View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal compiler

Gilles Duboscq[1] Thomas Würthinger[2] Hanspeter Mössenböck[1]
[1]Institute for System Software, Johannes Kepler University Linz, Austria [2]Oracle Labs, Linz, Austria
{duboscq, moessenboeck}@ssw.jku.at
thomas.wuerthinger@oracle.com

Abstract

Speculative optimizations are used in most Just In Time (JIT) compilers in order to take advantage of dynamic runtime feedback. These speculative optimizations usually require the compiler to produce meta-data that the Virtual Machine (VM) can use as fallback when a speculation fails. This meta-data can be large and incurs a significant memory overhead since it needs to be stored alongside the machine code for as long as the machine code lives. The design of the Graal compiler leads to many speculations falling back to a similar state and location. In this paper we present *deoptimization grouping*, an optimization using this property of the Graal compiler to reduce the amount of meta-data that must be stored by the VM without having to modify the VM. We compare our technique with existing meta-data compression techniques from the HotSpot Virtual Machine and study how well they combine. In order to make informed decisions about speculation meta-data, we present an empirical analysis of the origin, impact and usages of this meta-data.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization, Run-time environments

General Terms Algorithms, Languages, Performance

Keywords java virtual machine; just-in-time compilation; speculative optimization; metadata

1. Introduction

Deoptimization [6, 8] is a concept used in modern VMs that allows program execution to fall back from an optimized state to a less optimized state, e.g., from compiled code to interpreted code. The ability to deoptimize the running program at almost any position allows the JIT compiler of a VM to perform aggressive optimizations based on speculative assumptions. If those assumptions turn out to be wrong later, the optimized compiled program falls back to an unoptimized interpreted execution or a baseline compiler[1].

However deoptimization usually imposes memory costs because it require meta-data to be stored alongside the optimized machine code. While virtual machines may encode this data in a way that minimizes these costs, we believe that compilers can also improve on the amount of meta-data (i.e., *deoptimization information*) they handout to the VM. In this paper, we present *deoptimization grouping*, a VM-independent compiler optimization that helps reducing the amount of generated deoptimization information without compromising any speculative optimizations performed by the compiler.

We also want to gather empirical data on reduction of meta-data overhead using our technique and using existing compression techniques from the HotSpot VM. In particular, we want to see if both techniques can be combined constructively. We also collect empirical data about deoptimization information throughout the VM: its production by the compiler, its memory overhead while stored and finally its usage in the VM.

To summarize, the contributions of this paper are:

- An analysis of the origin of deoptimization, its impact on associated deoptimization information and its usage in the Java HotSpot VM.
- A VM-independent compiler optimization that reduces the amount of deoptimization information that needs to be stored by the VM.

2. Background

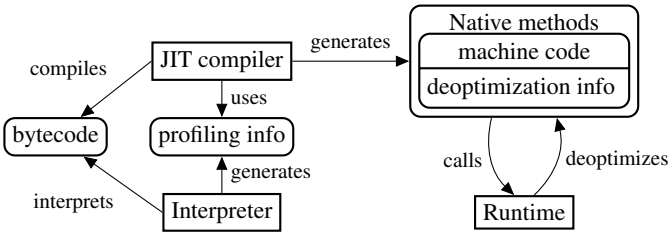


Figure 1: System overview

We implement deoptimization grouping as part of the Graal OpenJDK project [10] and its Graal compiler. The Graal compiler is written in Java and is designed to produce highly optimized code. It can be used to replace the standard compilers of the Java HotSpot VM.

[1] Deoptimizing to a baseline compiler is usually a special case of On Stack Replacement (OSR).

[Copyright notice will appear here once 'preprint' option is removed.]

Figure 1 shows the overall context of the VM. The Java HotSpot VM uses mixed-mode execution: all methods are initially interpreted, and frequently executed methods are scheduled for just-in-time (JIT) compilation. Thus execution of Java bytecodes starts in the interpreter, which is slow but has low startup costs and generates profiling information during interpretation. When a method has been executed for a certain number of times it is passed to the compiler, which uses the collected profiling information to generate optimized machine code for it (Figure 1).

The Java HotSpot VM has two JIT compilers: the Client compiler and the Server compiler. The Client compiler [9] aims at fast compilation speed, while the Server compiler [11] aims at better optimization at the expense of slower compilation speed. Both use speculative optimizations and deoptimization.

The Graal compiler is modular and has a clear interface with the VM such that it can be used in other VMs (e.g., in the Maxine VM [20]).

2.1 Deoptimization

The Graal compiler produces highly optimized code through extensive use of speculative optimizations. An optimization is speculative when the compiler makes an assumption that it cannot guarantee during compilation. Instead, it requires the VM to monitor the assumption and discard the machine code when the assumption no longer holds. For example, the compiler can replace a virtual method call with a static call and then inline a method if there is currently only one implementation available for it. If later class loading adds another implementation, the assumption no longer holds and the code is deoptimized.

Deoptimization can also be triggered by the machine code directly. For example, if profiling shows that the receiver of a virtual call is always of a specific type, the compiler can speculate that the receiver will always be of this type. It inserts a *guard* to check the type and can then devirtualize the call. If the type check from the guard fails, the guard will explicitly trigger a deoptimization and transfer execution to the interpreter.

To this end, the VM uses deoptimization information generated by the compiler to reconstruct the state of the interpreter (i.e., the state of the virtual machine) from the state of the physical machine running the compiled code. In the Java VM, this state consists of the contents of the local variables and the operand stack. In the context of escape analysis [16], where allocations of method-local objects are eliminated, the deoptimization information also contains the mapping necessary to reconstruct such objects on the heap. Figure 1 shows that this data is directly associated with the machine code of a compiled method.

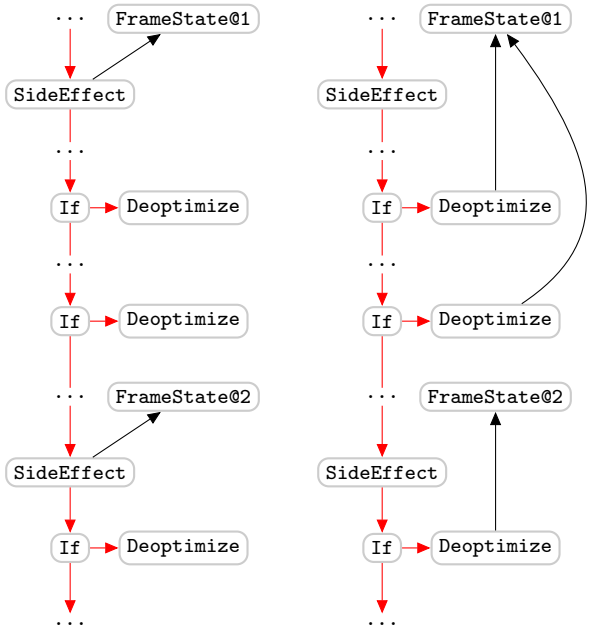
2.2 FrameState Assignment

Deoptimization can only happen at specific points in the program, e.g., when a guard fails. The corresponding intermediate representation (IR) nodes [4], called *deoptimizing nodes*, need deoptimization information, which is stored in so-called *`FrameState` nodes* in Graal. These nodes store the method and bytecode index to deoptimize to and for each value of the local variables and the expression stack, they reference the corresponding Static Single Assignment (SSA) IR node.

The Graal compiler generates `FrameState` nodes during bytecode parsing [5]. However, instead of attaching them to deoptimizing nodes immediately, it rather attaches them first only to nodes that can have side-effects (see Figure 2a). Note that we define *side-effect* as anything that changes a state that is observable by other threads or that is not fully captured by `FrameState` nodes. Between side-effecting nodes the state of the program cannot change and so all deoptimizing nodes can use the `FrameState` of their most recent side-effecting node.

Keeping the `FrameState` only in side-effecting nodes has the advantage that deoptimizing nodes can be moved during compiler optimizations. When a deoptimizing node is moved across a side-effecting node it takes the `FrameState` from the new most recent side-effecting node.

After optimizations and before the IR is passed to the register allocator, `FrameState` nodes are detached from the side-effecting nodes and attached to the proper deoptimizing nodes (see Figure 2b).



(a) Before (b) After

Figure 2: `FrameState` assignment: `FrameState` nodes are transfered from side-effecting nodes to deoptimizing nodes.

3. Deoptimization Grouping

Because of the way in which `FrameState` assignment is done in Section 2.2, many deoptimizing nodes will share the same `FrameState`. While this means that the frame they deoptimize to is the same, the low-level deoptimization information that needs to be stored is not necessarily the same. Indeed, the physical locations of the values may be different. For example, as illustrated in Figure 3, two `Deoptimize` nodes use the same `FrameState`. At both `Deoptimize` nodes, the same SSA values need to be restored for the locals and the expression stack but they are found in different physical registers or stack slots. This is important because the deoptimization information that needs to be produced by the VM is the low-level one.

This is the point where deoptimization grouping comes into play. The idea of deoptimization grouping is to combine all `Deoptimize` nodes with the same logical `FrameState` into a single `Deoptimize` node. This single `Deoptimize` node will then result in a single deoptimization site in the generated code which needs only one low-level translation of this `FrameState`.

To achieve this, once `FrameState` nodes have been assigned to deoptimizing nodes, the compiler groups `Deoptimize` nodes which use the same `FrameState`. As Figure 4 illustrates, for each group containing more than one `Deoptimize` node, a control-flow `Merge` node is created, followed by a single `Deoptimize` node. All

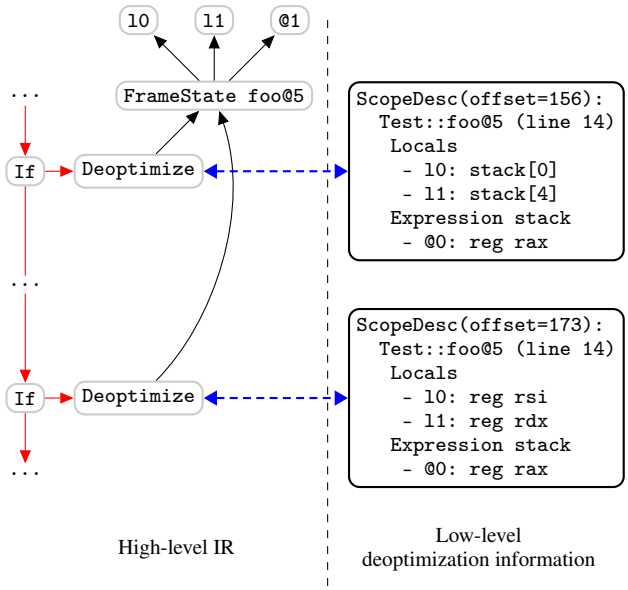


Figure 3: Low-level representation of the same `FrameState` used at two different positions. Note that the locals are in different physical locations.

previous branches leading to `Deoptimize` nodes of this group flow into this new `Merge` node.

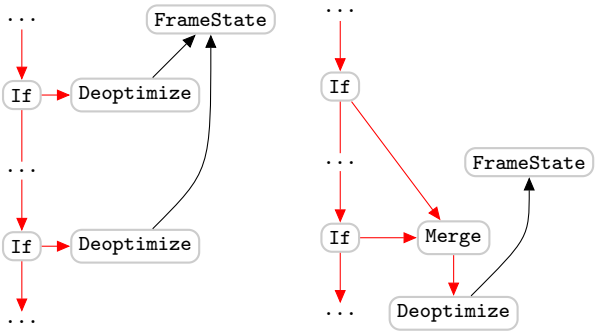


Figure 4: Merging deoptimization control flow (high-level IR).

Note that this transformation is a pure compiler transformation and is completely independent of the virtual machine in which this compiler is used.

With this transformation we expect a slight decrease of the generated code size because there are less runtime calls for deoptimization. But more importantly we expect a more substantial decrease of the size of deoptimization information.

Note that this transformation only covers cases where there is explicit control flow leading to a `Deoptimize` node in the IR. Currently, this is not the case for all deoptimization in the Graal IR. There are other deoptimizing nodes such as `Invoke` nodes or `SafePoint` nodes for which the control flow to a potential deoptimization is implicit. However it would be possible to add a successor to those nodes to explicitly represent the deoptimization continuation and thus make them participate in deoptimization grouping. In the current state, experiments show that about 38% of the final deoptimization information is associated with a `Deoptimize` node and explicit control-flow and are thus amenable to this optimization.

4. Related Work

4.1 Deoptimization Information Compression

The HotSpot VM can compress low-level deoptimization information while storing it. This is normally done only when some debugging features of the VM are used and thus more deoptimization information is produced by the compiler. Compression works by finding common byte sequences in the serialized deoptimization information and sharing them. In HotSpot, this compression is called "Debug Information Sharing" [1]. It is done for specific chunks of serialized meta-data, namely the list of all values of the expression stack, the list of all values of the locals, the list of all locked monitors and one full frame. This ad-hoc compression is rather effective (see Section 5.5). Especially, since it separates the expression stack from the locals it is able to share information at a smaller granularity than the full stack of frames. However, it can not find common sequences when the values have moved to different physical locations between two usages of the same `FrameState`. Also, it has quadratic complexity since every time a new chunk is recorded, it needs to compare it with the previous ones. In order to limit the time taken by this search, HotSpot only looks at the 50 previous chunks. Comparatively, since Graal maintains skip-lists for some node types and def-use edges [4], deoptimization grouping has a linear complexity in terms of `Deoptimize` nodes.

4.2 Delta encoding

Based on the insight that deoptimization information is the result of the abstract interpretation of the compiled program, some VMs use delta encoding for storing deoptimization information. The idea is that the delta between two instances of deoptimization information that follow each other in the control-flow should be rather small. This was described by Schneider and Bolz [14] for RPython. This is particularly effective for trace compilers where the compilation unit is essentially linear with side exits.

In RPython, the deoptimization information that is serialized is split into two parts: "resume data" which corresponds to Graal's high-level `FrameSate` nodes and "backend maps" which give the low level information by mapping SSA values to their physical location. This makes the delta encoding more efficient because it is independent of the physical location of values.

The LuaJIT compiler, like most trace compilers, uses deoptimization intensively for trace exits. It stores deoptimization information in so-called "snapshots" [12, 13]. Similar to Graal's handling of `FrameState` nodes, snapshots are only taken if there has been a side-effect since the last snapshot or if an exit is likely to be taken. Similar to RPython, snapshots reference the IR values and thus are independent of the value's physical location. Instead of using backend maps, since LuaJIT keeps the IR in memory, it looks for special `RENAME` IR nodes which indicates that the register allocator moved the value from one physical location to an other.

4.3 Deoptimization Information Size

The SELF VM [8, 19] is an ancestor of the HotSpot VM and contains lot of the infrastructure that was later used for deoptimization in the HotSpot VM. They report that compared to machine code, deoptimization information takes almost as much space (1.2×) for their baseline compiler and almost twice as much space (2.3×) for their optimizing compiler.

Using the delta encoding described above, the RPython implementation is able to compress "resume data" (the biggest part of RPython's deoptimization information) by up to 82%. After com-

pression the total deoptimization information takes $2.6\times$ to $5\times$ as much space than the machine code.

4.4 Meta-data vs. Code

While most systems that support deoptimization do so by using meta-data, it is also possible to support it by using only code. Instead of emitting meta-data and calling a centralized deoptimization handler to rebuild the deoptimized state, specialized code which does not need the meta-data can be emitted for each deoptimization point.

For example, with the Hotpath VM, Gal et al. [7] have explored both options and report that using specialized code is better for performance. However they do not report any findings in terms of memory overhead.

5. Evaluation

5.1 Methodology

All the data from this section was recorded using Graal VM, a modified version of the HotSpot VM which supports the Graal compiler as well as the original compilers. It is based on version `eff84c561a95` of the GraalVM code repository[2] which is extended with the necessary instrumentation code. We used the Java library from Oracle's 1.8.0 JDK. The machine that was used is based on an Intel Xeon E5-2690 CPU with 96GB of RAM running Ubuntu 12.04. HotSpot's "tiered compilation" mode was disabled to prevent mixing numbers from the Client compiler and from the Server or Graal compilers.

We used the DaCapo 9.12 [2, 3], the Scala-DaCapo [15], SPECjvm2008 [18] and SPECjbb2005 [17] benchmark suites. For DaCapo, Scala-DaCapo and SPECjvm2008 the different benchmarks of the suites were run separately in distinct VM processes. We then obtain our results for each suite by summing the numbers. For SPECjvm2008, the `startup` benchmarks were not included because we are not interested in startup performances in this paper. The `compiler.sunflow` benchmark from SPECjvm2008 and the `eclipse` benchmark from DaCapo are not included because of compatibility issues with Java 8.

The reported numbers are the average of at least 10 runs of the benchmarks. Each run happens in a different VM process. For each DaCapo and Scala-DaCapo benchmark, 29 warm-up iterations were used for a total of 30 iterations. For each SPECjvm2008 benchmark, a warm-up time of 60 s and an iteration time of 120 s was used. For SPECjbb2005, since the machine has 16 logical CPUs, we used 32 "warehouses".

The Graal compiler is written in Java and thus compiles itself along the application in the JVM. Thus, the measurements for Graal may be slightly polluted by compilation of Graal method's or the methods it calls in the Java standard library. To minimize this effect, we bootstrap the compiler at the start of the VM before running the application and discard any measurement acquired during this bootstrap period.

Besides discarding the measurement of Graal's bootstrap, the measurement are accumulated during the complete run. This is important because most compilations happen while the benchmarks are warming up.

5.2 Deoptimization Information Origin

First, we would like to study where deoptimization information is coming from and establish a baseline for our evaluation. To do this, we have added instrumentation code to three compilers: the Graal compiler and HotSpot's Client and Server compilers. The instrumentation reports how many machine code locations have been linked with deoptimization information and why it was needed.

Table 1 shows how the different compilers use deoptimization information. At this point, deoptimization grouping or HotSpot's debug information sharing are not enabled yet. We divide the total number of deoptimization information produced in each category by the number of compiled methods to obtain an average per method. Note that a compiled method may include more than one Java method because of inlining. The different categories are described under Table 1.

Overall the Graal compiler produces more deoptimization information than the two other compilers. Some of the difference can be explained by the different typical sizes of the compilation units due to different inlining policies. Indeed, in Table 2 we can see that on average, the Graal compiler produces bigger compiled methods than the Server compiler ($1.5\times$ to $2.1\times$ bigger) which in turn produces larger compiled methods than the Client compiler ($1.3\times$ bigger). Some of the difference can also be explained by the intensive usage of deoptimization for exception handling in the Graal compiler [4].

5.3 Deoptimization Information Memory Overhead

Deoptimization information needs to be stored and we now want to study its memory cost. We slight extended existing instrumentation from the HotSpot VM to gather those statistics. The information is collected when the result of a compilation is inserted in the VM's code cache. This means that it shows how much data has been generated by the compiler over the complete lifetime of the VM and not statistics about the content of the code cache at a specific point in time. Table 2 shows how much memory is used to store deoptimization information.

We can see that overall, meta-data has a significant overhead: $1.7\times$ to $1.9\times$ the amount of machine code for the Graal compiler, $1.2\times$ to $2.0\times$ for the Server compiler and $1.2\times$ to $1.5\times$ for the Client compiler. These numbers are similar to the numbers put forward by Hölzle et al. [8] for the SELF VM. They are much smaller than those from Schneider and Bolz [14] for RPython, even when using delta-encoding. However even if Graal uses speculation more intensively than the Server or Client compiler, it does not reach the density of guards found with RPython's tracing JIT.

5.4 Deoptimization Grouping

Using deoptimization grouping allows us to reduce the amount of low-level deoptimization information that the compiler produces. To observe the effect of this technique, we collected data about deoptimization information usage for the Graal compiler with deoptimization grouping enabled. Table 3 reports those numbers and show the change as a percentage of the numbers without deoptimization grouping. The amount of deoptimization information emitted by the compiler is reduced by about a quarter which confirms our intuition that a significant amount of `Deoptimize` nodes use the same `FrameState` nodes.

Benchmark	Grouping		Change
	Disabled	Enabled	
DaCapo	1,197,277	880,136	-26.49%
Scala-DaCapo	928,651	695,735	-27.74%
SPECjvm2008	937,947	694,378	-25.97%
SPECjbb2005	49,148	35,512	-25.08%

Table 3: Effect of deoptimization grouping on deoptimization information usage in the Graal compiler.

In Table 5, the "Sharing Disabled" section shows the effect of grouping on the memory overhead of deoptimization information.

[2] `http://hg.openjdk.java.net/graal/graal/`

Benchmark	Compiled Methods	Average Low-level Deoptimization Information Produced per Compiled Method					
		Invokes	Exceptions	Runtime Calls	Speculations	Safepoints	Total
Graal							
DaCapo	32,966	6.83	17.77	2.75	7.84	1.13	36.32
Scala-DaCapo	35,449	5.39	12.19	2.23	5.68	0.70	26.20
SPECjvm2008	21,240	8.90	22.20	3.29	8.39	1.38	44.16
SPECjbb2005	1,069	8.23	22.73	4.18	9.43	1.41	45.98
Server							
DaCapo	20,427	4.76	7.54	1.31	2.60	0.39	16.60
Scala-DaCapo	25,540	3.56	5.76	1.52	1.89	0.22	12.95
SPECjvm2008	10,321	4.40	8.92	1.55	2.28	0.41	17.57
SPECjbb2005	531	3.59	7.34	1.77	2.51	0.56	15.78
Client							
DaCapo	31,196	4.87	5.40	1.97	0.16	0.42	12.82
Scala-DaCapo	45,970	3.55	3.11	1.67	0.05	0.12	8.49
SPECjvm2008	14,719	3.92	5.07	2.04	0.29	0.50	11.82
SPECjbb2005	709	4.57	5.90	2.68	0.36	0.59	14.09

Table 1: Deoptimization information origin for different benchmarks and compilers. Deoptimization grouping and HotSpot's debug information sharing are disabled. The different origins are categorized as follow: *Invokes* is for call sites to other Java methods; *Exceptions* is for exception handling either explicitly using deoptimization or implicitly through hardware traps; *Runtime Calls* is for call sites to the runtime that require deoptimization information; *Speculations* is for deoptimizations used by speculative optimizations other than for speculative exception handling; *Safepoints* is for safepoints used by the VM to preempt execution for GC, deoptimization etc.

Benchmark	Compiled Methods	Average Memory Footprint per Compiled Method (bytes)				
		Code	Meta-data			Total
			Deoptimization	Shared	Other	
Graal						
DaCapo	32,965	2,190.1	2,996.5	311.2	482.0	5,993.9
Scala-DaCapo	35,438	1,589.9	2,293.3	298.5	460.9	4,658.1
SPECjvm2008	21,274	2,647.6	3,771.0	426.4	527.2	7,385.7
SPECjbb2005	1,075	2,810.4	3,957.1	386.6	524.2	7,691.7
Server						
DaCapo	20,424	1,432.3	1,294.5	76.0	635.2	3,451.0
Scala-DaCapo	25,543	991.1	1,308.4	86.2	581.9	2,981.1
SPECjvm2008	10,321	1,506.0	1,344.5	72.1	639.2	3,574.6
SPECjbb2005	531	1,484.1	1,159.3	63.3	588.2	3,308.3
Client						
DaCapo	31,194	1,040.0	747.2	34.1	532.4	2,368.7
Scala-DaCapo	46,102	750.8	674.6	34.8	449.9	1,925.2
SPECjvm2008	14,716	957.1	675.3	29.3	502.2	2,178.4
SPECjbb2005	710	1,177.9	853.2	30.0	508.3	2,584.9

Table 2: Memory footprint in bytes for deoptimization information for different benchmarks and compilers. Deoptimization grouping and HotSpot's debug information sharing are disabled. The *Shared* column is for constants that can be referenced either from deoptimization information or from the code. The *Other* column accounts for relocation information as well as other data needed for the VM's bookkeeping of the code cache.

We can see that the reduction by a quarter in emitted deoptimization information translates almost directly into a reduction by a quarter of the deoptimization meta-data size. It also translates into a slight decrease in code size as we expected because of the lower number of runtime call sites to the VM's deoptimization handler.[3] Interestingly, it also leads to a decrease for other meta-data which contains relocation information for those runtime call sites.

[3] The observed decrease in code size is significant, it is outside the 99% confidence interval.

5.5 Debug Information Sharing

As described in Section 4.1, the HotSpot VM supports a compression technique for deoptimization information. Table 4 shows the change in memory usage with sharing enabled. Note that for this experiment, deoptimization grouping is not enabled such that we can see the effect of sharing alone.

Debug Information Sharing is very effective at compressing deoptimization meta-data and results in a 40% to 50% decrease in the memory needed to store deoptimization meta-data. However since this sharing is done while deoptimization information is recorded and after the compilation is finished, it has no influence on any-

thing else.[4] Overall, it results in a 11% to 25% decrease in code cache occupation and brings the overhead of meta-data down to between $0.9\times$ and $1.3\times$ for all compilers and to between $1.1\times$ and $1.2\times$ for the Graal compilers in particular.

Benchmark	Code	Meta-data			Total
		Deopt.	Shared	Other	
Graal					
DaCapo	-0.7%	-47.1%	-0.4%	-0.2%	-23.8%
Scala-DaCapo	+0.3%	-48.6%	+0.4%	+0.1%	-23.8%
SPECjvm2008	-0.2%	-45.4%	-0.2%	-0.3%	-23.3%
SPECjbb2005	-1.7%	-48.5%	-0.9%	-0.9%	-25.7%
Server					
DaCapo	-0.5%	-41.7%	-0.1%	-0.1%	-15.8%
Scala-DaCapo	-0.2%	-50.4%	-0.2%	-0.1%	-22.2%
SPECjvm2008	+0.3%	-41.0%	+0.2%	+0.2%	-15.3%
SPECjbb2005	+1.1%	-35.4%	+0.3%	+0.5%	-11.8%
Client					
DaCapo	+0.0%	-40.0%	+0.0%	+0.0%	-12.6%
Scala-DaCapo	-0.3%	-50.3%	-0.5%	-0.5%	-17.9%
SPECjvm2008	+0.1%	-37.7%	+0.1%	+0.1%	-11.6%
SPECjbb2005	-0.1%	-41.1%	-0.1%	-0.2%	-13.7%

Table 4: Change in memory footprint of deoptimization information for different benchmarks and compilers when HotSpot's debug information sharing feature is enabled. The *Total* column corresponds to the total footprint (code & meta-data). The numbers are relative to the baseline established in Table 2

5.6 Combining Sharing and Grouping

We now want to study how debug information sharing and deoptimization grouping work together. Table 5 shows the change in memory usage of deoptimization information when grouping is enabled.

The results show that deoptimization grouping is still able to reduce code and meta-data size on top of the gains from debug information sharing. The reduction of footprint for code, shared and other meta-data is conserved from the situation where sharing is disabled and grouping further reduces the size of deoptimization meta-data by 22% to 25%. This is important because it shows that both techniques can be used together and combine well. Overall, the overhead of meta-data is now down to between $0.9\times$ and $1.1\times$ for the Graal compiler.

5.7 Runtime Accesses to Deoptimization Information

Storage of deoptimization information has to trade-off between size and ease of access. We want to look at how often this deoptimization information is accessed at runtime and for which reason. We have added instrumentation to the HotSpot VM that counts every access to deoptimization information. These accesses are categorized into different access reasons. Table 6 shows the number of these accesses for the three compilers. It also shows the runtime of the benchmarks to put the number of accesses into perspective.

Overall, accesses due to deoptimization are a minority, they represent at most 0.5% of all accesses. This is interesting because deoptimization is the only kind of access which needs all of the deoptimization information. Other accesses only inspect the method and bytecode index but not the values contained in each frame. We suspect that the method and bytecode index only make up a small part of the deoptimization information overhead. This means that most of the data from deoptimization information could be compressed more aggressively since it is only rarely decoded.

We can also see that deoptimization happens more often with the Graal compiler. This confirms that Graal replies more on speculative optimization than the Server and Client compilers.

[4] The observed variations for the other values are not significant, they lie within the 99% confidence interval.

Benchmark	Code	Meta-data			Total
		Deopt.	Shared	Other	
Sharing Disabled					
DaCapo	-4.3%	-25.4%	-10.6%	-3.8%	-15.1%
Scala-DaCapo	-3.7%	-23.5%	-8.5%	-2.7%	-13.7%
SPECjvm2008	-2.5%	-24.7%	-11.6%	-3.1%	-14.4%
SPECjbb2005	-5.3%	-27.1%	-10.8%	-4.0%	-16.7%
Sharing Enabled					
DaCapo	-4.0%	-58.7%	-10.4%	-3.6%	-31.6%
SPECjbb2005	-4.7%	-60.2%	-11.5%	-3.6%	-33.5%
SPECjvm2008	-3.5%	-57.9%	-12.1%	-3.5%	-31.8%
Scala-DaCapo	-3.4%	-59.7%	-8.1%	-2.4%	-31.3%

Table 5: Change in memory footprint of deoptimization information for different benchmarks when Graal's deoptimization grouping feature is enabled. The *Total* column corresponds to the total footprint (code & meta-data). The numbers are relative to the baseline established in Table 2

Benchmark	Time	Accesses		
		Deopt.	Stack-trace	Other
Graal				
DaCapo	794 s	9,734	277,678,887	28,198,368
Scala-DaCapo	1612 s	28,588	32,286,983	8,635,354
SPECjvm2008	3600 s	31,773	114,096,022	777,256,453
SPECjbb2005	4050 s	448	5	94,801
Server				
DaCapo	739 s	1,909	298,273,177	21,217,367
Scala-DaCapo	1711 s	3,252	28,407,976	11,437,794
SPECjvm2008	3600 s	1,371	112,355,484	524,800,317
SPECjbb2005	4050 s	86	0	17,229
Client				
DaCapo	1052 s	59	470,958,570	31,239,622
Scala-DaCapo	4884 s	72	61,995,783	2,518,778
SPECjvm2008	3600 s	62	80,820,622	738,751,894
SPECjbb2005	4050 s	1	522	38,075

Table 6: Runtime accesses to deoptimization information. The *Stack-trace* category is used when the VM builds a stack-trace for an exception object. The *Other* category is used when the VM walks the stack for the implementation of security or class loading.

6. Future Work

Currently, we do not try to group deoptimizations using `FrameState` nodes that are structurally equivalent but have different input SSA values. While this is not a very common case it can appear as a result of optimizations that duplicate code such as loop peeling, unrolling or unswitching. We would need to analyze the costs and benefits of grouping these deoptimization.

It would also be interesting to analyze the performance and memory overhead trade-offs associated with using specialized code instead of meta-data to implement deoptimization. One of the potential advantages of implementing deoptimization through spe-

cialized code is that it reduces the need for the VM to support advanced deoptimization features such as scalar-replaced objects, delayed loads and stores etc. The implementation of these compiler optimizations would then be less VM-dependent and better factored into the compiler.

An other interesting path to explore would be to make all deoptimization explicit in the Graal IR by adding a special successor to all the deoptimizing nodes for which deoptimization is currently implicit. This would make all deoptimizations candidate to deoptimization grouping and thus improve the gains of this optimization.

7. Conclusions

In this paper, we have empirically confirmed that the overhead of machine code meta-data is significant in the HotSpot VM and in particular the meta-data used for deoptimization. We have first studied how the HotSpot VM successfully compresses some of the meta-data associated with deoptimization. Then, we have empirically confirmed that the compiler can also help when it comes to reducing the meta-data overhead.

Using deoptimization grouping we were able to reduce this overhead without having to modify the VM which means that this improvement will be effective in any VM where the Graal compiler is used. Furthermore, our compiler optimization combines well with meta-data compression techniques that can be found in virtual machines.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback.

This work was performed in a research collaboration with Oracle Labs.

References

[1] `src/share/vm/code/debugInfoRec.cpp`. URL `http://hg.openjdk.java.net/jdk8/jdk8/hotspot`.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190. ACM Press, Oct. 2006. .

[3] DaCapo Project. *The DaCapo Benchmark Suite*, 2012. URL `http://dacapobench.org/`.

[4] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[5] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the ACM workshop on Virtual Machines and Intermediate Languages*, 2013. .

[6] 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*, pages 241–252. IEEE Computer Society, 2003. ISBN 0-7695-1913-X.

[7] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. pages 144–153. ACM Press, 2006. ISBN 1-59593-332-8. .

[8] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992. ISBN 0-89791-475-9. .

[9] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1), May 2008. ISSN 1544-3566. .

[10] OpenJDK Community. *Graal Project*, 2012. URL `http://openjdk.java.net/projects/graal/`.

[11] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology*, pages 1–12. USENIX, 2001.

[12] M. Pall. LuaJIT 2.0 intellectual property disclosure and research opportunities, 2009. URL `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`.

[13] M. Pall. `src/lj_snap.c`, 2009. URL `http://luajit.org/git/luajit-2.0.git`.

[14] D. Schneider and C. F. Bolz. The efficient handling of guards in the design of RPython's tracing JIT. In *Proceedings of the ACM workshop on Virtual Machines and Intermediate Languages*, pages 3–12. ACM Press, 2012. ISBN 978-1-4503-1633-0. .

[15] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da Capo con Scala: design and analysis of a Scala benchmark suite for the Java virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 657–676. ACM Press, 2011.

[16] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for Java. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 165–174. ACM Press, 2014. ISBN 978-1-4503-2670-4. .

[17] Standard Performance Evaluation Corporation. SPECjbb2005, . URL `http://www.spec.org/jbb2005/`.

[18] Standard Performance Evaluation Corporation. SPECjvm2008, . URL `http://www.spec.org/jvm2008/`.

[19] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242. ACM Press, 1987. ISBN 0-89791-247-0. .

[20] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Transactions on Architecture and Code Optimization*, 9(4): 30:1–30:24, Jan. 2013. ISSN 1544-3566. .