Compilation Queuing and Graph Caching for Dynamic Compilers [*]

Lukas Stadler   Gilles Duboscq
Hanspeter Mössenböck
Johannes Kepler University Linz, Austria
{stadler, duboscq, moessenboeck}@ssw.jku.at

Thomas Würthinger
Oracle Labs
thomas.wuerthinger@oracle.com

Abstract

Modern virtual machines for Java use a dynamic compiler to optimize the program at run time. The compilation time therefore impacts the performance of the application in two ways: First, the compilation and the program's execution compete for CPU resources. Second, the sooner the compilation of a method finishes, the sooner the method will execute faster.

In this paper, we present two strategies for mitigating the performance impact of a dynamic compiler. We introduce and evaluate a way to cache, reuse and, at the right time, evict the compiler's intermediate graph representation. This allows reuse of this graph when a method is inlined multiple times into other methods. We show that the combination of late inlining and graph caching is highly effective by evaluating the cache hit rate for several benchmarks.

Additionally, we present a new mechanism for optimizing the order in which methods get compiled. We use a priority queue in order to make sure that the compiler processes the hottest methods of the program first. The machine code for hot methods is available earlier, which has a significant impact on the first benchmark.

Our results show that our techniques can significantly improve the start up performance of Java applications. The techniques are applicable to dynamic compilers for managed languages.

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

General Terms Algorithms, Languages, Performance

[*] This work was supported by Oracle.

Keywords Java, Virtual Machine, Compilation, Caching, Queuing, Optimization, Performance

1. Introduction

Managed language runtimes start the execution of a program by interpreting its methods. The runtime dynamically detects *hot* methods that form a significant part of the program's execution time. Figure 1 shows the life cycle of such a hot method. The first invocations are slowly executed in the interpreter. Then the method is compiled and subsequently executed fast in native machine code. While the compilation of a method is often performed in parallel to the running program, it still competes with the program for CPU cycles.

There are two important metrics that affect the program's overall performance: First, the number of invocations of a hot method before it gets compiled. Second, the time necessary to compile a hot method.

In this paper, we present techniques for reducing those two metrics. The compilation of a method uses significant CPU resources such that it does not pay off to compile rarely executed methods [10]. Therefore, it is not sufficient to just reduce the number of invocations before a method gets compiled. Instead the runtime must make sure that the hottest methods are compiled earlier, because their compilation results in the biggest improvement on the program's execution speed. The aggressive inlining of dynamic compilers results in many methods being parsed several times (see Figure 4). This makes caching strategies for the method's intermediate representation an interesting target for optimizing the com-

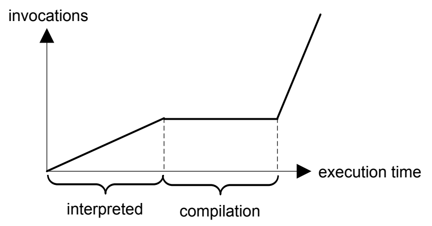

Figure 1. Executing a method in a managed runtime.

pilation time without major compiler modifications. This paper contributes the following:

- We present a new approach for managing the compilation queue of a dynamic compiler.
- We introduce an algorithm that combines late inlining and graph caching for reducing the compilation time.
- We show the effectiveness of the graph caching by evaluating the cache hit rate.
- We give an extensive evaluation of our techniques in the context of different compilation thresholds.
- We demonstrate that the combined application of our techniques significantly improves the startup performance.

2. System Overview

We implemented our system in the context of the Graal OpenJDK project [12]. The Graal VM is a modification of the Java HotSpot^{TM} VM where the compiler and the compilation queue are replaced with an implementation in Java. Figure 2 gives a schematic view of the system components that are relevant for the techniques described in this paper. The HotSpot^{TM} VM starts executing the Java program in the interpreter. When a method gets hot, the VM inserts it into the compilation queue with a priority value that determines the hotness. If the hotness changes over time, the VM has the ability to update the priority value.

The compiler uses a configurable number of worker threads that poll this queue and consecutively remove the topmost method to compile it. After the compilation finishes, the resulting machine code is sent back to the runtime, which installs it in its code cache. The runtime makes sure that subsequent calls to that method immediately jump to the compiled machine code instead of the interpreter.

Whenever the compiler needs the intermediate representation graph for a method, it first queries the graph cache whether it already exists for this method. Only if there is no cache hit, the compiler performs the expensive parsing of the method's bytecodes, resolves the bytecodes' references into the constant pool, and builds a static single assignment (SSA) form [5] representation. Otherwise, the cached graph is directly used. The Graal compiler performs inlining by replacing an invocation compiler node by the compiler graph of the called method.

The cached graphs can be built using optimistic assumptions about the current state of the application. Such assumptions can be invalidated by subsequent changes in application behavior. Therefore, the runtime must be capable of removing graphs from the cache if one of their assumptions no longer holds. This invalidation can also be necessary for installed machine code. If machine code associated with an invalid assumption is executed, it is *deoptimized* [8], and execution continues in the interpreter.

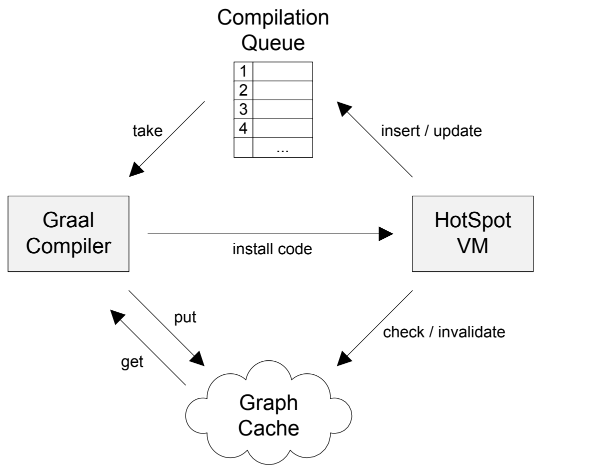

Figure 2. System architecture of the Graal VM.

The modifications necessary to implement our techniques to the Graal VM are limited: We changed the way methods are selected for compilation, created a priority compilation queue, and added the graph caching mechanism. Therefore, the techniques can be applied to any managed runtime that includes a compilation queue and the ability to store and reuse the compiler graphs.

We will first describe our compilation queuing system, then we will discuss graph caching and its implementation. We will then evaluate the impact of these changes on the performance of various benchmarks.

3. Compilation Queuing

Since the compilation of methods in a system using a dynamic compiler happens concurrently to the running application, it is important that the methods with the most influence on application run-time performance are compiled first. Additionally, the system needs to decide which methods are important enough to be compiled at all, and which methods will only ever be executed in the interpreter.

3.1 Detecting Hot Methods

In order to compile methods at the optimal point in time, knowledge would be required about when methods will be called during execution. In dynamic environments, like a Java VM, the system cannot foresee the future, and therefore all its decisions need to be based on a heuristic that predicts method usage patterns using previous events.

There are two basic ways in which the dynamic behavior of an application can be used to determine when a method should be considered for compilation:

Invocation Counters are kept for each method, and incremented each time the method is called. In order to give weight to long-running loops, the counter is usually also incremented on loop back edges within a method.

When a sufficiently large number of invocations has been recorded (i.e., the so-called *compilation threshold* has been reached), a method is considered to be hot, and therefore scheduled for compilation.

Stack Sampling periodically records the top frame of each thread's stack. When a method is contained within a sufficient amount of these recordings, it is considered to be hot.

This sampling needs to happen many times per second, otherwise the system will either take too long to detect important methods or be too imprecise to detect the right methods.

The Graal VM uses invocation counters to detect hot methods, similar to the HotSpot^{TM} VM, on which it is based.

3.2 Method Hotness Model

In order to be able to determine the importance of a specific method, a measure of the future usage of the method would be required. Since this cannot be measured, an approximation of the method's future importance can be based on how many times it has already been executed. Once a method is considered hot (e.g., because its invocation count reached the compilation threshold), there are different models to assign importance to methods:

FIFO. The relative importance of methods is determined solely by the order in which they reach their compilation threshold.

Size-Based. Smaller methods are more important.

Kulkarni's Method. Kulkarni [10] introduces an algorithm that scales the method invocation count by a global counter which is incremented for every method invocation. This technique favors methods that have recently been invoked very frequently.

All of these have significant drawbacks: The FIFO technique does not react to changes in application behavior, as it is not able to favor methods that have recently become very hot over less important methods that have been added to the compilation queue before. The size-based technique also does not react to changes in application behavior, and the size of a method does not predict the method's influence on performance in the presence of inlining. Kulkarni's method is able to react to changes in application behavior, but only as long as the method's priority is correct when it reaches the compilation threshold the first time. Also, the global invocation counter is a very coarse scaling factor that depends on many unrelated elements.

We therefore introduce a new system to measure the hotness of a method that calculates the actual speed at which the invocation counter increases. As long as a method is not compiled yet the runtime environment will periodically determine the method's priority by relating the change in invocation count with the elapsed time:

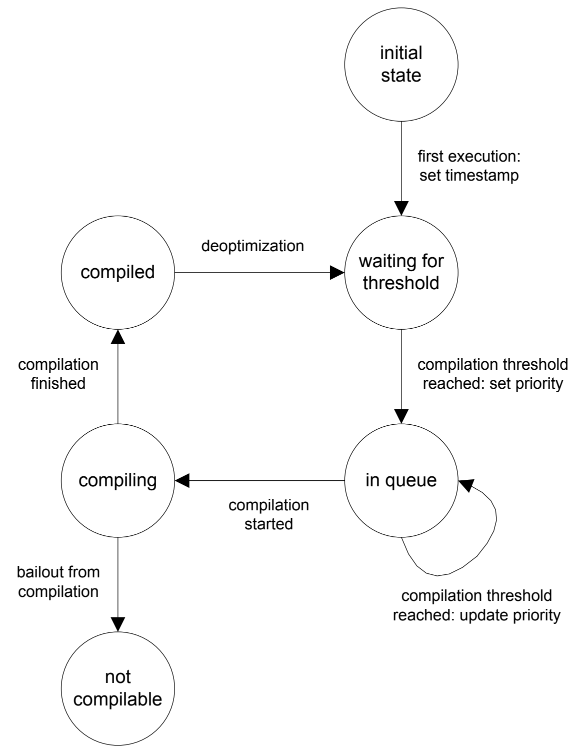

Figure 3. Overview of the states and transitions of methods.

$$\text{Priority} = \frac{\Delta \text{ method invocation count}}{\Delta \text{ timestamp}}$$

A simple compilation threshold is used to determine when the method is first considered for compilation.

This approach provides a good initial approximation of method importance and can react quickly if a method suddenly becomes more important. This happens, for example, when an application enters a new phase [11].

We decided to use the speed-based approach, because it is reasonably simple to implement, provides an accurate approximation of method hotness and can react to changes in application behavior quickly.

Figure 3 shows the states that methods can have in our system and the transitions between these states: As long as a method has never been executed, it is in its *initial state*. When it is invoked for the first time the method's timestamp will be set, and the method then waits until its invocation counter reaches the compilation threshold (*waiting for threshold*). As soon as the threshold is reached, the speed at which the invocation counter increases is calculated using the above formula and used as priority for the compilation queue (*in queue*). Also, the invocation counter will be reset so that the priority can be updated as soon as the compilation threshold is reached again.

When the compilation of a method starts it is removed from the queue (*compiling*). If the compiler determined that for some reason it is not able to compile the method, it will switch to the *not compilable* state. If the compilation finished successfully, the method will be in the *compiled* state.

Later on the method might be deoptimized because an assumption was violated (see Section 4.3), in which case the system will set the method's timestamp and then wait for the method's invocation counter to reach the compilation threshold again.

3.3 Compilation Queue with Priorities

We implemented the compilation queue in such a way that it is ordered according to the priorities of the methods to be compiled:

- The compilation queue itself is an efficient thread-safe priority queue, along with worker threads that take elements from the queue and process them. The number of worker threads defaults to the number of available cores in the system.
- Each time a method is ready to be compiled a *compilation task* is created and put into the compilation queue.
- The compilation tasks are ordered according to the priorities (i.e., the invocation counter speeds) of their methods.
- Compilation tasks may be reordered when the priority of a method improves. Since reordering within the compilation queue is a time-consuming task it is only performed if the priority changes significantly (by at least a factor of two). The original compilation task is canceled (by simply setting a flag on it) and a new one is inserted with the new priority.

  Note that the compilation tasks are only reordered if a method's priority improves, because also reordering on priority decreases can lead to situations where methods that are used in bursts will be compiled very late.

4. Graph Caching

Inlining is the process of replacing a call to a method with the method's implementation. It is one of the most important optimizations that compilers perform in order to increase the run-time performance of applications. Dynamic compilers, like the HotSpot client and server compilers and the Graal compiler, usually perform a large amount of inlining during compilation.

The same method is often inlined multiple times in different places. The cache hit ratio in Figure 4 shows the average probability that a method that is about to be inlined has been inlined before, for a typical benchmark (the DaCapo benchmark suite, see Section 5). Overall, the probability is above 90%, and most methods that are inlined are small (98% are smaller than 100 bytecodes, and 75% are smaller than 25 bytecodes).

Given the right infrastructure the compiler could cache intermediate results (i.e., compiler graphs) for methods that are inlined multiple times. The high reuse will translate to the high cache hit rates shown in Figure 4. However, such

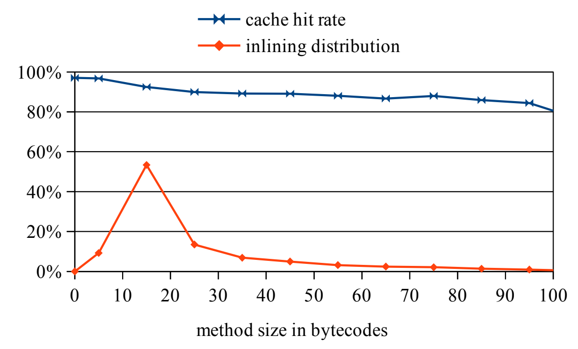

Figure 4. Cache hit rate and inlining distribution during typical benchmarks, in relation to the inline method's size.

a system needs to fulfill some specific requirements and it also needs to be aware of how some parts of the compiler interact with the cache. These requirements and interactions are explained in the following sections.

4.1 Late Inlining

Traditionally, dynamic compilers (e.g., the HotSpot client and server compilers) perform inlining during their bytecode parsing phase, by parsing the bytecodes of the inlined method as if they were part of the caller. However, this has several disadvantages:

- The compiler needs to make the inlining decision very early. This forces certain optimizations, such as global value numbering and canonicalization, to be done already during bytecode parsing, which makes parsing more complicated.
- Optimizations that happen later, like escape analysis [2], cannot perform inlining even if they see that it would be beneficial.

Graal, on the other hand, does *late inlining* (like, for example, JRockit [13]). It does not contain any facilities to perform inlining during the bytecode parsing step. This significantly decreases the complexity of Graal's bytecode parsing component, because it does not need to deal with multiple method scopes at once.

Graal's inlining system parses the method that needs to be inlined into a separate graph and copies the contents into the target graph. Copying the graphs is a fast and simple operation that takes only a negligible percentage of the compile time for typical benchmarks.

Separating bytecode parsing from inlining not only makes the compiler simpler and easier to maintain, it also has the effect that bytecode parsing itself resembles a *pure function*, whose results(i.e., the compiler graph that resulted from parsing the method to be inlined) can be cached for subsequent inlining operations. Inlining during bytecode parsing uses, and changes, the state of the parent method compilation, and therefore no intermediate results can be reused.

4.2 Garbage Collection

Most Java JIT compilers use some variant of explicit region memory allocation [6], also called zone allocation or arena allocation. This means that all temporary data structures allocated during compilation are freed en bloc at the end of the compilation process.

While this works well in limiting the life time of allocated memory, it also means that all compilation results need to be rescued explicitly in order to survive the destruction of the compilation's memory region. HotSpot, for example, copies the generated machine code of every method and a serialized version of the associated meta data (debug information, relocation info, etc.) to a global code cache before freeing what it calls a *resource area*.

Similarly, cached intermediate results also need to be rescued to a different memory area in a compiler that uses zone allocation.

Graal lets the JVM's garbage collector free the memory allocated during compilations. Therefore, data structures referenced by a global cache system will automatically be kept alive, and can use Java's soft pointers to respond to memory pressure appropriately.

4.3 Assumptions

Modern JIT compilers will perform aggressive optimizations based on assumptions about the future state of an application and use deoptimization in case one of the assumptions does not hold at a later point in time. There are two types of assumptions:

Static Assumptions deal with state surrounding a method, like the list of loaded classes.

They are used, for example, to optimize potentially polymorphic calls. If class hierarchy analysis guarantees that only one class has been loaded as the potential receiver type, the call can be replaced by a static call. If other potential receiver classes are loaded later on, the assumption is violated and the code depending on the assumption needs to be invalidated.

Dynamic Assumptions depend on the dynamic behavior of an application. Dynamic assumptions are usually facts that cannot be statically proven, but are hinted at by profiling feedback gathered by the interpreter.

For example, a compilation might make the assumption that a branch is never taken. In this case the branching condition still needs to be checked, in order to see if the assumption holds.

Static assumptions are generated during and after inlining, and therefore after intermediate results have been put into the cache. This means that the graphs that are put into the cache do not yet incorporate any static assumptions, so there is no need for the graph caching system to deal with them.

Dynamic assumptions, however, can be taken during bytecode parsing. For example, the compiler might completely omit a branch when the profiling information suggests that it will never be taken. This means that any cached intermediate result will encode a specific behavior of the application that might or might not still conform to the actual behavior later on.

When the behavior of a method that has been compiled changes, some dynamic assumptions might not hold any more. In case the execution reaches such an assumption, the only thing a system without a graph cache needs to do is to invalidate the compiled version of the method. When using a graph cache, however, this also means that if the dynamic assumption originates from within a cached graph, this graph needs to be evicted from the cache. Subsequent compilations would otherwise reuse a cached graph that represents outdated behavior, possibly leading to repeated deoptimizations.

4.4 Graph Caching Implementation

Our system implements caching of intermediate results within the compiler in the following way:

- Whenever the compiler performs an inlining, it first checks if there is a cached version of the method to be inlined. If there is a cached version, it will be used, otherwise the method's bytecodes will be parsed.
- Whenever a method is parsed during inlining, it will be put into the graph cache.
- The graph cache is a data structure that is global to the compiler, so it needs to be thread-safe because the Graal compiler uses multiple compiler threads within one compiler instance. This is easily achieved in Java by using readily available synchronized data structures.
- Our graph cache is implemented as a least-recently-added cache with a fixed maximum size. Insertion instead of access ordering was chosen in order to lower the contention when multiple threads access the cache. The influence of different cache sizes is evaluated in Section 5.2.4.
- In order to be able to track deoptimizations back to the inlining graph they originate from, each node in the compiler graph that can later on lead to a deoptimization is associated with the inlining graph it was created in. This way the graph cache can evict the correct graphs from the cache when a deoptimization happens. Globally unique IDs are associated with inline graphs to avoid having to keep object references that potentially prevent garbage collection of graphs.
- The compiler hands over the ID of the graph that caused a deoptimization to the runtime system as part of the debugging information associated with that deoptimization.

5. Evaluation

All benchmarks were executed on an Intel Core i5 750 quad-core 2.67GHz CPU running Ubuntu 11.10 (Linux 3.0.0-17). Graal was built using revision d5cf399e6637 from the official OpenJDK Graal repository available at http://hg.openjdk.java.net/graal/graal.

We use the DaCapo benchmark suite [1] in the current version (9.12-bach) in order to evaluate the impact of our caching and queuing optimizations. The DaCapo suite, which is split into 14 sub-benchmarks, is widely used and therefore well understood. It also has a useful notion of benchmark runs, which allows us to measure the impact of optimization on both first and best run performance.

The charts in this section are all arranged on a linear, unbiased, lower-is-better scale. Depending on the context, the unit of measurement is either run time in milliseconds or run time as percentage of the run time of a normal, unoptimized run. The relative representation is necessary when different sub-benchmarks are shown in one chart because of the wide range of absolute run time measurements (from 1.5 to 30 seconds).

All benchmark results in this section were produced by 10 runs of the benchmark suite with the same parameters. Averages for specific benchmarks were calculated as arithmetic mean, and the associated charts show the standard deviation of the results. Averages over multiple benchmarks were calculated using the geometric mean, because the different DaCapo sub-benchmarks have significantly different run times.

The evaluation contains results for four different VM configurations: *Normal* is the unmodified compilation queue policy, without graph caching. *Cache* is the unmodified compilation queue policy, with graph caching. *Prio* is the priority queue compilation policy, without graph caching. *Prio_cache* is the priority queue compilation policy, with graph caching.

5.1 Peak Performance

Figure 5 shows the overall peak performance of the benchmark (expressed as the geometric mean of all the sub-benchmarks) for different VM configurations, clustered by compilation threshold. The peak performance is measured after a large number of iterations, by which time all important methods have been compiled and the benchmark results stabilize. Our optimizations do not have a statistically significant influence on the peak performance of the benchmark.

For the compilation queue optimizations this is to be expected, since by the time the peak performance is measured all important methods will have been compiled, and the order in which they are compiled does not influence the outcome.

The graph caching optimizations, however, could theoretically have a negative influence on peak performance. By caching graphs the compiler always uses only the information about the running application that was available when the graph's method was inlined the first time. This could lead to suboptimal compilation results, e.g., the dynamic assumptions incorporated into the graph might be inaccurate. In the benchmarks, however, this does not seem to be the case. This means that by the time a method is old enough to be inlined the first time, it is mature enough so that caching the graph does not have a negative effect on peak performance.

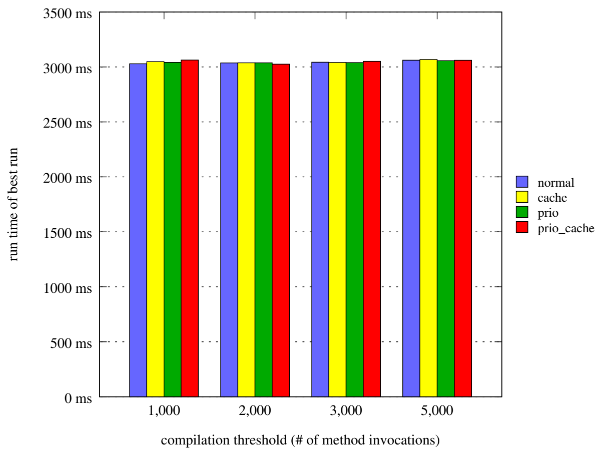

Figure 5. Geometric mean of the best runs of all benchmarks, for different VM configurations, clustered by compilation threshold.

It is important to note that the compilation threshold also has no significant influence on peak performance. On the one hand this is not surprising, since by the time the best run is measured all important methods will have been compiled, regardless of the compilation threshold. On the other hand, this means that the additional maturity of the methods gained by the higher compilation threshold does not improve peak performance. This is an interesting observation, because Graal relies very heavily on profiling information to produce optimized compilation results.

5.2 First Run Performance

The area where the compilation queue and graph caching optimizations should have the largest influence is the performance of the first run of the benchmarks. Therefore we explicitly measured the first run performance of the DaCapo benchmark suite for different VM configurations over a large range of compilation thresholds to determine:

- Do the graph caching optimizations have a positive influence on first run performance?
- Do the compilation queuing optimizations have a positive influence on first run performance?
- How does the influence of the optimizations change when the compilation threshold is increased or decreased?
- Which is the optimal combination of VM configuration and compilation threshold?

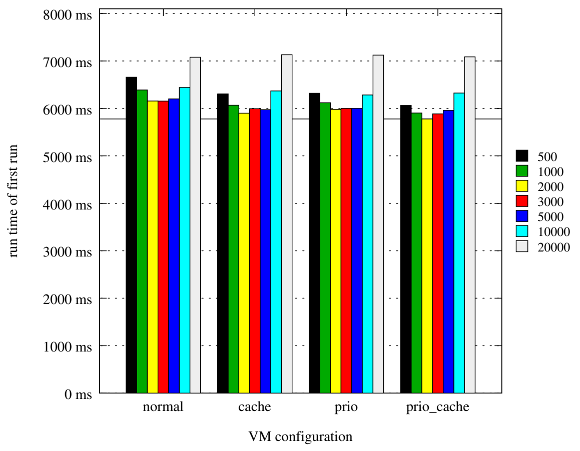

Figure 6. Geometric mean of the first run of all benchmarks, for different compilation thresholds, clustered by VM configuration.

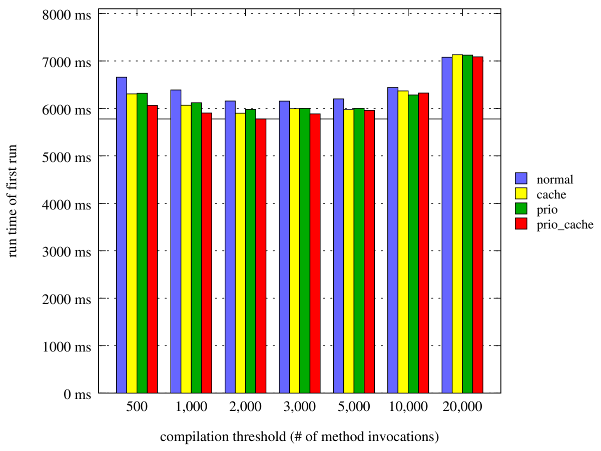

Figure 7. Geometric mean of the first run of all benchmarks, for different VM configurations, clustered by compilation thresholds.

5.2.1 Overall Influence on First Run Performance

Figure 6 shows the overall first run performance of the benchmark (expresses as the geometric mean of all sub-benchmarks) for different compilation thresholds, clustered by VM configuration.

The best results overall are achieved at a compilation threshold of 2,000 using the ”prio_cache” configuration at 5,777 ms, which is a speedup of 6% over the best result for the ”normal” configuration (6,154 ms), which is achieved at a compilation threshold of 3,000.

Figure 6 also shows that the compilation queue and graph caching optimizations vastly improve performance for low compilation thresholds. A lower compilation threshold leads to more methods being put into the compilation queue, which in turn means that it is more important to prioritize and quickly compile these methods.

Figure 7 shows the overall first run performance of the benchmark for different VM configurations, clustered by compilation threshold. This again shows that our optimizations have the most influence on performance for low compilation thresholds. At a compilation threshold of 20,000 there is virtually no difference in performance between VM configurations. At this threshold all configurations perform equally bad because the threshold fails to catch important methods during the first benchmark run.

5.2.2 Detailed Influence on First Run Performance

Figures 8 through 10 show the detailed influence of our optimizations on the DaCapo sub-benchmarks.

In Figure 8 the compilation threshold is at 500 invocations. Here the difference in the effects of the compilation queue and graph caching optimizations on the sub-benchmarks is most visible.

The largest gains can be seen on sub-benchmarks that have a small set of important methods (fop, luindex, pmd). In these cases it is important to quickly select the right methods for compilation.

There are benchmarks that do not improve due to our optimizations, but overall the combination of graph caching and compilation queue optimizations is always beneficial.

With an increasing compilation threshold the overall benefit from our optimizations decreases, as seen in Figures 9 and 10. Fewer methods will be compiled during the first run, and the compiler therefore has fewer opportunities to select and quickly compile the correct methods.

5.2.3 Compilation Queue Optimizations

Adjusting the order in which methods are compiled according to their importance leads to methods that have a large impact on application performance being compiled earlier. This allows for a lower compilation threshold, which would, without this reordering, pollute the compilation queue with less important methods.

5.2.4 Graph Caching

Figure 11 shows the influence of different graph cache sizes on first run performance. While the graph cache already shows an improvements for very small cache sizes, the best results are achieved at a size of 1000.

The main effect of the graph caching is that it reduces the compilation time for methods. During the first run this has the effect that the compilation queue will be processed quicker, so that important methods will be compiled earlier.

Later on, the main effect of graph caching is that it lowers the CPU load generated by the compiler, which improves performance only on benchmarks that use all available cores and only as long as methods are getting compiled.

We measured the total time spent compiling methods during all DaCapo sub-benchmarks. The total time without graph caching is 225 seconds, while the total time with graph

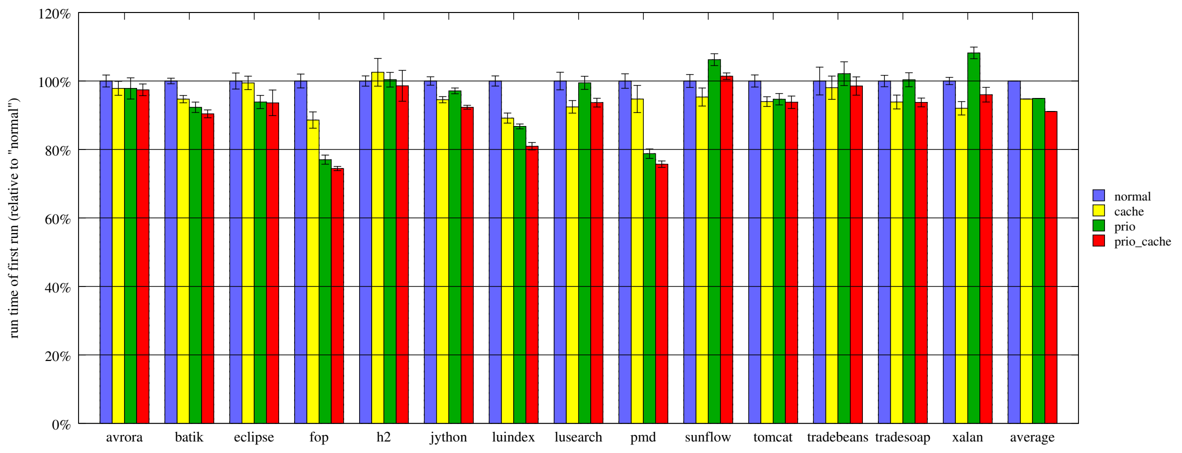

Figure 8. Benchmark results for the first run with a compilation threshold of 500 invocations.

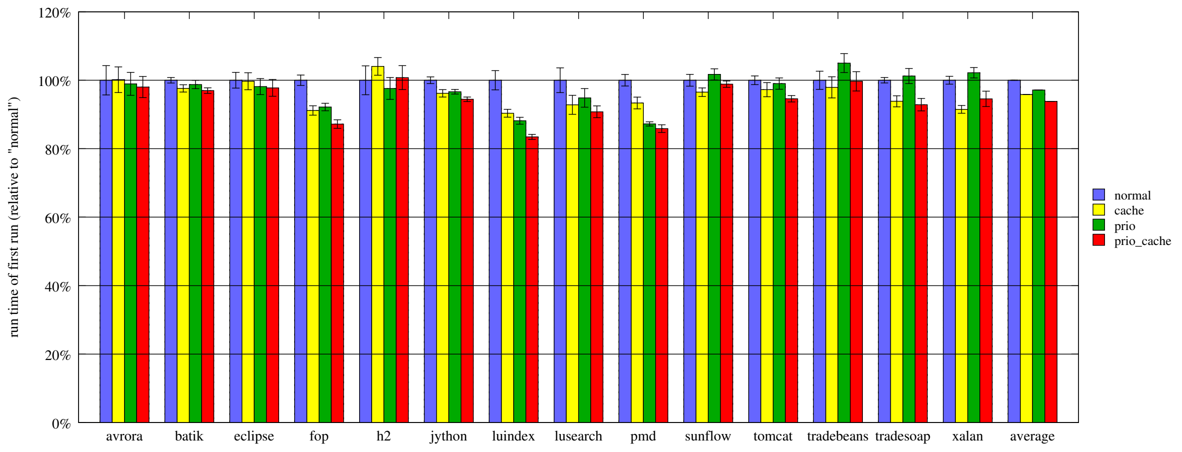

Figure 9. Benchmark results for the first run with a compilation threshold of 2,000 invocations.

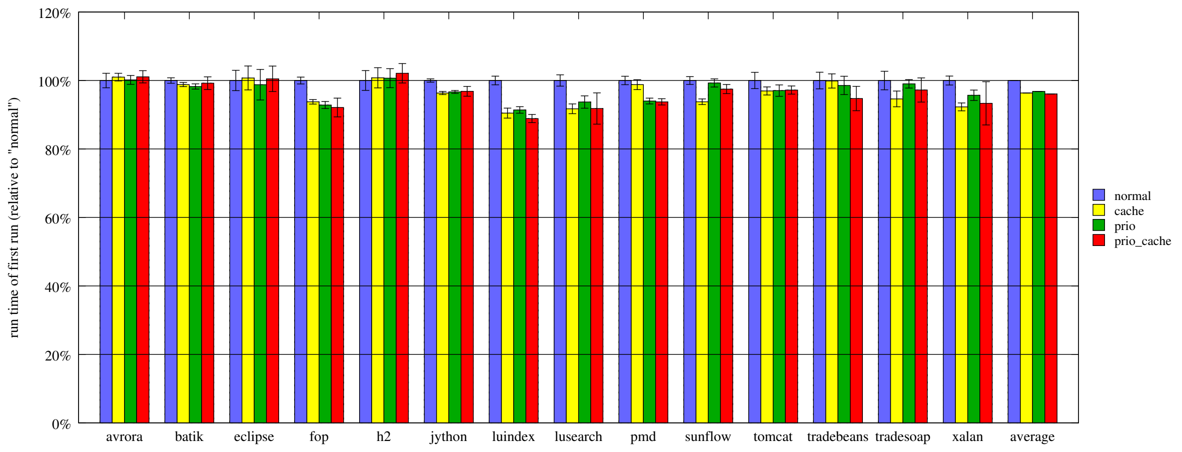

Figure 10. Benchmark results for the first run with a compilation threshold of 5,000 invocations.

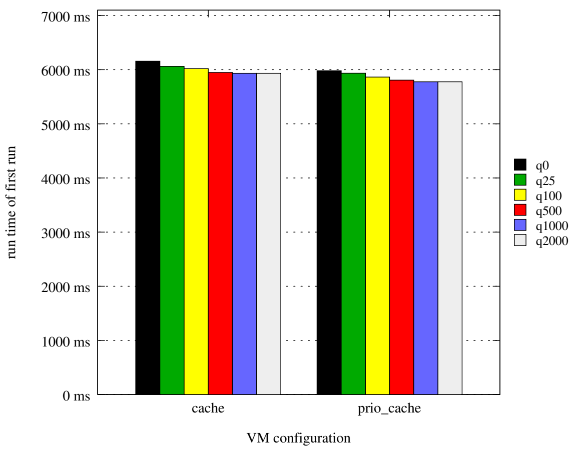

Figure 11. Geometric mean of the first run of all benchmarks, for different graph cache sizes, clustered by VM configuration.

caching is 185 seconds, which amounts to an 18% reduction in compilation time.

6. Related Work

The cost of runtime compilation and the mitigation of this cost have been studied and attacked in different ways.

It is generally assumed that there at least 2 modes of execution: a baseline mode which requires very little time to start but runs rather slowly and one or more optimized modes which take more time to start, but run faster afterwards. Such a system can be efficient because applications usually spend most of their time running only a small portion of their code. For example, SELF-93 [7] uses a fast baseline compiler and sends only a selected number of methods to a better, but slower, optimizing compiler. This technique is used in almost all JIT systems when both performance and interactivity are required.

Such systems can be further improved by running the compiler on a separate thread, in order to avoid pauses in the application code an thus make the startup phase of the application even less noticeable. This background compilation has been proposed by Plezbert and Cytron [15], is used in the HotSpot JVM [14] and has also been studied in the Jalapeño JVM [9]. In the case of background compilation there is often more than one compilation thread, in order to drain the compilation queue faster. This is especially beneficial in a multi-processor environment where it is more likely that the compilation really happens in parallel to the application's execution.

The most common way of selecting methods for compilation is by looking at how much they are being used by the application at runtime. However, doing so means that when these method are compiled they have already been running a lot in a slow mode. Various systems have been proposed to predict which methods will be beneficial to compile before they become hot and thus further reduce the cost of startup. For example, Campanoni et al. [4] inspect the code of the methods and use analysis such as static branch prediction to chose the method which are the most likely to be needed. Kulkarni [10] also proposes to use profiling information from previous runs of the same application to be able to immediately start to compile the methods which were hot in previous runs.

Kulkarni has also studied policies for selecting the methods which should be optimized and the effect of these policies in single- and multi-processor environments. These policies include different compilation thresholds and immediate compilation of all methods that were compiled in previous runs. He also assesses the effect of the number of background compilation threads for his different policies. Then he studies the ordering of the compilation queue with two strategies : first by using execution counts from a previous run in order to compile the hottest methods first, and then by using a heuristic to determine how quickly the method became hot prior to being queued for compilation. The study shows that ordering the compilation queue helps to counter the adverse effects of a low compilation threshold and this can be very beneficial in a multi-processor environment with multiple compilation threads.

In the context of trace compilation and binary translation, Böhm et al. [3] have also studied using a priority compilation queue and its impact on overall runtime, especially when using a low compilation threshold. Their approach orders compilation of traces based on their recency and frequency. Since the compilation threshold directly affects the amount of time that is spent compiling, they use an adaptive threshold based on the current length of the compilation queue. Their implementation also uses multiple compiler threads. The result of their study shows that ordering of compilations has a direct impact on the overall runtime of applications.

Further stressing the importance of these studies, Nagpurkar et al. [11] has shown that bursts of compilations do not only happen at startup but also when the application enters a new phase of its execution.

On the subject of graph caching, we could not find any related work in terms of goal or scope.

7. Future Work

The algorithm for measuring the hotness of methods presented in this paper uses a relatively simple invocation counter threshold to detect hot methods. Although this is the technique used by most VMs, it would be interesting to see how the hotness measurement interacts with other means to detect hot methods, e.g., stack sampling.

Further work in integrating the two optimizations presented in this paper could, for example, let the presence of an inline method's graph in the cache influence the decision if the method should be inlined or not.

While they are only briefly mentioned in this paper, the effects of the meta-circularity on the compilation queue system need to be studied in more detail.

Graph caching reduces the time required to compile methods, which in turn lowers the CPU load generated by the compiler given a specific set of methods to be compiled. The effects of this reduced CPU load will likely be very important in systems that are saturating most of their CPU cores. This should be studied in more detail.

8. Conclusions

While most compiler optimizations are targeted at the peak performance of applications, startup performance is important as well, for example in interactive applications or applications that frequently generate new code.

We have shown that compilation queuing and graph caching optimizations significantly increase the startup performance of Java applications, as measured by the first run performance of the DaCapo benchmark. Both the compilation queuing and the graph caching contribute to this improvement.

Our detailed measurements have shown that, while the effect of our improvements differ from benchmark to benchmark, they never have a negative influence on performance. This means that they can be enabled, without causing regressions on some benchmarks.

Lastly, the Graal VM proved to be an ideal vehicle for our experiments, since its compiler is written in Java, and therefore easy to modify. Only a small portion of the implementation had to be written outside of Java code. Also, the Graal VM is based on the high-performance HotSpot^{TM} VM, which makes the results of our measurements applicable to all compilers running on the HotSpot^{TM} VM.

Acknowledgments

This work was performed in a research collaboration with Oracle Labs. We would like to thank Christian Wimmer, Doug Simon and the Maxine team for their support.

References

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190. ACM Press, 2006. doi: 10.1145/1167473.1167488.

[2] B. Blanchet. Escape analysis for Java^{TM}: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003. doi: 10.1145/945885.945886.

[3] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 74–85. ACM Press, 2011. doi: 10.1145/1993498.1993508.

[4] S. Campanoni, M. Sykora, G. Agosta, and S. Crespi Reghizzi. Dynamic look ahead compilation: A technique to hide JIT compilation latencies in multicore environment. In *Proceedings of the International Conference on Compiler Construction*, pages 220–235. Springer Verlag, 2009. doi: 10.1007/978-3-642-00722-4_16.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. doi: 10.1145/115372.115320.

[6] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323. ACM Press, 1998. doi: 10.1145/277652.277748.

[7] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. In *ACM Transactions on Programming Languages and Systems*, pages 355–400. ACM Press, 1996. doi: 10.1145/233561.233562.

[8] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992. doi: 10.1145/143095.143114.

[9] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, 2001. doi: 10.1002/spe.384.

[10] P. A. Kulkarni. JIT compilation policy for modern machines. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 773–788. ACM Press, 2011. doi: 10.1145/2048066.2048126.

[11] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online Phase Detection Algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–123. IEEE Computer Society, 2006. doi: 10.1109/CGO.2006.26.

[12] OpenJDK Community. Graal project, 2012. URL http://openjdk.java.net/projects/graal/.

[13] Oracle. Oracle JRockit JVM, 2012. URL http://www.oracle.com/technetwork/middleware/jrockit/overview/.

[14] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} server compiler. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–12. USENIX, 2001.

[15] M. P. Plezbert and R. K. Cytron. Does "just in time" = "better late than never"? In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 120–131. ACM Press, 1997. doi: 10.1145/263699.263713.