Improving Trace-Based JIT Optimisation using Whole-Program Information

Maarten Vandercammen Coen De Roover
Vrije Universiteit Brussel, Brussels, Belgium
{mvdcamme, cderoove}@vub.ac.be

Abstract

Trace-based just-in-time compilers use program analyses to optimise execution traces. These analyses are limited in scope to the parts of the program that have been traced. In this position paper, we conjecture that trace optimisations can benefit from extending the scope of their enabling analyses beyond these traces to the set of possible future execution states of the program. This by incorporating a static analysis which, however, necessarily over-approximates this set. We therefore propose to continuously refine the set of future states computed by an initial, ahead-of-time analysis using run-time information about the current execution state of the program. Additional static analyses launched at run time could further refine the information about the future of the current and *all* possible states. We expect that the resulting, hybrid program view of static and dynamic information may enable additional optimisations on collected traces and that these optimisations may overcome the computational overhead of keeping the view up-to-date.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation, Optimization; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

Keywords JIT Compilation, Static Analysis, Incremental Analysis, Hybrid Analysis

1. Introduction

Traditional ahead-of-time (AOT) compilers perform optimisations such as loop invariant code motion, common subexpression elimination, loop unrolling, inlining, constant propagation, register allocation etc., that are enabled by static analyses. However, these analyses often struggle [3, 23] with features that are common in dynamic languages, such as the presence of dynamic typing, reflection and dynamic code generation [13]. Furthermore, in a dynamic language that enables redefining primitives, such as Scheme, it would be difficult to even apply constant propagation on the following snippet of code: `(define a (+ 1 2))`. As the operator `+` may have been redefined elsewhere in the program, a static analysis may only soundly conclude that `a` should equal `3` if it can first guarantee that the operator `+` is not redefined before reaching this expression. Such an analysis would become even more convoluted in the presence of dynamic code generation, e.g., via `eval` expressions.

Execution environments for dynamic languages therefore often incorporate just-in-time (JIT) compilation, as JIT compilers can observe and subsequently optimise a program's execution. This information can guide optimisations, such as type specialisation, which is in general difficult to acquire for these languages using AOT compilation [11]. To retrieve this knowledge, JIT compilers perform analyses. However, due to the limited time budget that is available to JIT compilers, these analyses are usually limited to the small part of the program that is to be compiled. They therefore lack a holistic view of the program and are thus limited to applying local optimisations on the compiled program segments. This makes it impossible to e.g., soundly determine whether the value of a variable remains constant throughout the program, even though this information would aid local optimisations on these segments. In contrast, AOT compilers can afford to spend the time required for building up a whole-program view of the program via interprocedural static analyses, enabling whole-program optimisations.

We therefore propose to combine both types of compilation into a hybrid, two-stage technique:

1. Perform an ahead-of-time static, but necessarily imprecise, analysis of the program.
2. Update the results of this analysis with information that is gathered at run time and, if necessary, rerun parts of the analysis. These updated results can be used to further improve performance of the dynamically compiled code.

We propose to incorporate this technique in trace-based JIT compilers specifically, as their compilation scope is larger than the one of the more common method-based JIT compil-

VMIL'16, October 31, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4645-0/16/10...$15.00
http://dx.doi.org/10.1145/2998415.2998418

ers [14], increasing the number of optimisation opportunities that become available thanks to the added static information.

In the following section, we give some background information on trace-based JIT compilation. In Section 3, we give a more detailed overview of our proposed technique. In Section 4, we describe an initial prototype implementation. We present related work in Section 5 and we conclude in Section 6.

2. Trace-Based JIT Compilation

Trace-based JIT compilation [11] is an alternative to the more common method-based JIT compilation. It builds on two basic assumptions: most of the execution time of a program is spent in loops, and several iterations of the same loop are likely to take the same path through the program [5]. Trace-based JIT compilers therefore do not limit themselves to the compilation of methods, like method-based JIT compilers, but they trace and compile frequently executed, "hot" loops.

Runtimes incorporating a trace-based JIT compiler usually do so through mixed-mode execution. Initially, an interpreter executes the program and profiles loops to identify hot ones. When a hot loop is detected, the runtime starts *tracing* the execution of this loop: the operations that are performed by the interpreter while in this loop are recorded into a trace. Tracing continues until the interpreter has completed a full iteration of the loop. The recorded trace is then compiled and optimised. Subsequent iterations of this loop will execute the compiled trace.

Because a trace represents a single execution path, it must ensure that the conditions that held while the trace was being *recorded* still hold when it is *executed*. These assumptions are checked by inserting *guards* encoding the corresponding conditions in each trace. When a guard *fails*, execution of the trace is aborted and the interpreter resumes normal interpretation of the program from that point onward. The point where trace execution is aborted and interpretation restarts is called a *side-exit*. Side-exits give rise to a performance penalty, because execution of the optimised trace must be aborted and evaluation must proceed through regular interpretation of the program. Most tracing compilers therefore include a mechanism, called *trace bridges*, to switch execution from one trace to another, once a guard has failed in the first trace [21].

Example Figure 1a depicts a Scheme program containing a loop. The loop's corresponding trace, consisting of pseudocode, is depicted in Figure 1b. As the expression `(= n 0)` evaluated to `false` while tracing the first iteration of the loop, the tracer inserted a guard `guard_false()` that will check whether this condition still evaluates to `false` during trace execution.

3. Hybrid Analysis

Our proposed technique is based on the hypothesis that performance of programs can be improved by extending the scope of information available to a trace-based JIT compiler beyond the recorded traces themselves. This additional,

```
(define (sqr x)
  (* x x))

(define (loop n result)
  (if (= n 0)
      result
      (loop (- n 1)
            (+ result
               (sqr n)))))

(loop 100 0)
```

(a) The program to be traced.

```
loop:
  t0 = n
  t1 = result
  int_cmp_equal(t0, 0)
  guard_false()
  t2 = int_subtr(t0, 1)
  t3 = int_mult(t0, t0)
  t4 = int_add(t1, t3)
  n = t2
  result = t4
  jump(loop)
```

(b) The corresponding trace.

Figure 1: A loop in a program and its corresponding trace.

whole-program information may then enable further local optimisation of traces. Current trace-based JIT compilers cannot perform optimisations that require such a whole-program view, or they can only perform them by executing additional run-time checks. AOT compilers for static languages manage to build up such a view through heavy-weight interprocedural analyses. However, their use for dynamic languages is problematic, due to sources of imprecision, such as dynamic typing and dynamic code generation, that are common in dynamic languages.

The lack of a global view of the program has a far-reaching impact on trace-based JIT compilation. For example, were the `loop` function in Listing 1 to be traced, a JIT compiler could not substitute the constant `1` for the variable `a` in the trace, as `a` may be re-assigned elsewhere in the program. Re-executing the trace after this re-assignment would produce incorrect values.

```
1 (let ((a 1))
2   (define (loop n)
3     (if (< n 0)
4         "done"
5         (loop (- n a))))
6   (loop 100)
7   ...)
```

Listing 1: Variable `a` at line 5 may be replaced by `1` if `a` is not changed elsewhere in the program.

Existing JIT compilers for dynamic languages either forego these value specialisation optimisations entirely, rely on run-time profiling of variables to determine which variables seemingly remain constant [20], or depend on programmer annotations to *promote* variables which are likely to remain constant [6]. The last two approaches require the execution environment to verify that these variables have indeed remained constant, as it cannot be proven that this is the case in general.

3.1 Run-Time Static Analysis

Performing a static analysis ahead-of-time for a dynamic language may solve this issue in theory, but imprecision in its results might preclude their use for optimisations in non-trivial applications. We therefore propose to run static, whole-program analyses *at run time*. By running such an analysis at run time, we can incorporate observed information about the execution of the program, which in turn should greatly improve its precision. This increase in precision might produce an accurate whole-program view, to the benefit of the optimisations employed by JIT compilers.

In the remainder of this paper, we focus on applying a constant propagation analysis at run time to find additional constant expressions in the program. However, other analyses and trace optimisations may benefit from our technique. Candidates include a type inference analysis [12] and analyses that detect which parts of the program are independent [17, 24], i.e., which program segments do not have a race condition in between them, and hence which parts of the program may be executed in parallel.

3.2 Minimising Run-Time Overhead

To realise speedups with our technique, it will be crucial to minimise the overhead of performing these analyses at run time. To this end, the analyses will necessarily have to be *incremental* [19]. This can be accomplished by running an initial, heavyweight static analysis before executing the program, as is done by AOT compilers, and letting the subsequent run-time analyses use and supplement the results from this initial analysis and subsequent run-time analyses. Instead of having to construct all analysis information from scratch, an incremental analysis can reuse the information that was computed by some previous analysis, and update it with new concrete information about the execution of the program. Incremental analyses have been studied among others by Arzt and Bodden [4] and Conway et al. [8]. Both of these projects were in the context of static analyses that update the work of previous analyses in response to changes to the source code of the program. We intend to perform analyses that are incremental with respect to updates about the observed information that becomes available about the execution of the program.

To further decrease the run-time overhead, we aim to run these analyses, when possible, in parallel to the execution of the program, on CPU cores that would otherwise remain idle. This lets us take maximal advantage of the now widespread multi- and manycore CPU architectures. Moreover, if at some point it can be determined that a program trace cannot be optimised any further, its analyses may be aborted, thereby further reducing run-time overhead. As a final point, we believe this new approach may especially be of use for long-running applications, such as on servers, where the time spent analysing, optimising and compiling the code is only a fraction of the overall time spent executing the program.

3.3 Challenges

We identify the following challenges for this technique to realise speedups:

Launch Point of the Analysis The ideal moment to launch an analysis must be determined. Continuously updating the information, such as a program state graph, computed by an earlier analysis while the program is executing will likely cause too much an overhead. Instead, the analysis can be launched upon reaching particular points of interest in the program's execution, such as the completion of a trace recording. Analyses may also be launched upon passing a barrier placed by a previous analysis, as is done in related work [12]. However, if the analysis is launched too early, later dynamic features in the program, such as `eval`, can cloud the precision of the analysis, as is the case in traditional static analyses. If the analysis is too late, compiled traces may benefit less from the enabled optimisations.

Scope of the Analysis The analyses should produce a precise whole-program view. As mentioned before, to minimise run-time overhead, the analyses must be incremental. This means that ideally, the run-time analyses would only range over those parts of the program about which run-time information has changed since the last time they were analysed. Detecting which parts of the program must be re-analysed will necessarily depend on the kinds of analyses and optimisations that the compiler will employ.

Incorporating Run-Time Values The key insight of our technique is to improve precision of static analyses by performing them at run time such that observed values can be incorporated. However, depending on the scope of the analysis, not all run-time information may be relevant. We must therefore investigate which run-time facts improve precision, and are therefore relevant, and which are not.

4. Initial Implementation

We have extended a trace-based JIT compiler for Scheme, named STRAF[1] so that it runs both an initial static analysis before executing the program and multiple full, static analyses at run time. It should be noted that these analyses are not yet incremental: they are relaunched in full, from each point in the program at which the compiler has completed recording a trace. As can be expected, the corresponding run-time overhead far exceeds any performance gains that can be achieved by additional optimisations. However, our initial implementation already enables studying which kind of optimisations now become possible, or which existing trace optimisations can be improved using whole-program information.

The STRAF trace-based compiler integrates in Scala-AM [25], a static analysis framework adhering to the abstracting abstract machines (AAM) approach [26] to abstract interpretation.

This approach enables developing one artefact, the so-called AAM, that can execute both the concrete and the abstract semantics of a program. In the first case, the AAM operates on concrete values, such as regular numbers or strings, and the AAM hence functions as a concrete interpreter for the program. In the latter case, the AAM operates on abstract values, such as the type `Integer` as an abstraction of the set of all concrete integers, and therefore functions as a static anal-

[1] Publicly available at `https://github.com/mvdcamme/scala-am`

ysis of the program. Launching a static analysis from the current point in the execution of a program can then be accomplished by converting a concrete AAM machine state to the corresponding abstract machine state and continuing execution under the abstract semantics from there.

4.1 Initial Analysis

The initial static analysis that is currently launched by the compiler aims to find variables, i.e., local variables, global variables or even function parameters, in the program that can be proven constant ahead-of-time. Note that the analysis currently employed by our prototype only considers a variable constant if its value can be predicted by the analysis; variables initialised to a random value, or whose value depends on user-input, are therefore automatically excluded.

When a trace has been recorded and is being optimised, the optimiser can consult the set of constant variables that was computed by the analysis and cross-reference it with the set of variables looked up in the trace; any variable that is looked up in the trace and which appears in the set of constant variables can soundly be replaced by just its constant value. This approach already enables us to replace the variable `a` in the trace corresponding with the `loop` function in Listing 1 with its constant value `1`.

4.2 Subsequent Run-Time Analyses

Although this initial analysis may already expose some constants in the program, it might fail on less trivial applications. Consider the program depicted in Listing 2. As the value of `a` is a random value, it cannot be known ahead-of-time by the initial analysis and the constant propagation optimisation that was applicable on the trace earlier is no longer available. As stated in Section 3, a traditional JIT compiler can observe the run-time value of `a`, but as it cannot know whether `a` will remain constant throughout the execution of the program, it must place guards in the trace to verify that the value of `a` has remained the same.

```
1  (let ((a (random 10)))
2    (define (loop n)
3      (if (< n 0)
4          "done"
5          (loop (- n a))))
6    (loop 1000)
7    ...)
```

Listing 2: The value of `a` cannot be computed ahead-of-time.

Our prototype launches a full, complete static analysis anew when it has completed the recording of a trace. This run-time analysis functions identically to the initial analysis. In the case of the application in Listing 2, suppose the compiler starts tracing the `loop` function, and that during this recording it executes the alternative branch on line 5. It will then jump to the beginning of the function on line 3 and stop recording there, as it has completed a full iteration of the loop. At this point, the compiler will launch a static analysis of the remaining parts of the program. As the value of `a` is now known, and if the analysis determines that this variable will not be changed at any point further in the program, the lookup of this variable in the trace can be replaced by its concrete value. Our prototype thus combines static, whole-program information, namely that `a` remains constant, with dynamic, run-time information, namely the concrete value of `a`.

4.3 Optimising the Analyses

In some cases, it may be sound to abort the subsequent analyses earlier, reducing the overhead of our technique. The analysis can conclude immediately that a variable does not remain constant once it is assigned. However, as long as no assignments of this variable have been detected, the analysis must continue, as it may find one further on in the program.

By reducing the set of variables that needs to be checked, the analysis may be aborted early: if an assignment has been found for each of its variables, the analysis can be aborted. As the static analysis is only used to replace lookups of variables that appear in the trace, the set of variables that must be checked can be limited to only those variables that appear in the trace. This set can be further constrained by removing all variables which have already been proven to be constant by the initial analysis.

4.4 Results

To illustrate the potential of our technique, we apply it to the program depicted in Listing 3. This program consists of a Scheme loop of which each iteration decreases the loop index by both a variable `x` and a variable `y`. The former equals `1` while the latter is a random number between `0` and `9`. Once initialised, neither `x` nor `y` is changed.

Listing 4 depicts the traces, consisting of the pseudo-bytecode employed by STRAF, of the `loop` function from Listing 3 that are produced by three different variations of the STRAF compiler. The first trace is produced by the STRAF compiler when it employs neither an initial analysis nor any run-time analyses, i.e., as is the case in current state-of-the art trace-based JIT compilers. The second trace corresponds to a setting where an initial analysis of the program is performed, but no run-time analyses are launched. The last trace corresponds to our actual prototype technique: the compiler performs an initial analysis and launches another run-time analysis, after completing the recording of a trace for `loop`, to refine the results of the initial one.

Each case represents an improvement on the previous case. Whereas in the first trace, neither `x` nor `y` were substituted for their respective constant values, in the second trace, the variable `x` has been replaced by the value `1`, as the initial analysis correctly determined that `x` remains constant throughout the program. However, this initial analysis could not predict the value of `y` ahead-of-time. In the third case, the run-time analysis employed by the compiler has observed the run-time value of `y`, and as this value never changes, the run-time analysis enables substituting its constant value for the variable lookup operation.

```
(let ((x 1)
      (y (random 10))) ; becomes 4
  (define (loop n)
    (if (< n 0)
        "done"
        (loop (- n x y))))
  (loop 1000))
```

Listing 3: The program to be traced.

```
...
LookupVariable(n)
PushVal()
LookupVariable(x)
PushVal()
LookupVariable(y)
PushVal()
PrimCall(3,-)
...
```

(a) Perform neither an initial nor a run-time analysis.

```
...
LookupVariable(n)
PushVal()
ReachedValue(1)
PushVal()
LookupVariable(y)
PushVal()
PrimCall(3,-)
...
```

(b) Perform an initial but no run-time analysis.

```
...
LookupVariable(n)
PushVal()
ReachedValue(1)
PushVal()
ReachedValue(4)
PushVal()
PrimCall(3,-)
...
```

(c) Perform both an initial and a run-time analysis.

Listing 4: Three traces corresponding to the above loop, with the initial and run-time analysis either enabled or disabled.

```
1   (let ((x (random 10))
2         (y (random 10))
3         (z (random 10)))
4     (define (loop1 n)
5       (if (< n 0)
6           "done"
7           (loop1 (- n x y z))))
8     (loop1 1000)
9     (set! y (random 10))
10    (set! z (random 10))
11    (define (loop2 n)
12      (if (< n 0)
13          "done"
14          (loop2 (- n x y z))))
15    (loop2 1000)
16    (set! z (random 10))
17    (define (loop3 n)
18      (if (< n 0)
19          "done"
20          (loop3 (- n x y z))))
21    (loop3 1000)
22    (loop1 1000)
23    (loop2 1000)
24    (loop3 1000))
```

Listing 5: Three random values and three separate loops.

Analysis nr.	Constant variables found
Initial analysis	{ }
Run-time analysis 1	{ x }
Run-time analysis 2	{ x, y }
Run-time analysis 3	{ x, y, z }

Table 1: The constant variables found by each analysis.

Listing 5 illustrates the potential of our prototype when launching multiple run-time analyses. The program consists of three variables x, y and z, and three different loops, each of which are called from two separate locations. As our prototype launches a run-time analysis after completing the recording of a trace, the compiler will launch three run-time analyses: after starting loop1 at line 8, after starting loop2 at line 15 and after starting loop3 at line 21, in addition to an initial analysis. However, as each of the three variables is initialised to a random value, the ahead-of-time analysis cannot predict any of the variables' values. Additionally, the first run-time analysis, i.e., the analysis that is launched after completing the recording of a trace for loop1, only finds the variable x constant, as the variables y and z are re-assigned at lines 9 and 10. Note that the trace for loop1 will be re-executed when reaching line 22, so if the variables y or z were to be substituted for their run-time values in the trace now, the second execution of this trace would produce incorrect values. Similar to the first one, the second run-time analysis cannot conclude that z is constant, as it is again re-assigned at line 16. Only the last run-time analysis can soundly substitute the values of all three variables in the trace of loop3, as shown in Table 1.

This example shows that in some situations it may be beneficial for our prototype to launch several run-time analyses successively. The advantage will become even greater once the analyses become incremental and can reuse each others' results.

4.5 Discussion

Our prototype represents a first step towards the realisation of our proposed technique: it already extends the scope of available information for the trace optimiser with a static, whole-program view. This enables experimenting with newly enabled optimisations, and with improvements to existing ones.

Much work remains to be done however. The amount of time spent in the run-time static analysis currently outweighs any performance improvements that can be made by possible new optimisations, because each analysis is restarted anew, instead of incrementally building on the work computed by an earlier analysis. We have focused on implementing a constant propagation optimisation and are currently investigating which additional optimisations may be added.

In conclusion, we can say that the current implementation approaches the three challenges that were identified in Section 3.3 in the following way:

Launch Point of the Analysis The analysis is started when the compiler finishes recording a trace.

Scope of the Analysis Starting from the current location in the program, each analysis keeps running until it reaches the end of the program. The current analysis is therefore not incremental.

Incorporating Run-Time Values As the scope of the run-time analyses can effectively be the entire program, all run-time, concrete values are converted to abstract values, because this naive analysis cannot predict which values will be relevant and which will not.

5. Related Work

5.1 JaegerMonkey

Our work is similar to the work of Hackett and Guo [12], who extended Mozilla's JaegerMonkey Javascript JIT compiler with a hybrid type inference technique to type specialise the dynamically compiled code. They perform an initial static analysis on a Javascript application, which generates an unsound set of type constraints for all expressions and statements in the program. To handle the imprecision caused by polymorphism in the code, the initial analysis generates run-time type barriers between constraints to restrict the set of types that could flow from one constraint to another. These type barriers are triggered at run time. If they are passed with a type they have not seen before, the new type value is propagated further through the constraints at run time. Constraints can also be generated by the analysis at run time. The approach leads to performance improvements of up to 50% on some benchmarks.

There are some differences between our technique and JaegerMonkey. As Hackett and Guo employ this technique to execute Javascript programs, the client program only becomes available when a website is loaded. As a result, initial analysis of the program necessarily has to be kept lightweight, so as to minimise the analysis time spent before execution. In contrast, our prototype operates on Scheme code. This enables us to run heavyweight analyses on client programs which can maximise the amount of information available beforehand, and minimise the amount of information that must be computed anew at run time. Furthermore, although the JaegerMonkey analysis can potentially range over the entire program, the optimisations only employ local type information of the program, instead of the optimisations that use a global view of the program like we propose. JaegerMonkey's technique is based on inclusion constraints, our technique is based on abstract interpretation via the AAM approach. The AAM approach potentially facilitates incorporating run-time information in the static analysis, as the static analysis works on a direct abstraction of the concrete semantics of the program. Lastly, JaegerMonkey is a method-based compiler, instead of the trace-based JIT compilers that we target. We believe the increased scope that is available to traces [14] can enhance the benefit of new optimisations that will become available.

5.2 Static Analysis and JIT Compilation

Although many, if not most, JIT compilers apply a lightweight analysis at run time over the code that is being compiled, Jaegermonkey is the only JIT compiler, to the best of our knowledge, which runs a static analysis over potentially the entire program at run time and which reuses the work computed by an initial analysis.

There are some JIT compilers that perform an initial analysis followed by a run-time, but local, analysis. However, in such compilers, the run-time analyses do not reuse the work of an earlier analysis, if there was one, and are also limited to a specific, local part of the program. An example is the McVM JIT compiler for MATLAB which was extended by Lameed and Hendren [16] with a two-stage static analysis for optimising copying of arrays. In the first stage, the compiler performs a simple static analysis over the program ahead-of-time to remove redundant copies. In the second stage, the compile combines two static analyses at run time to further optimise copying.

5.3 Dynamic Languages

Several recent empirical studies have investigated to what extent developers use dynamic features in dynamic languages, such as reflection and dynamic code generation [2, 7, 13, 18]. Although it is often said that these features are not commonly used by developers, these studies have partially refuted this claim. The results of these studies are important, as they indicate to which degree precision of static analyses can be improved by running the analyses at run time. It can be concluded from the studies that call-site monomorphism is high, even in dynamic languages: up to 81% in Javascript [18] and up to 97% in Python [1]. This means that, once the type of a receiver object is known, e.g., by looking at the concrete run-time type, this type will generally remain stable, facilitating further static analysis.

The level of other dynamic activity, i.e., reflection and dynamic code generation, depends on the language under consideration, with Javascript seemingly being used in much more dynamic manner [18] than Smalltalk [7].

To address the challenge of finding an ideal launchpoint of a static analysis, we look at studies which have investigated the amount of dynamic activity that takes place throughout the execution of a program. Holkner and Harland [13] reported that interactive Python programs show a remarkable higher level of dynamic activity in the startup phase of the program, compared with after this startup phase. This indicates that it might be optimal to analyse the program after completing the startup phase. However, these findings were disputed by Åkerblom et al. [2]. They also do not seem to be applicable to Javascript programs [18].

Sherwood et al. [22] have presented several clustering algorithms that attempt to explore phase-changes in the large-scale behaviour of a program. These algorithms may also be of use for us for finding launchpoints of run-time analyses.

5.4 Analysis at Run Time

Ernst [10] has noted similarities between dynamic analyses, operating on concrete values, and static analyses, operating on abstract values. He proposes to develop hybrid static-dynamic analyses, but does not present concrete examples of such analyses. Jenkins et al. [15] developed a theoretical framework, along with a practical implementation, for mixing concrete and abstract interpretation, but no concrete analysis is presented within this framework. Dufour et al. [9] developed a *blended* analysis for measuring the lifespan of objects created in framework-based Java applications. The blended analysis combines dynamic and static analysis by performing an offline static escape analysis on a call-graph that was obtained by dynamic analysis of the program. This approach enables avoiding the run-time overhead of dynamic monitoring of the program, while still being able to operate on a precise call-graph.

6. Conclusion

In this position paper, we outlined an approach for improving performance of trace-based JIT compilation by extending the local scope of information that is available to the trace-based JIT compiler with a global view of the entire program. The extended scope of information may expose additional sources of optimisations on traces by e.g., identifying sources of constants, soundly inferring types or finding independent, and therefore parallelisable, program segments. To achieve this global program view, we perform static, whole-program analyses over the program. To handle imprecision in static analyses for dynamic languages, we perform an initial, ahead-of-time static analysis and we update its results at run time by incorporating concrete run-time information, possibly in the process launching new analyses. To minimise the run-time overhead, such run-time analyses should be incremental, building on the work computed by both the initial, ahead-of-time analysis and previous run-time analyses. To further decrease overhead, we aim to run the analyses in parallel to the actual program execution whenever possible, on CPU cores that would otherwise remain idle.

We have made an initial step towards a full realisation of this technique by launching full static analyses over a program at run time, after completing the recording of a trace. These analyses are not yet incremental, but must be started completely anew. Nevertheless, they were already successful in finding constants that could not be predicted ahead-of-time.

References

[1] B. Åkerblom and T. Wrigstad. Measuring polymorphism in python programs. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, 2015.

[2] B. Åkerblom, J. Stendahl, M. Tumlin, and T. Wrigstad. Tracing dynamic features in python programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, 2014.

[3] E. Andreasen and A. Møller. Determinacy in static analysis for jquery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, 2014.

[4] S. Arzt and E. Bodden. Reviser: Efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, 2014.

[5] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proc. of the 4th ICOOOLPS Workshop*, 2009.

[6] C. F. Bolz, A. Cuni, M. Fijlakowski, M. Leuschel, S. Pedroni, and A. Rigo. Runtime feedback in a meta-tracing jit for efficient dynamic languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '11, 2011.

[7] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger. How developers use the dynamic features of programming languages: the case of smalltalk. In *Proceedings of the International Working Conference on Mining Software Repositories*, MSR'11, 2011.

[8] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In K. Etessami and S. K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification*, 2005.

[9] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, 2007.

[10] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, WODA '03, 2003.

[11] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proc. of the 30th ACM SIGPLAN PLDI Conf.*, 2009.

[12] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, 2012.

[13] A. Holkner and J. Harland. Evaluating the dynamic behaviour of python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09, 2009.

[14] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, 2011.

[15] M. Jenkins, L. Andersen, T. Gilray, and M. Might. Concrete and abstract interpretation: Better together. In *Proceedings of the 2014 Scheme and Functional Programming Workshop*, SFP '14, 2015.

[16] N. Lameed and L. Hendren. Staged static techniques to efficiently implement array copy semantics in a matlab jit compiler. In *International Conference on Compiler Construction*, CC '11, 2011.

[17] J. Nicolay, C. De Roover, W. De Meuter, and V. Jonckers. Automatic parallelization of side-effecting higher-order scheme programs. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '11, 2011.

[18] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. *SIGPLAN Not.*, 45(6), June 2010.

[19] D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, 2005.

[20] H. N. Santos, P. Alves, I. Costa, and F. M. Quintao Pereira. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, 2013.

[21] D. Schneider and C. F. Bolz. The efficient handling of guards in the design of rpython's tracing jit. In *Proc. of the 6th ACM VMIL Workshop*, 2012.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5), Oct. 2002.

[23] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of javascript. In *European Conference on Object-Oriented Programming*, 2012.

[24] Q. Stievénart, J. Nicolay, W. De Meuter, and C. De Roover. Detecting concurrency bugs in higher-order programs through abstract interpretation. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, 2015.

[25] Q. Stievénart, M. Vandercammen, J. Nicolay, W. De Meuter, and C. De Roover. Scala-am: A modular static analysis framework. In *Proceedings of the 16th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '16, 2016.

[26] D. Van Horn and M. Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, 2010.