Published in IET Software
Received on 14th August 2011
Revised on 10th February 2012
doi: 10.1049/iet-sen.2011.0144

ISSN 1751-8806

Heuristic optimisation algorithm for Java dynamic compilation

Y. Liu[1] *A.S. Fong*[2]

[1]*Faculty of Computer, Guangdong University of Technology, University Mega Center, Guangzhou 51006, People's Republic of China*
[2]*Department of Electronic Engineering, City University of Hong Kong, 83 Tat Chee Avenue, Kowloon, Hong Kong*
E-mail: yjliu2002@163.com

Abstract: Dynamic compilation increases Java virtual machine (JVM) performance because running compiled codes is faster than interpreting Java bytecodes. However, inappropriate decision on dynamic compilation may degrade performance owing to compilation overhead. A good heuristic algorithm for dynamic compilation should achieve an appropriate balance between compilation overhead and performance gain in each method invocation sequence. A method-size and execution-time heuristic algorithm is proposed in the study. The key principle of the algorithm is that different method-sizes necessitate different compile thresholds for optimal performance. A parameter search mechanism using a genetic algorithm for dynamic compilation is proposed to find optimised multi-thresholds in the algorithm. This heuristic algorithm is evaluated in an openJDK Java Server JVM using SPEC JVM98 benchmark suite. The algorithm shows an overall advantage in performance speedup when testing benchmarks and gain speedup by 19.1% on average. The algorithm also increases the performance of original openJDK by 10.2% when extended to the whole benchmark suite.

1 Introduction

A traditional Java virtual machine (JVM) executes Java bytecodes as an interpreter, which emulates and decodes bytecodes into native computer machine code through software at runtime. Since interpreting bytec odes into machine codes is rather straightforward, a Java interpreter is simpler in design and requires less resource of CPU and memory, than conventional compilers. However, the performance of interpretation is slow, compared with compiling with native machine code. To improve the performance of a JVM, a just-in-time (JIT) compiler is included which compiles bytecodes to underlying machine codes. During compiling, some instruction-level optimisations can be made to increase performance. Moreover, the compiled machine codes of the executed program segments are saved, so that if a program segment needs to be executed again, the respective saved machine codes are used for direct execution, saving the time of interpreting. Compiling bytecodes costs considerable time overhead, while executing machine code rather than interpreting bytecodes saves time in each invocation. Therefore the selection of interpreting and compiling should be based on appropriate profiling of the attributes of program segments [1, 2]. Decisions on whether to compile bytecodes are usually made on a per-method basis. Intuitively, frequently executed methods should be compiled. Investigation must be made to determine if there are any other attributes of a method to be used as heuristics to make such a decision.

In this paper, a heuristic optimisation algorithm is proposed for Java dynamic compilation. We use method-size and execution-time as heuristic parameters to decide whether to compile a method or not. In the algorithm, method-sizes will determine their respective execution-time thresholds. Only if the execution-time of a method exceeds the respective threshold is the method compiled. Otherwise, it executes in an interpreting way. The threshold parameters of method-sizes and thresholds are searched by analysing Java benchmark programs. A genetic algorithm (GA) is used to minimise the search space of parameter numbers and parameter values during the process of finding optimised thresholds.

The remainder of the paper is organised as follows: Section 2 reviews related works. Section 3 discusses which method attributes should be considered when making a decision of dynamic compilation, and proposes a method-size and execution-time heuristic algorithm for optimisation. Section 4 proposes an optimised heuristic parameter search mechanism using a genetic algorithm. Section 5 describes the experiment set-up and methodology and the experiment results. Finally, Section 6 summarises the work and the findings.

2 Related works

JIT compilers, such as Caffeine compiler [3] and native executable translation (NET) compiler [4], compile Java methods when they are invoked at the first time. JIT compilers outperform the interpreters in speed when there are a large amount of reuses of methods. However, compiling consumes extra time overhead. If programs

contain lots of method codes rarely reused, the overall performance may be degraded. Dynamic or hotspot compilers, such as IBM's Jalapeno JVM [5] and Sun's HotSpot JVM [6], are designed to increase overall performance of Java execution. Java dynamic compilers use different heuristic approaches and different profiling techniques to identify frequently used methods and compile them on the fly. Good detection and profiling techniques balance the trade-off between compilation time and running time. Some heuristics were proposed to identify the most suitable heuristics for dynamic compilation.

It is important to estimate which methods are often being executed, which are called 'hot' or frequently used methods. The most accurate estimation is the accumulated execution-time of each method by summing the execution time from start to exit of every invocation, but accurate measurement during run-time Java execution imposes a big overhead. To minimise overhead, some profiling information is used to estimate each method's execution time to justify whether it is likely to be called again subsequently. Some compilation heuristics are used to identify these hot spots and to trigger JIT compiler. The most popular heuristics are:

- Call stack sampling
- Counters

IBM JVM and Jikes RVM [7] use call stack sampling to select compile candidate. A sampling-based profiler [8] gathers information about program thread execution through keeping track of methods where the application threads are consuming the most CPU time by periodically sneaking the program counters of all the threads to identify which methods are currently being executed. The profiler then increments a hotness counter associated with each method [9, 10].

Counters are simple to implement and are commonly used by JVMs. The estimated execution time of a method is described by a number of counters. The estimated execution-time of each method is predicted by the current number of invocations on the corresponding method. The invocation number is profiled by a counter associated with each method. Compared to the sampling method, method invocation counter provides a simpler method for comparable estimation but it has the disadvantage of overhead increases proportionally with the invocation number during interpretation. The algorithm by counter uses a basic inequality to determine hot methods that are to be compiled. It is a simple heuristic algorithm and most JVMs use it as a fundamental mechanism to trigger the dynamic compiler. The most popular one is HotSpot JVM [6] developed by Longview Technologies, which was acquired by Sun Microsystems. We also adapt a similar counter-based heuristic in the proposed algorithm.

Sun HotSpot JVM uses software counters as the heuristic. Methods identified as hot methods when the values of the corresponding counters are larger than the compile threshold. HotSpot JVM also uses backward branch count as a metric to capture frequently executed methods since backward branch count mimics the number of loops a method executed. The compile threshold value is a static number. HotSpot JVM can be used in a server or a client computer, where the original server JVM has a predefined compile threshold of 10 000 and the client JVM has a compile threshold of 1500. HotSpot JVM uses the following equation to estimate the execution time of a method

$$\text{Execution\_time}(\text{method}) = \text{Invocation\_count}(\text{method}) + \text{backward\_branch\_count}(\text{method})$$

The decision-making algorithm of Sun hotspot JVM is illustrated by Fig. 1. If the execution time of a method exceeds a previously defined threshold, it is to be compiled. This is a *time-only* heuristic.

Simple heuristic proposed by Jonathan L. Schilling includes method-size as a heuristic metric [11]. Method-size and method invocation count are used as a heuristic which is implemented in SCO JDK 1.1.7B. Simple heuristic does not consider backward branch count as a metric since it is difficult to implement in the SCO/Caldera JIT. The heuristic mechanism is characterised by Fig. 2. Schilling believes that if the size of a method or its execution-time exceeds a certain number, the method should be compiled. The boundary points shown in Fig. 2 are labelled as JIT_MIN_SIZE and JIT_MIN_TIMES. The slope between these points is a dividing line of decision-making. If a method is categorised within the area below the line, it should not be compiled; otherwise, it should be compiled. Schilling picked some arbitrary numbers by trial to define the values of JIT_MIN_SIZE and JIT_MIN_TIMES (JIT_MIN_SIZE = 150 and JIT_MIN_TIMES = 40).

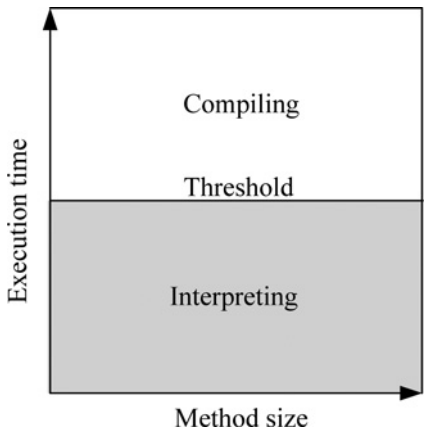

Fig. 1 *Time-only heuristic used in Sun HotSpot JVM*

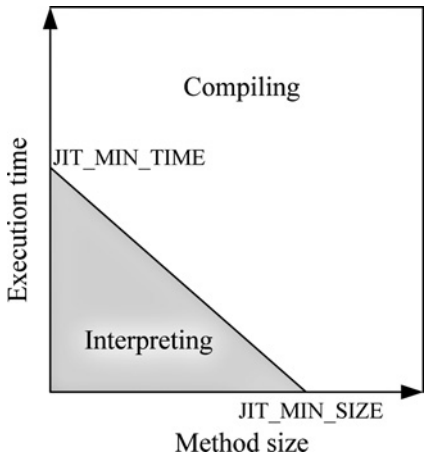

Fig. 2 *Simple heuristic used in SCO JDK*

doi: 10.1049/iet-sen.2011.0144

Through the experiment on SPEC JVM98 benchmarks [12], the simple heuristic outperforms the 'No-JIT' heuristic and 'Size-only' heuristic. However, it is slower than the unconditional compilation – 'Always-JIT' at the tests of all benchmarks and slower than 'Time-only' heuristic for some benchmarks in JVM98 benchmark suite.

3 Heuristic algorithm for dynamic compilation

Dynamic compilation should be done with appropriate trade-off between compilation time overhead and performance gain in each cycle of method invocation using compiled native machine codes. A decision is made by comparing the time difference between the two scenarios of interpreting and compiling as defined in (1)

$$\Delta T_{\text{benefit}} = T_{\text{itp}} - T_{\text{cpl}} \tag{1}$$

In the equation, $\Delta T_{\text{benefit}}$ is the time benefit obtained from compiling bytecodes rather than interpreting them; T_{itp} is the time used in executing a program method by interpreting; T_{cpl} is the time used in executing the method by compiling. Assuming the method is executed for n times, then $T_{\text{itp}} = n(T_i + T_{\text{mio1}})$, where T_i is the execution-time to run the method for one time by interpreting, T_{mio1} is the time overhead of each method evocation. $T_{\text{cpl}} = T_c + n(T_n + T_{\text{mio2}}) + T_{\text{overhead}}$, where T_{c} is the time used to compile the method to native code; T_{n} is the execution-time to run native machine code of the method for one time; T_{mio2} is the time overhead of each method call; T_{overhead} is the time overhead used by making decision whether to compile the method, including the overheads of method profiling and selecting, or not. Then (1) becomes (2)

$$\Delta T_{\text{benefit}} = n(T_i + T_{\text{mio1}}) - [T_{\text{c}} + n(T_{\text{n}} + T_{\text{mio2}}) + T_{\text{overhead}}] \tag{2}$$

Let us assume $T_{\text{mio1}} = T_{\text{mio2}}$, (2) becomes (3)

$$\Delta T_{\text{benefit}} = n(T_i - T_{\text{n}}) - T_{\text{c}} - T_{\text{overhead}} \tag{3}$$

Therefore there is a reasonable linear relationship between the size of a method (l) and its compiling time (T_{c}) through investigating SPEC JVM98 benchmarks. However, there is no apparent linear relationship between method-size and its interpreted execution-time (or compiled code execution-time) [11]. Intuitively, execution-time of a method has a relationship with its size (s) and the number of internal loops (l). T_{overhead} is mainly caused by profiling mechanism and selection algorithm used by a JVM. T_{overhead} normally has no relationship with method-size (s), but has a relationship with n and l since these parameters should be counted during run time. Through this analysis, $\Delta T_{\text{benefit}}$ is a function of at least three parameters, namely n, l and s. Maybe some other parameters also affect $\Delta T_{\text{benefit}}$, but we want to keep the algorithm simple.

From the analysis, we conclude that decision of dynamic compilation can be made by profiling execution-time (or frequency) of a method (influenced by n and l) and its bytecode size (defined as s). For a given method, the bytecode size is fixed, we need to find a threshold of execution frequency to make $\Delta T_{\text{benefit}}$ positive. Methods with different sizes have different thresholds. If the execution-time of a method exceeds a corresponding threshold, it will be compiled, else it will be interpreted. In practice, we put method-sizes into several categories. Each category has a corresponding compile threshold. A purposed heuristic algorithm is illustrated in Fig. 3. The figure shows that method-sizes are classified into four categories. The grey part in the coordinate system indicates those methods that do not need to compile, and the white part represents the methods need to compile. MSTH[i] represents a method-size threshold. MSTH[i-1] and MSTH[i] define method category i. ETTH[i] means the execution-time threshold of category i. A method falls in category i based on its method-size. For example if a method has a method-size (ms) greater than MSTH[1] and less than MSTH[2] and its execution-time (et) greater than ETTH[2], it is to be compiled. Since there are four execution-time threshold parameters for comparison in the figure, we call the algorithm '4-level heuristic algorithm'. Consequently, a two-level heuristic algorithm has three threshold parameters (ETTH[0, MSTH[0], ETTH[1]); a three-level algorithm has five parameters (ETTH[0], MSTH[0], ETTH[1], MSTH[1], ETTH[2]); a four-level algorithm has seven parameters as shown in Fig. 3.

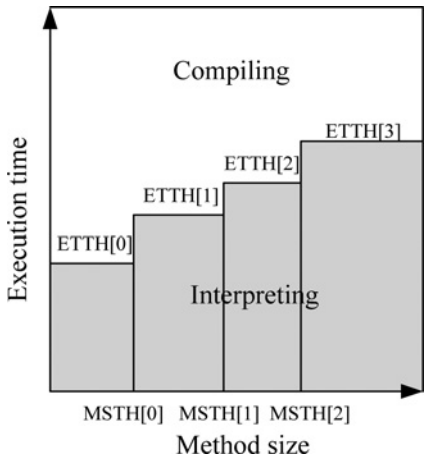

Fig. 3 *Proposed four-level heuristic algorithm for dynamic compilation*

4 Optimised heuristic parameter search using genetic algorithm

The heuristic algorithm proposed in the preceding section creates a problem which needs to be solved critically. Since we need more metric combinations for compile decision, increase of parameter count causes the search space of finding parameter values expand exponentially. Compared to both HotSpot and the simple heuristic in which the numbers of static parameters are only one (compile threshold) and two (JIT_MIN_TIME and JIT_MIN_SIZE), respectively, the proposed algorithm has at least three threshold parameters for two-level comparison, including ETTH[0], MSTH[0] and ETTH[1]. The parameter number expands to five for three-level comparison, seven for four-level and more when the level of decision increases. It is hard to find optimised threshold parameters simply by picking up some static values arbitrarily by trial, since it has a huge set of values with so many combinations to be selected for optimal decisions. An evolutionary heuristics optimisation algorithm is proposed to encounter the problem and search the optimised threshold parameter values.

The evolutionary algorithm [13] does an iterative process to generate a population of candidacy, and then uses search techniques to find the optimised solutions. In this work, a GA [14] is used to solve the problem of huge search space during the process of searching optimised threshold parameters.

Two issues should be defined in the genetic algorithm:

1. a genetic representation (chromosome) of the solution domain;
2. a fitness function to evaluate the solution domain.

For the first issue, a chromosome refers to a candidate solution and we encode it in a bit string. A chromosome is a block of bits that encodes several parameters. We choose a one-dimensional genome array (GAArray) with 16-bit unsigned integers to represent threshold values. Each extendable array consists of at least three elements for the heuristic using two-level comparison. The entire array format represents the feasible solution as shown in Fig. 4. Since each threshold has its constrained value, we further define the constraints for different thresholds. For examples execution-time thresholds (ETTH[i]) are bounded in values between 0 and 50 000, and method-size thresholds (MSTH[i]) are bounded in values between 0 and 5000, and also with another constraint of MSTH[0] < MSTH[1] < MSTH[2] < · · · , etc. Fig. 4 illustrates the chromosome format for three-level heuristic algorithm with three thresholds.

For the second issue, a fitness function is defined to evaluate the solution domain. GA is a method to optimise a particular function, where the goal is to pick the optimised parameter values that maximise or minimise the multi-parameter fitness function. An appropriate fitness function is crucial to capture the best candidate. Our objective is to calculate the best heuristic metrics for dynamic compilation in JVM in order to minimise the overall application execution-time. We include Java benchmarks in the fitness function to estimate the total execution-time. The fitness function for an individual benchmark (p_i) is defined by (4).

$$f(p_i) = \text{execution\_time}\,(p_i) \tag{4}$$

The fitness function for an individual benchmark is the execution time of the benchmark program under the tests using different chromosomes. The optimised parameters can be decoded from the chromosome making the fitness function (execution time) minimal.

However, when trying to find the optimised threshold parameters using a benchmark suite (p) containing several benchmark programs (p_i), the sum of the execution-time of different benchmarks, defined by (5), is not suitable because it may have partiality for the programs having long execution time.

$$f(p) = \sum_{p_i \in p} \text{execution\_time}\,(p_i) \tag{5}$$

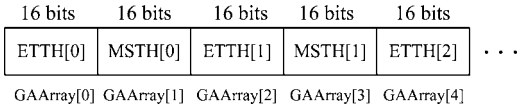

Fig. 4 *Chromosome representation of the proposed genetic algorithm*

The execution time of different benchmarks should be normalised. The normalised fitness function for a suite including several benchmarks is defined by (6).

$$f(p) = \sum_{p_i \in p} \frac{\text{execution\_time}\,(p_i)}{\text{norm\_time}\,(p_i)} \tag{6}$$

The norm_time (p_i) is calculated in order to obtain the optimised threshold parameters of individual benchmark (p_i). We store the smallest 100 execution-time values of benchmark p_i and the mean value of the100 minimal 100 execution-time values is used as norm_time (p_i).

A system may mainly be utilised to perform some particular functions and the corresponding benchmarks should have more weight to the fitness function. For this, we add a weight metric to (6) and it becomes (7).

$$f(p) = \sum_{p_i \in p} w_i \frac{\text{execution\_time}\,(p_i)}{\text{norm\_time}\,(p_i)} \tag{7}$$

where w_i is the weight of benchmark p_i. For example if a JVM runs on a database server in which approximately 70% of the time is used to execute database operations, we will set w_i to 0.7 for database benchmark. In our experiment, all benchmarks have the same weight ($w_i = 1/n$, where n is the benchmark program count in the suite).

5 Experiment set-up and result

5.1 Experiment set-up

The objective of the experiment is to measure the performance with various thresholds to achieve optimised results, with different benchmarks under a popular platform.

The proposed heuristic algorithm is implemented in OpenJDK [15] for testing purpose. A modified OpenJDK is invoked from the fitness function of the GA program to test and verify the proposed heuristic optimisation algorithm. We choose OpenJDK as an implementation basis because among the available open-source JVMs in the market, Sun HotSpot JVM is a popular one and it is open-source. OpenJDK is promising for the free software community. The hardware and software environment is listed as follows

Hardware:

- *Processor:* Intel® Pentium® Dual CPU 3.00 GHz
- *Memory:* 3.0 GB
- *Available disk space:* 19.3 GB

Software:

1. *Operating System*:
   - Fedora Release 9 (Sulphur)
   - Kernel Linux 2.6.25-14.fc9.i686
   - GNOME 2.22.1
2. *C/C++ compiler:* GNU g++ (GCC) 4.3.0
3. *IDE:* NetBeans IDE 6.5

We choose Server JVM rather than Client JVM in the experiment. To minimise overheads of background operating processes such as networking, graphical user interfaces, multithreading, input–output drivers etc. the whole evaluation process of the proposed algorithm is

doi: 10.1049/iet-sen.2011.0144

executed in single user and single CPU mode. The GNOME is not started. Moreover, the clock speed of CPU is tuned down to 1.5 GHz to avoid the sudden throttle of the CPU clock in case of getting overheated for the long period of iterative execution during optimisation. The actual implementation of the experiment method involves relationships among different components shown in Fig. 5.

According to the figure, the proposed heuristic algorithm is implemented in OpenJDK. We choose SPEC JVM98 as the benchmark suite to integrate in the heuristic optimisation algorithm. The benchmark suite is used in the tuning of best combination of heuristic in the JVM system. The GA is designed and implemented using the GAlib libraries [16] of version 2.4.7. The proposed objective fitness function and the genome (chromosome) representation are discussed previously. In every GA iteration process, the objective fitness function is invoked. It passes the generated genome array to the modified OpenJDK. The JDK runs the benchmarks with the heuristic algorithm proposed in Section 3. The execution-time values of the benchmarks are fed back to the GA for further evaluation. The iterative process keeps on running till the optimised parameters are found.

5.2 Experiment results

The SPEC JVM98 [12] are chosen as the benchmark suite, including seven benchmarks – _201_compress, _202_jess, _209_db, _213_javac, _222_mpegaudio, _227_mtrt and _228_jack. First, we search the optimised threshold parameters using individual benchmarks. The performance of the original openJDK with a system-predefined compile threshold of 10 000 (server) is used as a standard for comparisons. Three levels of comparison algorithms are used in the experiment. One-level comparison is a time-only heuristic, where when execution-time of a method exceeds the defined compile threshold it is compiled. Two-level comparison has three thresholds – (ETTH[0], MSTH[0], ETTH[1]). 3-level comparison has five thresholds – (ETTH[0], MSTH[0], ETTH[1], MSTH[2], ETTH[2]). The optimised threshold parameters searched by the GA are listed in Table 1. The performance speedup using the proposed heuristic algorithm with the optimised threshold parameters is shown in Fig. 6. As shown, the GA identifies the optimised threshold parameters and the proposed heuristic algorithm using the searched optimised thresholds speedups original openJDK by an average of 15.7%.

We then use the GA to find the optimised parameters on the whole JVM98 benchmark suite. The optimised parameters in different levels are (6773), (7191, 1375, 28 708), (2490, 1316, 18 132, 4098) and (6282, 1918, 19 725, 2061, 12 674, 3005, 29 449). Fig. 7 shows the speedup results compared to the original openJDK with a predefined compileThreshold of 10 000 for Server JVM. The one-level optimisation with optimised compile threshold of (6773) achieves 4.3% speedup; the 2-level heuristic algorithm using optimised parameters (7191, 1375, 28 708) achieves a speedup of 10.2%; the three-level algorithm using parameters (2490, 1316, 18 132, 4098) achieves a speedup of 6.1%; the four-level algorithm using parameters (6282, 1918, 19 725, 2061, 12 674, 3005, 29 449) achieves a speedup of 6.3%. From the experiment results, the

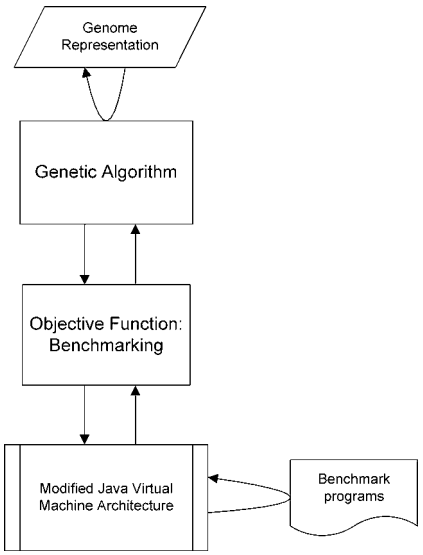

Fig. 5 *Relationships among related components on genetic algorithm*

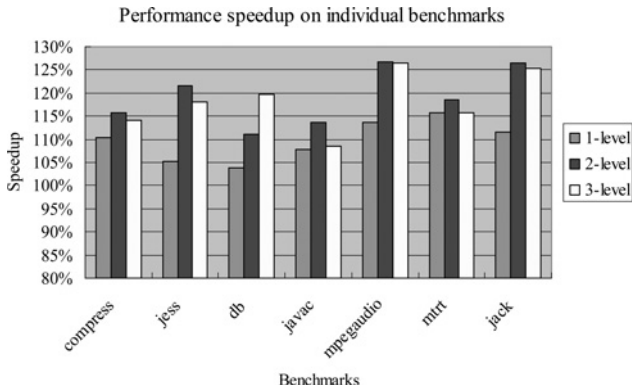

Fig. 6 *Speedup using the proposed heuristic algorithm on individual benchmarks*

Table 1 Optimised threshold parameter found by GA under different scenario

Benchmark program	Optimised thresholds, Time only	Optimised thresholds, 2-level comparison	Optimised thresholds, 3-level comparison
_201_compress	14 021	(3538, 503, 13 519)	(3856, 12, 7400, 481, 3851)
_202_jess	6284	(5883, 1439, 16 429)	(7642, 1689, 17 116, 4877, 8439)
_209_db	4348	(6477, 3705, 24 836)	(29 266, 2171, 29 185, 3304, 25 685)
_213_javac	2341	(2723, 1366, 9561)	(2897, 2463, 16 458, 2669, 19 987)
_222_mpegaudio	24 908	(22 318, 274, 5866)	(24 702, 326, 1157, 4306, 21 794)
_227_mtrt	2676	(2839, 40, 2080)	(2210, 2980, 16 068, 3605, 18 678)
_228_jack	2842	(1368, 1132, 24 869)	(3875, 519, 9030, 3871, 29 992)

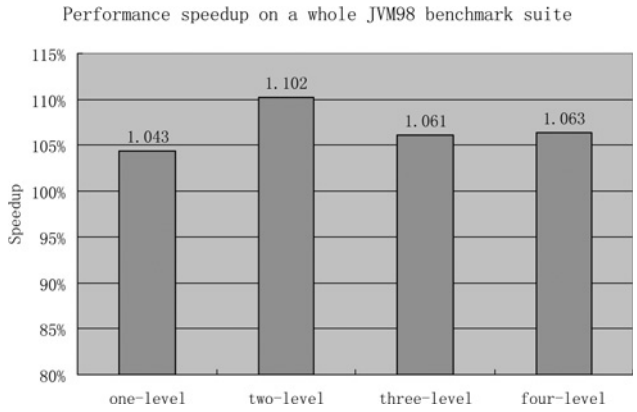

Fig. 7 *Speedup using the proposed heuristic algorithm on JVM98 suite*

two-level comparison heuristic algorithm is shown to perform the best in the whole benchmark suite analysis.

5.3 Experiment result analysis

From the analysis of the experiment results on individual benchmarks and the whole JVM98 benchmark suite, it is shown that a significant speedup can be achieved using the proposed adaptive heuristic algorithm with the optimised threshold parameters searched through the genetic algorithm. Moreover, the heuristic algorithm in two-level scenario with three thresholds (ETTH[0], MSTH[0], ETTH[1] – 7191, 1375, 28 708) performs better than in other scenarios. Implemented in Server openJDK JVM, the two-level algorithm increases the performance of original JVM by 19.1% using individual benchmarks and 10.2% using the whole benchmark suite.

6 Conclusion

Dynamic compilation in Java should be done after considering trade-off between compiling overhead and performance gain using compiling machine code. In this paper, a method-size and execution-time heuristic algorithm is proposed to make decisions on dynamic compilation. Methods are put in different categories based on their bytecode sizes. Different categories have different compile thresholds. Only if the execution-time of a method exceeds the corresponding thresholds is it compiled. This is a multiple-parameter heuristic algorithm and search space of finding optimised thresholds is huge. A GA method is proposed to reduce the search space of multiple parameter combinations in finding the optimised threshold values. The proposed two-level heuristic algorithm using the searched optimised threshold parameters speeds up the openJDK Server averagely by 19.1% for individual benchmarks, and 10.2% for a whole SPEC JVM98 suite.

7 Acknowledgments

The authors would like to thank the members of Project HISC team, in particular, C. H. Yau. The work described in this paper was partially supported by Strategic Research Grant (no. 7002602) from the City University of Hong Kong and was also supported by National Natural Science Foundation of China (no. 61106019).

8 References

1 Sundaresan, V., Maier, D., Ramarao, P., Stoodley, M.: 'Experiences with multi-threading and dynamic class loading in a Java just-in-time compiler'. Int. Symp. on Code Generation and Optimization, CGO 2006, March 2006, pp. 26–29
2 Goetz, B.: 'Java theory and practice: garbage collection in the 1.4.1 JVM', DeveloperWorks, IBM, available at http://www.ibm.com/developerworks/library/j-jtp12214/, December 2004
3 Hsieh, C.-H.A., Gyllenhaal, J.C., Hwu, W.W.: 'Java bytecode to native code translation: the Caffeine prototype and preliminary results'. Proc. 29th Annual IEEE/ACM Int. Symp. on Microarchitecture 1996, MICRO-29, 2–4 December 1996, pp. 90–97
4 Hsieh, C.-H.A., Conte, M.T., Johnson, T.L., Gyllenhaal, J.C., Hwu, W.-M.W.: 'Compilers for improved Java performance', *Computer*, 1997, **30**, (6), pp. 67–75
5 Michael, G., Jong-Deok, C., Stephen, F., *et al.*: 'The Jalapeño dynamic optimizing compiler for Java'. ACM 1999 Java Grande Conf., June 1999, pp. 129–141
6 Sun Microsystems, Inc.: 'The Java HotSpot performance engine architecture', available at http://java.sun.com/products/hotspot/docs/whitepaper/, April 1999
7 Jikes RVM, available at http://jikesrvm.org/, Feb. 2012
8 Whaley, J.: 'A portable sampling-based profiler for java virtual machines'. Proc. ACM 2000 Java Grande Conf., June 2000, pp. 78–87
9 Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., Nakatani, T.: 'Design and evaluation of dynamic optimizations for a java just-in-time compiler', *ACM Trans. Program. Lang. Syst.*, 2005, **27**, (4), pp. 732–785
10 Toshio, S., Toshiaki, Y., Toshio, N.: 'A region-based compilation technique for dynamic compilers', *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2006, **28**, (1), pp. 134–174
11 Schilling, J.L.: 'The simplest heuristics may be the best in java JIT compilers', *SIGPLAN Not.*, 2003, **38**, (2), pp. 36–46
12 SPEC.: 'JVM98 Benchmark suits', available at http://www.spec.org/jvm98, Aug. 1998
13 Michalewicz, Z.: 'Genetic algorithm + data structure = evolution program' (Springer-Verlag, 1996, 3rd edn.)
14 Man, K.F., Tang, K.S., Kwong, S.: 'Genetic algorithms' (Springer-Verlag, 1999)
15 OpenJDK, available at http://openjdk.java.net/, Jul. 2011
16 Wall, M.: 'GALib: A C++ library for genetic algorithm components', version 2.4, Documentation Revision B, MIT, 1996

doi: 10.1049/iet-sen.2011.0144