Comparing Points-to Static Analysis with Runtime Recorded Profiling Data

Codruţ Stancu*† Christian Wimmer* Stefan Brunthaler† Per Larsen† Michael Franz†

*Oracle Labs †University of California, Irvine

c.stancu@uci.edu christian.wimmer@oracle.com s.brunthaler@uci.edu perl@uci.edu franz@uci.edu

Abstract

We present an empirical study that sheds new light on static analysis results precision by comparing them with runtime collected data. Our motivation is finding additional sources of information that can guide static analysis for increased application performance.

This is the first step in formulating an adaptive approach to static analysis that uses dynamic information to increase results precision of frequently executed code. The adaptive approach allows static analysis to (i) scale to real world applications (ii) identify important optimization opportunities. Our preliminary results show that runtime profiling is 10% more accurate in optimizing frequently executed virtual calls and 73% more accurate in optimizing frequently executed type checks.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Algorithms, Experimentation, Measurement

Keywords static program analysis, runtime profiling, feedback directed optimizations, performance optimization

1. Motivation

Object-oriented languages such as Java are usually just in time (JIT) compiled. The dynamic nature of object-oriented languages makes ahead of time (AOT) compilation less effective. Having access to concrete information about the application execution patterns the JIT compiler improves performance through optimization of frequently executed code. The motivation of our work is to make AOT compilation for Java *efficient*. The first challenge in reaching this goal is to statically determine the reachable methods and inferring as much as possible about applications' runtime behavior. We rely on points-to static analysis to discover the application, Java standard libraries, and third party libraries reachable code and discard the rest. The precision is however limited. Prior work on Java static analysis [16, 23, 33] shows that the analysis can quickly become intractable for real world applications. In designing a static analysis one must find the trade-off between analysis cost, execution time and memory use, and results precision that is best suited to their context.

Static analysis uses various information to statically model application behavior over all possible executions. In particular, points-to analysis uses the object allocation site correlated with various context refinements such as call stack, object recency, heap connectivity information, and enclosing type [16, 21]. However, it traditionally ignores one important source: runtime information such as method execution frequency and concrete receiver types. When targeting performance this knowledge is an essential source of optimization.

A static analysis unaware of dynamic code behavior tries to infer facts with the highest precision by allocating equal resources over the entire code space. However, it is known that programs follow the 90/10 rule, that is, 90% of the execution time is spent in only 10% of the code. From a performance perspective the 90/10 rule suggests that inferring precise information for 90% of the *cold* code only has marginal utility. To maximize application performance an analysis should focus on increasing the precision of frequently executed code. Taking into account execution frequencies the analysis can automatically find the best suited precision/cost trade-offs at a finer granularity. An adaptive analysis approach can pick different abstraction refinements and precision settings for different parts of the application.

In this study we identify the differences and similarities between static analysis and runtime profiling and discuss how they complement each other. Our system targets the Java programming language, however, the technique can be adopted by other languages as well. Figure 1 shows an overview of the proposed methodology. We begin by analyzing the code using our points-to analysis implementation for Java and extract analysis inferred facts. The analysis is purely static, does not use any dynamic information. We execute the same code in the Java HotSpot VM [26] and extract runtime profiling information collected by the interpreter. Next we compare the static analysis and runtime profiling information and interpret the results.

We extract virtual calls degree of morphism, i.e., number of resolved methods, and type check decidability, i.e., does the type check always hold or fail, from both static analysis and runtime profiling. Accurate information about virtual calls and type check receiver types is essential in optimizing object-oriented languages. Additionally we extract execution frequencies from runtime profiling. We use these metrics to inspect the analysis precision distribution over different execution frequency ranges and to project the impact of the static analysis accuracy on runtime performance, focusing the attention on frequently executed code.

The contributions of our study are as follows:

- We discuss the importance of using runtime profiling information as an additional source of information to statically model application execution when motivated by performance.

PPPJ '14, September 23–26, 2014, Cracow, Poland.

ACM 978-1-4503-2926-2/14/09. . . $15.00.
http://dx.doi.org/10.1145/2647508.2647524

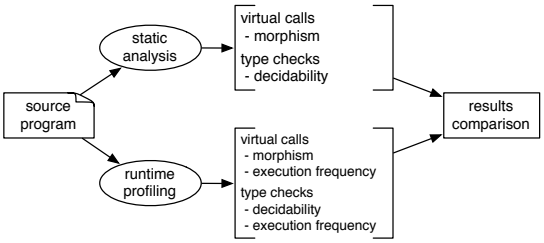

Figure 1. System overview.

- We present an empirical study that measures the precision of a points-to static analysis by comparing the results with runtime extracted profiling data.
- We find that the runtime profile is able to decide that 10% more frequently executed virtual calls are monomorphic and that 73% more frequently executed type checks are decidable when comparing to static analysis results.

2. Static Analysis

Static analysis is used to infer the runtime behavior of the code without knowledge about the input data. It abstracts the execution of the program conservatively to cover all execution scenarios. It uses abstract interpretation of the analyzed program statements operating on an abstract representation of runtime data.

We focus on object-oriented languages whose features and idioms, e.g., late function binding and encapsulation, make precise static analysis results hard to obtain [23]. The focus of our work is points-to analysis, a fundamental static analysis used by optimizing compilers to precisely determine the set of objects that a statement can access or modify at runtime. The points-to analysis is used as a basis for various optimizations such as virtual call or type check resolution. An efficient virtual call resolution must accurately infer the actual types of the receiver objects and is essential for constructing an accurate control flow graph (CFG).

The points-to analysis associates a points-to set to each reference variable and reference field. The elements of the points-to sets are abstractions of heap allocated objects. The analysis is modeled using flow transfer functions between program statements. In general, type state propagation happens at direct assignments, instance field reads and writes, and method invocations.

The two dimensions that define the design space of static analysis are *results precision* and *analysis cost*. Choosing a design point in the static analysis space is equivalent to finding the trade-off between precision and cost best suited to the context in which the analysis results are used.

Previous work has shown that an efficient points-to analysis for object-oriented languages like Java must be context sensitive, i.e. a method must be analyzed separately for each invocation context. The two main choices of context in literature are method invocation site, called *call-site sensitivity* [29, 32], and receiver object abstraction, called *object-sensitivity* [23].

From the introduction of object-sensitivity by Milanova et al. [23] it has been shown that it closely models the runtime behavior of object-oriented languages and yields a superior precision at a cost comparable to that of call-site sensitivity [6, 18–20, 24]. In addition, heap object abstractions can be modeled context sensitively by tagging object abstractions with the context of the allocator method; this is called heap sensitivity. Orthogonal to the choice of context is the context depth at which the analysis operates. That is, how many invocation levels are considered when grouping points-to sets of method variables or when abstracting heap allocated objects. A higher context depth yields more precise results but it reaches the limits of practicability quickly. Call-site sensitivity with a depth greater than 1 is typically considered impractical [23]. Object-sensitivity reaches the limit of practicality quickly too, at a depth of 2 with a heap sensitivity of 1 [33].

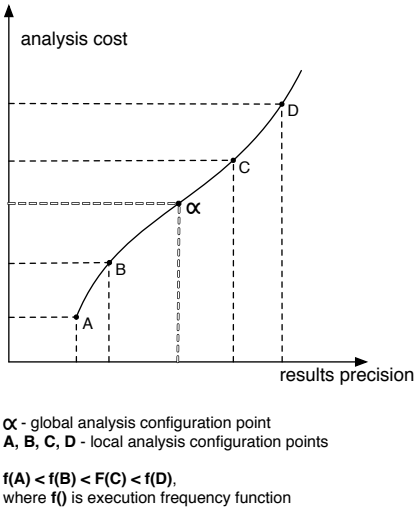

Figure 2. Points-to analysis design space.

Trading precision for cost using a fixed point in the design space for the entire application is rigid when targeting application performance. Ideally the static analysis would flexibly allocate more resources (i.e., use a more accurate object abstraction or an increased context depth) to obtain more accurate results for frequently executed code and analyze the rest of the application less precise (i.e., fall back to call-site sensitivity or disable heap sensitivity) to balance cost. To make an informed decision the static analysis requires knowledge about the runtime behavior of the code. It needs to know what the hot spots of the application are. Hence the design space for static analysis needs a third dimension, execution frequency. This knowledge can be used to automatically increase precision for frequently executed code at a cost penalty and to estimate the impact of results precision on runtime performance.

As shown in Figure 2 the execution frequency function determines local configuration points that vary from the global configuration point adaptively, trading analysis cost for results precision and vice versa. Selectively varying the context abstraction and depth throughout the program requires the use of a demand-driven analysis.

2.1 Example

Figure 4 shows an example of type check optimization based on static analysis inferred facts. The data flow graph is an abstract representation of the code in Figure 3. Method `foo` takes a parameter of type `Object` and returns an object of type `I`. Let's assume that `I` is an interface that is implemented by two classes `A` and `B`. The

```
1 static I foo(Object param) {
2   Object result = param;
3   if (...) {
4     result = new A();
5   }
6   return (I) result;
7 }
```

Figure 3. Code example

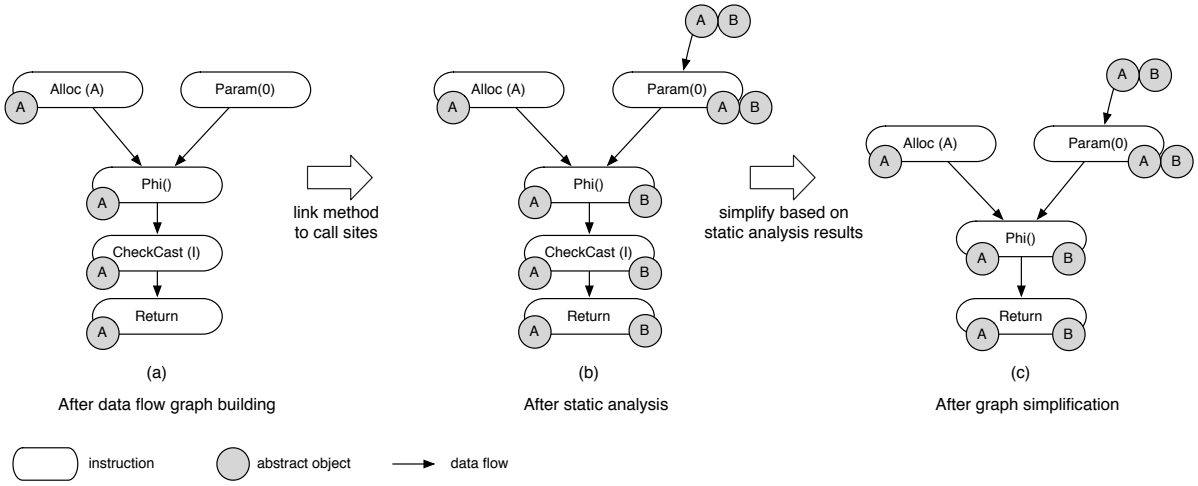

Figure 4. Static analysis data flow graph.

inputs in the data flow graph come from the object creation node (`Alloc(A)`) and from the formal parameter node (`Param(0)`). The control flow graph on which the data flow is built on is in static single assignment (SSA) form [9], which means that for every variable there is just a single point in the program where a value is assigned to it. The type sets of the two input nodes, coming from the two branches of a conditional statement, are merged in the `Phi()` node. In SSA form `Phi()` functions are placed at control-flow joins to indicate that the value of variables may come from either of the flow paths and it creates a new point of definition for every variable that is modified on either of the paths.

Figure 4(a) depicts the object states after data flow graph building. The state of the node which allocates `A` contains an abstract object of type `A`. The state of the allocation node is propagated to the `Phi()` node, it passes through the `CheckCast` statement since the two types are compatible and flows into the return type of `foo`.

When `foo` gets linked to a call site, Figure 4(b), the object state of the formal parameter node `Param(0)` gets updated to the object state of the actual parameter, abstract objects of type `A` and `B` in this case. This leads to a chain of object state updates. The `Phi()` node state is a union of its previous state and the newly incoming state. Since the `Phi()` node state is updated it propagates the new state further through the `CheckCast` to the `Return` node. Assuming that no abstract objects of other types flow into the `Param(0)` the static analysis infers that the `CheckCast` node can be removed and that `foo` can return only types `A` and `B`, as shown in Figure 4(c).

A cheap but imprecise class hierarchy analysis based on the statically declared types would have failed to detect that the `CheckCast` can be removed and would have inferred that `foo` can return any `Object`. A more precise object-sensitive analysis could as well fail to optimize this code if objects incompatible with type `I` flow in `Param(0)` due to limitations in the maximum context depth. If `foo` is a frequently executed method the analysis should allocate best effort (e.g., use a more precise context abstraction and a higher context depth) to optimize it. Failing to disambiguate the types that flow into `Param(0)` would otherwise result in a penalty on runtime performance.

3. Runtime Profiling

Feedback-directed optimizations are commonly used together with dynamic compilation. The code is transformed at runtime using recently recorded execution profiles. The profiling information can range from method calls and backward branch execution counts to concrete types of objects.

In general the code is initially executed using an interpreter or baseline compiler that is instrumented for profiling. Once the profiling information reaches maturity, i.e., enough samples have been collected and the execution count of a call or backward branch has reached a certain threshold, the code is compiled to an optimized version. However, the profiling phase has a limited lifetime and the collected information reflects only the facts recorded up until the profile reaches maturity. Dynamic profiling can formulate optimistic assumptions about the code based on the recorded facts, but it cannot guarantee that those assumptions hold for the entire duration of the execution. Thus the compiler must insert deoptimization guards that verify the correctness of the assumptions. In the event that the assumptions fail the guards trigger deoptimization and recompilation. Deoptimization is done by halting the execution of the compiled code containing the failing assumption and resuming execution in baseline mode where the exceptional case is executed and profiling restarted. Deoptimization requires a mechanism for transferring the execution state from optimized compiled code to baseline code.

Runtime feedback is especially important for object-oriented languages. Apart from traditional compiler optimizations, efficient compilation of object-oriented languages must use optimizations that target specific features and idioms that come with object orientation. Heavy use of polymorphic class hierarchies can impact performance unless special care is taken to optimize virtual calls.

Although there are special cases of virtual calls that can be statically dispatched, i.e., calls to methods that can be unambiguously resolved such as final methods or methods of final classes, virtual calls are in general dynamically dispatched based on the concrete types of receiver objects. Thus method binding is usually deferred to runtime when the concrete types of receiver objects can be recorded. The compiler constructs a polymorphic inline cache (PIC) that dispatches the call to the right callee based on the recently recorded receiver types [14]. The compiler can decide to apply aggressive optimizations based on the optimistic assumption that no other types flow into the receiver. It could for example decide to inline the target of a frequently executed virtual call; inlining reduces function call overhead and provides opportunities for other compiler optimizations by increasing the context, i.e., size of continuous code. However, the compiler cannot guarantee that the assumption always holds so it has to insert guards that transfer the control to a runtime

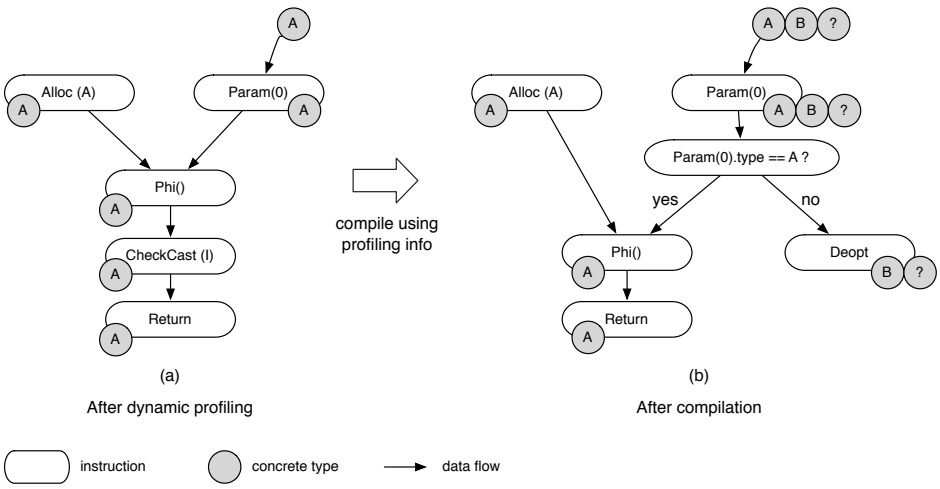

Figure 5. Runtime profiling data flow graph

routine that patches the call site and potentially leads to deoptimization. Deoptimization occurring inside inlined methods is especially complicated since the compiler has to keep track of virtual call stack frames in order to reconstruct the interpreter native call stack.

Despite the fact that runtime profiling has only a statistical accuracy, yielding a view of the application runtime behavior that is limited by the span of the interpretative execution phase and the space allocated for profile metadata, it drives efficient compilation decisions [14]. However, to find virtual and interface calls that can be statically bound and type checks that can be statically decidable, compilers use a simple class hierarchy analysis (CHA) [27]. Enhancing the dynamic compilation with a more precise analysis would reduce the necessity of using guards and would enable more aggressive optimizations. Furthermore, current dynamic compilation techniques make method inlining decisions without distinguishing the profiling meta data for different invocation sites. Using a context sensitive dynamic profiling can lead to better inlining decisions.

3.1 Example Continued

Figure 5 reexamines the example of type check optimization, this time based on runtime profiling. The runtime profile cannot infer that the `CheckCast` node can be removed by simply inspecting the types of the values that flow in. Although for the duration of the profile the `CheckCast` may never fail, the compiler must still insert a type check guard.

However, the runtime profile has access to concrete runtime values. Assume that for the duration of the profiling the type of `Param(0)` is always `A`, as in Figure 5(a). This leads to an optimization that inserts a guard checking the exact type of `Param(0)` and removes the `CheckCast` node, as shown in Figure 5(b).

The exact type check guard is cheaper than the full dynamic type check that the `CheckCast` node would have performed. If the assumption fails, i.e., the objects that flow into `Param(0)` are of type `B` or a type that was not recorded during profiling, the guard triggers deoptimization. Inferring that `foo` can only return objects of type `A` can lead to further optimizations in `foo`'s caller too. For example, if `foo` is inlined into its caller method, the compiler can propagate the more precise type information within the caller.

Although the profiling information is not more accurate than the static analysis it has access to concrete runtime objects and can drive more aggressive optimizations. However, with the additional knowledge offered by the static analysis the code could have been compiled to a more optimized version. The `CheckCast` could have been completely removed and no deoptimization guard would have been necessary.

4. Implementation Details

We implemented our entire system, including the static analysis, in Java. The core of our compilation system is the Graal compiler [25]. The static analysis phase is integrated in the compilation pipeline and it operates on the Graal internal representation [10]. At the same time we extended the Java HotSpot VM profiling capabilities to extract detailed profiling information. By comparing the profiling information with the points-to analysis results we project the impact of the analysis precision on application performance.

4.1 Ahead-of-Time Compilation for Java

At a high level, the goal of our project is to ahead-of-time compile Java code to machine code. This decision is mainly motivated by the limited resources of the target machine. Our solution is to identify methods that are reachable using static analysis, covering application code, third party libraries, and the entire Java Development Kit (JDK).

Our system does not support reflection and dynamic class loading. These dynamic features cannot be supported for two reasons. First, the resulting executable image does not contain code to load and link Java classes, therefore all the code must be available ahead of time. Second, in presence of dynamic class loading and reflection the static analysis would need to make overly conservative assumptions. These assumptions greatly increase the size of the code and defeat our goals. There are various solutions that could be used to address these limitations and increase the spectrum of the supported language features. We cover those in the related work section.

4.2 Points-to Analysis

Our implementation follows closely the rules described by Smaragdakis et al. [33], a state of the art specification to points-to static analysis for object-oriented languages. Our points-to analysis implementation is context sensitive and flow insensitive. It implements a context-sensitive heap abstraction, i.e., it distinguishes between allocation sites of an object in different contexts. It is field-sensitive, i.e., distinguishes between different fields of an object and distinguishes the fields among different objects. It is array sensitive in the sense that it distinguishes between the elements points-to sets of different array objects, however, it is array-element insensitive, it does not

consider different points to sets for each index of an array object. It is subset-based, preserving the directionality of assignments unlike equivalent-based analysis. It discovers the reachable world on-the-fly using a fixed-point approach. We define the reachable world as the call graph plus the collection of fields that are read or written.

At a macro level the analysis evolves in an iterative manner. The first iteration starts analyzing the entry method, e.g., the `main` method of the application. After the first iteration reaches a fixed-point the analysis pauses and the newly discovered classes and their constant pools are added to the analysis space. Then the analysis resumes and updates the discovered universe according to the newly discovered facts. This execution pattern evolves until the analysis reaches a global fixed-point.

At a micro level the analysis operates on a data flow graph built on top of the Graal intermediate representation (IR) [10]. Each IR node has an associated abstract object state and two data dependent nodes are connected through an object flow edge. The abstract object state propagation is asynchronous and is implemented using synchronization free Java tasks that can be scheduled in parallel on all available hardware threads. The abstract objects are organized in unique type sets, thus redundant object state propagation is easily avoided.

We implement context sensitivity using function cloning, i.e., analyze each function separately for each different context. We only use the abstract value of the receiver object as context, and not the value of all parameters. As stated by Milanova et al. [23] this is a good enough approximation for object-oriented languages. Creating a new clone for every distinct receiver object abstraction avoids the redundancy of creating a clone for each distinct call path.

The output of the analysis is summarized in data structures similar to Graal profiling summaries for later use in subsequent compilation stages.

4.3 Choice Of Input: Graal IR vs Raw Java Bytecode

Our static analysis implementation takes as input the internal representation generated by the Graal compiler. Using the Graal IR instead of the raw Java bytecode has several advantages.

First, Graal discovers early the trivially statically bindable virtual calls, i.e., calls to final methods or to methods declared in final classes. Additionally it can optimize type checks. For example it removes type checks that always hold or always fail based on a simple inspection of receiver object declared type and the condition type. It can also fold two sequential checkcast instructions into one that checks for the more specific type if the first one has a less precise and compatible type. In the evaluation section we report the trivially statically bindable calls and the removed type checks separately and not as facts discovered by the analysis.

Second, Graal evaluates the constant objects when it parses the bytecodes and presents those as concrete Java objects to the analysis. Hence, the Graal IR gives a more concrete and easier to use representation of the analyzed code than the bytecode. The bytecode preprocessing step reduces the code complexity and enables the static analysis to save some iterations.

4.4 Profiling Information in the Java HotSpot VM

In order to speculate on the runtime importance of the statically inferred facts we compare those with profiling information collected during the application execution. For this purpose we exploit HotSpot VM profiling infrastructure.

The profiling information is collected by the interpreter. It includes virtual call site receiver types and resolved methods, as well as dynamic type checks (checkcast, instanceof, aastore) receiver types. The aastore bytecode stores an object reference into an array of objects verifying that the runtime type of the value is assignment-compatible with the type of the array.

HotSpot VM uses a template based interpreter to execute bytecodes and analyzes the code as it runs to detect the critical hot spots in the program [17, 27]. In addition the interpreter uses assembly level instrumentation to collect profile information: counts at method entry and backward branches, type profiles at call sites, never null object pointers for instanceof or checkcast bytecodes and branch frequencies.

When a method activation counter reaches a threshold, the method is compiled by the dynamic compiler using the profile information. The profile information is used for aggressive optimizations. It can drive dynamic call devirtualization, inlining decisions, checkcast elimination. When the VM shuts down, we dump the collected profiling information for all executed methods.

5. Evaluation

We implemented our whole-program static analysis in Java and extracted profiling metadata from the HotSpot VM. By comparing the two sets of results we want to answer the question: how much does increasing the analysis accuracy matter in the real world? Or, in other words, what would be the impact of increased static analysis accuracy on application performance?

5.1 Experimental Setup

We carefully configure the HotSpot VM execution to increase profiling information accuracy. We disabled tiered compilation and enabled interpreter profiling, options `-XX:-TieredCompilation` and `-XX:+ProfileInterpreter`, so that profiling happens only in the interpreter, without profile update in the client compiler. We increased the compilation threshold to extend the interpretative execution phase (option `-XX:CompileThreshold=100000`). We also increased the values of the parameters that control the size of the collected profiles (options `-XX:TypeProfileWidth=10` and `-XX:MethodProfileWidth=10`). We run a HotSpot VM build that includes the Graal extended profiling capabilities for more detailed profiling information.

We selected a variety of Java benchmarks to show how the two approaches compare in different scenarios. We analyzed:

- a number of DaCapo-9.12 [4] benchmarks, the largest in the literature on context-sensitive points-to analysis: avrora, luindex, lusearch;
- a number of SPECjvm2008 [30] benchmarks, medium sized computational kernels: compress, mpegaudio, scimark;
- a JavaScript engine implemented in Java;
- the Jolden [7] benchmarks, a port to Java of the pointer intensive Olden benchmarks for C [8];

We analyzed all benchmarks together with the standard Java library and the third party libraries they depend on.

The JavaScript engine benchmark is executed in two contexts. The first scenario is a JavaScript shell, i.e., the engine is built to load and run any script as an input. For the runtime profile extraction we execute it using the delta-blue benchmark. In the second scenario the engine is built as a test framework that executes the ECMAScript test262 [11] JavaScript conformance tests. The test framework is built in Java, hence the analyzed and executed code includes the engine code plus the test framework code in the second scenario.

Since most of the selected dacapo and spec benchmarks or their respective suite harnesses use reflection we patched the code to avoid reflection.

5.2 Methodology

We begin by analyzing the application and extracting facts about the analyzed code. The facts that we care about are virtual call

reachable methods #	dacapo			spec			js engine		jolden	
	avrora	luindex	lusearch	compress	mpegaudio	scimark sor	deltablue	js262	bh	voronoi
static analysis	2286	1126	1143	33	16	25	16807	15772	41	18
dynamic profiling	596	534	316	17	10	12	1626	4061	20	10
static/dynamic	3.8x	2.1x	3.6x	1.9x	1.6x	2.1x	10.3x	3.9x	2.1x	1.8x

Table 1. Reachable methods

sites receiver types and resolved methods, and dynamic type checks (checkcast, instanceof, aastore) receiver types.

We then run the application in HotSpot and before the VM shutdown we iterate over the discovered call graph and extract profiling information for the executed bytecodes.

We export the static analysis results and the runtime profiling information in files with similar structures. Then we parse the two resulting files for each benchmarked application and compare the results. In this process we not only collect the number summaries, but also detailed information about the type sets, resolved methods, etc. The detailed information can be used by a developer to inspect the inferred facts and interpret the results in detail.

The focus of this study is to analyze the results of the static analysis in the context of hot spots discovered by runtime profiling. To be on par with the runtime profiling results collected by HotSpot VM, which are not context sensitive, we present the results of the context insensitive analysis. Increasing the context depth would improve analysis accuracy as shown in previous work [33], but the comparison would be less precise.

To reduce the noise introduced by VM calls into the runtime we do not include results for the executed JDK classes. It is not possible to isolate the effect of VM internal calls into the runtime when extracting the profiles. Eliminating the effect of the noise without completely isolating the JDK classes would be possible if runtime profiling would be context sensitive.

For both the analysis and the runtime profile we filter the results based on the application and libraries package names:

- dacapo:avrora - avrora, org.dacapo, cck
- dacapo:luindex - luindex, org.dacapo, org.apache.lucene
- dacapo:lusearch - lusearch, org.dacapo, org.apache.lucene
- spec - spec
- jolden - jolden
- jsengine - com.engine.js

We present two sets of results. We first compare the size of the static and dynamic discovered call graphs. Then we discuss the projected runtime performance impact of the static analysis results, virtual calls and type checks.

5.3 Reachable Methods

Table 1 compares the number of static analysis reachable methods with the number of runtime profile reachable methods.

Because both the static analysis and the runtime profiling are context insensitive the statically discovered world is a super set of the runtime discovered world. This fact is hinted in the table by the size of the discovered world, but our system is also able to detect any executed methods that the static analysis would fail to discover.

The last line of the table shows the relative difference between the number of statically and dynamically discovered methods. It is as low as 1.6x for spec:mpegaudio benchmark and as high as 10.3x for jsengine:deltablue benchmark. The reason for the big difference in the jsengine:deltablue benchmark is that the static analysis has to be conservative and discover the entire possible reachable world, while the runtime profile is extracted from the execution of a single script which only uses a limited subset of the JavaScript functionality. The difference between the static and dynamic discovered worlds for the jsengine:js262 benchmark is less, 3.9x, because the execution of the js262 conformance tests covers more language features. The same observation applies to the other benchmarks too, the size of the dynamically discovered world is a function of the input data path coverage.

5.4 Detailed Results

In this section we only show virtual call and type check data for the methods that were discovered in both static analysis and runtime profiling. Being flow insensitive, the static analysis reports data for bytecodes that were never executed. We include those for completeness. However, the static analysis is able to detect dead code based on the precomputed values of constants or type checks, hence we excluded the respective bytecodes from both the static and runtime profiles.

The numbers are detailed in Table 2. The left side presents the virtual calls number summaries while the right side presents the type checks number summaries. The data for each benchmarks is grouped in four execution frequency ranges: never executed, executed less than 100 times, executed more than 100 and less than 9,000 times, and executed more than 9,000 times.

We summarize the important data in Figure 6: frequently executed virtual calls, Figure 7: virtual calls that are reached by the static analysis but never executed, and Figure 8: frequently executed type checks.

5.4.1 Virtual Calls

We correlate information from the static analysis: number of reachable methods at a given call site, with runtime profile information: number of reachable methods and profiled instruction execution count. We classify the virtual call sites by the degree of morphism. We include numbers for for the invokevirtual and invokeinterface bytecodes.

The first column of Table 2, tbind, in both the static and dynamic profiles represents the trivially bindable call sites. These are the call sites that can be devirtualized by a simple class hierarchy inspection, i.e., the resolved method is final or the class that declares it is final. The trivially statically bindable calls are the result of the preceding class hierarchy analysis carried by the Graal compiler, thus we do not count them as points-to analysis results. HotSpot VM profiling also identifies the trivially bindable methods and does not collect detailed receiver type and resolved method information.

The next columns, 1-m, 2-m, 3-m, and p-m classify the virtual call data in monomorphic, bimorphic, trimorphic, and respectively polymorphic virtual call sites. The differentiation between bimorphic, trimorphic, and polymorphic is important: although only the monomorphic call sites can be actually devirtualized, the bimorphic and trimorphic call sites enable more aggressive optimizations, e.g., polymorphic method inlining.

Looking at the summary of frequently executed calls, i.e., more than 9,000 execution counts, Figure 6, it is interesting to observe that most of the profiles are dominated by trivially bindable and monomorphic virtual calls. The preprocessing step carried by Graal

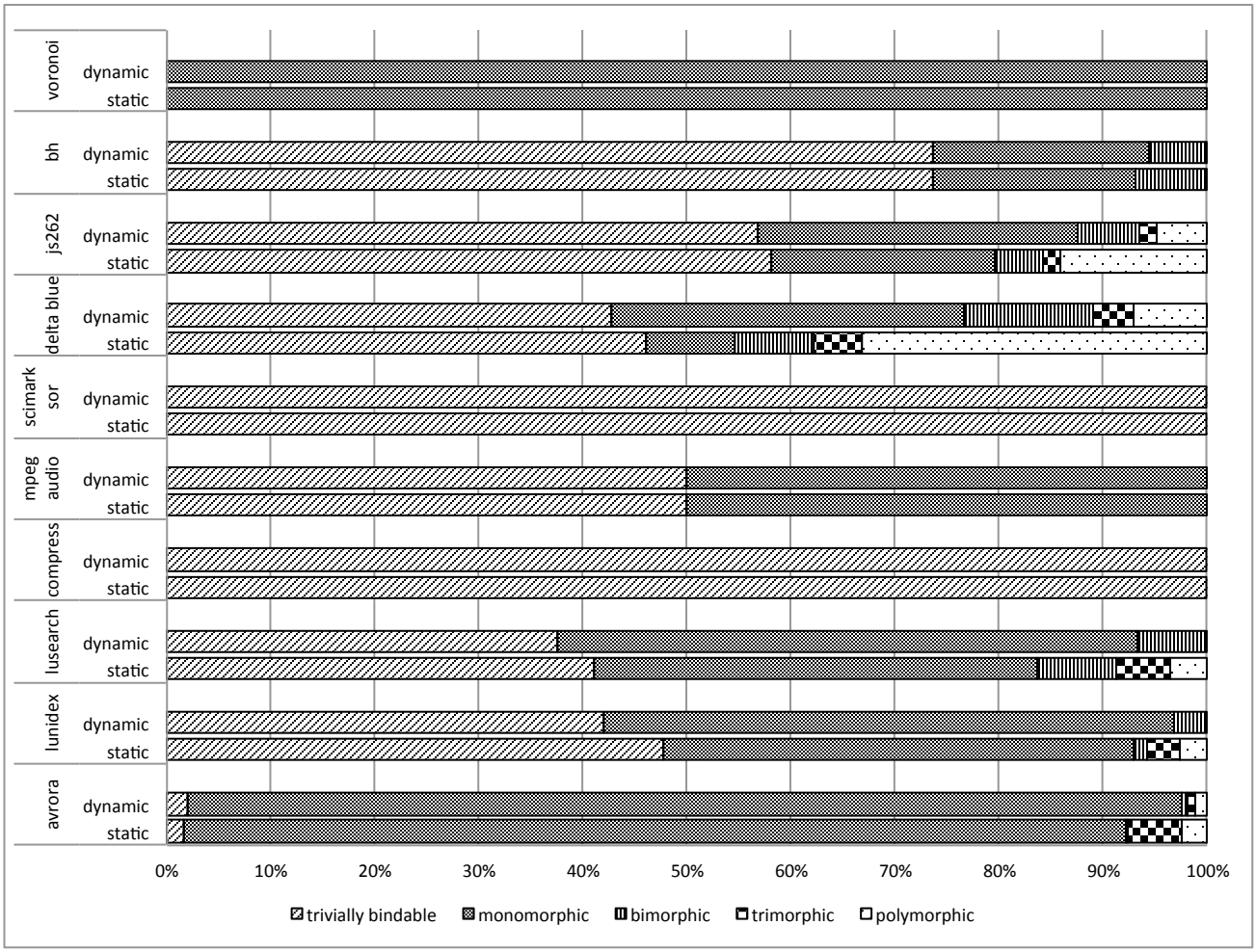

Figure 6. Frequently executed virtual calls, more than 9,000 times

is able to discover many of the optimizable calls relevant to runtime performance without the help of the points-to analysis.

Looking at the dacapo benchmarks we observe that the static analysis is able to discover a high percentage of the virtual call sites as monomorphic, very close to the accuracy of the runtime profile. Correlating this with the fact that the majority of the virtual calls are executed less than 9,000 times, for this set of benchmarks increasing the accuracy of the static analysis, i.e., inferring that more call sites are monomorphic, would have a marginal impact on the performance of the compiled code. It is also interesting to note that the number of trivially devirtualizable call sites is an important fraction of the total number of virtual calls for the luindex and lusearch benchmarks. Therefore we find that the preprocessing step of identifying trivially devirtualizable call sites is essential in reducing the complexity of the analysis.

Next we inspect a set of medium sized computational kernels extracted from the spec suite. The common characteristic of these benchmarks is that they are small but computationally intensive. Their use of object-oriented features is minimal and as a consequence they do not exhibit a high number of virtual calls. Hence the static analysis discovered facts about the virtual calls are as accurate as those discovered by runtime profiling.

Our JavaScript engine is an interesting benchmark for the study of pointer analysis implementations. It represents the programs internally as an abstract syntax tree (AST) and it relies heavily on the use of virtual call dispatch to select the correct implementation of an operation. Interestingly the majority of the virtual calls are trivially devirtualizable for both engine configurations, delta-blue and js262. However, given the complexity of the analyzed class hierarchy, the context insensitive analysis does a poor job at inferring that a number of frequently executed virtual calls are monomorphic, almost as much as 4 times virtual calls are counted by the static analysis as being polymorphic compared to the runtime profile. Yet the accuracy in discovering bimorphic and trimorphic call sites is comparable to that of the runtime profile. Enabling context sensitivity for this set of benchmarks would improve the accuracy of the results.

The next set of benchmarks, jolden, are a Java port of the C language pointer intensive Olden benchmarks. As in the case of the spec benchmarks the static analysis is able to create an accurate image of the runtime profile. This is due to a limited amount of polymorphism. Looking at the voronoi benchmark we note that the majority of the virtual calls are frequently executed, the consequence of virtual calls in long running kernels.

Previous work on static analysis classifies virtual calls in monomorphic and polymorphic without distinguishing between trivially devirtualizable calls. We believe that distinguishing trivially devirtualizable calls is important to give a more accurate understanding of the real impact of static analysis.

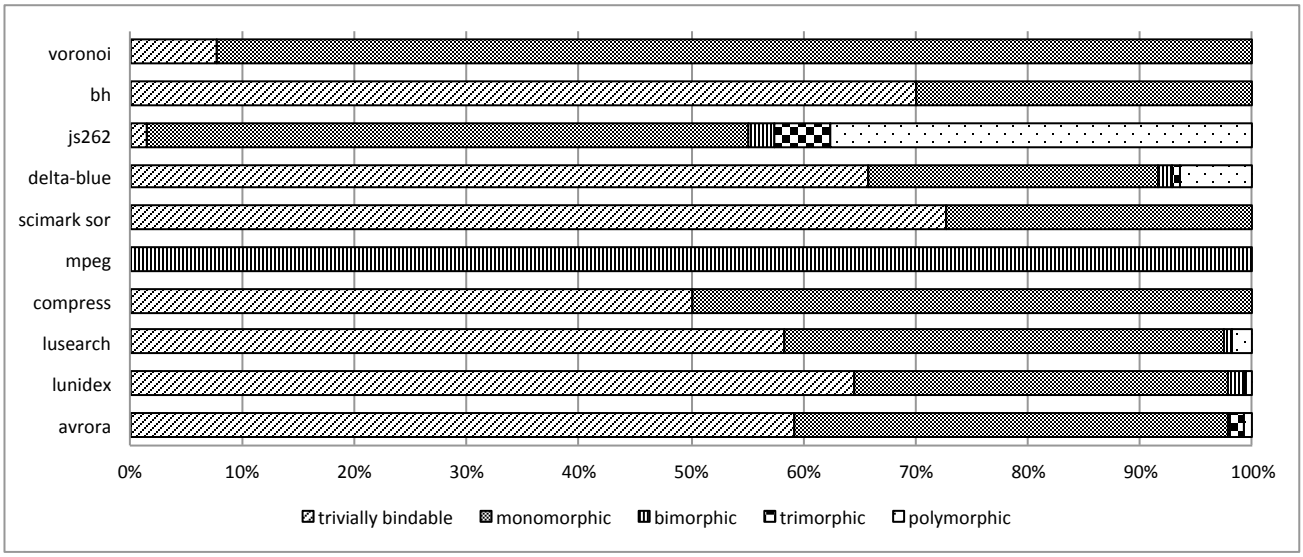

Figure 7. Never executed virtual calls

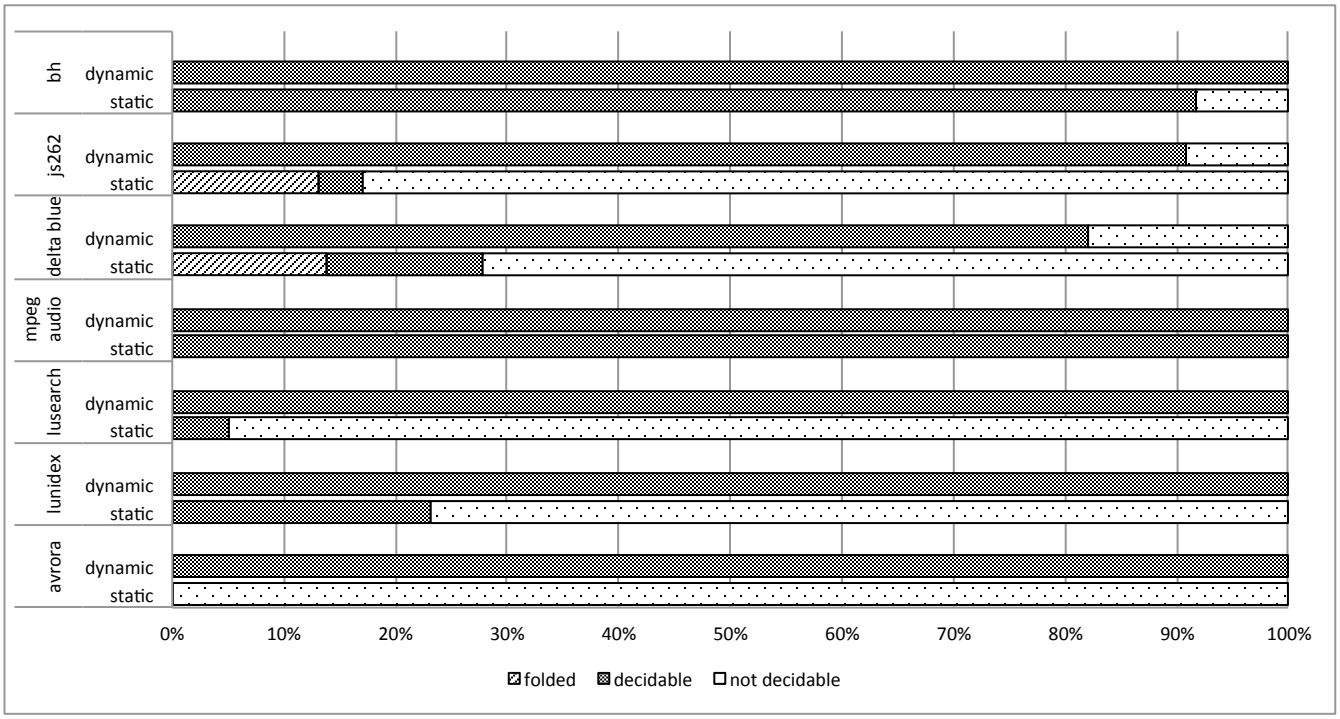

Figure 8. Frequently executed type checks, more than 9,000 times

5.4.2 Type Checks

The right half of Table 2 presents the numbers found for type checks, checkcast and instanceof bytecodes. The facts inferred by the static analysis, type sets recorded at a given type check, are correlated with runtime profile information, type check instructions execution count and receiver types. We divide the type checks in decidable and not decidable according to the recorded receiver types. A type check is decidable if it either passes or fails for all of the recorded types. Proving that a type check instruction is decidable enables more aggressive compiler optimizations.

Since the input of our analysis is Graal IR we present in a separate column the type checks that are removed by the canonicalization preprocessing step. The canonicalization step can decide to remove type checks using a simple inspection of the type check receiver object declared type and of the condition type. For example for an instanceof if the receiver object declared type is a subtype of the condition type and the receiver object cannot be null then the type check can simply be removed. For checkcast the same rule applies with the amendment that the receiver object can be null.

			virtual calls										type checks				
			static					dynamic					static			dynamic	
		exec #	tbind	1-m	2-m	3-m	p-m	tbind	1-m	2-m	3-m	p-m	folded	decid	!decid	decid	!decid
dacapo	avrora	= 0	366	239	1	8	5	-	-	-	-	-	0	4	8	-	-
		<100	162	595	10	19	1	153	632	2	0	0	1	9	38	48	0
		<9k	30	165	6	2	4	30	170	1	0	6	0	2	11	13	0
		>9k	6	286	1	16	8	6	302	2	2	4	0	0	8	8	0
	luindex	= 0	818	423	19	3	7	-	-	-	-	-	0	10	18	-	-
		<100	625	690	40	15	8	570	800	8	0	0	1	28	78	107	0
		<9k	95	99	22	2	1	71	135	12	1	0	0	1	9	10	0
		>9k	74	70	2	5	4	65	85	5	0	0	0	3	10	13	0
	lusearch	= 0	414	279	5	0	13	-	-	-	-	-	0	13	22	-	-
		<100	169	257	13	2	1	169	273	0	0	0	1	6	11	18	0
		<9k	63	79	8	2	1	62	87	4	0	0	0	1	9	10	0
		>9k	103	107	19	13	9	94	140	17	0	0	0	1	19	20	0
spec	compress	= 0	3	3	0	0	0	-	-	-	-	-	0	0	0	-	-
		<100	30	20	0	0	0	30	20	0	0	0	0	0	1	1	0
		<9k	2	0	0	0	0	2	0	0	0	0	0	0	0	0	0
		>9k	19	0	0	0	0	19	0	0	0	0	0	0	0	0	0
	mpeg audio	= 0	0	0	2	0	0	-	-	-	-	-	0	0	0	-	-
		<100	13	9	0	0	0	13	9	0	0	0	0	0	1	1	0
		<9k	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		>9k	2	2	0	0	0	2	2	0	0	0	0	1	0	1	0
	scimark sor	= 0	8	3	0	0	0	-	-	-	-	-	0	0	0	-	-
		<100	11	11	0	0	0	11	11	0	0	0	0	0	2	2	0
		<9k	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		>9k	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
js engine	delta blue	= 0	1663	650	34	15	165	-	-	-	-	-	70	46	668	-	-
		<100	1499	503	85	31	216	1488	782	42	8	14	259	59	1573	1843	48
		<9k	286	132	58	13	35	284	181	30	0	29	12	30	125	96	71
		>9k	223	41	37	22	161	207	164	60	19	34	10	10	52	59	13
	js262	= 0	3019	673	28	64	472	-	-	-	-	-	71	30	773	-	-
		<100	3249	1191	125	61	749	3195	1870	197	63	50	60	100	771	870	61
		<9k	1614	598	100	48	479	1575	951	116	48	149	35	78	413	407	119
		>9k	1533	565	124	44	371	1499	810	156	45	127	253	76	1597	1747	179
jolden	bh	= 0	7	3	0	0	0	-	-	-	-	-	0	1	0	-	-
		<100	29	7	1	0	0	29	8	0	0	0	0	0	0	0	0
		<9k	8	2	1	0	0	8	3	0	0	0	0	1	0	1	0
		>9k	53	14	5	0	0	53	15	4	0	0	0	11	1	12	0
	voronoi	= 0	1	12	0	0	0	-	-	-	-	-	0	0	0	-	-
		<100	16	10	0	0	0	16	10	0	0	0	0	0	0	0	0
		<9k	0	13	0	0	0	0	13	0	0	0	0	0	0	0	0
		>9k	0	130	0	0	0	0	130	0	0	0	0	0	0	0	0

Abbreviations	
tbind	trivially devirtualizable virtual calls
1-m	monomorphic virtual calls
2-m	bimorphic virtual calls
3-m	trimorphic virtual calls
p-m	polymorphic virtual calls
folded	type checks removed by the preprocessing step
decid	type checks that are decidable
!decid	type checks that are not decidable

Table 2. Static vs. Dynamic results

Figure 8 presents the distribution according to decidability of the frequently executed type checks. Some benchmarks do not have any frequently executed type checks, hence they are removed from the graph. Only for the jsengine benchmarks a significant number of frequently executed type checks are eliminated by the class hierarchy analysis carried by Graal.

Overall we observe that the dynamic profile is more accurate than the static analysis with respect to type checks. The number of type check instructions that are decidable at the end of the profiling, and that enables JIT optimizations, is higher than the number of type checks that are guaranteed as removable by the static analysis.

For the selected dacapo benchmarks most of the type checks have a low execution count and the preprocessing step only removes a single type check. The spec benchmarks have few type checks too.

For the jsengine benchmarks the preprocessing removes a good number of type checks, many of them with a high execution count. However, the discrepancy between static and dynamic profile is even more pronounced. The static analysis infers that only a small number of type checks can be removed in comparison to the number of type checks that the runtime profile finds decidable. The projected impact of this inaccuracy is significant since most of the checkcast have a high execution count, above 9,000.

Previous work classifies type checks in may or may not fail. Type check static decidability is a super set of the type checks that may not

fail, including those that always fail, and enables JIT compilers to do more aggressive optimizations. Previous work does not mention easily removable type checks.

6. Related Work

6.1 Static Analysis

Our exploration of static points-to analysis is orthogonal to most of the previous work on this topic and complements previous results.

Prior work by Milanova et al. [23] has shown that an accurate points-to analysis for object-oriented languages such as Java must be context sensitive, i.e., a method must be analyzed separately for each calling context. A context insensitive analysis which considers a single copy for every method for all possible invocations is imprecise. The imprecision comes from object-oriented languages features and programming idioms such as encapsulation, inheritance and use of complex data structures (i.e., collections and maps).

The choice of context abstraction is also crucial for an accurate analysis. The trivial choice for context abstraction is invocation site, or chain of invocation sites. However, given that instance methods work on encapsulated data using the implicit parameter `this` to read or write to instance fields a good context abstraction is an abstract representation of the receiver object [23]. A points-to static analysis that does not distinguish the different object structures accessed through the receiver object effectively merges the states of different objects and any access is reflected in all other objects across the same class. Empirical results by Smaragdakis et al. [33] show that in practice object-sensitivity has more impact on precision than call-site sensitivity. Since static methods do not have a receiver object, a hybrid approach where static method's context is based on chains of call sites was proposed and shows good precision at a low cost [16].

To further increase precision the depth of the calling contexts can be increased too. However, in practice a context sensitivity of 2 is the limit for an object-sensitive analysis [33] while a call site sensitive analysis of depth greater than 1 reaches the limits of practicability [23].

The quality of an object-sensitivity analysis is largely determined by its heap abstraction. Previous work by Liang et al. [21] has explored various static heap abstractions trying to find the design point that maximizes analysis precision. The base for all heap abstraction refinements is object allocation site to which additional bits of information are added.

Call stack heap abstraction adds the chain of the k most recent call sites on the stack of the thread creating the object. This is known as k-CFA with *heap cloning* [31]. Further refinements of *call stack* heap abstraction use the abstraction of the allocator method receiver object instead of the call site itself [23]. Another refinement of heap abstraction is *object recency*; it adds the recency index of an object to the allocation site to distinguish the last r objects created at an allocation site [3]. Heap abstraction based on *heap connectivity* tries to distinguish objects by their connectivity properties in the heap. It tries to associate objects with other objects reachable through the heap graph [28]. The impact of the various refinements of heap allocation sites on analysis precision is however dependent on the client that uses the analysis results. There is no single abstraction that performs efficiently for all clients.

Type-sensitive points-to analysis [33] trades precision for analysis cost to obtain better scalability by using coarser approximations of objects as context. It makes an unconventional use of types as context: the context types are not dynamic types of objects involved in the analysis, but instead upper bounds on the dynamic types of their allocator objects.

Increased precision is not always possible due to cost restrictions or due to application complexity that can lead to analysis explosion. To address scalability issues of points-to analysis Milanova et al. [23] proposed a coarse grain parametrization technique: i.e., different context depth for call and heap sensitivity, or selecting a subset of reference variables that should be analyzed context sensitively (e.g., implicit parameter `this` and return variables). The authors also propose using different context depths to optimize frequently used patterns. For example allocation sites in container classes (e.g., the array of hash entries in HashTable) could be analyzed with increased context depth for more precise results to avoid *sharing* of objects stored in different containers. Our work proposes a more flexible, fine grained analysis sweet spot selection based on the runtime feedback.

Smaragdakis et al. [34] addressed the analysis cost issue by introducing a set-based preprocessing step that puts the program in a normal form optimized for points-to analysis. The preprocessing step computes constraints at the points-to set level that result in a reduced analysis space by removing local variables and instructions

Java programs are distributed in a Java bytecode format. However, optimizing the Java bytecode directly is difficult due to the fact that bytecode instructions operate directly on an operand stack, and thus have implicit uses and definitions of stack locations. A representation in which a statement refers explicitly to the variables it uses is better suited for static analysis and other optimizations. The Soot [36, 37] Java optimization framework uses Jimple [35], a 3-address intermediate representation that has been designed to simplify analysis and transformation of Java bytecode. Soot also supports a static single assignment (SSA) form variation of Jimple. Our static analysis operates on the Graal internal representation [10], a graph-based, SSA form IR that models both control-flow and data-flow dependencies between nodes. The input to the analysis phase is a canonicalized IR. In the canonicalization process Graal discovers and simplifies easily optimizable code patterns, e.g., trivially statically bindable virtual calls and statically decidable type checks.

Doop [6] is a widely used static analysis framework for Java. It implements a range of algorithms, including context insensitive, call-site sensitive, and object-sensitive analyses. Doop expresses the various analyses declaratively using the Datalog language. Declarative logic programming allows for a natural expression of static analysis rules. However, our analysis step is designed as a phase in the compilation pipeline and since it operates on the Graal IR we decided to implement it completely in Java. An imperative programming language presents some challenges in dealing with type set operations. Our internal data structures are highly optimized and thread safe. Plus our implementation organizes abstract objects in unique type sets, thus redundant object state propagation is easily avoided.

6.1.1 Dynamic Features

At this stage of our project we do not support Java dynamic class loading and reflection. Our static analysis implementation operates under a closed world assumption: the code that is visible at compile time must be a superset of the executed code. Dynamic loading may load code that is not available for analysis before the program starts. However, there are various approaches that could be used to overcome these limitations and increase the range of supported language features.

Bodden et al. [5] have extended the scope of the static analysis to dynamically loaded classes by running the application before compilation and recording the dynamically loaded classes. The discovered classes are included in the analyzed code and the analysis operates under the assumption that no other classes can be loaded at runtime. Livshits et al. [22] proposed a static reflection resolution algorithm, which approximates the target of reflective calls as part of call graph construction, complemented by user provided specifications for reflective calls that rely on application input. Hirzel et al. [12, 13] presented an online version of Andersen's points-to

analysis [2] that executes alongside the program, as an extension to the Jikes RVM [1], an open-source Java Research Virtual Machine. Thus the analysis has access to the new code as its loaded.

6.2 Feedback Directed Optimizations

We propose using runtime feedback to tune points-to analyses for increased application performance. Feedback directed optimizations are largely used in JIT compilers.

Runtime profiling gives a concrete view of the application execution patterns as it runs enabling the compiler to apply efficient optimizations. Hölzle et al. [14] described a dynamic optimization technique that feeds back type information from the runtime system to the compiler. Using the concrete types the compiler can efficiently devirtualize and inline dynamically dispatched calls.

Feedback directed optimizations cannot however guarantee that the assumptions based on the profiled execution hold for the entire life span of the application. Hölzle et al. [15] described an efficient dynamic deoptimization technique that can be used to roll aggressively optimized code back to interpreter execution. The deoptimization only affects the procedure that needs to be deoptimized; all other code runs at full speed.

Our system exploits the runtime profiles collected by the Java HotSpot virtual machine. The Java HotSpot VM improves performance through optimizing frequently executed application code. It collects runtime profiles to guide optimizations of the client (C1) and server (C2) compilers. The Client Compiler [17] is used by default for interactive desktop applications where low startup and pause times are more important than peak performance. The Server Compiler [27] is tuned for peak performance and it applies more aggressive optimizations. The two compilers share the same runtime environment and utilize the same profile recording system.

7. Conclusions

This paper emerged from our own experience with implementing a static analysis phase in our compilation pipeline and discusses how we think that the benefits of a static analysis step can be maximized. This work is a necessary first step to reach our goal of efficiently compiling Java ahead of time. We believe that switching from a whole-program analysis to a demand-driven analysis that exploits runtime profiling information is the key to achieve better runtime performance while keeping the analysis costs under control.

We presented a comparison between points-to static analysis and runtime profiling and identified the differences and similarities. We hope that this study can stand as a guidance to JIT developers that could utilize these insights for better optimizations.

Acknowledgments

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000. .

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[3] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the International Static Analysis Symposium*, pages 221–239. Springer-Verlag, 2006. .

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, 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. .

[5] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the International Conference on Software Engineering*, pages 241–250. ACM Press, 2011. .

[6] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 243–262. ACM Press, 2009. .

[7] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291. IEEE Computer Society, 2001.

[8] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38. ACM Press, 1995. .

[9] 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. ISSN 0164-0925. .

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

[11] Ecma TC39. ECMAScript test262. URL `http://test262.ecmascript.org`.

[12] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 96–122. Springer-Verlag, 2004. .

[13] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007. .

[14] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336. ACM Press, 1994. .

[15] 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. .

[16] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 423–434. ACM Press, 2013. .

[17] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1): 7:1–7:32, 2008. .

[18] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, 2006.

[19] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–53, 2008. .

[20] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 6–12. ACM Press, 2005. .

[21] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *oopsla*, pages 411–427. ACM Press, 2010. .

[22] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Proceedings of the Asian Conference on Programming Languages and Systems*, pages 139–160. Springer-Verlag, 2005. .

[23] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005. .

[24] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319. ACM Press, 2006. .

[25] OpenJDK. Graal Project, . URL `http://openjdk.java.net/projects/graal`.

[26] OpenJDK. HotSpot, . URL `http://openjdk.java.net/groups/hotspot`.

[27] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology*. USENIX Association, 2001. .

[28] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 105–118. ACM Press, 1999. .

[29] M. Sharir and M. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[30] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. Specjvm2008 performance characterization. In *Proceedings of the SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35. Springer-Verlag, 2009. .

[31] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174. ACM Press, 1988. .

[32] O. Shivers. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[33] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 17–30. ACM Press, 2011. .

[34] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis. Set-based pre-processing for points-to analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 253–270. ACM Press, 2013. .

[35] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations, 1998.

[36] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java optimization framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135. IBM Press, 1999.

[37] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction*, pages 18–34. Springer-Verlag, 2000.