Exploring Optimal Compilation Unit Shapes for an Embedded Just-In-Time Compiler

Derek Bruening
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
iye@mit.edu

Evelyn Duesterwald
Hewlett-Packard Laboratories
Cambridge, MA 02142
duester@hpl.hp.com

Abstract

This paper investigates strategies for finding optimal compilation unit shapes for a Java Just-in-Time (JIT) compiler. The standard compilation unit for a JIT compiler is a whole method. However, compiling all executing methods is an ill-suited solution in the embedded domain where space constraints limit the storage available to hold the compiled code. A more effective strategy for an embedded JIT compiler is to only compile the frequently executed core of the application while less important portions continue to be interpreted. We explore in this paper various code selection strategies that are targeted at minimizing compiled code size while maximizing the time spent in compiled code versus interpretation. Our evaluation shows that using methods as the only permissible compilation unit, even when using profiling support to compile only hot methods, cannot yield optimal results. We develop in this paper hybrid compilation unit strategies that select methods and either dynamic code traces or loops for compilation. The combination of methods with traces or loops significantly outperforms the solely method-based strategy by allowing for reductions in compiled code size without sacrificing execution coverage of the compiled code.

1 Introduction

Recently there has been a growing interest in using Java on embedded portable devices. As deployment of Java applications on these devices increases, performance will become a more important issue. To achieve higher performance, Java Virtual Machines (JVMs) are typically enhanced with Just-In-Time (JIT) compilers. The standard compilation unit for a JIT compiler is a whole method [7, 10]. While compiling whole methods can provide an effective solution for a JVM running in a desktop environment, it is ill-suited for an embedded JVM where space constraints limit the storage available to hold the compiled code. If there is only room for compiling a fraction of the running application, the decision as to which code fraction to compile and which fraction to continue interpreting becomes crucial.

We conducted a preliminary study to evaluate the effectiveness of various alternative compilation unit shapes for an embedded JIT compiler. The goal of our study was to find a shape that minimizes compiled code size but still gives good code coverage in terms of execution time. It has been well established that programs typically follow the "90/10" rule, that is, 90% of the execution time is spent in only 10% of the code. The ideal compilation unit shape would approach the 90/10 rule, such that 90% of the execution time is spent in the compiled code while only 10% of the complete code is compiled (i.e., the compiled code size should be only 10% of the compiled code size when every method is compiled).

To pursue the 90/10 rule we target the hot regions inside methods for compilation. However, when compiling code regions smaller than entire methods, care has to be taken to not preclude valuable opportunities for optimizing call overhead. Executing Java method calls is expensive. Figure 1 shows the percentage of execution time spent in call overhead for twelve Java benchmarks. The data was collected using HP's Chai embedded JVM [3] but is also representative of the call overhead in other JVMs. The overhead ranges from 4% to 40% with an average of 28%.

The high call overhead is primarily due to the expensive execution of the JVM's invoker routine. For method calls from one compiled method to another, the call overhead can be reduced substantially by compiling the call itself and thereby bypassing the invoker routine. The compiled callee can then be invoked more efficiently on the native stack following a C-style calling convention. Compiling method calls compli-

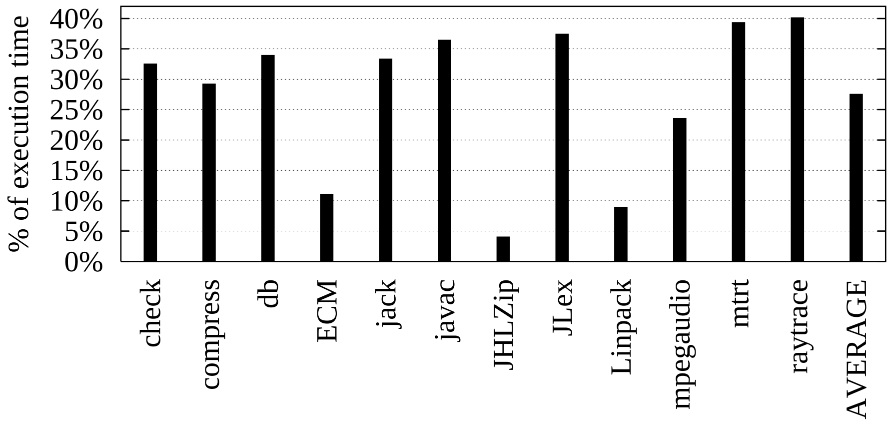

Figure 1: Method call overhead for twelve Java benchmarks and the average (arithmetic) overhead.

cates and lengthens the compiled code. For example, exception handling code must be generated to traverse the native stack instead of the Java stack kept in the interpreter. However, the savings in call overhead are often worth the additional compilation cost.

The importance of eliminating call overhead is one factor in favor of compiling whole methods. However, compiling rarely invoked methods only increases the compiled code size without much benefit from reduced calling overhead.

Most published work on JIT compilation [1, 6, 11] has focused on code generation and largely ignored compilation unit selection. Previous approaches that limit the amount of code being compiled still rely on whole methods as the compilation unit but use profile information to selectively compile only hot methods [5].

In this paper we consider compiled code size reduction along an additional dimension by varying the compilation unit shape. Besides whole methods we consider loops and traces as compilation units and explore various hybrid compilation strategies. The use of traces in our study was motivated by the previous success of using traces in dynamic optimization [2] and in static compilation [4]. Trace compilation is also the target of future work in the Latte JVM [12].

To measure the efficacy of each unit shape we use a cost/benefit model, where the cost of a unit shape is the resulting compiled code size and the benefit is the amount of execution time spent in compiled code. Our study was performed using HP's Chai JVM. Chai does not have a JIT compiler, so we estimate compiled code sizes and simulate the execution of compiled code using the interpreter.

We were motivated to study alternative compilation units due to code cache size constraints in an embedded target platform. Our results, however, also generalize to other platforms and languages.

2 Compilation Unit Shapes

This section presents the various compilation unit shapes that we consider in our study. Our base case is to compile every method upon first invocation. The base case establishes the foundation for the 90/10 rule in that the amount of code compiled (every executing method) defines 100% compiled code size and the amount of time spent in executing compiled code establishes our target of 100% execution time. We can now evaluate other compilation strategies by determining how close they get to the 90/10 rule with respect to the code size and execution time values of compiling every method.

We consider two variations on the whole-method strategy to reduce compiled code size. The first variation is based on a hit threshold, such that a method is compiled only when its execution frequency has exceeded the hit threshold. We obtain a second variation by only compiling methods that are smaller than a given size threshold. Finally, we consider a hybrid of the two, in which a method that has become hot is only compiled if it is also smaller than the size threshold.

2.1 Traces

An alternative to a whole-method compilation strategy is to use dynamic execution traces as the unit for compilation. Traces provide an attractive compilation unit since they are simpler to process and optimize than whole methods due to their linear control flow. A dynamic trace is a single-entry, multiple-exit region of code. A trace may begin (i) at a hot start of a method, (ii) at the target of a backward branch in order to capture loops, or (iii) at the exit of a previously compiled trace. A trace may contain method calls but never descends into the callee. Instead the trace continues in the calling method until one of the following termination conditions has been met: a backward taken branch or a method return has been encountered, an exception has been thrown, or a maximum size or maximum number of branches has been reached.

Upon exit from a compiled trace, the interpreter's state must be restored. However, if a trace exit branch directly targets another trace's entrance, the exit branch is directly linked to the second trace so that there is no need to restore interpreter state in between executing the two traces.

In a similar manner to the whole-method case, we can adapt the trace selection scheme to only compile hot traces using a hit threshold. We keep execution counters for each potential trace starting point and when a count reaches the hit threshold a trace is created.

If a trace contains a method call, the call can only be compiled if the callee is completely compiled.* To enable the compilation of calls, we consider a hybrid strategy that compiles both traces and whole methods. For frequently executed methods, in particular if they are small, it may be more beneficial to compile the whole method rather than a collection of traces through it. To integrate methods in our trace compilation strategy we add another (method) hit threshold for which we keep execution counts on method entries.

2.2 Loops

Another alternative compilation unit we consider is a loop. In contrast to the trace strategy that targets single paths through a loop, the loop strategy compiles the entire loop body. Compiling whole loops reduces the number of exits from the compilation unit and, therefore, the amount of compensation code that must be generated. Also, compiling entire loops avoids the tail duplication that results when peeling out a trace from the code. However, in order to compile loops the entire control flow graph for the enclosing method must first be built and analyzed to identify all loops.

We currently rule out loops inside a *try* statement due to the complications of compiling exception handling for units smaller than a method. Loops are selected for compilation based on a hit threshold of the trip count. As with traces, we also consider a combination of loops and methods to enable the compilation of calls to entirely compiled methods.

*Otherwise, the handling of an early exit from the callee's trace requires the costly mapping of the callee frame that has been compiled on the native stack back to the Java stack. Due to the expense in both time and size we assume calls to partially compiled methods always go through the invoker. This expense of exiting across compiled call boundaries is also the reason that traces do not descend into callees.

3 Methodology

We evaluated the effectiveness of the method-, trace-, and loop-based compilation strategies using HP's Chai JVM. Since Chai does not contain a JIT compiler we currently simulate the compilation of the various unit shapes. Thus, the entire client program is always interpreted but interpretation time for code that is assumed to be compiled under the current compilation strategy is considered execution time in the compiled code cache. While this simulation may not be reliable for predicting the actual performance of one particular compilation strategy, it provides a relative measure to effectively compare the code coverage of different compilation units.

We used the following Java benchmarks: Linpack, JHLZip, EmbeddedCaffeineMark (ECM), JLex, and the SPECJVM98 [8] benchmarks (except jess, which cannot be executed by Chai because it references unsupported classes).

We determined an estimate of compiled code size in terms of StrongARM [9] instructions. To estimate the size for a particular compilation unit we accumulate estimated sizes for each bytecode in that unit. The estimated size for a bytecode is the number of StrongARM instructions that handle the bytecode in the Java interpreter code. These estimates result in compiled bytecode sizes ranging from 0 to 118 StrongARM instructions with a mean of 15–20. This estimate corresponds to a naive cut-and-paste JIT compiler, where the compiled code is simply assembled by pasting together the appropriate object code for each bytecode. Thus, as an absolute measure our estimates are pessimistic since JIT optimizations across bytecodes are ignored, but as a relative measure they are effective.

We estimate the size of the compensation code that restores the interpreter state upon a non-method-return exit from a compiled fragment to be 25 StrongARM instructions. We arrived at this estimate as follows. The restored interpreter state consists of local variables and stack operands. We measured the average maximum stack depth to be 2.9 and the average number of local variables to be 2.8. Allowing for up to four instructions to restore each local variable and each stack operand gives a conservative total of about 25 instructions.

To measure the time benefit of compiling code, we simulate the effects of the JIT compiler by placing interpretation time into different timing buckets based on whether the currently executing code is assumed to be compiled or not. For uncompiled method calls, the call overhead time is considered interpretation time. To simulate the effect of compiling method calls, a call from compiled code to a compiled method has its call overhead time placed in the compiled-code bucket.

4 Experimental Results

We first evaluate in Figure 2 the whole-method-only compilation strategy. The figure is composed of four graphs. The top two graphs show the benefit (i.e., execution time covered by compiled code) and the bottom two graphs show the cost (i.e., compiled code size as a percentage of the compiled code size in the base case, where every method is compiled). In the two graphs on the left the hit threshold is varied in order to reduce the compiled code size. The two graphs on the right show the effects of only compiling methods that are below a certain size threshold.

The left two graphs show that a hit threshold of 0 compiles every method before it is executed, resulting in the base case. Raising the hit threshold to 10 reduces the time too far below our 90% goal but has not even reduced the size below 20%. Controlling compiled code size by a method size threshold is even less effective as shown on the right side of the figure. In order to reach a cache size near 10% the compiled execution time drops to less than 40% coverage. Thus, using methods as the sole compilation unit cannot satisfy our goal.

Figure 3 shows the performance of trace-based compilation units. The top two graphs show the execution time covered by compiled code and the bottom two graphs show the compiled code size. The two graphs on the left show the performance based on solely compiling traces. The two graphs on the right show a combination of traces and methods. To ensure that we maintain our 10% code size goal after adding whole methods, we consulted Figure 2 to identify an appropriate hit threshold to trigger whole-method compilation. The graphs on the right are based on a fixed method hit threshold of 500 and a varying trace hit threshold. Varying the method size threshold had negligible effects.

Using traces alone does not yield an execution time coverage higher than 70% on average. This is due to the inability of a trace-only strategy to compile method calls to reduce the high call overhead. Adding methods allows us to compile some of the hot calls which substantially improves the execution time coverage. The graphs on the right show that the combination of traces and methods approaches our goal. For trace hit threshold 500 we obtain 90% execution time coverage with about 15% compiled code size on average.

Figure 4 shows the performance of loop-based compilation units. The figure is composed of four graphs analogous to Figure 3. Solely compiling loops performs even more poorly than solely compiling traces. Unlike traces, loops cannot cover enough of the code, missing important non-looping code in frequently executed methods. The combination of loops and methods is shown in the right two graphs. Again, methods are integrated with a fixed hit threshold of 500. Compared to Figure 3, there is less variation across the loop hit thresholds since loops often contain more than 500 iterations. The combination of loops and methods also reaches the 90/10 rule, and even performs slightly better than the trace/method combination.

Table 1 summarizes the results of this section. The table uses a score to rate each unit shape according to benefit (*%Time*) minus cost (*%Size*). The table also gives the percentage of the total compilation units that are whole methods.

The table shows again that only compiling whole methods cannot achieve both high code coverage and low cache size. In contrast, both traces and loops when combined with methods can reach the 90/10 goal. The performance of loop-based compilation is slightly better than that of trace-based compilation because loops avoid the code duplication of traces, leading to smaller compiled code sizes.

5 Conclusions

We explored in this paper several compilation unit shapes targeted at minimizing compiled code size while maximizing compiled execution time. Our results clearly indicate that a solely method-based strategy is not able to approach the 90/10 rule. Superior performance can be achieved by using different compilation unit shapes in combination with methods. Both loops and traces provide efficient alternative compilation units if combined with methods. While the performance of the loop-based combination is slightly better than the performance of the trace-based combination, traces have several advantages over loops. Traces are faster to compile due to their linear control flow. Although loops provide a larger optimization scope than traces, traces often result in better quality code since optimization focuses on one path. Most

Units	Hit Threshold(s)	Score	%Time	%Size	%Methods
Methods	0	0.0	100.0	100.0	100.0
	50	55.9	73.8	17.9	100.0
Traces	500, —	52.9	64.6	11.7	0.0
	50, 500	64.3	95.8	31.5	19.3
	500, 500	77.9	92.5	14.6	53.5
Loops	500, —	31.2	34.2	3.0	0.0
	50, 500	80.2	92.4	12.2	51.9
	500, 500	81.2	91.3	10.1	63.2

Table 1: Performance comparison of the compilation unit shapes. For traces and loops, the second hit threshold is the method hit threshold. For all strategies shown there is no method size threshold. *Score* equals *%Time* minus *%Size*. *%Methods* is the percentage of compiled code fragments that are entire methods. For reference, the total size of the method strategy when compiling every method is 3701 KB.

importantly, traces can be collected without having to build potentially large control flow graphs for every method. Traces are simple to determine using the trace definition of Section 2.1; a hit counter for each backward branch and each method are all that is needed. Thus, we argue that the simplicity of traces makes them the compilation unit of choice.

In the future we plan to obtain actual sizes and execution times by fully implementing a loop- and trace-based JIT. We hope to be able to design and efficiently implement the compilation of calls to partially compiled code. With such a design, we may not have to consider whole-method compilation at all and a simplified and slimmer trace JIT compiler that only deals with straight-line code may become feasible.

References

[1] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the SIGPLAN'98 Conference on Programming Language Design and Implementation*, June 1998.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the SIGPLAN'00 Conference on Programming Language Design and Implementation*, June 2000.

[3] ChaiVM. Hewlett-Packard Company. `http://www.chai.hp.com/`.

[4] Stefan Freudenberger, Thomas R. Gross, and P. Geoffrey Lowney. Avoidance and suppression of compensation code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems*, 16(4):1156–1214, July 1994.

[5] The Java HotSpot performance engine architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

[6] A. Krall and R. Grafl. CACAO — a 64 bit Java VM just-in-time compiler. In *Proceedings of the ACM PPoPP'97 Workshop on Java for Science and Engineering Computation*, June 1997.

[7] Microsoft SDK for Java JIT. Microsoft Corporation. `http://www.microsoft.com/Java/sdk/40/pg/pg_vm_cmplrs_5.htm`.

[8] SPEC JVM Client98 benchmark. Standard Performance Evaluation Corporation. `http://www.spec.org/osg/jvm98`.

[9] StrongARM processors. Intel Corporation. `http://developer.intel.com/design/strong/`.

[10] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1), 2000.

[11] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, October 1999.

[12] B.-S. Yang. Private communication, 2000.

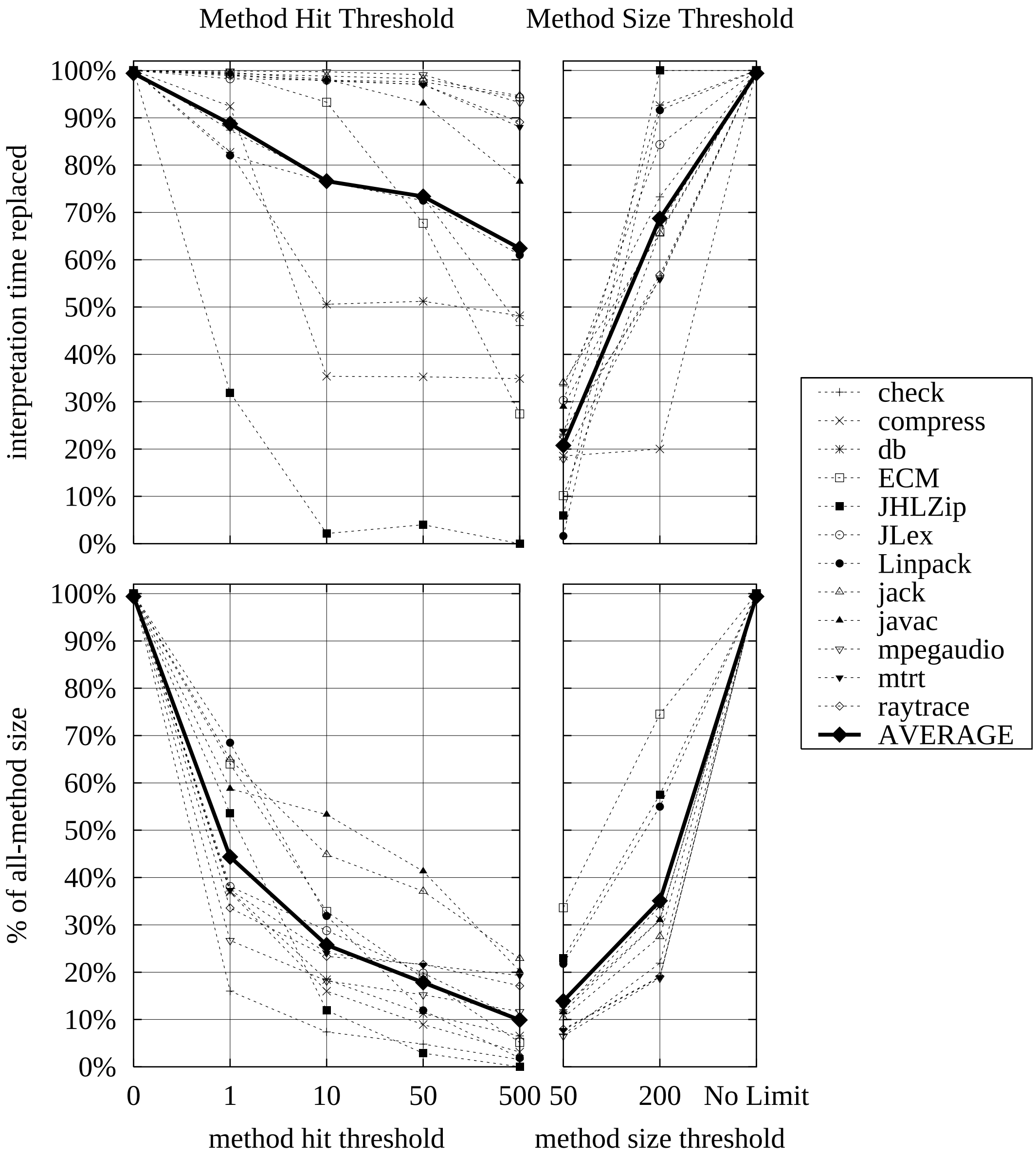

Figure 2: The effects on execution time (top) and cache size (bottom) of varying the hit threshold (left) and the size threshold (right) for the whole-method-only strategy. The size threshold is given as the method code size in bytes. The average, shown with a dark line, is the arithmetic mean of all twelve benchmarks.

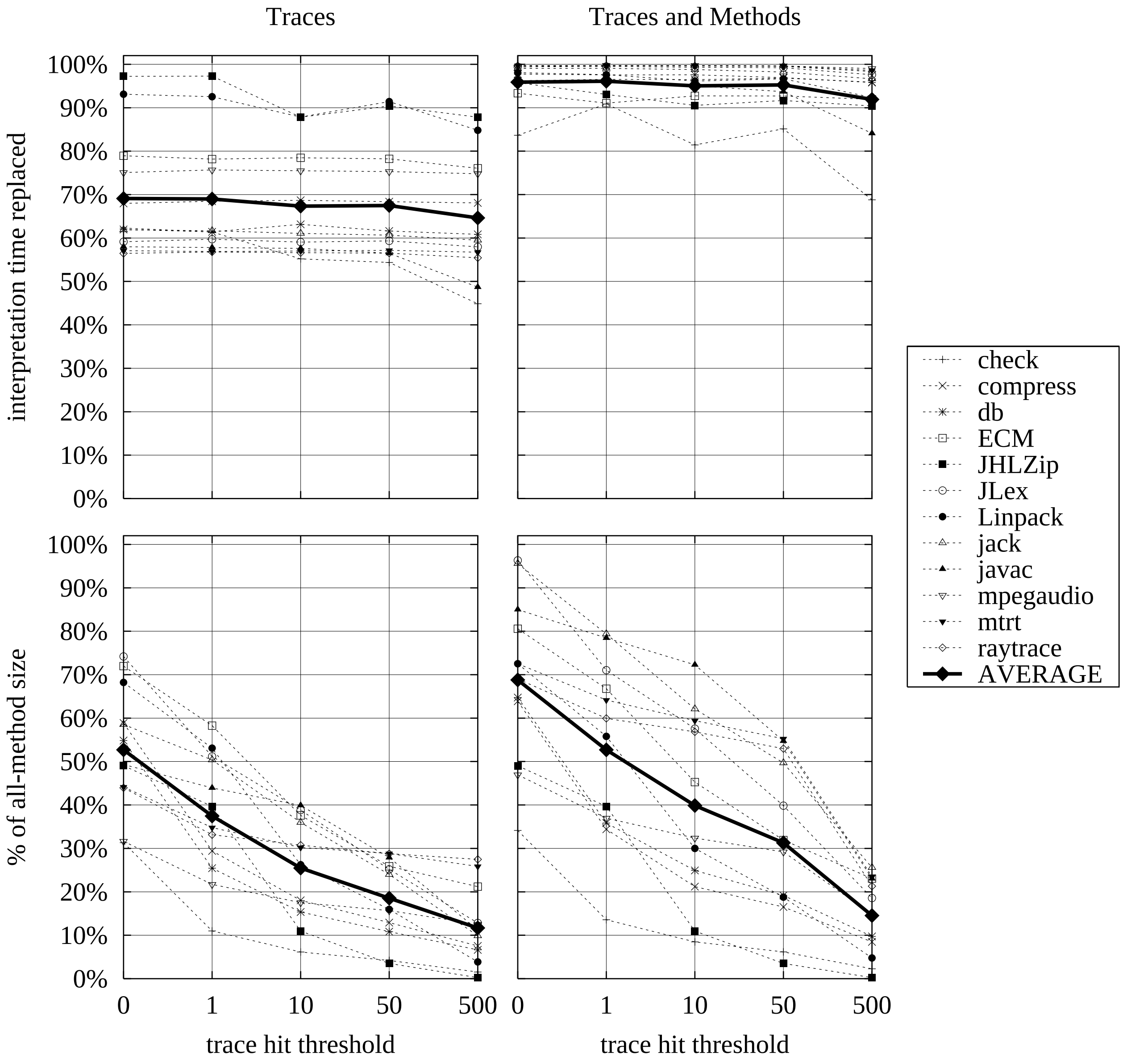

Figure 3: Time (top) and size (bottom) performance for traces by themselves (left) and for traces combined with methods (right). The methods have a hit threshold of 500 and no size threshold. The average, shown with a dark line, is the arithmetic mean of all twelve benchmarks.

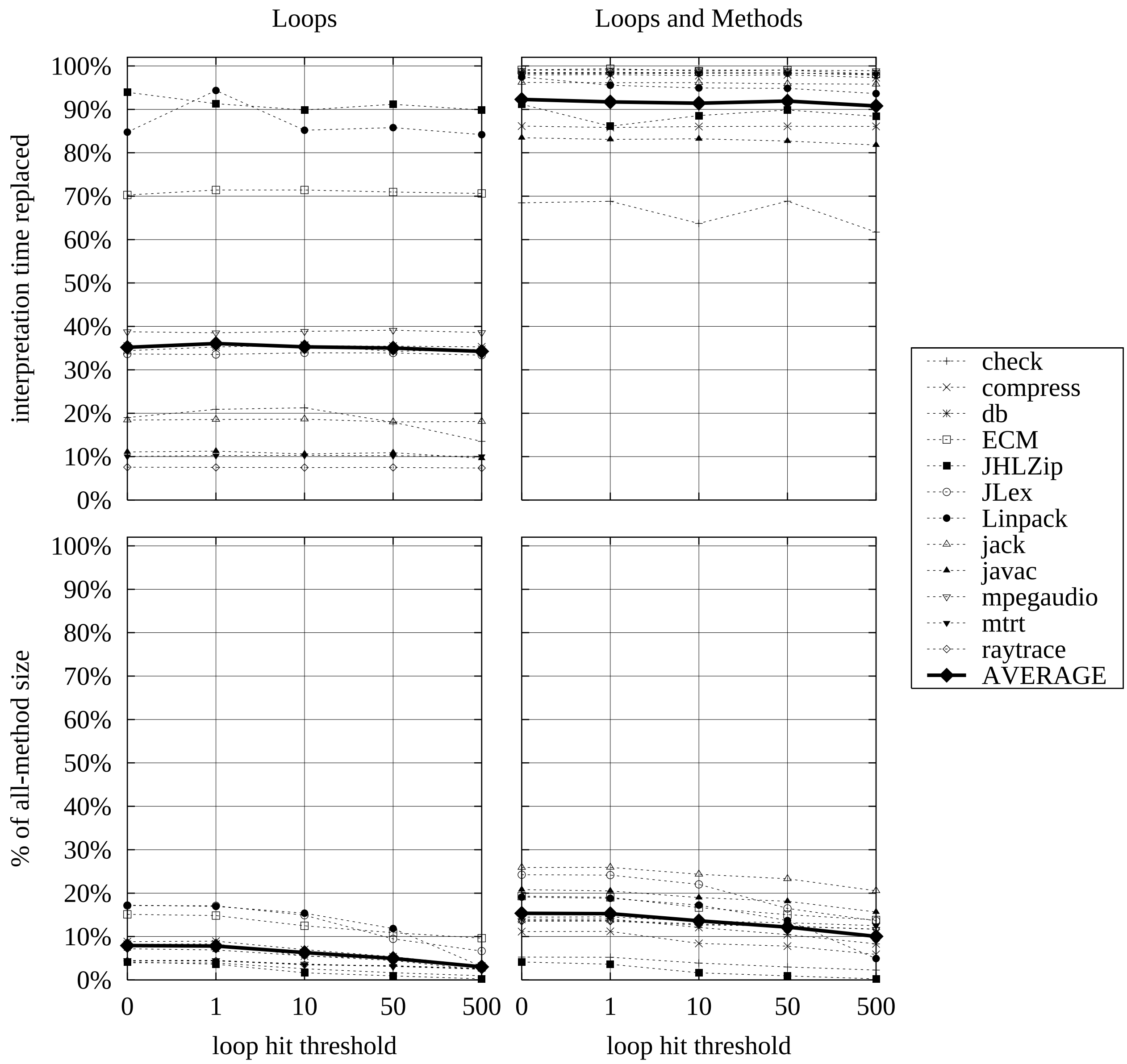

Figure 4: Time (top) and size (bottom) performance for loops by themselves (left) and for loops combined with methods (right). The methods have a hit threshold of 500 and no size threshold. The average, shown with a dark line, is the arithmetic mean of all twelve benchmarks.