ByteSize counter_offset()               { return byte_offset_of(InvocationCounter, _counter); }
   static void reinitialize(bool delay_overflow);
+  static void initialize(int threshold);
+  static void increment_threshold();
+  static void decrease_threshold();
+
+ public:
+  static int          average;
 
  private:
   static int          _init  [number_of_states];  // the counter limits
```

```
Only in /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter: .invocationCounter.hpp.swp
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/runtime/globals.hpp /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/runtime/globals.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/runtime/globals.hpp  2009-06-29 14:17:02.000000000 +0430
+++ /home/rgougol/cs298/build/relative-openjdk/hotspot/src/share/vm/runtime/globals.hpp 2009-07-02 17:30:47.000000000 +0430
@@ -3221,7 +3221,26 @@
                                                                              \
   product(bool, UseVMInterruptibleIO, true,                                 \
          "(Unstable, Solaris-specific) Thread interrupt before or with "    \
-        "EINTR for I/O operations results in OS_INTRPT")
+        "EINTR for I/O operations results in OS_INTRPT")                    \
+                                                                            \
+  product(intx, RelativeTresholdFactorPower, 1,                             \
+        "This number to power of 2 is the factor that multiply average"     \
+       "invocation frequecy to get the compile treshold"                   \
+       " CompileThreshold = 2 ^ RelativeTresholdFactor "                   \
+                      " * average invocation frequency")                   \
+                                                                           \
+  product(intx, AverageHit, 1024,                                           \
+        "If the average invocation freuquency of the methods reach here"   \
+       " it will change the CompileThreshold")                             \
+                                                                           \
+  product(intx, MaskShift, 10,                                              \                "The number of bits that the average hit will be shift"          \
+        "to the right when the average invocation frequency reach it"      \
+       " so the average can reach this new average hit later ")            \
+                                                                           \
+  product(intx, ThresholdShift, 1,                                          \
+       "the number of bits to shift the CompileThreshold to right to"      \
+       " in order to decrease it.")
+


 /*
```

A.4 Method Grouping

```
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/cppInterpreter_x86.cpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/cpu/x86/vm/cppInterpreter_x86.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/cppInterpreter_x86.cpp      2009-06-29 14:16:58.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/cpu/x86/vm/cppInterpreter_x86.cpp 2009-07-13 18:30:11.000000000 +0430
@@ -562,8 +562,9 @@
    // profile_method != NULL == !native_call
    // BytecodeInterpreter only calls for native so code is elided.

-   __ cmp32(rcx,
-            ExternalAddress((address)&InvocationCounter::InterpreterInvocationLimit));
+  //__ cmp32(rcx,
+   //          ExternalAddress((address)&InvocationCounter::InterpreterInvocationLimit));
+  __ cmp32(rcx, Address(rbx, methodOopDesc:: interpreter_invocation_limit_offset()));
   __ jcc(Assembler::aboveEqual, *overflow);

 }
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp 2009-06-29 14:16:58.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp    2009-07-13 18:30:11.000000000 +0430
@@ -7,7 +7,7 @@
   * published by the Free Software Foundation.
   *
   * This code is distributed in the hope that it will be useful, but WITHOUT
- * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
+ * ANY WARRANTY; without eteInterpreter_x86_32.cppven the implied warranty of MERCHANTABILITY or
   * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
   * version 2 for more details (a copy is included in the LICENSE file that
   * accompanied this code).
@@ -324,6 +324,46 @@
    if (ProfileInterpreter) { // %%% Merge this into methodDataOop
      __ incrementl(Address(rbx,methodOopDesc::interpreter_invocation_counter_offset()));
    }
+
+  //Method Grouping
+  const Address id(rbx, methodOopDesc:: id_offset());
+  __ get_thread(rdx);                        //rdx : thread address
+  __ movl(rax, id);                          // rax :  method id
+  __ testl(rax, rax);
+  Label AlreadyHasId;
+  __ jcc(Assembler:: notZero, AlreadyHasId);
+  const Address static_id(rdx, JavaThread::id_generator_offset());
+  __ incrementl(static_id);
+  __ movl(rax, static_id);
+  __ movl(id, rax);
+  __ bind(AlreadyHasId);
+  __ push(rcx);
+  __ movl (rcx, Address(rdx, JavaThread:: saved_method_id_offset()));
+                        //rcx: id of previous method that was interpreted
+  Label NoPreviousId;
+  __ testl (rcx, rcx);
+  __ jcc(Assembler:: zero, NoPreviousId);
+  __ movl(rdx, rcx);
+  int sugar = (unsigned int) -1 >> (sizeof(int) * 8 - PairQueueLengthPower);
+  __ andl(rdx, sugar);
+             // rdx: remainder of dividing pervious method id by PairQueueLength
+             //(hash of id into PairQueue)
+  __ push(rax);                            /*rax : current method id */
+  __ movl(rax, Address(rbx, methodOopDesc:: pairs_offset()));
+  __ movl(Address(rax , rdx, Address::times_4), rcx);
+  __ pop(rax);                                   /*eax : id of current method*/
+
+  __ bind(NoPreviousId);
+  __ pop(rcx);
+  __ get_thread(rdx);
+  __ movl(Address(rdx, JavaThread:: saved_method_id_offset()), rax);
+
+  __ push(rcx);
+  __ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::is_paired), rax);
+                        // rax will indicate if the current method is paired
+  __ pop (rcx);
+  __ movl(Address(rbx, methodOopDesc:: paired_offset()), rax);  // save rax
+
   // Update standard invocation counters
   __ movl(rax, backedge_counter);                // load backedge counter

@@ -339,16 +379,62 @@

   if (ProfileInterpreter && profile_method != NULL) {
     // Test to see if we should create a method data oop
-    __ cmp32(rcx,
-             ExternalAddress((address)&InvocationCounter::InterpreterProfileLimit));
+    __ cmp32(rcx, Address(rbx, methodOopDesc::interpreter_profile_limit_offset()));
     __ jcc(Assembler::less, *profile_method_continue);

     // if no method data exists, go to profile_method
     __ test_method_data_pointer(rax, *profile_method);
   }

-  __ cmp32(rcx,
-           ExternalAddress((address)&InvocationCounter::InterpreterInvocationLimit));
+  __ push(rcx);
+  __ movl(rdx, Address(rbx, methodOopDesc:: paired_offset()));  // rdx: if method is predicted to be hot
+  Label NotPaired;
+  __ testl(rdx, rdx);
+  __ jcc(Assembler:: zero, NotPaired);
+  __ movl(rcx, invocation_counter);
+
+  if (!ProfileInterpreter) {
+    __ movl(Address(rbx, methodOopDesc:: interpreter_invocation_limit_offset()), rcx);
+    __ movl(Address(rbx, methodOopDesc:: interpreter_backedge_limit_offset()), rax);
+    __ pop(rcx);
+    __ jmp(*overflow);
+  }
+  __ shrl(rcx, 3);
+  __ imull(rax, rcx, 100);
+  __ push(rbx);
+  __ movl(rbx, (int) InterpreterProfilePercentage);
+  __ movl(rdx, 0);
+
+  Label NoNewCompileThreshold;
+  //__ cmpl(rbx, 0);
+  //__ jcc(Assembler:: zero, NoNewCompileThreshold);
+  __ idivl(rbx);
+  __ pop(rbx);
+  __ cmpl(rax, CompileThreshold);
+  __ jcc(Assembler:: aboveEqual, NoNewCompileThreshold);
+  __ shll(rax, 3);
+  __ shll(rcx, 3);
+  __ movl(Address(rbx, methodOopDesc:: interpreter_invocation_limit_offset()), rax);
+  __ movl(Address(rbx, methodOopDesc:: interpreter_profile_limit_offset()), rcx);
+  __ push(rbx);
+  __ shrl(rax, 3);
+  __ push(rax);
+  __ movl(rax, OnStackReplacePercentage);
+  __ subl(rax, InterpreterProfilePercentage);
+   __ movl(rbx, 100);
+   __ movl(rdx, 0);
+   __ idivl(rbx);
+   __ pop(rbx);
+   __ imull(rax, rbx);
+   __ pop(rbx);
+   __ shll(rax, 3);
+   __ movl(Address(rbx, methodOopDesc:: interpreter_backedge_limit_offset()), rax);
+
+   __ bind(NoNewCompileThreshold);
+   __ bind(NotPaired);
+   __ pop(rcx);
+
+   __ cmp32(rcx, Address(rbx, methodOopDesc:: interpreter_invocation_limit_offset()));
    __ jcc(Assembler::aboveEqual, *overflow);

 }
@@ -376,6 +462,8 @@
    // C++ interpreter only
    // rsi - previous interpreter state pointer

+   __ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::store_pairs));
+   // Predict all the methods in the group to be hot
    const Address size_of_parameters(rbx, methodOopDesc::size_of_parameters_offset());

    // InterpreterRuntime::frequency_counter_overflow takes one argument
@@ -384,7 +472,6 @@
    // if the compilation did not complete (either went background or bailed out).
    __ movptr(rax, (int32_t)false);
    __ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::frequency_counter_overflow), rax);
-
    __ movptr(rbx, Address(rbp, method_offset));   // restore methodOop

    // Preserve invariant that rsi/rdi contain bcp/locals of sender frame
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_32.cpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_32.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_32.cpp   2009-06-29 14:16:58.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/cpu/x86/vm/templateTable_x86_32.cpp   2009-07-13 18:30:11.000000000 +0430
@@ -1555,8 +1555,7 @@

      if (ProfileInterpreter) {
        // Test to see if we should create a method data oop
-       __ cmp32(rax,
-                ExternalAddress((address) &InvocationCounter::InterpreterProfileLimit));
+       __ cmp32(rax, Address(rbx, methodOopDesc:: interpreter_profile_limit_offset()));
        __ jcc(Assembler::less, dispatch);

        // if no method data exists, go to profile method
@@ -1564,8 +1563,7 @@

        if (UseOnStackReplacement) {
          // check for overflow against rbx, which is the MDO taken count
-         __ cmp32(rbx,
-                  ExternalAddress((address) &InvocationCounter::InterpreterBackwardBranchLimit));
+        __ cmp32(rbx, Address(rcx, methodOopDesc:: interpreter_backedge_limit_offset()));
          __ jcc(Assembler::below, dispatch);

          // When ProfileInterpreter is on, the backedge_count comes from the
@@ -1581,8 +1579,7 @@
      } else {
        if (UseOnStackReplacement) {
          // check for overflow against rax, which is the sum of the counters
-         __ cmp32(rax,
-                  ExternalAddress((address) &InvocationCounter::InterpreterBackwardBranchLimit));
+        __ cmp32(rax, Address(rcx, methodOopDesc:: interpreter_backedge_limit_offset()));
          __ jcc(Assembler::aboveEqual, backedge_counter_overflow);

        }
Only in /home/rgougol/cs298/build/grouping-openjdk/hotspot/src: grouping-.diff
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp   2009-06-29 14:17:02.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp   2009-07-13 18:30:11.000000000 +0430
@@ -317,11 +317,33 @@

    // print osr_bci if any
    if (is_osr) tty->print("␣@␣%d", osr_bci());
+
+   //print method id and group
+   tty->print("␣ID␣%d␣[", method->id());
+   for (int i = 0; i <  (1 << PairQueueLengthPower); i++)
+   {
```

```
+     int a;
+     if ((a = method->pair(i)) > 0)
+       tty->print("_%d", a);
+   }
+   tty->print("]");
+
+   tty->print("Counts_I_%d_B_%d_IT_%d",  method->invocation_counter()->count(), method->backedge_counter()->count(), method->interpreter_invocation_count()/*, method->compiled_invocation_count()*/);
+   tty->print("Limits_%d_%d_%d", method->interpreter_invocation_limit(), method->interpreter_backedge_limit(), method->interpreter_profile_limit());

    // print method size
    tty->print_cr("_(%d_bytes)", method->code_size());
-}

+  //print Grouping Hash List
+  tty->print("[");
+  for (int i = 0; i < (1 << PairHashLengthPower); i++)
+  {
+    int a;
+    if ((a = * (JavaThread::hash_pair(i))) != 0)
+      tty->print("%d_", a);
+  }
+  tty->print_cr("]");
+}

 // ------------------------------------------------------------------
 // CompileTask::log_task
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp 2009-06-29 14:17:06.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp  2009-07-13 18:30:11.000000000 +0430
@@ -704,6 +704,40 @@
 }
 #endif // !PRODUCT

+IRT_ENTRY(nmethod*, InterpreterRuntime::store_pairs(JavaThread* thread, address bcp))
+  ResourceMark rm;
+    methodOop method = thread->last_frame().interpreter_frame_method();
+      for (int i = 0; i < (1 << PairQueueLengthPower); i++) {
+           if (method->pair(i) > 0) {
+                   JavaThread::store_pair(method->pair(i));
+            }
+      }
+  /* tty->print("Stored pairs of method %d %s which were [", method->id() , method->name_and_sig_as_C_string());
+  for (int i = 0; i < (1 << PairQueueLengthPower); i++)
+  {
+       if (method->pair(i) != 0) tty->print(" %d", method->pair(i));
+  }
+  tty->print("] into [");
+  for (int i = 0; i < (1 <<PairHashLengthPower); i++)
+  {
+     if (*(JavaThread::hash_pair(i)) != 0) tty->print(" %d", *(JavaThread::hash_pair(i)));
+  }
+  tty->print_cr("]"); */
+  return NULL;
+IRT_END
+
+IRT_ENTRY(bool, InterpreterRuntime::is_paired(JavaThread* thread, int method_id))
+    return JavaThread:: is_paired(method_id);
+IRT_END
+
+IRT_ENTRY(void, InterpreterRuntime::trace_method(JavaThread* thread))
+  ResourceMark rm;
+  methodOop method = thread->last_frame().interpreter_frame_method();
+  tty->print("ID_%d_Counts_I_%d_%d_%d", method->id(), method->invocation_counter()->count(), method->invocation_counter()->state(), method->invocation_counter()->carry());
+  tty->print("_B_%d_%d_%d_", method->backedge_counter()->count(), method->backedge_counter()->state(), method->backedge_counter()->carry());
+  tty->print_cr("Limits_%d_%d_%d",  method->interpreter_invocation_limit(), method->interpreter_backedge_limit(), method->interpreter_profile_limit());
+IRT_END
+
 IRT_ENTRY(nmethod*,
           InterpreterRuntime::frequency_counter_overflow(JavaThread* thread, address branch_bcp))
   // use UnlockFlagSaver to clear and restore the _do_not_unlock_if_synchronized
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp 2009-06-29 14:17:06.000000000 +0430
```

```
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.hpp  2009-07-13 18:30:11.000000000 +0430
@@ -118,6 +118,10 @@

   // Interpreter's frequency counter overflow
   static nmethod* frequency_counter_overflow(JavaThread* thread, address branch_bcp);
+
+  static nmethod* store_pairs(JavaThread* thread, address bcp);
+  static bool is_paired(JavaThread* thread, int method_id);
+  static void trace_method(JavaThread*);

   // Interpreter profiling support
   static jint    bcp_to_di(methodOopDesc* method, address cur_bcp);
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/invocationCounter.cpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/interpreter/invocationCounter.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/invocationCounter.cpp  2009-07-11 21:36:11.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/interpreter/invocationCounter.cpp   2009-07-13 18:30:11.000000000 +0430
@@ -39,14 +39,15 @@
   set_state(wait_for_compile);
 }

-void InvocationCounter::set_carry() {
+void InvocationCounter::set_carry(int method_compile_threshold) {
+  ResourceMark rm;
   _counter |= carry_mask;

   // The carry bit now indicates that this counter had achieved a very
   // large value.  Now reduce the value, so that the method can be
   // executed many more times before re-entering the VM.
   int old_count = count();
-  int new_count = MIN2(old_count, (int) (CompileThreshold / 2));
+  int new_count = MIN2(old_count, (int) (method_compile_threshold / 2));
   if (old_count != new_count)  set(state(), new_count);
 }

@@ -107,7 +108,8 @@

 static address do_nothing(methodHandle method, TRAPS) {
   // dummy action for inactive invocation counters
-  method->invocation_counter()->set_carry();
+  ResourceMark rm;
+  method->invocation_counter()->set_carry(method->interpreter_invocation_limit());
   method->invocation_counter()->set_state(InvocationCounter::wait_for_nothing);
   return NULL;
 }
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/invocationCounter.hpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/interpreter/invocationCounter.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/interpreter/invocationCounter.hpp  2009-07-11 21:34:31.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/interpreter/invocationCounter.hpp   2009-07-13 18:30:11.000000000 +0430
@@ -78,7 +78,7 @@
   void set_state(State state);                    // sets state and initializes counter correspondingly
   inline void set(State state, int count);        // sets state and counter
   inline void decay();                            // decay counter (divide by two)
-  void set_carry();                               // set the sticky carry bit
+  void set_carry(int);                            // set the sticky carry bit

   // Accessors
   State  state() const                            { return (State)(_counter & state_mask); }
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/methodKlass.cpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/oops/methodKlass.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/methodKlass.cpp 2009-06-29 14:17:06.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/oops/methodKlass.cpp  2009-07-13 18:30:11.000000000 +0430
@@ -88,7 +88,10 @@
   m->set_interpreter_invocation_count(0);
   m->invocation_counter()->init();
   m->backedge_counter()->init();
+  m->init_invocation_limit();
   m->clear_number_of_breakpoints();
+  m->setup_pairs();
+  m->set_id(0);
   assert(m->is_parsable(), "must be parsable here.");
   assert(m->size() == size, "wrong size for object");
   // We should not publish an uprasable object's reference
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/methodOop.cpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/oops/methodOop.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/methodOop.cpp   2009-07-11 21:31:34.000000000 +0430
```

```
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/oops/methodOop.cpp  2009-07-13 18:30:11.000000000 +0430
@@ -298,6 +298,12 @@
   }
 }

+void methodOopDesc::init_invocation_limit() {
+    _interpreter_invocation_limit =  InvocationCounter::InterpreterInvocationLimit;
+    _interpreter_profile_limit = InvocationCounter::InterpreterProfileLimit;
+    _interpreter_backedge_limit = InvocationCounter::InterpreterBackwardBranchLimit;
+}
+
 void methodOopDesc::cleanup_inline_caches() {
   // The current system doesn't use inline caches in the interpreter
   // => nothing to do (keep this method around for future use)
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/methodOop.hpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/oops/methodOop.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/oops/methodOop.hpp   2009-06-29 14:17:07.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/oops/methodOop.hpp  2009-07-13 18:30:11.000000000 +0430
@@ -94,6 +94,10 @@
   constantPoolOop   _constants;                   // Constant pool
   methodDataOop     _method_data;
   int               _interpreter_invocation_count; // Count of times invoked
+  int               _interpreter_invocation_limit;
+  int               _interpreter_profile_limit;
+  int               _interpreter_backedge_limit;
+
   AccessFlags       _access_flags;                // Access flags
   int               _vtable_index;                // vtable index of this method (see VtableIndexFlag)
                                                   // note: can have vtables with >2**16 elements (because of inheritance)
@@ -110,6 +114,10 @@
   u2                _number_of_breakpoints;       // fullspeed debugging support
   InvocationCounter _invocation_counter;          // Incremented before each activation of the method - used to trigger frequency-based optimizations
   InvocationCounter _backedge_counter;            // Incremented before each backedge taken - used to trigger frequencey-based optimizations
+  int *              _pairs;
+  unsigned int      _id;                         // method id for keeping track of methods
+  bool              _paired;
+
 #ifndef PRODUCT
   int               _compiled_invocation_count;  // Number of nmethod invocations so far (for perf. debugging)
 #endif
@@ -206,6 +214,16 @@
     _method_size = size;
   }

+  void set_id(unsigned int id)                           { _id = id; }
+  unsigned int id() const                                { return _id;}
+  int pair(int index) {
+       return _pairs[index];
+  }
+  void setup_pairs() {
+       _pairs = (int *) calloc(1 << PairQueueLengthPower, sizeof(int));
+        _paired = false;
+  }
+
   // constant pool for klassOop holding this method
   constantPoolOop constants() const              { return _constants; }
   void set_constants(constantPoolOop c)          { oop_store_without_check((oop*)&_constants, c); }
@@ -284,8 +302,12 @@
   static void build_interpreter_method_data(methodHandle method, TRAPS);

   int interpreter_invocation_count() const       { return _interpreter_invocation_count; }
+  int interpreter_invocation_limit() const { return _interpreter_invocation_limit >> 3; }
+  int interpreter_profile_limit() const { return _interpreter_profile_limit >> 3; }
+  int interpreter_backedge_limit() const { return ProfileInterpreter ? _interpreter_backedge_limit : _interpreter_backedge_limit >> 3; }
   void set_interpreter_invocation_count(int count) { _interpreter_invocation_count = count; }
   int increment_interpreter_invocation_count() { return ++_interpreter_invocation_count; }
+  void init_invocation_limit();

 #ifndef PRODUCT
   int  compiled_invocation_count() const         { return _compiled_invocation_count; }
@@ -485,10 +507,16 @@
   static ByteSize code_offset()                  { return byte_offset_of(methodOopDesc, _code); }
   static ByteSize invocation_counter_offset()    { return byte_offset_of(methodOopDesc, _invocation_counter); }
   static ByteSize backedge_counter_offset()      { return byte_offset_of(methodOopDesc, _backedge_counter); }
```

```
+  static ByteSize interpreter_invocation_limit_offset() { return byte_offset_of(methodOopDesc, _interpreter_invocation_limit); }
+  static ByteSize interpreter_profile_limit_offset() {return byte_offset_of(methodOopDesc, _interpreter_profile_limit); }
+  static ByteSize interpreter_backedge_limit_offset() {return byte_offset_of(methodOopDesc, _interpreter_backedge_limit); }
   static ByteSize method_data_offset()          {
     return byte_offset_of(methodOopDesc, _method_data);
   }
   static ByteSize interpreter_invocation_counter_offset() { return byte_offset_of(methodOopDesc, _interpreter_invocation_count); }
+  static ByteSize id_offset()                    { return byte_offset_of(methodOopDesc, _id); }
+  static ByteSize pairs_offset()                 { return byte_offset_of(methodOopDesc, _pairs); }
+  static ByteSize paired_offset()                { return byte_offset_of(methodOopDesc, _paired);}
 #ifndef PRODUCT
   static ByteSize compiled_invocation_counter_offset() { return byte_offset_of(methodOopDesc, _compiled_invocation_count); }
 #endif // not PRODUCT
diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/runtime/compilationPolicy.cpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/runtime/compilationPolicy.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/runtime/compilationPolicy.cpp 2009-06-29 14:17:02.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/runtime/compilationPolicy.cpp 2009-07-13 18:30:11.000000000 +0430
@@ -97,9 +97,10 @@
   // as would be the case for native methods.

   // BUT also make sure the method doesn't look like it was never executed.
-  // Set carry bit and reduce counter's value to min(count, CompileThreshold/2).
-  m->invocation_counter()->set_carry();
-  m->backedge_counter()->set_carry();
+  // Set carry bit and reduce counter's value to min(count, m->interpreter_invokation_limit() / 2)).
+  ResourceMark rm;
+  m->invocation_counter()->set_carry(m->interpreter_invocation_limit());
+  m->backedge_counter()->set_carry(m->interpreter_invocation_limit());

   assert(!m->was_never_executed(), "don't_reset_to_0_--_could_be_mistaken_for_never-executed");
 }
@@ -113,15 +114,16 @@
   // Don't set invocation_counter's value too low otherwise the method will
   // look like immature (ic < ~5300) which prevents the inlining based on
   // the type profiling.
-  i->set(i->state(), CompileThreshold);
+  ResourceMark rm;
+  i->set(i->state(), m->interpreter_invocation_limit());
   // Don't reset counter too low - it is used to check if OSR method is ready.
-  b->set(b->state(), CompileThreshold / 2);
+  b->set(b->state(), m->interpreter_invocation_limit() / 2);
 }

 // SimpleCompPolicy - compile current method

 void SimpleCompPolicy::method_invocation_event( methodHandle m, TRAPS) {
-  assert(UseCompiler || CompileTheWorld, "UseCompiler_should_be_set_by_now.");
+  assert(UseCompiler || CompileTheWorld, "UseCompiler_m->interpreter_invocation_limit()_should_be_set_by_now.");

   int hot_count = m->invocation_count();
   reset_counter_for_invocation_event(m);
@@ -436,7 +438,7 @@
   if (m->code_size() <= MaxTrivialSize) return NULL;
   if (UseInterpreter) {     // don't use counts with -Xcomp
     if ((m->code() == NULL) && m->was_never_executed()) return (_msg = "never_executed");
-    if (!m->was_executed_more_than(MIN2(MinInliningThreshold, CompileThreshold >> 1))) return (_msg = "executed_<_MinInliningThreshold_times");
+    if (!m->was_executed_more_than(MIN2(MinInliningThreshold, m->interpreter_invocation_limit() >> 1))) return (_msg = "executed_<_MinInliningThreshold_times");
   }
   if (methodOopDesc::has_unloaded_classes_in_signature(m, JavaThread::current())) return (_msg = "unloaded_signature_classes");

diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/runtime/globals.hpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/runtime/globals.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/runtime/globals.hpp  2009-06-29 14:17:02.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/runtime/globals.hpp 2009-07-13 18:30:11.000000000 +0430
@@ -3221,7 +3221,15 @@
                                                                             \
   product(bool, UseVMInterruptibleIO, true,                                 \
           "(Unstable,_Solaris-specific)_Thread_interrupt_before_or_with_"   \
-          "EINTR_for_I/O_operations_results_in_OS_INTRPT")
+          "EINTR_for_I/O_operations_results_in_OS_INTRPT")                  \
+                                                                            \
+  product(intx, PairHashLengthPower, 3,                                      \
+          "Length␣Power␣of␣Pair␣Hash,␣hash␣size␣=␣2␣^␣PairHashLengthPower") \
+                                                                             \
+  product(intx, PairQueueLengthPower, 3,                                      \
+                "Length␣Power␣of␣Pair␣Queue,␣length␣=␣2␣^␣n")
+
+


 /*
 diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/runtime/thread.cpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/runtime/thread.cpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/runtime/thread.cpp   2009-06-29 14:17:01.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/runtime/thread.cpp  2009-07-13 18:30:11.000000000 +0430
@@ -1206,6 +1206,9 @@
    _popframe_preserved_args = NULL;
    _popframe_preserved_args_size = 0;

+   _id_generator = 0;
+   _saved_method_id = 0;
+
    pd_initialize();
  }

 diff -r -u /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/runtime/thread.hpp /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/runtime/thread.hpp
--- /home/rgougol/cs298/build/openjdk/hotspot/src/share/vm/runtime/thread.hpp   2009-06-29 14:17:01.000000000 +0430
+++ /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/runtime/thread.hpp  2009-07-13 18:30:11.000000000 +0430
@@ -633,6 +633,8 @@
   private:
    JavaThread*    _next;                          // The next thread in the Threads list
    oop            _threadObj;                     // The Java level thread object
+   int            _id_generator;                   // id generator for methods
+   int           _saved_method_id;                  //id of method being interpreted to pair with next method

 #ifdef ASSERT
   private:
@@ -1167,6 +1169,8 @@
    static ByteSize suspend_flags_offset()          { return byte_offset_of(JavaThread, _suspend_flags       ); }

    static ByteSize do_not_unlock_if_synchronized_offset() { return byte_offset_of(JavaThread, _do_not_unlock_if_synchronized); }
+   static ByteSize id_generator_offset()              { return byte_offset_of(JavaThread, _id_generator); }
+   static ByteSize saved_method_id_offset()           { return byte_offset_of(JavaThread, _saved_method_id); }

    // Returns the jni environment for this thread
    JNIEnv* jni_environment()                        { return &_jni_environment; }
@@ -1445,6 +1449,15 @@
    // clearing/querying jni attach status
    bool is_attaching() const { return _is_attaching; }
    void set_attached() { _is_attaching = false; OrderAccess::fence(); }
+
+   static int * hash_pair(int pair_id) {
+       static int * _pairs = (int *) calloc(1 << PairHashLengthPower, sizeof(int));
+       return & _pairs[pair_id % (1 << PairHashLengthPower)];
+   }
+   static bool is_paired(int pair_id) {
+       return  pair_id == * hash_pair(pair_id);
+   }
+   static void store_pair(int pair_id)  { * hash_pair(pair_id) = pair_id;}
  };

  // Inline implementation of JavaThread::current
```

A.5 Tiered Grouping

```
 diff -r -u /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp /home/rgougol/cs298/build/postpone-grouping-openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp
--- /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp   2009-01-15 17:11:06.000000000 -0800
+++ /home/rgougol/cs298/build/postpone-grouping-openjdk/hotspot/src/cpu/x86/vm/templateInterpreter_x86_32.cpp   2008-11-16 21:44:01.000000000 -0800
@@ -7,7 +7,7 @@
   * published by the Free Software Foundation.
   *
   * This code is distributed in the hope that it will be useful, but WITHOUT
```

```
- * ANY WARRANTY; without eteInterpreter_x86_32.cppven the implied warranty of MERCHANTABILITY or
+ * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
  * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
  * version 2 for more details (a copy is included in the LICENSE file that
  * accompanied this code).
@@ -339,6 +339,14 @@
    __ bind(AlreadyHasId);
    __ push(rcx);
    __ movl (rcx, Address(rdx, JavaThread:: saved_method_id_offset()));
+   __ movl(Address(rdx, JavaThread:: saved_method_id_offset()), rax);
+
+   // Is the method in the highest compilation tier
+   __ movl(rdx, CompLevel_highest_tier);
+   __ cmp32(rdx, Address(rbx, methodOopDesc::highest_tier_compile_offset()));
+   Label Postpone;
+   __ jcc(Assembler:: notEqual, Postpone);
+
                        //rcx: id of previous method that was interpreted
    Label NoPreviousId;
    __ testl (rcx, rcx);
@@ -348,21 +356,18 @@
    __ andl(rdx, sugar);
              // rdx: remainder of dividing pervious method id by PairQueueLength
             //(hash of id into PairQueue)
-   __ push(rax);                              /*rax : current method id */
    __ movl(rax, Address(rbx, methodOopDesc:: pairs_offset()));
    __ movl(Address(rax , rdx, Address::times_4), rcx);
-   __ pop(rax);                                    /*eax : id of current method*/

    __ bind(NoPreviousId);
    __ pop(rcx);
-   __ get_thread(rdx);
-   __ movl(Address(rdx, JavaThread:: saved_method_id_offset()), rax);

    __ push(rcx);
    __ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::is_paired), rax);
                         // rax will indicate if the current method is paired
-   __ pop (rcx);
    __ movl(Address(rbx, methodOopDesc:: paired_offset()), rax);  // save rax
+   __ bind(Postpone);
+   __ pop (rcx);

    // Update standard invocation counters
    __ movl(rax, backedge_counter);                // load backedge counter
@@ -387,18 +392,11 @@
    }

    __ push(rcx);
-   __ movl(rdx, Address(rbx, methodOopDesc:: paired_offset()));  // rdx: if method is predicted to
      be hot
+   __ movl(rax, Address(rbx, methodOopDesc:: paired_offset()));  // restore rax
    Label NotPaired;
-   __ testl(rdx, rdx);
+   __ testl(rax, rax);
    __ jcc(Assembler:: zero, NotPaired);
    __ movl(rcx, invocation_counter);
-
-   if (!ProfileInterpreter) {
-     __ movl(Address(rbx, methodOopDesc:: interpreter_invocation_limit_offset()), rcx);
-     __ movl(Address(rbx, methodOopDesc:: interpreter_backedge_limit_offset()), rax);
-     __ pop(rcx);
-     __ jmp(*overflow);
-   }
    __ shrl(rcx, 3);
    __ imull(rax, rcx, 100);
    __ push(rbx);
@@ -462,8 +460,6 @@
    // C++ interpreter only
    // rsi - previous interpreter state pointer

-   __ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::store_pairs));
-   // Predict all the methods in the group to be hot
    const Address size_of_parameters(rbx, methodOopDesc::size_of_parameters_offset());

    // InterpreterRuntime::frequency_counter_overflow takes one argument
@@ -472,6 +468,7 @@
    // if the compilation did not complete (either went background or bailed out).
    __ movptr(rax, (int32_t)false);
    __ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::frequency_counter_overflow),
        rax);
+   __ call_VM(noreg, CAST_FROM_FN_PTR(address, InterpreterRuntime::store_pairs));
    __ movptr(rbx, Address(rbp, method_offset));   // restore methodOop

    // Preserve invariant that rsi/rdi contain bcp/locals of sender frame
 diff -r -u /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/compiler/compileBroker.
     cpp /home/rgougol/cs298/build/postpone-grouping-openjdk/hotspot/src/share/vm/compiler/
     compileBroker.cpp
```

```
--- /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp 2008-11-15 18:36:13.000000000 -0800
+++ /home/rgougol/cs298/build/postpone-grouping-openjdk/hotspot/src/share/vm/compiler/compileBroker.cpp 2008-11-15 10:38:09.000000000 -0800
@@ -328,7 +328,7 @@
     }
     tty->print("]");

-    tty->print("Counts␣I␣%d␣B␣%d␣IT␣%d",  method->invocation_counter()->count(), method->backedge_counter()->count(), method->interpreter_invocation_count()/*, method->compiled_invocation_count()*/);
+    tty->print("Counts␣I␣%d␣B␣%d␣IT␣%d␣",  method->invocation_counter()->count(), method->backedge_counter()->count(), method->interpreter_invocation_count()/*, method->compiled_invocation_count()*/);
     tty->print("Limits␣%d␣%d␣%d", method->interpreter_invocation_limit(), method->interpreter_backedge_limit(), method->interpreter_profile_limit());

     // print method size
 diff -r -u /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp /home/rgougol/cs298/build/postpone-grouping-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp
--- /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp   2008-11-18 20:13:51.000000000 -0800
+++ /home/rgougol/cs298/build/postpone-grouping-openjdk/hotspot/src/share/vm/interpreter/interpreterRuntime.cpp 2008-11-18 20:15:01.000000000 -0800
@@ -712,7 +712,7 @@
                         JavaThread::store_pair(method->pair(i));
                 }
         }
-    /* tty->print("Stored pairs of method %d %s which were [", method->id() , method->name_and_sig_as_C_string());
+    /*tty->print("Stored pairs of method %d %s which were [", method->id() , method->name_and_sig_as_C_string());
     for (int i = 0; i < (1 << PairQueueLengthPower); i++)
     {
         if (method->pair(i) != 0) tty->print(" %d", method->pair(i));
@@ -722,7 +722,7 @@
     {
       if (*(JavaThread::hash_pair(i)) != 0) tty->print(" %d", *(JavaThread::hash_pair(i)));
     }
-    tty->print_cr("]"); */
+    tty->print_cr("]");*/
     return NULL;
  IRT_END

 diff -r -u /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/oops/methodOop.cpp /home/rgougol/cs298/build/postpone-grouping-openjdk/hotspot/src/share/vm/oops/methodOop.cpp
--- /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/oops/methodOop.cpp   2009-01-10 13:41:44.000000000 -0800
+++ /home/rgougol/cs298/build/postpone-grouping-openjdk/hotspot/src/share/vm/oops/methodOop.cpp 2008-11-13 21:21:52.000000000 -0800
@@ -298,12 +298,6 @@
     }
  }

-void methodOopDesc::init_invocation_limit() {
-     _interpreter_invocation_limit =  InvocationCounter::InterpreterInvocationLimit;
-     _interpreter_profile_limit = InvocationCounter::InterpreterProfileLimit;
-     _interpreter_backedge_limit = InvocationCounter::InterpreterBackwardBranchLimit;
-}
-
 void methodOopDesc::cleanup_inline_caches() {
    // The current system doesn't use inline caches in the interpreter
    // => nothing to do (keep this method around for future use)
 diff -r -u /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/oops/methodOop.hpp /home/rgougol/cs298/build/postpone-grouping-openjdk/hotspot/src/share/vm/oops/methodOop.hpp
--- /home/rgougol/cs298/build/grouping-openjdk/hotspot/src/share/vm/oops/methodOop.hpp   2009-01-10 13:41:44.000000000 -0800
+++ /home/rgougol/cs298/build/postpone-grouping-openjdk/hotspot/src/share/vm/oops/methodOop.hpp 2008-11-16 21:58:34.000000000 -0800
@@ -304,10 +304,14 @@
    int interpreter_invocation_count() const        { return _interpreter_invocation_count; }
    int interpreter_invocation_limit() const { return _interpreter_invocation_limit >> 3; }
    int interpreter_profile_limit() const { return _interpreter_profile_limit >> 3; }
-   int interpreter_backedge_limit() const { return ProfileInterpreter ? _interpreter_backedge_limit : _interpreter_backedge_limit >> 3; }
+   int interpreter_backedge_limit() const { return _interpreter_backedge_limit >> 3; }
    void set_interpreter_invocation_count(int count) { _interpreter_invocation_count = count; }
    int increment_interpreter_invocation_count() { return ++_interpreter_invocation_count; }
-   void init_invocation_limit();
+   void init_invocation_limit() {
+     _interpreter_invocation_limit =  invocation_counter()->get_InvocationLimit() << 3;
+     _interpreter_profile_limit = invocation_counter()->get_ProfileLimit() << 3;
+     _interpreter_backedge_limit = invocation_counter()->get_BackwardBranchLimit() << 3;
+   }

 #ifndef PRODUCT
```

```
   int   compiled_invocation_count() const         { return _compiled_invocation_count; }
@@ -517,6 +521,7 @@
   static ByteSize id_offset()                    { return byte_offset_of(methodOopDesc, _id); }
   static ByteSize pairs_offset()                 { return byte_offset_of(methodOopDesc, _pairs); }
   static ByteSize paired_offset()                { return byte_offset_of(methodOopDesc, _paired);}
+  static ByteSize highest_tier_compile_offset()  { return byte_offset_of(methodOopDesc, _highest_tier_compile); }
 #ifndef PRODUCT
   static ByteSize compiled_invocation_counter_offset() { return byte_offset_of(methodOopDesc, _compiled_invocation_count); }
 #endif // not PRODUCT
```