An Empirical Study on Deoptimization in the Graal Compiler

Yudi Zheng[1], Lubomír Bulej*[2], and Walter Binder[1]

1 **Faculty of Informatics, Università della Svizzera italiana (USI), Switzerland**
`firstname.lastname@usi.ch`
2 **Faculty of Mathematics and Physics, Charles University, Czech Republic**
`lubomir.bulej@d3s.mff.cuni.cz`

Abstract

Managed language platforms such as the Java Virtual Machine or the Common Language Runtime rely on a dynamic compiler to achieve high performance. Besides making optimization decisions based on the actual program execution and the underlying hardware platform, a dynamic compiler is also in an ideal position to perform speculative optimizations. However, these tend to increase the compilation costs, because unsuccessful speculations trigger deoptimization and recompilation of the affected parts of the program, wasting previous work. Even though speculative optimizations are widely used, the costs of these optimizations in terms of extra compilation work has not been previously studied. In this paper, we analyze the behavior of the Graal dynamic compiler integrated in Oracle's HotSpot Virtual Machine. We focus on situations which cause program execution to switch from machine code to the interpreter, and compare application performance using three different deoptimization strategies which influence the amount of extra compilation work done by Graal. Using an adaptive deoptimization strategy, we managed to improve the average start-up performance of benchmarks from the DaCapo, ScalaBench, and Octane benchmark suites, mostly by avoiding wasted compilation work. On a single-core system, we observed an average speed-up of 6.4% for the DaCapo and ScalaBench workloads, and a speed-up of 5.1% for the Octane workloads; the improvement decreases with an increasing number of available CPU cores. We also find that the choice of a deoptimization strategy has negligible impact on steady-state performance. This indicates that the cost of speculation matters mainly during start-up, where it can disturb the delicate balance between executing the program and the compiler, but is quickly amortized in steady state.

1998 ACM Subject Classification D.3.4 Programming Languages, Processors — Compilers, Optimization

Keywords and phrases Dynamic compiler; profile-guided optimization; deoptimization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.92

1 Introduction

Managed language platforms such as the Java Virtual Machine (JVM) or the Common Language Runtime provide memory-safe and portable execution environments targeted by many object-oriented programming languages. On these platforms, programs are initially executed by an interpreter which collects and uses profiling information to schedule frequently executed methods (or code paths) for compilation into machine code to speed up the execution

* Major part of the work was conducted while Lubomír Bulej was with Faculty of Informatics, Università della Svizzera italiana (USI), Switzerland.


© Yudi Zheng, Lubomír Bulej and Walter Binder;
licensed under Creative Commons License CC-BY

of the program. The compilation is handled by a dynamic optimizing compiler (or a hierarchy of compilers if compilation is *tiered*). By making a program run faster, the dynamic compiler frees up computational resources that can be used to perform more optimizations. However, to actually benefit from faster program execution, the compiler should only consume a fraction of the computational resources that it has freed up. Because the effects of dynamic compilation accumulate over time, the goal is to speed up the program as soon as possible, but without slowing it down by the compilation work. Deciding *what* to compile, *when*, and *how* then becomes an optimization problem of its own [17].

Besides producing machine-code for the underlying hardware platform, the dynamic compiler is also in an ideal position to perform speculative optimizations based on the collected profiling information. While *profile-driven* and *feedback-driven* optimizations are not exclusive to managed platforms with dynamic compilers, a dynamic compiler works with profiles that reflect the actual behavior of the currently executing program. This provides the compiler with a more accurate view of the common-case behavior which the compiler should optimize. If a certain assumption about program behavior turns out to be wrong, the affected code can be recompiled to reflect the new behavior. This allows the dynamic compiler to pay less attention to uncommon execution paths, replacing them with traps that switch from program's machine code back to the virtual machine's (VM) runtime which then decides how to handle the situation. As a result, the compiler needs to do less work and produces higher-density code for the common code paths. Combined with aggressive inlining and code specialization based on receiver type feedback, a dynamic compiler can optimize away a significant portion of the abstraction overhead commonly found in object-oriented programs that make heavy use of small methods and dynamic binding.

The pioneering work by the authors of the SmallTalk-80 [6] and the Self-93 [11] systems has laid down the foundations of modern dynamic compilers, and sparked an enormous body of research [1] on techniques that make managed language platforms fast, such as selective compilation [11, 15, 4, 16, 17], profiling for feedback-directed optimization and code generation [22, 2, 25, 21], or dynamic deoptimization and on-stack replacement [10, 19, 7, 14, 12]. As a result, adaptive compilation and speculative optimization techniques are now widely used. Ideally, speculative optimizations will always turn out to be right and provide performance gains that outweigh the one-time cost in terms of compilation time before the program terminates. In reality, some speculations in the machine code will be wrong, and trigger *deoptimization*. Besides switching to interpreted (or otherwise less optimized) execution mode, deoptimization may also trigger recompilation of the affected code, thus wasting previous compilation work and adding to the overall cost of compilation.

How often does this happen and for what reason? How much compilation effort is wasted and what is the cost of speculative optimizations? What happens when the compiled code triggers deoptimization? In fact, these aspects of speculative optimizations have not been previously studied in the literature—unlike, e.g., the trade-offs involved in selective compilation. We therefore analyze the deoptimization behavior of code compiled by the Graal [18] dynamic compiler and the behavior of the VM runtime in response to the deoptimizations. Even though Graal has not (yet) replaced the classic C2 server compiler, it is integrated in Oracle's HotSpot Virtual Machine and serves as the basis for the Truffle framework for self-optimizing interpreters [24]. Truffle allows executing programs written in modern dynamic languages on the JVM and generally outperforms the original interpreters. Similarly to the classic C2 compiler, Graal performs feedback-directed optimizations and generates code that speculates on receiver types and uncommon paths, but is more aggressive about it. Unlike the C2 compiler, when Graal reaches a deoptimization site in the compiled

code, it switches back to interpreted mode and discards the machine code with the aim to generate it again using better profiling information. The C2 compiler is more conservative and in many cases discards the compiled code only after it triggers multiple deoptimizations. The obvious question is then: which of the two approaches is better, and how often programs actually violate the assumptions put in the code by the dynamic compiler?

In this paper, we make the following contributions:

1. We characterize the deoptimization causes in the code produced by Graal for the DaCapo [3], ScalaBench [20]), and Octane [8] benchmark suites (Section 3). We show that only a small fraction (∼2%) of deoptimization sites is triggered, most of which (∼98%) cause reprofiling. We investigate the causes of two types of repeatedly triggered deoptimizations that appear in the profile.
2. We provide two alternative deoptimization strategies for the Graal compiler. A *conservative* strategy, which defers invalidation of compiled code until enough deoptimizations are observed (default HotSpot behavior not used by Graal), and an *adaptive* strategy which switches among various deoptimization actions based on a precise deoptimization profile (Section 4).
3. We evaluate the performance of both deoptimization strategies and compare them to the default strategy used by Graal (Section 5). We show that the *conservative* strategy may cause extra compilation work, while the *adaptive* strategy reduces compilation work and provides statistically significant benefits to startup performance on a single-core system with both static and dynamic languages.

Before presenting our main contributions, we provide a summary of related work and the necessary background on deoptimization (both general and Graal-specific).

2 Related Work and Background

Dynamic deoptimization as a way to transfer execution from compiled code to interpreted code was introduced in the Self system to facilitate full source-level debugging of optimized code [10]. It also introduced techniques such as on-stack-replacement, which were since adopted and improved by others [19, 7, 14, 12].

Being more interested in the use of deoptimization in the implementation of speculative optimizations, we trace their origins to partial and deferred compilation in Self [5]. To reduce compilation time, program code that was predicted to execute infrequently was compiled only as a stub which invoked the compiler when a particular code path was first executed, thus deferring the compilation of uncommon code paths until they were actually needed. Many of the techniques found in Self, such as adaptive compilation, dynamic deoptimization, and speculative optimizations using deoptimization, were later adopted by Java [19, 14]. Further improvements to the HotSpot VM target selective compilation [15, 4, 16, 17], phase-based recompilation [9], and feedback-directed optimization [22, 2, 25, 21].

In general, deoptimization switches to a less optimized execution mode, e.g., interpreted execution, or execution of machine code generated by a baseline compiler. In Self, deoptimization was primarily used to defer compilation and to execute uncommon code in the interpreter. In a modern HotSpot JVM, especially with Graal enabled, deoptimization represents a key recovery mechanism for speculative optimizations. However, despite the role of deoptimization in the implementation of speculative optimizations, we are not aware of a study that characterizes the actual deoptimization behavior of programs compiled by a speculating dynamic compiler, and the impact of the deoptimizations on the compiled code.

That does not mean that deoptimization does not receive any attention. In recent work [23], the authors present a VM implementation technique that allows a deoptimization triggered in aggressively optimized code to resume execution in (deoptimized) machine code generated by the same compiler at a different optimization level. In contrast to an interpreter or baseline compiler, both of which rely on a fixed stack-frame layout, using a single compiler allows using an optimized stack layout for both the optimized and deoptimized code. This approach helps reduce the complexity of a VM implementation, because neither an interpreter nor a baseline compiler are needed.

In the remainder of this section we first provide more background on the use of deoptimization in speculative optimizations, and then complement it with details specific to the Graal compiler.

2.1 Speculation and Deoptimization

Speculative optimizations are aimed at optimizing for the common case, which is approximated using profiling data collected during program execution. Common speculative optimizations include implicit null checks, uncommon conditional branch elision, and type specialization. If a speculation turns out to be wrong, deoptimization allows the VM to ensure that the program always executes correctly, albeit more slowly.

Deoptimizations are usually triggered synchronously with program execution, either explicitly by invoking a deoptimization routine of the VM runtime, or implicitly, by performing an operation which causes a signal (e.g., segmentation fault in the case of a null pointer) to be sent to the VM, which handles the signal and switches execution to the interpreter. Deoptimizations can be also triggered asynchronously at the VM level, when the program invalidates assumptions under which it was compiled, e.g., when the second class implementing an interface is loaded.

The ability to trigger deoptimization from compiled code allows the compiler to avoid generating code that will be rarely used, e.g., code that constructs and throws exceptions, because exceptions should be rare in well-written programs. This applies both to explicitly thrown exceptions as well as exceptions that can be thrown implicitly by operations such as array access or division by zero. Based on the profiling feedback, the dynamic compiler can apply a similar approach to conditional jumps, replacing low-probability branches with a deoptimization trigger. Hence, the compiler saves computing resources by avoiding code generation for the uncommon paths. Moreover, this approach helps speed up global optimizations thanks to the reduced program state, and makes the generated machine code more compact, resulting in better instruction cache performance.

Another common kind of speculative optimization relies on type feedback, which allows the compiler to specialize code to most commonly used types. For instance, the targets of a virtual method invocation may be inlined (or the invocation can be devirtualized) if only a limited number of receiver types has been observed at a particular callsite. The type-specific code will be guarded by type-checking conditions, while a generic code path representing an uncommon branch may trigger deoptimization to handle the invocation in the interpreter.

While deoptimization is handled by the VM runtime, the compiler needs to provide the VM with details on how to handle it. This information is typically provided in form of parameters passed to the invocation of the deoptimization trigger routine in the generated code. For example, if recompilation of the code that triggers a deoptimization is unlikely to make it any better, the VM is instructed to just switch to the interpreter and leave the compiled code as-is. If a deoptimization does not depend on profiling data and could be eliminated by recompiling the code, the code is invalidated and the corresponding compilation

Table 1 Deoptimization actions in the Graal compiler.

Graal Deopt Action	*Description*	*HotSpot Deopt Action*
None	Do not invalidate the compiled code.	Action_none
RecompileIfTooManyDeopts	Do not invalidate the compiled code and schedule a recompilation if enough deoptimizations are seen.	Action_maybe_recompile
InvalidateReprofile	Invalidate the compiled code and reset the invocation counter.	Action_reinterpret
InvalidateRecompile	Invalidate the compiled code and schedule a recompilation immediately.	Action_make_not_entrant
InvalidateStopCompiling	Invalidate the compiled code and stop compiling the outermost method of this compilation.	Action_make_not_compilable

unit is immediately scheduled for recompilation. If a deoptimization was caused by insufficient profiling information, besides invalidating the machine code, the VM also attempts to reprofile the method thoroughly and recompile it later based on the updated profile. To avoid an endless cycle of recompilation and deoptimization for pathological cases, per-method counters are used to stop recompilation of a method if it has been recompiled too many times (yet did not eliminate the deoptimization).

In state-of-the-art dynamic compilers the mapping between a deoptimization reason and the corresponding deoptimization action is hard-coded. This makes perfect sense for certain cases, when there is only a single suitable deoptimization action. However, determining the most suitable action for situations in which the deoptimization is caused by an incomplete profile is difficult. For instance, when the compiler inlines potential callee methods based on the receiver type profile, it inserts a reprofiling deoptimization trigger in the uncommon (generic) path to cope with previously unseen receiver types. When encountering a very rare receiver type, deoptimization (including reprofiling) is triggered. However, due to the (usually) limited receiver type profile space[1], the newly collected profiling information might not include the rare case at the time of recompilation. The dynamic compiler will then either use the original invocation as the uncommon path (if megamorphic inlining is supported), or not inline the callsite at all. In both cases, the reprofiling and recompilation effort is wasted, and the recompiled code may become even worse.

2.2 Deoptimization in the Graal Compiler

The Graal compiler is integrated into the HotSpot runtime via the JVM Compiler Interface (JVMCI)[2] and replaces the HotSpot VM's C2 server compiler when enabled. It makes heavy use of profile-directed speculative optimizations and is thus more likely to exhibit deoptimizations. Because Graal only provides the last-level compiler, it can only instruct the HotSpot runtime what action to perform during deoptimization. The deoptimization actions used internally by Graal can be therefore directly mapped to the deoptimization actions defined in the HotSpot runtime.

[1] -XX:TypeProfileWidth in the Oracle JVM, defaults to 2 in standard HotSpot runtime or 8 in the Graal compiler.
[2] http://openjdk.java.net/jeps/243

Table 2 Deoptimization reasons in the Graal compiler.

Deoptimization Reason	*Description*	*Associated Action*
None	Absence of a relevant deoptimization.	-
NullCheckException	Unexpected null or zero divisor.	None InvalidateRecompile InvalidateReprofile
BoundsCheckException	Unexpected array index.	InvalidateReprofile
ClassCastException	Unexpected object class.	InvalidateReprofile
ArrayStoreException	Unexpected array class.	InvalidateReprofile
UnreachedCode	Unexpected reached code.	InvalidateRecompile InvalidateReprofile
TypeCheckedInliningViolated	Unexpected receiver type.	InvalidateReprofile
OptimizedTypeCheckViolated	Unexpected operand type.	InvalidateRecompile InvalidateReprofile
NotCompiledExceptionHandler	Exception handler is not compiled.	InvalidateRecompile
Unresolved	Encountered an unresolved class.	InvalidateRecompile
JavaSubroutineMismatch	Unexpected JSR return address.	InvalidateReprofile
ArithmeticException	A null_check due to division by zero.	None InvalidateReprofile
RuntimeConstraint	Arbitrary runtime constraint violated.	None InvalidateRecompile InvalidateReprofile
LoopLimitCheck	Compiler generated loop limits check failed.	InvalidateRecompile
TransferToInterpreter	Explicit transfer to interpreter.	-

The possible deoptimization actions are summarized in Table 1. Apart from the None action, which only switches execution to the interpreter, all other options influence the compilation unit which triggered the deoptimization in some way. Most of them invalidate the compilation unit's machine code immediately, with the exception of the RecompileIfTooManyDeopts action, which depends on a profile of preceding deoptimizations, and only invalidates the compiled code if too many deoptimizations are triggered at the same site or within the compilation unit.

Even though the deoptimization action is fixed in the compiled code, the HotSpot runtime rewrites the actual action either to force reprofiling or to avoid endless deoptimization and recompilation cycles. If a recompilation is scheduled for the second time for the same deoptimization site with the same reason, the HotSpot runtime rewrites the action to InvalidateReprofile, which resets method's hotness counters and causes it to be reprofiled. If the total number of recompilations of any method exceeds a threshold, the HotSpot runtime rewrites the action to InvalidateStopCompiling to prevent further recompilation of the method.

To illustrate how Graal uses these deoptimization actions, Table 2 shows the deoptimization reasons along with the associated actions as defined and used throughout the Graal code base. The table reveals that the actions RecompileIfTooManyDeopts and InvalidateStopCompiling are not used as of Graal v0.17[3]. This suggests that the compiler tries to keep full control over invalidation of compiled code, and that it tries not to give up any optimization opportunity until the HotSpot runtime enforces certain actions.

[3] https://github.com/graalvm/graal-core/tree/graal-vm-0.17

Some of the deoptimization reasons are used with multiple actions, depending on the situation in which the deoptimization is invoked. For instance, the OptimizedTypeCheckViolated reason is used when inlining the target of an interface with a single implementation, and when optimizing instanceof checks. In the former case, if a guard on the expected receiver type fails, the compiler invokes the InvalidateRecompile action with the reason OptimizedTypeCheckViolated, because it has produced the compiled code under the assumption that there is only a single implementation of a particular interface. In the latter case, the compiler checks against types derived from the given type that have been observed so far. Because the occurrence of a previously unseen type indicates an incomplete type profile, the compiler invokes the InvalidateReprofile action to get a more accurate type profile. If the compiler knew that the previously unseen type was a very rare case, it could invoke the None action. However, because encountering a new type may also signify a phase change in the application, Graal uses the InvalidateReprofile action.

Nevertheless, the mapping between deoptimization reasons and deoptimization actions in the Graal compiler is hard-coded and represents the trade-offs between startup and steady-state performance made by the compiler developers. In the following sections, we provide quantitative and qualitative analyses of how these decisions influence the actual deoptimization behavior of the Graal compiler.

3 Study of Deoptimization Behavior

In this section we analyze the deoptimization behavior of the HotSpot VM with the Graal compiler when executing benchmarks from the DaCapo [3], ScalaBench [20], and Octane [8] benchmark suites. The individual benchmarks are based on real-world programs written in Java and Python (DaCapo), Scala (ScalaBench), and JavaScript (Octane), slightly modified to run under a benchmarking harness suitable for experimental evaluation. The Python workloads are executed by Jython, a Python interpreter written in Java, the Scala workloads are compiled to Java bytecode, and the JavaScript workloads are executed by Graal.js, a JavaScript runtime written in Java on top of the Truffle framework [24].

We first analyze the kind of deoptimization sites emitted by Graal and the frequency with which they are triggered during execution, and then investigate two specific cases in which the same deoptimizations are triggered repeatedly.

3.1 Profiling Deoptimizations

To collect information about deoptimizations, we use a deoptimization profiler based on the accurate profiling framework integrated in Graal [26]; that framework ensures that profiling does not perturb the compiler optimizations. The profiler instruments each deoptimization site and reports the number of deoptimizations triggered at that site during execution.

The identity of each deoptimization site consists of the deoptimization reason, action, the originating method and bytecode index, and (optionally) a context identifying the compilation root if the method was inlined. The information encoded in the site identity along with the number of deoptimizations triggered at the site allow us to perform qualitative and quantitative analysis of the deoptimizations triggered in the compiled code produced by Graal. To this end, we profile selected benchmarks[4] from the DaCapo 9.12 suite, all benchmarks

[4] The eclipse and tomcat benchmarks were excluded due to their incompatibility with Java 8.

from the ScalaBench suite, and selected benchmarks[5] from the Octane suite on a multi-core platform[6].

We present the resulting profile from different perspectives. First we provide a static break-down of the deoptimization sites and deoptimization actions found in the code emitted by Graal (Section 3.1.1). This is complemented by a dynamic view of deoptimization sites that are actually triggered during execution (Section 3.1.2). Finally, we look at the most frequent repeatedly-triggered deoptimizations, because these are potential candidates for wasted compilation work (Section 3.1.3).

3.1.1 Deoptimizations Sites Emitted

The profiling results are summarized in Table 3. The top-level column groups represent the actions used at the deoptimization sites. We only track three of the five possible deoptimization actions, because Graal does not make use of the other two (c.f. Section 2.2). The bottom-level columns correspond to the number of deoptimization sites invoking a particular action, the fraction of the total number of sites, and the fraction of the total number of deoptimization sites emitted at which at least one deoptimization was triggered.

In general, the number of deoptimization sites emitted during a benchmark's lifetime varies significantly, ranging from 2000 to 23 000. For the DaCapo benchmarks, 94.17% of the total deoptimization sites invoke the InvalidateReprofile action, 3.23% just switch to the interpreter (action None), and 2.60% invoke the InvalidateRecompile action. For the ScalaBench benchmarks, the compiler emits a slightly higher proportion (95.44%) of the InvalidateReprofile deoptimization sites and a lower proportion (1.16%) of the InvalidateRecompile sites. We attribute this to the fact that the Scala language features are compiled into complex call chains in the Java bytecode. During dynamic compilation, these callsites are optimized with type guards that lead to InvalidateReprofile deoptimization sites. To summarize, in standard Java/Scala applications the Graal compiler favors speculative profile-directed optimizations, which invoke the InvalidateReprofile deoptimization action in their guard failure paths.

For the Octane benchmarks, the compiled code of the Graal.js self-optimizing interpreter contains a higher proportion (4.74%) of the InvalidateRecompile deoptimization sites. One of the reasons for this difference is that language runtimes implemented on top of the Truffle framework heavily utilize the Truffle API. Because this API consists of many interfaces with a single implementation, the compiled code for callsites invoking the Truffle API uses guarded devirtualized invocations. Consequently, the (many) corresponding guard failure paths invoke the InvalidateRecompile deoptimization action with OptimizedTypeCheckViolated as the reason (c.f. Section 2.2). The second reason for the higher proportion of InvalidateRecompile deoptimization sites is that the Truffle framework encourages aggressive type specialization in the interpretation of abstract syntax tree (AST) nodes of the hosted language. Internally, Truffle uses Java's exception mechanism to undo type specialization, and because at the time the type specialization occurs the exception handler has never been executed (otherwise the type specialization would not happen in the first place), the dynamic compiler considers the exception handler to be uncommon and replaces it with a deoptimization site which invokes the InvalidateRecompile action with NotCompiledExceptionHandler as the reason. This

[5] The pdf.js benchmark was excluded due to an internal exception.

[6] Intel Xeon E5-2680, 2.7 GHz, 8 cores, 64 GB RAM, CPU frequency scaling and Turbo mode disabled, hyper-threading enabled, Oracle JDK 1.8.0_101 b13 HotSpot Server VM (64-bit), Graal VM 0.17, running on Ubuntu Linux Server 64-bit version 14.04.1

Table 3 The number and percentage of deoptimization sites with a particular action emitted and triggered during the first benchmark iteration of the DaCapo and ScalaBench workloads, and during the warmup phase of the Octane workloads.

	Benchmark	None			Reprofile			Recompile		
		#	%	$\%_{Hit}$	#	%	$\%_{Hit}$	#	%	$\%_{Hit}$
DaCapo	avrora	94	3.2	.00	2813	94.2	1.04	79	2.7	.00
	batik	147	3.5	.00	3991	94.5	2.70	86	2.0	.02
	fop	186	3.8	.00	4639	95.6	1.52	30	0.6	.00
	h2	208	2.6	.00	7516	94.0	2.49	275	3.4	.05
	jython	337	2.9	.00	10 837	94.5	1.89	289	2.5	.03
	luindex	196	6.4	.00	2839	92.8	1.37	26	0.9	.00
	lusearch	204	6.7	.00	2785	91.2	0.75	64	2.1	.00
	pmd	163	2.6	.00	5942	93.2	1.11	270	4.2	.09
	sunflow	92	4.1	.00	2123	94.7	1.12	26	1.2	.00
	tradebeans	267	2.7	.00	9307	93.4	2.25	394	4.0	.02
	tradesoap	608	2.8	.00	20 866	94.6	1.71	593	2.7	.02
	xalan	225	3.7	.00	5880	95.3	0.34	63	1.0	.00
	Total	2727	3.2	.00	79 538	94.2	1.68	2195	2.6	.02
ScalaBench	actors	116	2.5	.00	4418	95.2	1.87	108	2.3	.09
	apparat	230	3.8	.00	5751	94.7	2.45	91	1.5	.12
	factorie	133	4.0	.00	3153	94.4	1.71	54	1.6	.00
	kiama	178	4.9	.00	3423	94.3	2.26	31	0.9	.00
	scalac	289	1.8	.00	15 525	97.6	3.15	90	0.6	.01
	scaladoc	288	2.5	.00	10 909	96.1	2.93	155	1.4	.00
	scalap	133	5.2	.00	2428	94.2	1.59	18	0.7	.00
	scalariform	189	4.3	.00	4198	94.7	1.11	44	1.0	.00
	scalatest	215	5.0	.00	4083	94.2	1.25	37	0.9	.05
	scalaxb	166	4.4	.00	3547	94.7	1.68	31	0.8	.03
	specs	212	5.4	.00	3672	93.6	1.27	39	1.0	.05
	tmt	191	4.0	.00	4535	94.0	1.97	99	2.0	.00
	Total	2340	3.4	.00	65 642	95.4	2.27	797	1.2	.03
Octane	box2d	195	2.2	.00	8162	91.8	1.43	538	6.1	.46
	code-load	699	2.9	.00	23 041	93.8	1.43	827	3.4	.37
	crypto	142	2.1	.00	6325	92.6	1.46	364	5.3	.16
	deltablue	136	2.3	.00	5395	91.8	1.19	347	5.9	.15
	earley-boyer	172	2.4	.00	6740	92.5	1.65	376	5.2	.36
	gbemu	200	1.9	.00	9743	92.9	2.74	546	5.2	.28
	mandreel	367	3.3	.00	10 197	92.4	1.72	470	4.3	.37
	navier-stokes	129	2.4	.00	4930	92.8	1.88	256	4.8	.06
	raytrace	133	2.2	.00	5696	92.4	1.28	334	5.4	.32
	regexp	221	2.3	.00	9050	93.1	2.27	449	4.6	.11
	richards	113	2.2	.00	4834	91.9	1.48	316	6.0	.19
	splay	123	2.0	.00	5664	93.1	1.87	296	4.9	.12
	typescript	294	2.0	.00	13 403	93.1	1.58	694	4.8	.38
	zlib	204	2.9	.00	6458	92.8	1.98	298	4.3	.17
	Total	3128	2.4	.00	119 638	92.8	1.71	6111	4.7	.28

mechanism allows Truffle to attempt aggressive type specialization and recompile with generic types if a type-related exception occurs.

3.1.2 Deoptimization Sites Triggered

Of all the sites emitted for the DaCapo benchmarks, only 1.68% were actually triggered and invoked the InvalidateReprofile deoptimization action during execution. The proportion increases to 2.27% in the ScalaBench benchmarks for the same reason that affects the total number of emitted sites. Similarly, only 0.02% of the sites in the DaCapo benchmarks and 0.03% of the sites in the ScalaBench benchmarks were triggered and invoked the InvalidateRecompile action. This indicates that in ordinary Java/Scala applications, deoptimization sites that do not rely on profiling feedback represent only a small fraction of the total number of deoptimization sites and are rarely triggered. In addition, these

sites tend to be eliminated by the recompilation they force, therefore they rarely cause repeated deoptimizations. In total, over 98% of all triggered deoptimizations invoke the InvalidateReprofile action, while only less than 2% invoke the InvalidateRecompile action. This suggests that in the code produced by the Graal compiler, deoptimizations are dominated by those that force reprofiling of the affected code.

Compared to DaCapo and ScalaBench, the number and the proportion of the InvalidateRecompile deoptimizations triggered during execution of the Octane benchmarks on top of Graal.js is significantly higher. As discussed earlier, this is because the Truffle code that undoes type specialization in the hosted language is implicitly replaced by deoptimization. Nevertheless, similarly to DaCapo and ScalaBench, the most frequently triggered deoptimization action in the Octane benchmarks is InvalidateReprofile (88.83%).

3.1.3 Deoptimizations Triggered Repeatedly

In Table 4 we show the number of sites which trigger a particular deoptimization more than once during benchmark execution. In the DaCapo benchmarks, these sites account for 11.67% of deoptimization sites triggered at least once, and for 26.64% of all triggered deoptimizations; the results for ScalaBench are similar. For the Octane benchmarks on Graal.js, the proportion of repeated deoptimization sites drops to 5.96%, which is caused by Truffle invalidating the type specialization code that triggered a deoptimization.

While it is possible for multiple threads to trigger the same deoptimization site in the same version of the compiled code, the majority of the repeated deoptimizations originate from recompiled code. This means that if recompilation does not eliminate these deoptimization sites, reprofiling either does not produce a profile that would change the optimization decisions, or that the profile is not provided in time for the recompilation.

To aid in investigating the reasons behind the worst-case repeated deoptimizations, Table 4 also lists the deoptimization sites that repeatedly trigger the most deoptimizations during the execution of a particular benchmark. All of the worst-case deoptimization sites invoke the InvalidateReprofile action, which is consistent with our findings so far.

We observe that the most frequently triggered deoptimization sites cause reprofiling for three main reasons: TypeCheckedInliningViolated, OptimizedTypeCheckViolated, and UnreachedCode. Deoptimizations specifying UnreachedCode as the reason result from conditional branches that were eliminated based on (assumed) zero execution probability according to the branch profile for the corresponding bytecode. The actors benchmark contains the most frequent deoptimization site of this type in method java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(), which contains a blocking thread synchronization operation. Deoptimizations that specify type-checking violations as the reason result from optimizations that rely on a type profile. Here, the compiler typically uses deoptimization in the failure path of a guard that ensures that type-specific code is only reached with proper types. Among the benchmarks that suffer from deoptimizations for these reasons, the scalac and scaladoc benchmarks share the same worst-case deoptimization site which triggers deoptimization 23 times.

In the case of the Octane benchmarks on Graal.js, a high number of repeated deoptimizations are triggered in the earley-boyer and typescript benchmarks. The underlying reason for repeated deoptimizations is the same as in the case of the DaCapo and ScalaBench suites—inaccurate profiling information caused by associating a profiling record with a deoptimization target (instead of origin), and subsequent sharing of this record by multiple deoptimization sites. Unfortunately, code obfuscation in the Graal.js binary release prevents us from presenting the situation in more detail at source code level.

Table 4 Number of deoptimization sites that were triggered repeatedly and the deoptimization reason used at the most frequently triggered InvalidateReprofile deoptimization site during the first benchmark iteration of the DaCapo and ScalaBench workloads, and during the warmup phase of the Octane workloads. Due to the obfuscated code in the binary release of Graal.js, some of the reported sites (marked with *) are identified by their deoptimization target instead of their precise location in the bytecode.

	Benchmark	*Repeated Deoptimizations*			*Most Frequent Site*	
		$\#_{Sites}$	$\%_{Sites}$	$\%_{Deopts}$	$\#_{Hit}$	Reason
DaCapo	avrora	6	19.4	41.9	4	UnreachedCode
	batik	4	3.5	7.5	3	TypeCheckedInliningViolated
	fop	2	2.7	10.0	6	UnreachedCode
	h2	27	13.3	28.5	6	OptimizedTypeCheckViolated
	jython	22	10.0	23.6	9	UnreachedCode
	luindex	0	0.0	0.0	-	-
	lusearch	2	8.7	36.4	10	TypeCheckedInliningViolated
	pmd	6	7.8	17.4	5	UnreachedCode
	sunflow	1	4.0	7.7	2	TypeCheckedInliningViolated
	tradebeans	34	15.0	36.8	10	TypeCheckedInliningViolated
	tradesoap	58	15.2	32.0	5	TypeCheckedInliningViolated
	xalan	1	4.8	16.7	4	TypeCheckedInliningViolated
	Total	163	11.7	26.6		
ScalaBench	actors	14	15.4	67.7	64	UnreachedCode
	apparat	15	9.6	24.6	3	TypeCheckedInliningViolated
	factorie	8	14.0	40.2	9	OptimizedTypeCheckViolated
	kiama	11	13.4	39.3	10	OptimizedTypeCheckViolated
	scalac	70	13.9	34.2	23	TypeCheckedInliningViolated
	scaladoc	41	12.3	35.1	23	TypeCheckedInliningViolated
	scalap	2	4.9	11.4	3	TypeCheckedInliningViolated
	scalariform	7	14.3	34.4	7	OptimizedTypeCheckViolated
	scalatest	3	5.4	10.2	2	TypeCheckedInliningViolated
	scalaxb	1	1.6	4.6	3	TypeCheckedInliningViolated
	specs	1	1.9	3.8	2	TypeCheckedInliningViolated
	tmt	7	7.4	21.4	9	OptimizedTypeCheckViolated
	Total	180	11.4	34.3		
Octane	box2d	16	9.5	20.4	* 6	TypeCheckedInliningViolated
	code-load	35	7.9	18.6	6	UnreachedCode
	crypto	2	1.8	6.0	5	UnreachedCode
	deltablue	5	6.3	12.9	3	TypeCheckedInliningViolated
	earley-boyer	5	3.4	74.3	* 398	TypeCheckedInliningViolated
	gbemu	17	5.4	11.5	4	UnreachedCode
	mandreel	12	5.2	13.4	5	UnreachedCode
	navier-stokes	4	3.9	10.0	* 5	TypeCheckedInliningViolated
	raytrace	2	2.0	4.0	2	TypeCheckedInliningViolated
	regexp	16	6.9	16.6	9	UnreachedCode
	richards	1	1.1	2.3	2	TypeCheckedInliningViolated
	splay	3	2.5	4.8	2	UnreachedCode
	typescript	24	8.5	66.3	* 237	TypeCheckedInliningViolated
	zlib	11	7.3	13.7	2	OptimizedTypeCheckViolated
	Total	153	6.0	33.7		

3.1.4 Deoptimizations per Iteration

Finally, Table 5 shows the number of deoptimizations triggered in subsequent benchmark iterations for the DaCapo and ScalaBench benchmarks. Most benchmarks encounter no more than 3 deoptimizations per iteration after the 4th iteration, because the compiled code for most of the hot methods stabilizes. However, there are a few cases of repeated deoptimizations, especially in the scalac benchmark, where on average 10 deoptimizations per iteration are triggered even past the 15th iteration. In most cases, TypeCheckedInlining-Violated is given as the reason, and half of the deoptimizations originate at the same bytecode (scala.collection.immutable.HashSet.elemHashCode(Object)#9) inlined in different methods.

Table 5 Number of deoptimizations per iteration when executing the DaCapo and ScalaBench benchmarks.

Iteration	1	2	3	4	5	6	...	15	16	17	18	19	20
avrora	43	17	4	1	2	0		0	0	0	0	0	0
batik	120	20	18	14	6	2		1	1	0	0	1	0
fop	80	23	5	4	2	0		0	1	1	0	0	0
h2	246	17	4	1	2	0		0	1	1	0	0	0
jython	259	27	2	1	0	0		1	1	2	1	0	1
luindex	42	14	1	0	0	0		0	0	0	0	0	0
lusearch	33	3	0	0	0	0	...	0	0	0	0	0	0
pmd	86	25	6	11	5	4		1	1	2	3	0	0
sunflow	26	3	1	0	0	0		0	0	0	0	0	0
tradebeans	304	7	4	0	0	0		0	0	0	0	0	0
tradesoap	475	34	3	0	2	2		0	1	0	0	0	1
xalan	24	1	1	0	0	0		0	0	0	0	0	0
actors	238	28	5	6	4	5		0	1	2	1	1	2
apparat	187	22	9	3	5	2		3	2	2	2	2	3
factorie	82	10	0	0	0	0		0	1	2	0	0	0
kiama	117	5	2	3	3	3		0	0	1	0	0	2
scalac	656	123	36	25	22	21		8	14	5	13	8	12
scaladoc	450	106	18	13	1	6		1	0	0	2	2	8
scalap	44	10	0	0	0	0	...	0	0	0	0	0	0
scalariform	64	23	9	5	4	3		1	0	0	0	0	0
scalatest	59	29	8	9	3	1		1	0	0	0	0	0
scalaxb	66	26	3	0	0	1		0	0	0	0	1	0
specs	53	10	9	5	5	7		6	6	4	2	3	4
tmt	112	6	4	3	2	1		2	1	2	2	2	1

This suggests that the receiver type profile may not be updated properly (or soon enough) after deoptimization and reprofiling.

3.2 Investigating Repeated Deoptimizations

Our findings in Section 3.1.3 indicate that certain deoptimizations are triggered repeatedly at the same site. If a particular deoptimization is triggered by multiple threads in one version of the compiled code, the subsequent recompilation should eliminate the deoptimization site. However, repeated deoptimizations triggered at the same site in multiple subsequent versions of the compiled code indicate a problem, because that site should have been eliminated by recompilations.

By analyzing the cases of repeatedly triggered deoptimization, we have discovered that this situation occurs because an outdated method profile is used during the recompilation. In the Graal compiler, this can happen because Graal inlines methods aggressively, but at the same time, deoptimization site in the inlined code can deoptimize to the caller containing the callsite of the inlined method (if no program state modification precedes the deoptimization site in the inlined code). After deoptimization, when the interpreter wants to invoke the (previously inlined) method at the callsite, the callee can be compiled either at a different level (without speculation and thus deoptimization), or with a different optimization outcome that did not emit a deoptimization site. In both cases, the profile for the callee is not updated, and subsequent recompilations of its inlined code will use an inaccurate profile, resulting in repeated deoptimizations.

We now illustrate the situations leading to repeated deoptimization for two specific cases: the UnreachedCode deoptimization in the actors benchmark, and the type-check related deoptimizations in the scalac benchmark.

3.2.1 Repeated Deoptimizations in the actors Benchmark

The results in Table 4 show that the actors benchmark contains a site which triggers the UnreachedCode deoptimization 64 times during the first iteration of the benchmark execution. Figure 1 shows a snippet of code containing this deoptimization site. The await() method invokes the checkInterruptWhileWaiting(Node) method (line 21), which returns a value depending on the result of the Thread.interrupted() method.

When compiling the await() method, Graal inlines the invocation of the (small and private) checkInterruptWhileWaiting(Node) method at the callsite (line 21). The ternary operator used in the return statement of that method is essentially a conditional branch compiled using the ifeq[7] bytecode, for which the VM collects a branch profile. Because thread interruption happens rarely, it is very likely that all invocations of Thread.interrupted() will return false, and the branch profile for the ifeq bytecode will tell the compiler that the branch was taken in 100% of the cases. By default[8], Graal removes the code in the (apparently) unreachable branch, and inserts a guard for the expected result of the Thread.interrupted() method with a failure path which invokes the InvalidateReprofile deoptimization with UnreachedCode as the reason.

In the await() method, threads may block in the park() method at line 19, which returns when a thread is unparked, or when a thread is interrupted. Any thread returning from the park() method will execute the condition at line 20, including the inlined optimized version of checkInterruptWhileWaiting(Node). If a thread was interrupted, the Thread.interrupted() method returns true contrary to the expectation, and causes the thread to trigger a deoptimization. The first thread to trigger the deoptimization will invalidate the compiled code of the await() method by making it *not entrant* (execution entering the compiled code will immediately switch to interpreter), and resume execution in the interpreter.

However, there may be more threads in the same situation, executing the (now invalidated) compiled code—the 64 repeated deoptimizations in the actors benchmark were caused by 64 different threads triggering the same deoptimization in the same version of the compiled code. While this kind of repeated deoptimization causes threads to execute in the interpreter, it only leads to a single recompilation and is relatively harmless. The branch profile for the ifeq bytecode will be updated during interpreted execution, and taken into account during recompilation of the await() method.

But the await() method contains another UnreachedCode deoptimization site that is problematic. In this case, Graal inlines the invocation of the (final) isOnSyncQueue(Node) method at the callsite (line 18). The null-check in the inlined code uses the ifnonnull bytecode (line 4), which is a conditional branch. Based on the associated branch profile indicating 100% *branch-taken* probability, Graal replaces the unreachable branch with a deoptimization which is triggered if node.prev is null.

If the deoptimization in the loop header is triggered, the code of the await() method will be invalidated and the interpreter will resume execution at beginning of the loop (line 18). The interpreter will then likely invoke the compiled version of the isOnSyncQueue(Node) method, which contains the same guard and deoptimization derived from the same ifnonnull branch profile. In the meantime, because the actors benchmark is highly multi-threaded, another thread may set node.prev to a non-null value. The compiled version of the isOnSyncQueue(Node) method will then execute normally, without retriggering the deoptimization.

[7] Branch if the value on top of the operand stack is zero, i.e., false.
[8] This can be disabled via -Dgraal.RemoveNeverExecutedCode=false.

```
1  public abstract class AbstractQueuedSynchronizer
2      extends AbstractOwnableSynchronizer implements java.io.Serializable {
3    final boolean isOnSyncQueue(Node node) {
4      if (node.waitStatus == Node.CONDITION || node.prev == null)
5        return false;
6      ...
7    }
8    public class ConditionObject implements Condition, java.io.Serializable {
9      private int checkInterruptWhileWaiting(Node node) {
10       return Thread.interrupted() ?
11         (transferAfterCancelledWait(node) ? THROW_IE : REINTERRUPT) : 0;
12     }
13
14     public final void await() throws InterruptedException {
15       ...
16       int savedState = fullyRelease(node);
17       int interruptMode = 0;
18       while (!isOnSyncQueue(node)) {
19         LockSupport.park(this);
20         if ((interruptMode = checkInterruptWhileWaiting(node))
21             != 0)
22           break;
23       }
24       ...
25     }
26   }
27 }
```

Figure 1 Excerpt from the source code of java.util.concurrent.locks.AbstractQueuedSynchronizer.

```
1  class HashSet[A] extends Set[A]
2          with GenericSetTemplate[A, HashSet] with SetLike[A, HashSet[A]] {
3    protected def elemHashCode(key: A) = if (key == null) 0 else key.##
4    protected def computeHash(key: A) = improve(elemHashCode(key))
5  }
```

```
9  // Java pseudo-code for the ## operation
10 int ##() {
11   if (this instanceof Number) {
12     return BoxesRunTime.hashFromNumber(this);
13   } else {
14     return hashCode();
15   }
16 }
```

Figure 2 Excerpt from scala.collection.immutable.HashSet.

Without that the isOnSyncQueue(Node) method will not be reinterpreted, and the branch profile for the ifnonnull bytecode will not be updated. When recompiling the await() method, the compiler will use an inaccurate branch profile and produce the same code that was previously invalidated. In our experiment, we observed 9 deoptimizations originating at the same site, but triggered in different versions of the compiled code. This kind of repeated deoptimizations is more serious, because it causes reprofiling of the await() method (requiring it to be executed in the interpreter more times) and subsequent recompilation, but does not improve the situation.

3.2.2 Repeated Deoptimizations in the scalac Benchmark

Another deoptimization anomaly that can be observed in the profiling results concerns several benchmarks that exhibit the same pattern of repeated deoptimizations, with either TypeCheckedInliningViolated or OptimizedTypeCheckViolated specified as the reason. This is also true for the steady-state execution of the scalac benchmark shown in Table 5, which we now investigate in more detail.

The code containing the deoptimization site is shown in Figure 2. At line 4 the computeHash(Object) method invokes the elemHashCode(Object) method, which in turn invokes the

```
1 if (key.type == String) {
2   // inlined code of String.hashCode
3 } else {
4   deoptimize(InvalidateReprofile, TypeCheckedInliningViolated,
5     HashSet.computeHash /* target method */,  0 /* target bytecode index */
6   ); // never returns
7 }
```

Figure 3 Pseudo-code of the Graal-compiled code for the ## operation.

operation on key. The ## operation is a Scala intrinsic which can be expressed as Java pseudo-code shown in lines 10–16. For every use of the ## operation, the Scala compiler directly inlines the corresponding bytecode sequence into the bytecode it produces.

Line 11 produces an instanceof bytecode which checks for the Number class, and is subject to type-profile-based optimizations in Graal. When compiling the instanceof bytecode into machine code, the compiler queries the recorded type profile associated with the particular bytecode, and generates tests against the profiled types instead of the operand type, and a failure path which will trigger deoptimization if all the type checks fail.

In our experiment, when compiling the computeHash(Object) method for the first time, the compiler receives a type profile containing only the String class, and generates machine code corresponding to the pseudo-code shown in Figure 3. The deoptimization in the else branch actually transfers execution to the beginning of the computeHash(Object) method, because the program state is not mutated between the invocation of the elemHashCode(Object) method and the deoptimization due to the inlined ## operation. When the interpreter reaches the invocation of the elemHashCode(Object) method again, it will likely find the method compiled, so the invocation will switch to machine code. However, with the default tiered compilation strategy, the elemHashCode(Object) method is very likely to be compiled by the level 1 compiler, which is intended for simple methods. As such, level 1 compilation does not use profile-directed optimizations for instanceof and the generated code does not update the profiling information. The compiled version of the elemHashCode(Object) method will therefore correctly handle the ## operation for all types, but the type profile for the inlined code of the ## operation will not be updated. When Graal compiles the computeHash(Object) method again, it will inline the elemHashCode(Object) method again, but the type profile for the instanceof bytecode will still contain only the String class. The recompiled elemHashCode(Object) method will therefore repeatedly trigger deoptimizations and recompilations.

Consequently, the anomaly occurs when a deoptimization due to an inlined method resumes in the caller and invokes a compiled version of the (previously inlined) callee. If the callee is compiled at level 1, it neither contains profile-directed optimizations nor updates profiling information. When the caller is recompiled (as it is a hot method) and the callee is inlined again, the compiler uses the inaccurate type profile for the code in the callee and generates code that triggers the same deoptimization.

We have also identified a similar problem when Graal devirtualizes method invocations. A devirtualized callsite uses a number of type checks against types from a callsite's receiver profile to invoke concrete methods on specific receiver types, and may trigger deoptimization if it encounters an unexpected receiver type (unless the callsite is megamorphic, which performs a virtual method dispatch). The problem occurs if a callsite is devirtualized in the ancestor of the direct caller of a method, which may happen when the direct caller is inlined. If such a devirtualized (non-megamorphic) callsite triggers a deoptimization and does not transfer execution to the direct caller, the receiver type profile used for devirtualization of the callsite may not be updated if the direct caller also has a standalone compiled version that neither

devirtualizes the callsite (and thus trigger the same deoptimization) nor collects profiling information. In general, this situation is caused by the weighted inlining mechanism in the Graal compiler, and the problem would be remedied by either disallowing deoptimization to cross the direct caller's method boundary, or by invalidating its compiled code.

4 Alternative Deoptimization Strategies

The deoptimization code produced by Graal mostly invokes the InvalidateReprofile action, hoping to trade extra work in the short term for a potentially better peak performance in the long term. Another reason for using this kind of deoptimization is to cope with application phase changes. These can manifest in the form of completely different execution and type profiles, rendering the compiled code based on profiles from the previous phase obsolete. Obviously, the compiler cannot tell ahead of time whether the actual benefits will outweigh the costs. However, as long as the costs are not excessive, they will be amortized in the long term even without huge performance gains.

With this strategy, the worst-case scenario for long-term performance is the occurrence of rare cases that trigger deoptimization. In this case, the ensuing reprofiling and recompilation will not provide a long-term benefit, but instead cause short-term performance degradation. Worse, during recompilation, the rare case may cause the compiler to abandon speculative optimizations that have worked well before the rare case occurred.

The solution is to introduce some tolerance for rare cases, delaying deoptimizations until the supposedly rare cases become more frequent. This notion is supported by the HotSpot runtime, as the presence of the action_maybe_recompile deoptimization action suggests. However, Graal does not use its own corresponding action (RecompileIfTooManyDeopts) in the deoptimization code it emits. Presumably, this is because Graal speculates aggressively and the Graal developers do not want to delay recompilation if the program violates optimization assumptions. In addition, because Graal focuses on achieving high peak performance, the cost associated with eager deoptimizations should be amortized in the longer run.

Because the effect of this approach on performance has not been previously studied, we modify Graal to support two additional strategies for handling deoptimizations and compare the performance achieved with the alternative strategies to the default strategy used by Graal. Unlike the default strategy, which always invokes the InvalidateReprofile action, the alternative strategies differ in the degree of tolerance for rare cases.

4.1 Conservative Deoptimization Strategy

The first strategy, referred to as *conservative*, replaces the use of the InvalidateReprofile action with the RecompileIfTooManyDeopts. This strategy relies on the existing mechanisms in the HotSpot runtime to determine when to invalidate the compiled code and when to reinterpret (and possibly reprofile) it. The runtime keeps an execution profile for each method, including information about deoptimizations. The deoptimization profile consists of a counter for each deoptimization reason as well as a recompilation counter. It also stores limited information associated with deoptimization targets (referred to as *traps*), i.e., the bytecode instructions at which the interpreter resumes execution after deoptimization. The per-trap information is keyed to the bytecode index of the target instruction in the target method, and stores

the reasons[9] for which the trap was targeted, and whether the method code was invalidated and recompiled due to this trap. The deoptimization reasons are split into two categories considered separately. The first category, referred to as *per-method*, represents reasons that are only considered at the method level, while the second category, referred to as *per-bytecode*, represents reasons that are only considered at the bytecode level.

When a deoptimization is triggered, the HotSpot runtime uses the method profile to make the following decisions: (1a) if the deoptimization reason belongs to the *per-bytecode* category, was previously observed at this trap, and the deoptimization count (taken from the method-level profile) for that reason exceeds a *per-bytecode* threshold[10], the compiled code is invalidated; (1b) if the deoptimization reason belongs to the *per-method* category and the deoptimization count for that reason exceeds a *per-method* threshold[11], the compiled code is invalidated; (2) for compiled code that is to be invalidated, if the per-trap information shows that the code has been previously recompiled for the same *per-bytecode* reason, or if the recompilation counter is greater than 0 for other reasons, the runtime resets the method execution and back-edge counters to facilitate reprofiling; (3) if the recompilation counter for a *per-bytecode* reason exceeds a *per-bytecode* threshold[12], or a *per-method* threshold[13] for *per-method* reasons, the deoptimizing method is made *not compilable*.

The per-trap information is inherently approximate. For example, it does not distinguish between two deoptimization sites sharing the same deoptimization target. But when Graal is enabled, it makes it even more approximate. While the trap bytecode index always refers to the instruction in the bytecode of the target method, updates to the per-trap information are stored in the profile of the method in which a deoptimization occurred (not the deoptimization target, as in the case of HotSpot without Graal). A deoptimization triggered by an inlined method will therefore update the per-trap information of the compilation root using an index associated with the bytecode in the target method. This is presumably to avoid spurious invalidation of the compiled code of methods that were inlined with speculative optimizations. However, if several methods inlined in the same compilation root contain a trap instruction, they may share the same slot in the per-trap profile of the compilation root. In addition, due to Graal's aggressive inlining, the deoptimization target may cross method boundaries—a deoptimization from an inlined method may target the returning bytecode of the previous callsite in the caller.

4.2 Adaptive Deoptimization Strategy

The second strategy, referred to as *adaptive*, uses a custom deoptimization profile to choose a deoptimization action both during dynamic compilation and during program execution. Unlike the HotSpot runtime or Graal (c.f. Section 4.1), we simply associate a deoptimization counter with each deoptimization site ID (c.f. Section 3), but disregard the stack trace for inlined methods. This means that methods inlined in different compilation roots will update the same deoptimization counters.

During compilation, whenever Graal intends to emit the InvalidateReprofile deoptimization at a particular site, we check the value of the counter corresponding to that site, and emit the

[9] To limit memory consumption, only one precise reason can be stored, otherwise the profile just indicates that there is more than one reason.
[10] -XX:PerBytecodeTrapLimit, defaults to 4.
[11] -XX:PerMethodTrapLimit, defaults to 100.
[12] -XX:PerBytecodeRecompilationCutoff, defaults to 200.
[13] -XX:PerMethodRecompilationCutoff, defaults to 400.

default deoptimization code (invoking InvalidateReprofile) if the value is between two thresholds, deoptsTolerated (exclusive, defaults to 1) and deoptsAllowed (inclusive, defaults to 100). If the counter exceeds the deoptsAllowed threshold, too many deoptimizations have been triggered at that particular site, and we instead emit code to invoke the InvalidateStopCompiling deoptimization. If the method containing the deoptimization site is being inlined, we mark the method as non-inlineable and emit the InvalidateRecompile deoptimization in the inlined code. Consequently, the method is inlined one last time in the compilation root being compiled, but will not be inlined in future recompilations of any method. If the counter does not exceed the deoptsTolerated threshold, the number of deoptimizations triggered at the site is considered tolerable, and we emit code that chooses between the None and InvalidateReprofile deoptimization actions at runtime. When such a deoptimization site is reached and the corresponding deoptimization counter still does not exceed the deoptsTolerated threshold, the deoptimization just switches to the interpreter and keeps the compiled code as-is (the None action). Otherwise the deoptimization invalidates the code and resets the hotness counters of the corresponding method to force reprofiling (the InvalidateReprofile action).

To avoid using a stale deoptimization profile during application phase changes, the counters for deoptimization sites involved in a particular compilation are aged in each compilation. Alternatively, we provide an option to age the deoptimization profile periodically, which allows tolerating deoptimizations based on rates, instead of absolute numbers.

5 Performance Evaluation

We now evaluate performance of the two additional strategies and compare them to the default strategy used by Graal. Using the same set of benchmarks and the same hardware platform as presented in Section 3, we evaluate the deoptimization strategies with a varying number of CPU cores available to the JVM. To minimize interference due to compilation of Graal classes, we enable bootstrapping of Graal[14] at JVM startup.

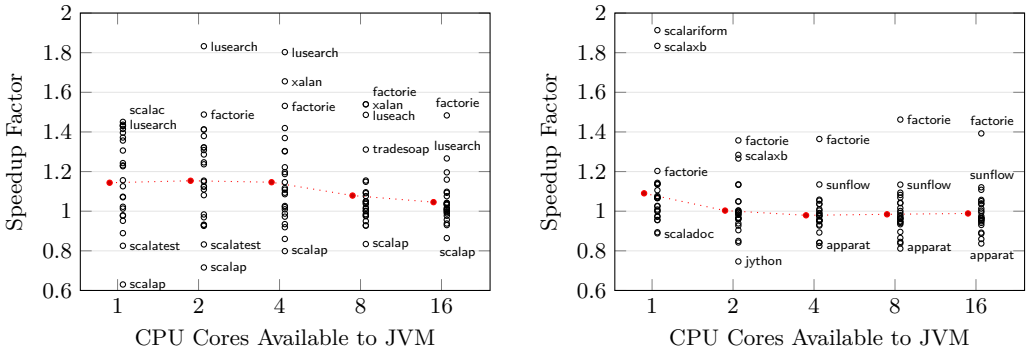
Because the DaCapo and ScalaBench benchmark suites are similar (ScalaBench uses the DaCapo benchmarking harness), we present the results for these two benchmark suites separately from the results for the Octane benchmarks on Graal.js, which are not directly comparable to the results from the other two suites. We also subject the results from the DaCapo and ScalaBench benchmark suites to more extensive evaluation, whereas the results for the Octane benchmarks are meant to illustrate the indirect impact of deoptimization strategies on the performance of the hosted language (JavaScript).

5.1 DaCapo and ScalaBench Evaluation

To evaluate the impact of the deoptimization strategies on the performance of the benchmarks from the DaCapo and ScalaBench benchmark suites, we collect[15] the following performance metrics: (1) startup time, i.e., the wall-clock time for the execution of the first benchmark iteration, (2) steady-state execution time, i.e., the wall-clock time for the execution of the last benchmark iteration, and (3) compilation time in each iteration, i.e., CPU time spent in compiler threads during benchmark iteration.

To present the results, we plot the speed-up factor of each benchmark against the baseline, as well as the geometric mean of speed-up factors for all benchmarks to illustrate the overall

[14] Enabled by the -XX:+BootstrapJVMCI option.
[15] Data from 10 benchmark runs, each benchmark executed for at least 10 iterations and 10 seconds.



(a) Startup **(b)** Steady-state

Figure 4 Startup and steady-state performance of the DaCapo and ScalaBench benchmarks on Graal, with C2 as the baseline. Black circles indicate the speed-up factor of individual benchmarks (ratio of mean execution time on Graal and C2). Value greater than 1 means that Graal outperforms C2. Red points represent the geometric mean of speed-up factors across all bechmarks. The line connecting the geometric means is intended only as a visual aid.

effect. When discussing average performance, we also report the range of speed-up factors for individual benchmarks contributing to the particular geometric mean.

5.1.1 Choosing the Baseline

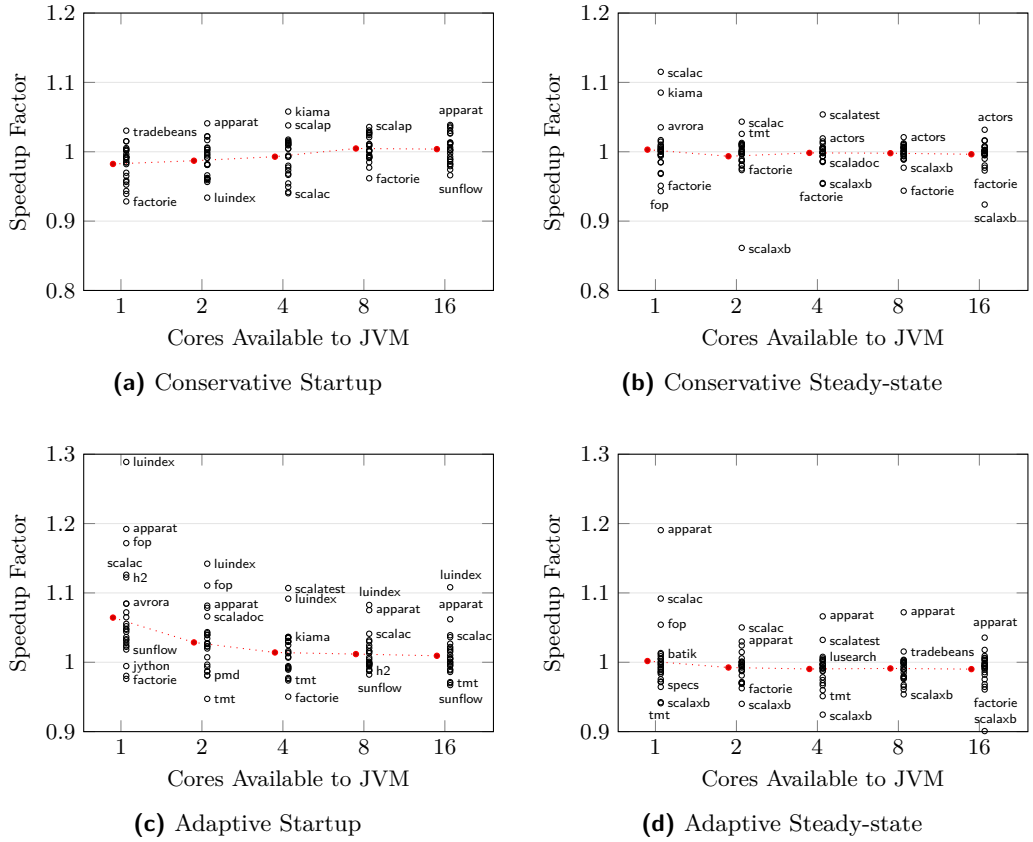
The choice of the baseline for evaluating the performance of the alternative deoptimization strategies in Graal deserves a justification. Because changes were made to the original Graal implementation, using HotSpot with Graal in place of the server compiler is our default choice. However, the production configuration of the HotSpot JVM still uses the C2 server compiler in the last compilation tier, which makes C2 a candidate for a performance baseline. Moreover, reporting changes against a well-known HotSpot configuration can help assessing the relevance of the presented changes.

A problem could arise if the Graal baseline was significantly slower than C2. Any performance improvements would be reported against a slow baseline, but the peak performance might not reach or exceed that of C2. To resolve this tension, we evaluate the relative performance of the two potential baselines, C2 and Graal, using the same benchmarks that will be used to evaluate the alternative deoptimization strategies.

The results of this comparison for different number of cores available to the JVM are shown in Figure 4. The plot of startup performance (Figure 4a) shows that on average, the Graal baseline outperforms the C2 baseline. We attribute this to the fact that we enable bootstrapping of the Graal compiler, which may also precompile frequently executed methods from the Java class library in addition to methods from the Graal compiler itself.

On the other hand, the plot of steady state performance (Figure 4b) shows that on average, the Graal baseline becomes slightly slower (2.1% in the worst case for 4 cores, with an average speed-up factor of 0.979 and individual speed-up factors from 0.824 to 1.364) than C2 as more CPU cores are made available to the JVM. The single-core case is an exception in which Graal outperforms C2 by 9% (average speed-up factor of 1.090, individual speed-up factors from 0.888 to 1.913).

In summary, Graal is a competitive compiler for our workload and this experiment validates our choice of Graal as the baseline.



(a) Conservative Startup **(b)** Conservative Steady-state

(c) Adaptive Startup **(d)** Adaptive Steady-state

Figure 5 Startup and steady-state performance of the alternative deoptimization strategies when executing the DaCapo and ScalaBench benchmarks. Black circles indicate the speed-up factor of individual benchmarks against the default Graal baseline. Red points represent the geometric mean of speed-up factors across all bechmarks. The line connecting the geometric means is intended only as a visual aid.

5.1.2 Start-up Performance

The result of evaluating the startup and steady-state performance of the alternative deoptimization strategies is presented in Figure 5. The default deoptimization strategy used in Graal represents the baseline. Figure 5a shows that in the single-core case the *conservative* strategy is on average 1.8% slower than the baseline (average speed-up factor of 0.982, individual speed-up factors from 0.941 to 1.190). As the number of CPU cores increases, the single-core slowdown becomes a slight speed-up for 16 cores. The *conservative* strategy apparently causes more compilation work and more cores allow it to hide the compilation latency. While tolerating some deoptimizations may provide a slight performance benefit, in this case it is completely outweighed by the extra compilation work.

In contrast, Figure 5c shows that the *adaptive* strategy is on average 6.4% faster in the single-core case (average speed-up factor of 1.064, individual speed-up factors from 0.630 to 1.451), and remains on average slightly faster. The *adaptive* strategy causes less compilation work, improving startup performance on average, but the benefit diminishes with the increasing number of available CPU cores, because the (baseline) default strategy can hide some of its compilation latency.

The two alternative strategies differ mainly in the level of tolerance for deoptimizations,

the accuracy of the deoptimization profile used to make decisions, and the deoptimization actions taken. The *conservative* strategy actually makes the compiler less sensitive to changes in profiling information during startup. On the one hand, the RecompileIfTooManyDeopts deoptimization used by the *conservative* strategy delays recompilation, but on the other hand it causes methods to be recompiled without being thoroughly reprofiled. Recall also that unlike the *adaptive* strategy, the *conservative* strategy associates deoptimization profile with the target of a deoptimization, not its origin. This impairs the ability to tolerate rare deoptimizations but deal with deoptimizations that are repeatedly triggered at the same deoptimization site. Due to inlining, the code triggering the deoptimizations may be duplicated in different methods and target different deoptimization traps, spreading the information about a single deoptimization site among different profiles.

The effect of tolerating deoptimizations is clearly workload dependent, and the results show a few interesting cases. The luindex benchmark clearly benefits from the *adaptive* strategy, as it exhibits a speed-up factor of 1.289 in the single-core case, and a speed-up factor of at least 1.083 throughout the experiment. Interestingly, it does not benefit from the *conservative* strategy, exhibiting a slow-down (speed-up factor of 0.965) in the single-core case, even though the compilation times for both strategies are similar.

In contrast to luindex, the factorie benchmark does not benefit from either of the strategies, exhibiting slowdowns (speed-up factors from 0.928 to 0.996) throughout the experiment. Further investigation shows that the slowdown results from an increased number of deoptimizations which may result in more time spent in the interpreter.

5.1.3 Steady-state Performance

The plots in Figure 5b and Figure 5d show the steady-state performance of both strategies. Even though the results for individual benchmarks differ slightly, on average the steady-state performance of the *conservative* strategy does not really differ from the baseline. In the case of the *adaptive* strategy, the overall speed-up factor remains slightly below 1 as the number of CPU cores increases. We attribute this to the fact that unlike the *conservative* strategy, which is supported by the HotSpot runtime and attempts to store all profiling data efficiently, the implementation of the *adaptive* strategy is far from optimized. It uses more memory to store profiling data, and emits conditional code and a volatile memory access at deoptimization sites that select deoptimization action at runtime. We expect this to impact performance, especially given the memory barriers associated with the volatile memory access and the increasing number of CPU cores.

For some benchmarks, the increased tolerance to deoptimizations provided by both strategies is beneficial even during steady-state execution. The scalac benchmark benefits from both strategies in single-core and dual-core configurations, exhibiting a performance improvement of 9.4% (single-core) and 5% (dual-core) with the *adaptive* strategy, and 11.6% (single-core) and 4.4% (dual-core) with the *conservative* strategy. The apparat benchmark benefits from the *adaptive* strategy even in 4-core and 8-core configurations, which we attribute to the aging of the deoptimization profile. On the other hand, benchmarks such as tmt exhibit an average 4% slow-down in steady-state performance for all core configurations. Short-running benchmarks (less than 300ms) such as fop and scalaxb have a tendency to amplify speed-ups and slow-downs, so they appear as outliers in the plots.

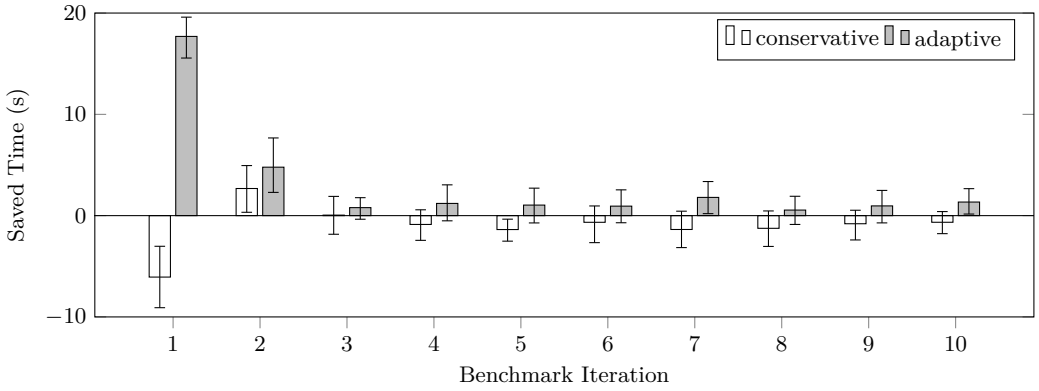
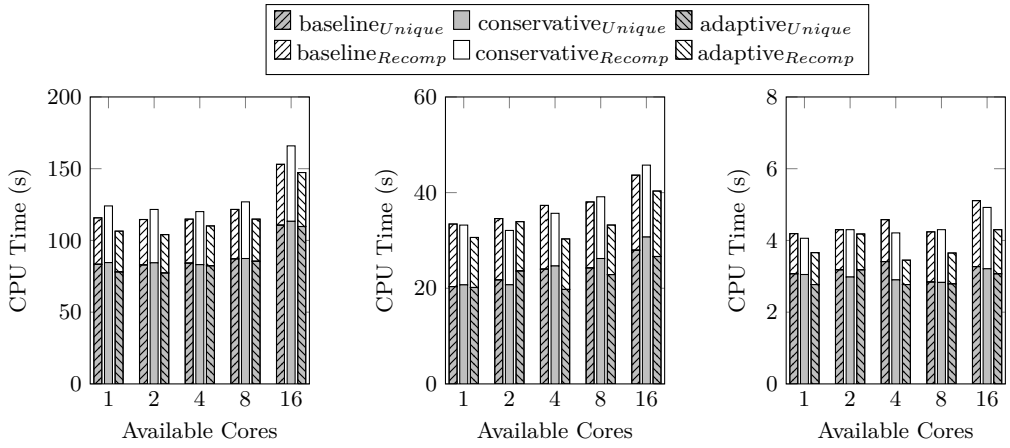


Figure 6 Total saved execution time for the selected 24 DaCapo and ScalaBench benchmarks in single-core setup. Negative values represent a slowdown.



(a) 1^{st} Iteration **(b)** 2^{nd} Iteration **(c)** 10^{th} Iteration

Figure 7 The total CPU time spent compiling (□+■) and recompiling (□) when executing the 24 selected DaCapo and ScalaBench benchmarks.

5.1.4 Overall Execution and Compilation Time

Figure 6 shows the amount of execution time saved for the 24 benchmarks from the DaCapo and ScalaBench suites together in a single-core configuration. When considering the total execution time, the execution time of each benchmark provides a weight to its respective speed-up or slow-down. With the *adaptive* strategy, the first iteration of all benchmarks finishes 17.7 seconds earlier than with the default strategy (which required 337 seconds in total), resulting in a speed-up factor of 1.053. With the *conservative* strategy, the first iteration takes 6 seconds longer than with the default strategy, resulting in a speed-up factor of 0.982. Note that these speed-up factors implicitly weigh the speed-up achieved for individual benchmarks by the execution time of each benchmark, giving a more conservative estimate than the geometric mean of speed-up factors, which treats all benchmarks with equal weight. The improvement observable with the *adaptive* strategy diminishes with the increasing number of available CPU cores, but the *adaptive* strategy still manages to save some time on each iteration, which would accumulate in the long run. Considering the overall execution time shows that the *adaptive* strategy does not necessarily hurt steady-state performance, as the results discussed in Section 5.1.3 may suggest.

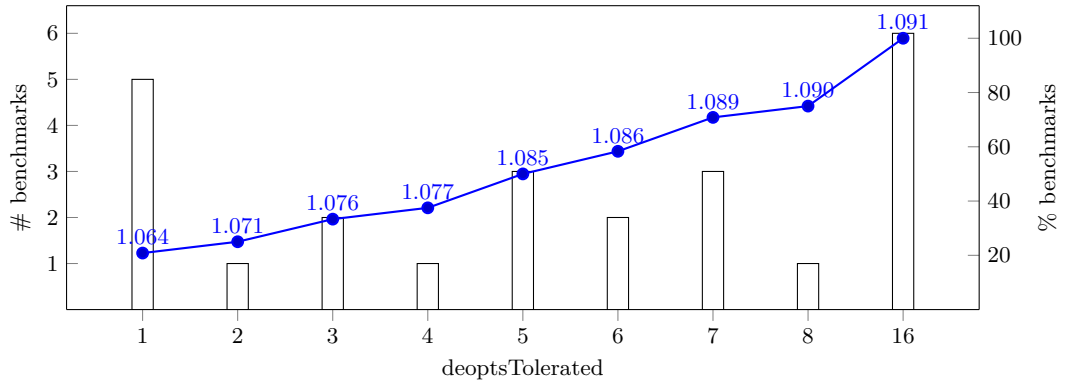


Figure 8 Theoretical speed-up with optimal values of deoptsTolerated for each of the 24 selected DaCapo and ScalaBench benchmarks. The bars represent the number of benchmarks for which the value was optimal. The line connecting the blue points represents the cumulative percentage of benchmarks for which the optimal threshold does not exceed the corresponding value. Associated with each blue point is the overall speed-up factor that would be achieved if we managed to choose an optimal threshold for each benchmark not exceeding the corresponding value.

Even though there are benefits in avoiding repeated deoptimizations, the influence of the increasing number of CPU cores on the performance results suggests that the differences in performance can be mostly attributed to compilation. To support this observation, Figure 7 provides a summary of the compilation log for all strategies. The data shows that indeed the *adaptive* strategy saves approximately 8% in the total compilation time compared to the default strategy in the first iteration of the single-core scenario, which benefits the most. The *alternative* strategy mostly saves time in all scenarios, but the impact on total execution time diminishes with increased number of cores available, and in steady-state execution. In contrast, the *conservative* strategy is apparently not a good fit for the first iteration, because it creates more compilation work. It saves some compilation time in later iterations, but too little too late.

5.1.5 Tolerance for Deoptimizations in the Adaptive Strategy

The tolerance of the *adaptive* strategy to deoptimizations can be adjusted by changing the deoptsTolerated and deoptsAllowed thresholds (c.f. Section 4.2). The results presented so far were obtained with the default values, but we are interested in how different levels of tolerance to deoptimizations impact performance of the strategy. Because the strategy had the most effect on the 1st benchmark iteration in the single-core configuration, we evaluated the performance of the *adaptive* strategy with the deoptsTolerated threshold set to 1–8, and 16. We analyzed the speed-up factors of individual benchmarks for all tested values of the deoptsTolerated threshold, and selected the threshold value resulting in maximal speed-up factor as optimal for each benchmark.

The tolerance to deoptimizations, and thus the value of the deoptsTolerated threshold, is clearly a property of a particular workload and represents a tuning parameter. If we were able to (quickly) determine the appropriate threshold based on the character of the workload being executed, the parameter could be adjusted in response to program behavior. To gauge the potential for improvement, Figure 8 shows the theoretical speed-up factor that could be achieved, if we managed to find the optimal deoptsTolerated threshold (within a given limit) for each benchmark. The plot shows that searching for an optimal threshold in the range of 1–5 would provide an optimal value for approximately 50% of benchmarks (given the upper

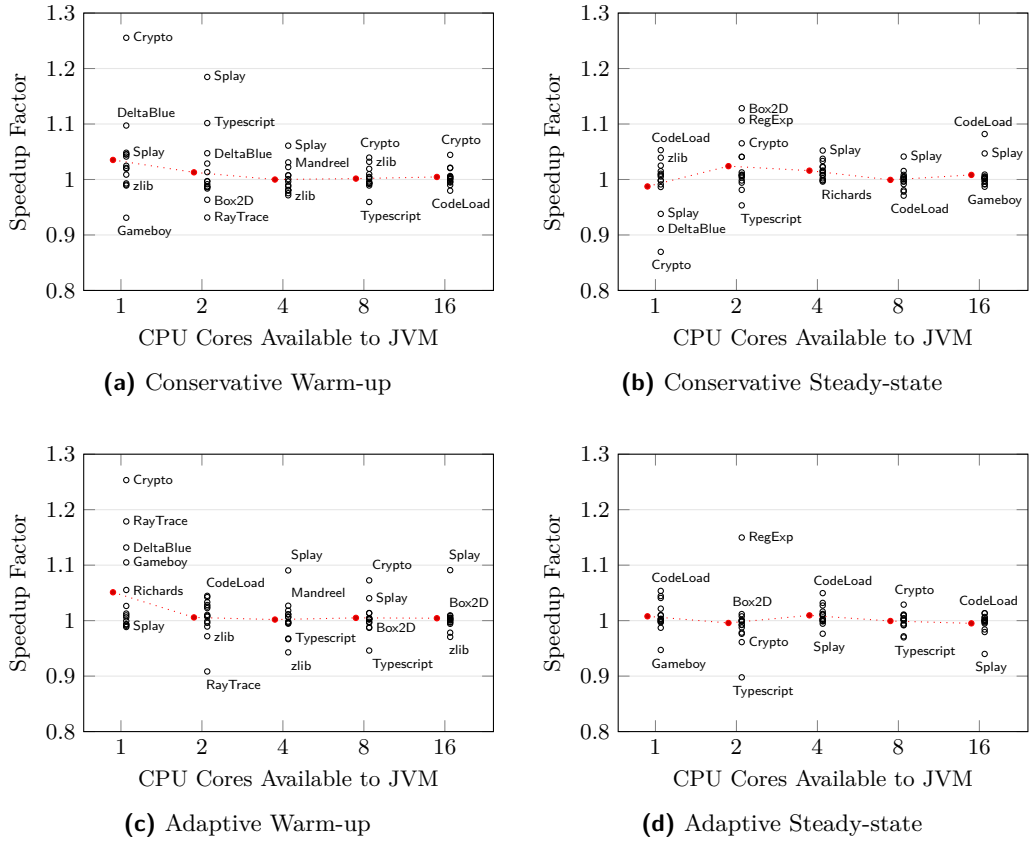


Figure 9 Warm-up and steady-state performance of the alternative deoptimization strategies when executing the Octane benchmarks on Graal.js. Black circles represent the speed-up factor of individual benchmarks against the default Graal baseline. Red points represent the geometric mean of speed-up factors across all benchmarks. The line connecting the geometric means is intended only as a visual aid.

bound of 16), and yield a speed-up factor of 1.085.

5.2 Octane on Graal.js Evaluation

To evaluate the performance of the Octane benchmarks running on Graal.js, we use the benchmarking harness for the Octane suite provided in the GraalVM binary release[16]. The harness uses benchmark-specific warm-up times ranging from 15 to 120 seconds, and a common steady-state period of 10 seconds. When finished executing a benchmark, the harness reports per-iteration execution times achieved during the warm-up and steady-state periods. We collect the per-iteration execution times for both phases and report the speed-up factors w.r.t. the default deoptimization strategy. Because the warm-up and steady-state phases are defined differently for the Octane suite than for the DaCapo and ScalaBench suites, we report the speed-up factors separately.

The plots in Figure 9a and Figure 9c show the warm-up performance of Octane benchmarks running on Graal.js with the *conservative* strategy and the *adaptive* strategy, respectively. We again show the speed-up factors for the individual benchmarks and the overall speed-up

[16] http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/index.html

factor calculated as a geometric mean of the individual speed-up factors. In the single core case, both alternative deoptimization strategies achieve better warm-up performance than the default strategy. On average, the *conservative* strategy is approximately 3.5% faster (average speed-up factor of 1.035, individual speed-up factors from 0.931 to 1.255) and the *adaptive* strategy is approximately 5.1% faster (average speed-up factor of 1.051, individual speed-up factors from 0.989 to 1.253) than the default strategy.

The results indicate that the JavaScript runtime implemented using the Truffle framework generally benefits from tolerating deoptimizations during startup due to the reduction of the compilation work. This is potentially beneficial for JavaScript workloads that mostly execute code once, instead of repeatedly. However, similarly to the DaCapo and ScalaBench benchmarks, the benefit diminishes as the number of available CPU cores increases. Even though JavaScript is a single-threaded language, the runtime may use additional CPU cores to hide compilation latency.

Finally the plots in Figure 9b and Figure 9d show the steady-state performance of Octane benchmarks with the *conservative* and *adaptive* strategies, respectively. Neither of them deviates from the performance of the default strategy in a significant way.

5.3 On the Scale of Performance Changes

The results of performance evaluation indicate that on average the *adaptive* deoptimization strategy provides moderate improvements to startup performance in a single-core scenario. As the number of cores and benchmark runtime increases, the effect wears off, until it disappears. Because the improvement is moderate, it is difficult to assess how it fits the overall picture. In his 1974 paper, Knuth notes that in established engineering disciplines, 12% improvement, easily obtained, is never considered marginal [13]. The improvements obtained here are roughly half of that, but still rather easily obtained, given the complexity of the other parts of the VM.

Arguably, the execution time aspect of the improvement diminishes with more CPU cores available to the JVM, but the computation saved remains. To provide a frame of reference, we evaluate the single-core performance of five different configurations of the HotSpot VM and compare it with their respective baselines. Two of the configuration changes swap entire VM subsystems, while three other changes alter the behavior of the dynamic compiler.

The first baseline is the default configuration of HotSpot with C2 as the top tier-compiler to which we compare the following configurations:

g1gc Replaces the default garbage collector in HotSpot with the Garbage First (G1) garbage collector.

noinline Disables inlining in C2.

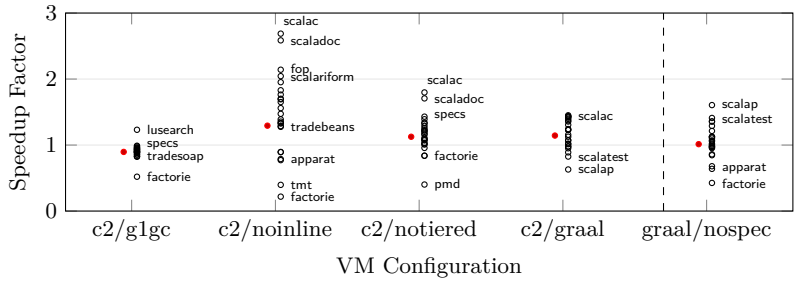
notiered Disables tiered compilation in HotSpot, i.e., disables C1 compiler.

graal Replaces the C2 server compiler with Graal.

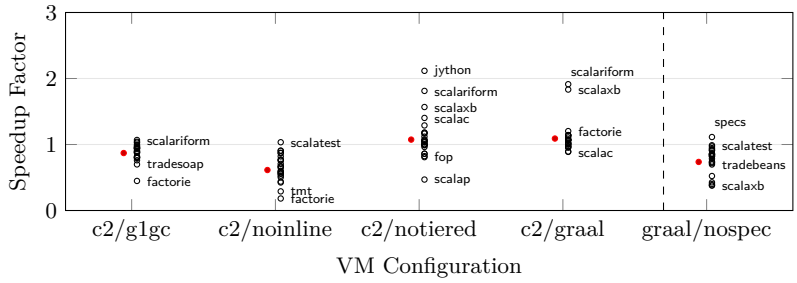
The second baseline is the default configuration of HotSpot with Graal as the top-tier compiler to which we compare the following configuration:

nospec Disables the majority of speculative optimizations relying on deoptimization in Graal, providing a rough estimate of performance gains enabled by deoptimization.

The results of the evaluation are shown in Figure 10. The subfigures correspond to startup (Figure 10a) and steady state (Figure 10b) performance, each showing average performance of the first four configurations compared to the C2 baseline, followed by the fifth configuration compared to the Graal baseline.



(a) Startup (1-core)



(b) Steady-state (1-core)

Figure 10 Startup and steady-state performance of different VM configurations executing the DaCapo and ScalaBench benchmarks. Black circles represent speed-up factor against the respective baseline, red points represent geometric mean of speed-up factors across all benchmarks.

In a single-core setting, the change of the GC algorithm caused a 10.4% degradation in startup performance, and a 12.9% degradation in steady-state performance of the *g1gc* configuration. We are aware that this results from different mode of operation of the G1 collector, which is typically recommended for heaps exceeding 6 GB. However, it illustrates the kind of performance impact a careless swap of a GC may have in a particular scenario.

The *noinline* and *nospec* configurations reduce the amount of compilation work, either due to avoiding redundant compilation of methods that could have been inlined, or due to compiling immediately using the top-tier compiler. Consequently, we observe a significantly better startup performance in the single-core scenario—29.4% improvement due to disabled inlining, and 12.6% due to disabled tiered compilation. In steady state, disabled tiered compilation retains a 7.4% performance improvement, but disabled inlining changes the situation dramatically. Because inlining is a critical optimization that increases optimization scope and effectively enables inter-procedural optimization, disabling inlining causes a 40% degradation in steady-state performance.

The *graal* configuration illustrates the effect of replacing C2 with Graal as the top-tier compiler. This situation is investigated more closely in Section 5.1.1, here we just note a 14.4% improvement in startup performance, and 9% improvement in steady-state performance.

Finally, the *nospec* configuration illustrates the effect of disabling various speculative optimizations in the Graal compiler. These include elimination of unreached branches, heuristic inlining, speculative `instanceof` test, elimination of unreached exception handlers, and elimination of safepoints within a loop. On average, this change appears to have neutral impact on startup performance, but has a significant impact later, resulting in a 26.1% degradation in steady-state performance.

This suggests that the above speculative optimizations pay off in the long term, but do not provide much benefit at startup. The *adaptive* strategy complements this by providing a moderate improvement in startup performance without adversely affecting performance in the long term.

6 Conclusion

Deoptimization is a key fallback mechanism for implementing speculative optimizations in modern dynamic compilers. While the existing literature covers the implementation aspects of deoptimization in great depth, the actual use of deoptimizations in compiled code has not been previously studied.

We present a study of deoptimization behavior in benchmarks executing on a Graal-enabled HotSpot VM. We profile deoptimization sites in the code produced by the Graal compiler, and provide a qualitative and quantitative analysis of deoptimization causes in benchmark suites such as DaCapo, ScalaBench, and Octane, which provide workloads derived from real applications and libraries written in Java, Python, Scala, and JavaScript. We show that only a small fraction of deoptimization sites actually trigger deoptimizations at runtime, and that most of the deoptimizations actually triggered in Graal-compiled code unconditionally invalidate and reprofile the method which caused a deoptimization.

To gain insight on the trade-offs made by Graal in its default deoptimization strategy, we modify Graal to add support for two alternative deoptimization strategies and evaluate benchmark performance using the three strategies. We show that by avoiding the *conservative* strategy provided by the HotSpot VM runtime, Graal gains better startup performance. However, we also show that certain tolerance to deoptimizations can provide performance benefits, if used with a precise deoptimization profile. The *adaptive* strategy, which switches among various deoptimization actions based on a precise deoptimization profile, manages to reduce the amount of method recompilations and eliminate certain repetitive deoptimizations. As a result, on a single-core system, it improves the average start-up performance by 6.4% in the DaCapo and ScalaBench benchmarks, and by 5.1% in the Octane benchmarks.

Finally, we show that tolerance to deoptimizations is a workload-specific parameter, and that finding correlation between some workload characteristics and the appropriate level of tolerance to deoptimizations can potentially provide additional performance benefits.

Acknowledgements

We thank Jan Vitek, Olga Vitek, Petr Tůma, and the anonymous ECOOP reviewers for their suggestions on how to improve the paper. We also thank Tom Rodriguez, Doug Simon, Gilles Duboscq and Thomas Würthinger for their support with the HotSpot VM and the Graal compiler. The research presented in this paper was supported by Oracle (ERO project 1332), by the European Commission (contract ACP2-GA-2013-605442), by the Charles University institutional funding (project SVV-260451), and by project no. LTE117003 (ESTABLISH) from the INTER-EUREKA LTE117 programme by the Ministry of Education, Youth and Sports of the Czech Republic.

References

1 Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

2 Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online Feedback-directed Optimization of Java. In *Proc. 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA 2002, pages 111–129. ACM, 2002.

3 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proc. ACM SIGPLAN International Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA 2006, pages 169–190. ACM, 2006.

4 Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. Using Hpm-sampling to Drive Dynamic Compilation. In *Proc. 22nd ACM SIGPLAN Conference on Object-oriented Programming, Systems and Applications*, OOPSLA 2007, pages 553–568. ACM, 2007.

5 Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Proc. ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA 1991, pages 1–15. ACM, 1991.

6 L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proc. 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL 1984, pages 297–302. ACM, 1984.

7 S. J. Fink and Feng Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *Proc. IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2003, pages 241–252. IEEE Computer Society, March 2003.

8 Google. Octane 2.0 JavaScript Benchmark. https://developers.google.com/octane/.

9 Dayong Gu and Clark Verbrugge. Phase-based Adaptive Recompilation in a JVM. In *Proc. 6th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2008, pages 24–34. ACM, 2008.

10 Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 1992, pages 32–43. ACM, 1992.

11 Urs Hölzle and David Ungar. Reconciling Responsiveness with Performance in Pure Object-oriented Languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, July 1996.

12 Madhukar N. Kedlaya, Behnam Robatmili, Cġlin Caşcaval, and Ben Hardekopf. Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines. In *Proc. 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2014, pages 103–114. ACM, 2014.

13 Donald E. Knuth. Structured Programming with Go to Statements. *ACM Comput. Surv.*, 6(4):261–301, December 1974.

14 Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008.

15 Chandra J. Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the Overhead of Dynamic Compilation. *Software: Practice and Experience*, 31(8):717–738, 2001.

16 Prasad Kulkarni, Matthew Arnold, and Michael Hind. Dynamic Compilation: The Benefits of Early Investing. In *Proc. 3rd International Conference on Virtual Execution Environments*, VEE 2007, pages 94–104. ACM, 2007.

17 Prasad A. Kulkarni. JIT Compilation Policy for Modern Machines. In *Proc. ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA 2011, pages 773–788. ACM, 2011.

18 Oracle. Graal project. http://openjdk.java.net/projects/graal/.

19 Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ Server Compiler. In *Proc. Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*, JVM 2001, pages 1–1. USENIX Association, 2001.

20 Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proc. ACM International Conference on Object Oriented Programming, Systems, Languages and Applications*, OOPSLA 2011, pages 657–676. ACM, 2011.

21 Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Design and Evaluation of Dynamic Optimizations for a Java Just-in-time Compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, July 2005.

22 John Whaley. Partial Method Compilation Using Dynamic Profile Information. In *Proc. 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA 2001, pages 166–179. ACM, 2001.

23 Christian Wimmer, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger. One Compiler: Deoptimization to Optimized Code. In *Proc. 26th International Conference on Compiler Construction*, CC 2017, pages 55–64. ACM, 2017.

24 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proc. ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204. ACM, 2013.

25 Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers. In *Proc. 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT 2003, pages 148–158. IEEE Computer Society, 2003.

26 Yudi Zheng, Lubomír Bulej, and Walter Binder. Accurate Profiling in the Presence of Dynamic Compilation. In *Proc. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 433–450. ACM, 2015.