2010 14th International Conference Information Visualisation

Visual Amortization Analysis of Recompilation Strategies

Stephan Zimmer and Stephan Diehl (Authors)
Computer Science Department
University of Trier
Trier, Germany
Email: zimmer23@web.de, diehl@uni-trier.de

***Abstract*—Dynamic recompilation tries to produce more efficient code by exploiting runtime information. Virtual machines like the Jikes RVM use recompilation heuristics to decide how to recompile the program, i.e. what parts are recompiled at what level of optimization. In this paper we present our post-mortem amortization analysis based on improved call stack sampling. Our tool presents the results of the analysis as an interactive visualizations to help both virtual machine implementors improve their recompilation strategies, as well as programmers assess whether these recompilation strategies pay off not only for their application as a whole, but also for individual methods.**

***Keywords*-software visualization; recompilation; dynamic analysis;**

I. Introduction

The Jikes RVM [1] has been developed as a research platform to experiment with different approaches to implement virtual machines. One of the most important features of the Jikes RVM is its adaptive optimization system [2], [3] which based on measurements of runtime performance decides when and how to recompile functions. Obviously, recompilation requires extra time at runtime, and thus a method should only be recompiled, if enough time is saved by future executions of the resulting, optimized code. In other words, the optimized code must be faster than the old code, and it must be executed a sufficient number of times later on. Unfortunately, in general the number of future executions of a method is not known at runtime, thus the Jikes RVM uses heuristics to control recompilation of methods.

From a programmer's point of view the question is not, whether these recompilation heuristics of the Jikes RVM pay off for many applications, but whether they really help to speed up his or her application. To this end we developed the post-mortem amortization analysis and visualization methods presented below and integrated these as a plugin into the Eclipse IDE.

Our approach to analyze the amortization of recompilation for a given application program consists of the following four steps:

1. Instrumenting the byte-code: The instrumentation computes the dynamic call graph and exact call counts for each method. In addition, the time spent in instrumentation code[1] is recorded.

2. Executing the code on an extended Jikes RVM: The extended virtual machine computes partial context trees.

3. Analyzing recorded data post-mortem: Based on the information in the partial context tree and the call counts, estimations of executions times can be computed after the execution of the application.

4. Visualizing the results: The overall amortization results are shown in the dynamic call graph, while recompilation diagrams present detailed information for each method.

For estimation of the execution times of each method we use call stack sampling. To this end we extended the Jikes RVM, such that at regular intervals a sampling thread interrupts the execution of the application and inspects its call stack. Our estimation is based on the assumption that *methods with short execution times are less likely to be found on the stack than those with longer execution times.*

II. Recompilation Heuristics

The Jikes RVM uses the following recompilation heuristics: *A method is recompiled, if the sum of the expected compilation time and the expected future execution time of the optimized method, is lower than the execution time of the method, so far*. More precisely:

$$C^{exp}_{m,i} + T^{exp}_{m,i} < T^{act}_{m,j} \quad \text{where } i,j \in \{0,1,2,3\} \text{ and } i > j \quad (1)$$

Here, $C^{exp}_{m,i}$ is the expected compilation time for method m at optimization level i. Level 0 corresponds to the Jikes `BASE` level compilation, 1 to `OPT0`, 2 to `OPT1`, and 3 to `OPT2`. $T^{exp}_{m,i}$ is the expected execution time of method m at optimization level i, and $T^{act}_{m,j}$ is the actual execution time of method m at the current optimization level j.

To compute the expected execution time, the Jikes RVM assumes that *the method is executed in the optimized code at least as often, as it has been executed, so far*.

III. Amortization of Recompilation

Once the execution of the application has finished, the recompilation of a method at level i actually paid off, if

[1]Note, that the instrumentation code itself is excluded from optimization.

1550-6037/10 $26.00 © 2010 IEEE
DOI 10.1109/IV.2010.76

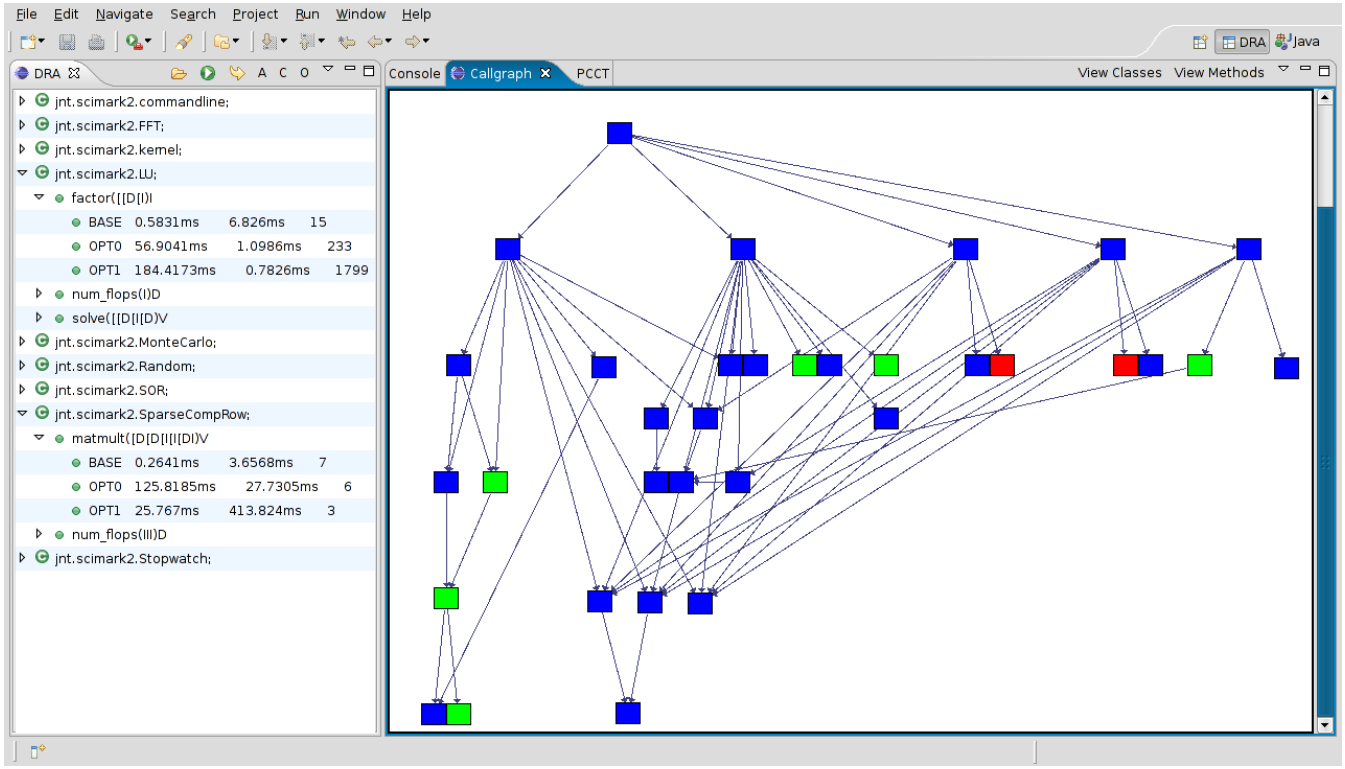

Figure 1. Dynamic call graph

its actual compilation and execution times instead of the expected times satisfy equation 1, i.e.:

$$C^{act}_{m,i} + T^{act}_{m,i} < T^{act}_{m,j} \tag{2}$$

The question is how to obtain these times. For the compilation times the answer is simple: The Jikes RVM actually records this information. To capture the execution times one could instrument the code at the start and end of each method to capture its execution. But, because other threads can interfere with the execution of a method, the computed times would be invalid. Instead, we use call stack sampling to estimate the execution times.

For our post-mortem amortization analysis we assume *that each invocation of a method takes the same amount of time*, thus we compute the product of the number of invocations and the average execution time of a method as an estimation of $T^{act}_{m,i}$:

$$C^{act}_{m,i} + (N^{act}_{m,i} * t^{post}_{m,i}) < (N^{act}_{m,j} * t^{post}_{m,j}) \tag{3}$$

$N^{act}_{m,i}$ is the number of times the method m at optimization level i is actually called. It is computed by the instrumentation code. $t^{post}_{m,i}$ is our post-mortem estimation of the average time for a single execution of method m at optimization level i.

IV. Partial Context Trees

To compute our post-mortem estimation $t^{post}_{m,i}$ we use a partial context tree [4] built by call stack sampling. In the context of this paper, a partial context tree $G = (V, E, r, \rho, \sigma)$ consists of

- a set V of nodes and a set E of edges, such that (V, E, r) forms a tree with root $r \in V$,
- a sampling count mapping $\sigma : V \to \mathbb{N}$,
- and a labeling $\rho : V \to M \times L$ where M is the set of method signatures and $L = \{0, 1, 2, 3\}$ the optimization levels.

The context of a node $v \in V$ consists of the labels along the path from the root to the node v, i.e. $context(v) = (\rho(v_1), \ldots, \rho(v_n))$ where $(v_1, \ldots, v_n)$ is the path from r to $v_n = v$. Furthermore, we define $parent(v) = w$ iff $\exists (w, v) \in E$.

To construct a partial context tree, a sampling thread interrupts the main application at regular intervals and traverses its call stack. As an example, consider the five call stack samples shown in Figure 2.

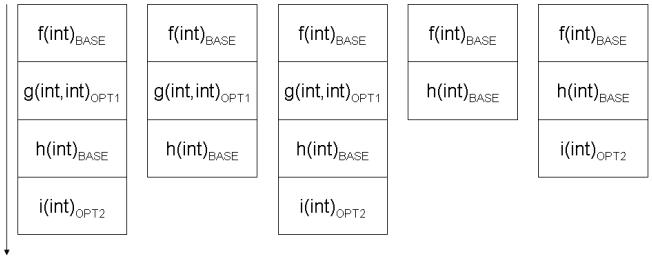

Figure 2. Stack samples

In these samples, every stack frame is annotated with the signature of the invoked function and its optimization level. From these samples the partial context tree shown in Figure 3 is constructed. Let v be the rightmost node in this tree, then $\sigma(v) = 1$, $\rho(v) = (i(int), 3)$, and $context(v) = ((f(int), 0), (h(int), 0), (i(int), 3))$.

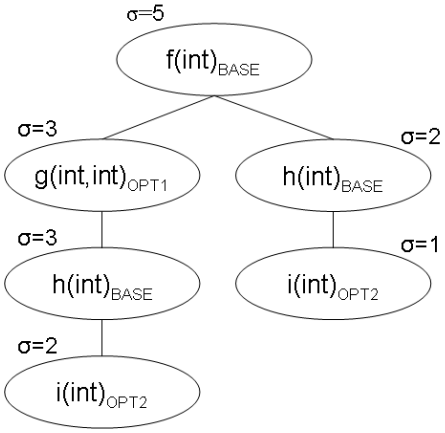

Figure 3. Partial context tree

Let T^{mt} be the total execution time of the main thread, i.e. the application, T^{instr} the time spent executing instrumentation code, and S the total number of samples, then

$$t_{unit} = (T^{mt} - T^{instr})/S \tag{4}$$

is the average time interval between two samples.

Our estimation algorithm first computes execution time estimates for every node in the partial context tree as follows:

- Initialization: $\forall v \in V : T^{post}(v) = undefined$
- For all leaves v of the tree, we set $T^{post}(v) = \sigma(v) * t_{unit}$.
- For all intermediate nodes v, we set $T^{post}(v) = \left(\sigma(v) - \sum_{(v,v') \in E} \sigma(v')\right) * t_{unit}$

For an intermediate node, the difference of the samples of the node and the sum of the samples of its children yields the number of samples where the context represented by this node exactly matches the stack of that sample. For our example, this algorithm yields the post-mortem times shown in Figure 5.

Based on the estimations of execution times of different nodes, and thus different contexts, we can now compute an estimation of the total execution time of a method m at optimization level i as the sum of the times of all nodes with label (m, i) in the partial context tree:

$$T^{post}_{m,i} = \sum_{v \in V \text{ and } (m,i) \text{ occurs in } context(v)} T^{post}(v) \tag{5}$$

Based on the times given in Figure 5, we get for example $T^{post}_{h(int),0} = 0.02$ ms and $T^{post}_{i(int),3} = 0.03$ ms. And finally, we estimate the average execution time by:

$$t^{post}_{m,i} = \frac{T^{post}_{m,i}}{N^{act}_{m,i}} \tag{6}$$

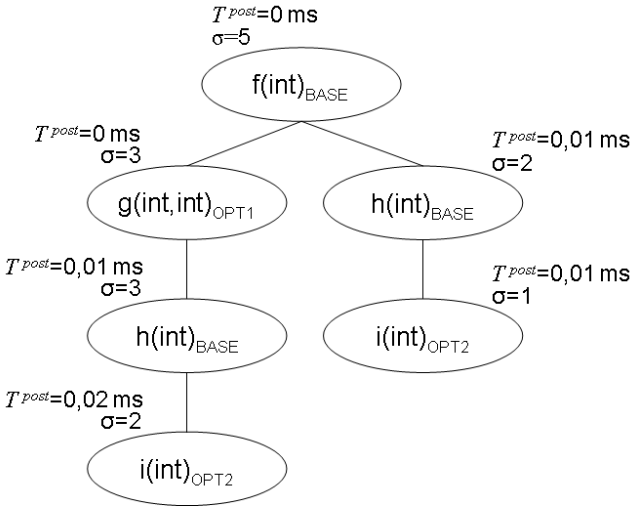

Figure 5. Partial context tree with post-mortem times assuming $t_{unit} = 0.01$ ms

Note, that call stack sampling does not allow to assess the number of actual method calls. For example, it is impossible to decide whether two stack frames of the same method in subsequent samples belong to the same method call. Instead we use byte-code instrumentation which in addition computes the dynamic call graph.

V. Improved Call Stack Sampling

One problem of the above approach is that it neither estimates an upper or lower bound of the execution times, if methods are inlined: First, the estimation of the execution time of the target method is too high, because it includes the execution time of the inlined code. Second, the estimation of the execution time of the inlined method is too low, because calls to this method have been replaced by the code of the method, and thus no stack frames are produced, and as a result, call stack sampling will not detect that the code of the method is actually executed.

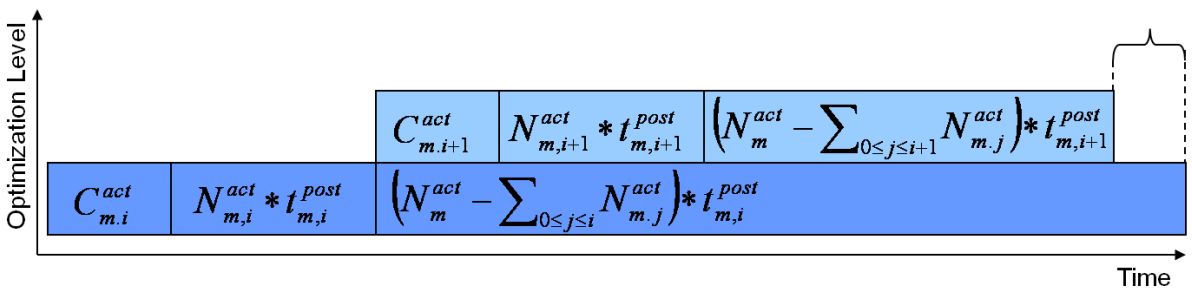

Figure 4. Recompilation diagram

Fortunately, for reasons such as inter-procedural analysis and debugging, the Jikes RVM provides a map from machine code offsets to inlining information, i.e. whether the code actually stems from the body of a method which was inlined. Using this information we extended our call stack sampling to also handle inlined methods correctly. The sampling thread first checks whether the current machine code instruction of the interrupted main thread stems from an inlined method[2]. In case, it proceeds as if a frame for the related method call would be on the stack: it adds a node to the partial context tree. As a result, also for inlined methods we have nodes in the partial context tree and the two problems mentioned above do no longer occur.

VI. Analysis and Visualization Tool

The above approach has been implemented as a plugin for the Eclipse IDE. To get an overview of the analysis results the user can look at a visualization of the dynamic call graph as shown in Figure 1. The nodes in these visualizations correspond to methods compiled at a certain optimization level. The color of a node indicates whether the method has not been recompiled at all (blue), or whether its recompilation paid off (green) or not (red). By clicking at a node of the graph the user can get detail information about the corresponding method. To visualize the various compilation and execution times we developed recompilation diagrams as shown in Figure 4.

For every optimization level three times are shown: the compilation time $C^{act}_{m,i}$, the estimated execution time $N^{act}_{m,i} * t^{post}_{m,i}$, and the remaining execution time $(N^{act}_{m} - \sum_{0 \le j \le i} N^{act}_{m,j}) * t^{post}_{m,i}$ at the same level, in case the method has been recompiled at a higher level again. Here N^{act}_{m} is the number of times any of the optimized instances of method m has been called, i.e. $N^{act}_{m} = \sum_{i \in \{0,1,2,3\}} N^{act}_{m,i}$.

VII. Example

To illustrate the use of our tool we chose the SciMark2.0 benchmark [5]. It consists of five scientific computations including fast fourier transform, LU factorization, and multiplication of dense matrices.

The screen dump in Figure 1 shows our Eclipse-plugin after the execution of the benchmark.

In the left view of all classes are listed. The user can browse through this list, expand the classes to see their methods and the related compilation and execution times.

In the right view the dynamic call graph is shown. In this graph the root node represents the `main()` method and its five children methods performing the above mentioned scientific computations. Eight methods have been recompiled (green and red nodes), for six of these the recompilation amortized later on (green nodes).

As an example of one of these successfully recompiled methods we take a closer look at the method `factor()` of the `LU` class. In the recompilation diagram in Figure 6 we see that the compilation times increase for higher levels of optimization, but that the execution times decrease dramatically – from 6.826 ms per method invocation at the BASE optimization level to 0.783 ms at level OPT1. In this example, the method was executed 15 times before it was recompiled for the first time, after another 233 invocations, it was recompiled again and executed another 1799 times. The pay off $\Delta_{0,1}$ for the first recompilation was much higher, than the pay off $\Delta_{1,2}$ for the second recompilation.

In Figure 1 we also see two red nodes indicating that there have been two methods for which the recompilation did not pay off. One of these methods is the method `bitreverse()` of the Fast Fourier Transform. The recompilation diagram for this case is shown in Figure 7. Here we see, that the recompilation at level `OPT0` only produced a negligible benefit, but even worse recompilation at level `OPT1` increased the overall execution time by 2,6 percent.

The second red node represents the method `matmult()` for which our amortization analysis yields a very surprising result as shown in Figure 8. After recompilation the execution time increases dramatically. The reason for this phenomenon is that for this method our assumption (and the assumption of most recompilation heuristics) does not hold, namely that overall all invocations of a method need about the same amount of time. In the SciMark benchmark the

[2]Actually, the approach works also for inlining sequences, i.e. in cases where a method was inlined in another method, that was itself inlined in yet another method, and so on.

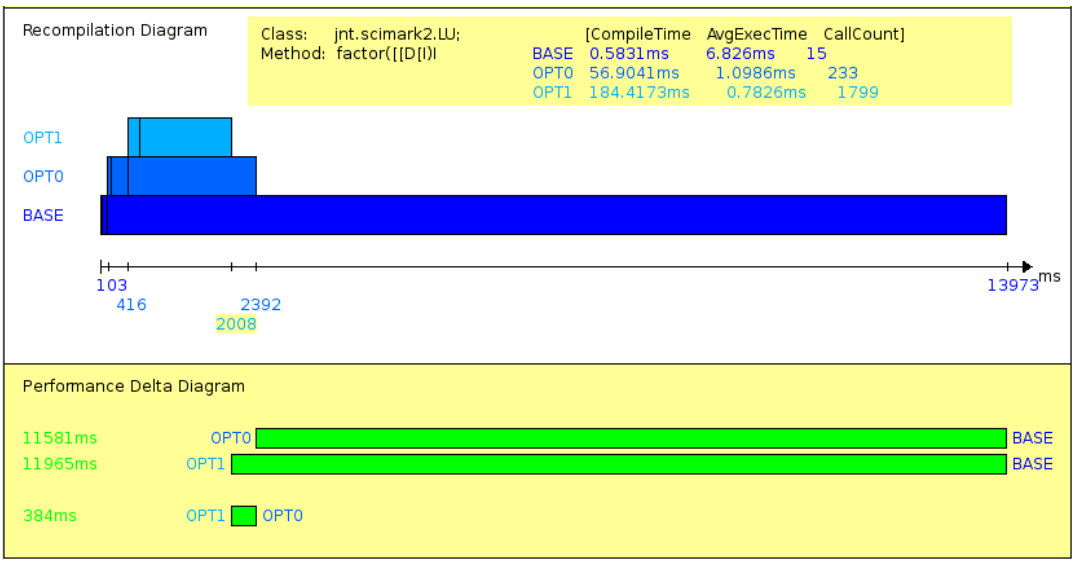

Figure 6. Recompilation diagram of method `factor()`

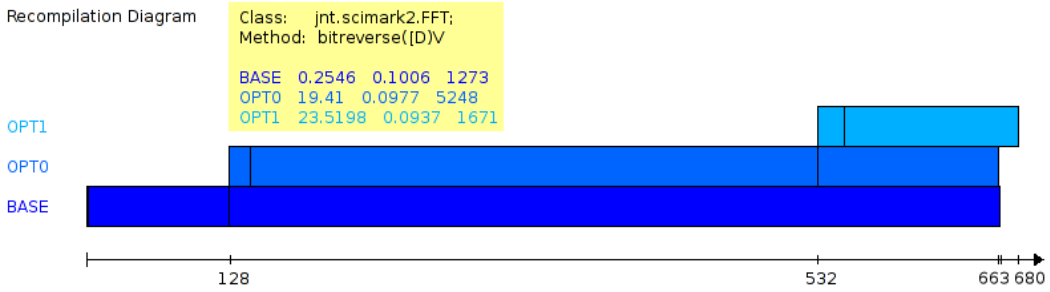

Figure 7. Recompilation diagram of method `bitreverse()`

method `matmult()` performs matrix-vector multiplication for n iterations. The benchmark invokes the method with increasing values of n, actually for the i-th invocation the value of n is 2^i. In other words, an invocation of the method requires twice the execution time of the previous invocation.

VIII. Related Work

Existing studies [6], [7], [8] report speedups at the level of applications or threads, but do not provide details for single methods. The goal of our work was not to perform studies, but to enable programmers and researchers to analyze dynamic recompilation of their own programs.

Visualization tools like TuningFork [9] or Performance Explorer [10] are very general visualization tools for events and performance data, in contrast our tool is tailored for a single purpose – inspecting the amortization of dynamic recompilation.

IX. Conclusions

In this paper we presented our approach to analyze the amortization of recompilation of individual methods. In our tool an overview of the analysis is shown by a color-coded dynamic call graph, whereas recompilation diagrams show details about single methods.

As our analysis is based on certain assumptions, it does not work for all kinds of programs. In these cases, other methods could be used to compute the compilation and execution times which are used as an input for the amortization analysis and visualizations presented here.

References

[1] The Jikes RVM Project, "Jikes RVM," http://jikesrvm.org, 2007.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "Adaptive optimization in the Jalapeño JVM," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications OOPSLA'00*. New York, NY, USA: ACM, 2000, pp. 47–65.

[3] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar,

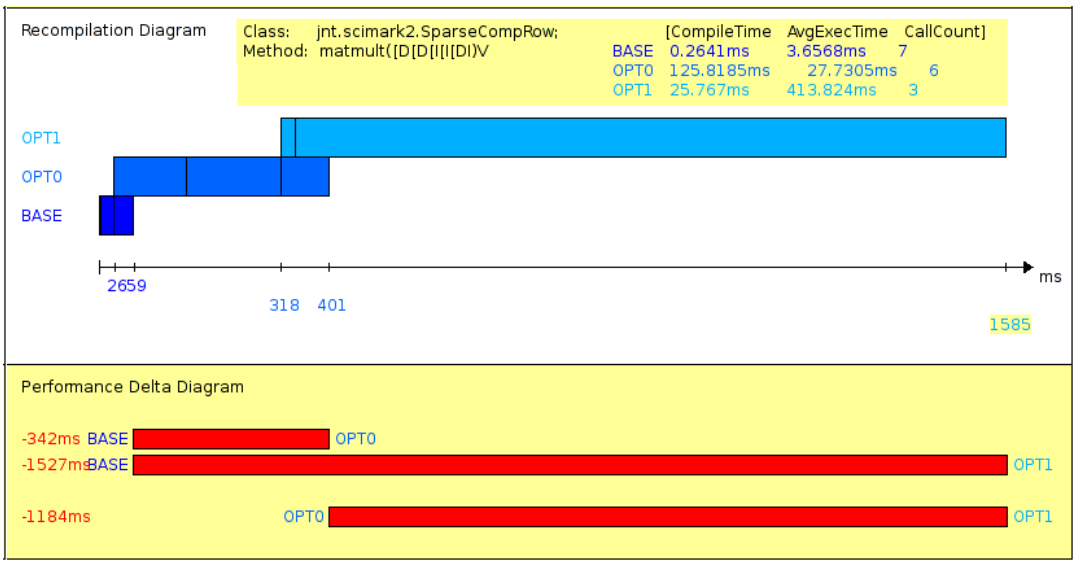

Figure 8. Recompilation diagram of method `matmult()`

"The Jikes research virtual machine project: building an open-source research community," *IBM Syst. J.*, vol. 44, no. 2, pp. 399–417, 2005.

[4] J. Whaley, "A portable sampling-based profiler for Java virtual machines," in *Proceedings of the ACM 2000 conference on Java Grande JAVA'00*. New York, NY, USA: ACM, 2000, pp. 78–87.

[5] Roldan Pozo and Bruce Miller, "SciMark 2.0," National Institute of Standards and Technology (NIST), http://math.nist.gov/scimark2/, 2007.

[6] P. Kulkarni, M. Arnold, and M. Hind, "Dynamic compilation: the benefits of early investing," in *Proceedings of the 3rd International Conference on Virtual Execution Environments VEE'07*. New York, NY, USA: ACM, 2007, pp. 94–104.

[7] M. Arnold, M. Hind, and B. G. Ryder, "An Empirical Study of Selective Optimization," in *Proceedings of 13th International Workshop on Languages and Compilers for Parallel Computing, Yorktown Heights, New York", August 10-12*, 2000.

[8] S. J. Fink and F. Qian, "Design, implementation and evaluation of adaptive recompilation with on-stack replacement," in *Proceedings of the International Symposium on Code Generation and Optimization CGO'03*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 241–252.

[9] D. F. Bacon, P. Cheng, D. Frampton, D. Grove, M. Hauswirth, and V. Rajan, "On-line visualization and analysis of real-time systems with TuningFork," in *Proceedings of the Fifteenth International Conference on Compiler Construction*, ser. Lecture Notes in Computer Science volume 3923, 2006, pp. 96–100.

[10] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind, "Using hardware performance monitors to understand the behavior of Java applications," in *Proceedings of the 3rd Conference on Virtual Machine Research and Technology Symposium VM'04*. Berkeley, CA, USA: USENIX Association, 2004.