The Efficient Handling of Guards in the Design of RPython's Tracing JIT

David Schneider Carl Friedrich Bolz

Heinrich-Heine-Universität Düsseldorf, STUPS Group, Germany

david.schneider@uni-duesseldorf.de cfbolz@gmx.de

Abstract

Tracing just-in-time (JIT) compilers record linear control flow paths, inserting operations called guards at points of possible divergence. These operations occur frequently in generated traces and therefore it is important to design and implement them carefully to find the right trade-off between deoptimization, memory overhead, and (partly) execution speed. In this paper, we perform an empirical analysis of runtime properties of guards. This is used to guide the design of guards in the RPython tracing JIT.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, incremental compilers, interpreters, run-time environments

General Terms Languages, Performance, Experimentation

Keywords tracing JIT, guards, deoptimization

1. Introduction

Tracing just-in-time (JIT) compilers record and compile commonly executed linear control flow paths consisting of operations executed by an interpreter.[1] At points of possible divergence from the traced path operations called guards are inserted. Furthermore, type guards are inserted to specialize the trace based on the types observed during tracing. In this paper we describe and analyze how guards work and explain the concepts used in the intermediate and low-level representation of the JIT instructions and how these are implemented. This is done in the context of the RPython language and the PyPy project, which provides a tracing JIT compiler geared at dynamic language optimization.

Our aim is to help understand the constraints when implementing guards and to describe the concrete techniques used in the various layers of RPython's tracing JIT. All design decisions are motivated by an empirical analysis of the frequency and the overhead related to guards.

It is important to handle guards well, because they are very common operations in the traces produced by tracing JITs. As we will see later (Figure 7) guards account for about 14% to 22% of the operations before and for about 15% to 20% of the operations after optimizing the traces generated for the different benchmarks used in this paper. An additional property is that guard failure rates are very uneven. The majority of guards never fail at all, whereas those that do usually fail extremely often.

Besides being common, guards have various costs associated with them. Guards are possible deoptimization points. The recorded and compiled path has to be left if a guard fails, returning control to the interpreter. Therefore guards need enough associated information to enable rebuilding the interpreter state. The memory overhead of this information should be kept low. On the other hand, Guards have a runtime cost, they take time to execute. Therefore it is important to make the on-trace execution of guards as efficient as possible. These constraints and trade-offs are what makes the design and optimization of guards an important and non-trivial aspect of the construction of a tracing just-in-time compiler.

In this paper we want to substantiate the aforementioned observations about guards and describe based on them the reasoning behind their implementation in RPython's tracing just-in-time compiler. The contributions of this paper are:

- An analysis of guards in the context of RPython's JIT,
- detailed measurements about the frequency and the memory overhead associated with guards, and
- a description about how guards are implemented in the high and low-level components of RPython's JIT and a description of the rationale behind the design.

The set of central concepts upon which this work is based are described in Section 2, such as the PyPy project, the RPython language and its meta-tracing JIT. Based on these

[1] There are also virtual machines that have a tracing JIT compiler and do not use an interpreter [4]. This paper assumes that the baseline is provided by an interpreter. Similar design constraints would apply to a purely compiler-based system.

VMIL'12, October 21, 2012, Tucson, Arizona, USA.

concepts in Section 3 we proceed to describe the details of guards in the frontend of RPython's tracing JIT. Once the frontend has traced and optimized a loop it invokes the backend to compile the operations to machine code, Section 4 describes the low-level aspects of how guards are implemented in the machine specific JIT-backend. The frequency of guards and the overhead associated with the implementation described in this paper is discussed in Section 5. Section 6 presents an overview about how guards are treated in the context of other just-in-time compilers. Finally, Section 7 summarizes our conclusions and gives an outlook on further research topics.

2. Background

2.1 RPython and the PyPy Project

The RPython language and the PyPy project[2] [22] were started in 2002 with the goal of creating a Python interpreter written in a high level language, allowing easy language experimentation and extension. PyPy is now a fully compatible alternative interpreter for the Python language. Using RPython's tracing JIT compiler it is on average about 5 times faster than CPython, the reference implementation. PyPy is an interpreter written in RPython and takes advantage of the language features provided by RPython such as the provided tracing just-in-time compiler described below.

RPython, the language and the toolset originally created to implement the Python interpreter have developed into a general environment for experimenting and developing fast and maintainable dynamic language implementations. Besides the Python interpreter there are several experimental language implementation at different levels of completeness, e.g. for Prolog [9], Smalltalk [8], JavaScript and R.

RPython can mean one of two things, the language itself and the translation toolchain used to transform RPython programs to executable units. The RPython language is a statically typed object-oriented high-level subset of Python. The subset is chosen in such a way to make type inference possible[1]. The language tool-set provides several features such as automatic memory management and just-in-time compilation. When writing an interpreter using RPython the programmer only has to write the interpreter for the language she is implementing. The second RPython component, the translation toolchain, is used to transform the interpreter into a C program.[3] During the transformation process different low level aspects suited for the target environment are automatically added to the program such as a garbage collector and a tracing JIT compiler. The process of inserting a tracing JIT is not fully automatic but is guided by hints from the interpreter author.

[2] `http://pypy.org`

[3] RPython can also be used to translate programs to CLR and Java bytecode [1], but this feature is somewhat experimental.

2.2 RPython's Tracing JIT Compiler

Tracing is a technique of just-in-time compilers that generate code by observing the execution of a program. VMs using tracing JITs are typically mixed-mode execution environments that also contain an interpreter. The interpreter profiles the executing program and selects frequently executed code paths to be compiled to machine code. Many tracing JIT compilers focus on selecting hot loops.

After profiling identifies an interesting path, tracing is started thus recording all operations that are executed on this path. This includes inlining functional calls. As in most compilers, tracing JITs use an intermediate representation to store the recorded operations, typically in SSA form [11]. Since tracing follows actual execution, the code that is recorded represents only one possible path through the control flow graph. Points of divergence from the recorded path are marked with special operations called *guards*. These operations ensure that assumptions valid during the tracing phase are still valid when the code has been compiled and is executed. Guards are also used to encode type checks that come from optimistic type specialization by recording the types of variables seen during tracing[13, 14]. After a trace has been recorded it is optimized and then compiled to platform specific machine code.

When the check of a guard fails, the execution of the machine code must be stopped and the control is returned to the interpreter, after the interpreter's state has been restored. If a particular guard fails often a new trace starting from the guard is recorded. We will refer to this kind of trace as a *bridge*. Once a bridge has been traced and compiled it is attached to the corresponding guard by patching the machine code. The next time the guard fails the bridge will be executed instead of leaving the machine code.

RPython provides a tracing JIT that can be reused for a number of language implementations [7]. This is possible, because it traces the execution of the language interpreter instead of tracing the user program directly. This approach is called *meta-tracing*. For the purpose of this paper the fact that RPython's tracing JIT is a meta-tracing JIT can be ignored. The only point of interaction is that some of the guards that are inserted into the trace stem from an annotation provided by the interpreter author [6].

Figure 1 shows an example RPython function that checks whether a number reduces to 1 with less than 100 steps of the Collatz process.[4] It uses an `Even` and an `Odd` class to box the numbers, to make the example more interesting. If the loop in `check_reduces` is traced when `a` is a multiple of four, the unoptimized trace looks like in Figure 2. The line numbers in the trace correspond to the line numbers in Figure 3. The resulting trace repeatedly halves the current value and checks whether it is equal to one, or odd. In either of these cases the trace is left via a guard failure.

[4] `http://en.wikipedia.org/wiki/Collatz_conjecture`

```
class Base(object):                                                  1
    def __init__(self, n):                                           2
        self.value = n                                               3
    @staticmethod                                                    4
    def build(n):                                                    5
        if n & 1 == 0:                                               6
            return Even(n)                                           7
        else:                                                        8
            return Odd(n)                                            9
                                                                     10
class Odd(Base):                                                     11
    def step(self):                                                  12
        return Even(self.value * 3 + 1)                              13
                                                                     14
class Even(Base):                                                    15
    def step(self):                                                  16
        n = self.value >> 2                                          17
        if n == 1:                                                   18
            return None                                              19
        return self.build(n)                                         20
                                                                     21
def check_reduces(a):                                                22
    j = 1                                                            23
    while j < 100:                                                   24
        j += 1                                                       25
        if a is None:                                                26
            return True                                              27
        a = a.step()                                                 28
    return False                                                     29
```

Figure 1. Example program

```
[j1, a1]                                                             -1
j2 = int_add(j1, 1)                                                  25
guard_nonnull(a1)                                                    26
guard_class(a1, Even)                                                28
i1 = getfield_gc(a1, descr='value')                                  17
i2 = int_rshift(i1, 2)                                               17
b1 = int_eq(i2, 1)                                                   18
guard_false(b1)                                                      18
i3 = int_and(i2, 1)                                                  6
i4 = int_is_zero(i3)                                                 6
guard_true(i4)                                                       6
a2 = new(Even)                                                       7
setfield_gc(a2, descr='value')                                       3
b2 = int_lt(j2, 100)                                                 24
guard_true(b2)                                                       24
jump(j2, a2)                                                         -1
```

Figure 2. Unoptimized trace, the line numbers in the trace correspond to the line numbers in Figure 3.

3. Guards in the Frontend

In this context we refer to frontend as the component of the JIT that is concerned with recording and optimizing the traces as well as storing the information required to rebuild the interpreter state in case of a guard failure. Since tracing linearizes control flow by following one concrete execution, the full control flow of a program is not observed. The possible points of deviation from the trace are denoted by guard operations that check whether the same assumptions observed while tracing still hold during execution. Similarly, in the case of dynamic languages guards can also encode type assumptions. In later executions of the trace the guards can fail. If that happens, execution needs to continue in the interpreter. This means it is necessary to attach enough information to a guard to reconstruct the interpreter state when that guard fails. This information is called the *resume data*.

To do this reconstruction it is necessary to take the values of the SSA variables in the trace to build interpreter stack frames. Tracing aggressively inlines functions, therefore the reconstructed state of the interpreter can consist of several interpreter frames.

If a guard fails often enough, a trace is started from it to create a bridge, forming a trace tree. When that happens another use case of resume data is to reconstruct the tracer state. After the bridge has been recorded and compiled it is attached to the guard. If the guard fails later the bridge is executed. Therefore the resume data of that guard is no longer needed.

There are several forces guiding the design of resume data handling. Guards are a very common operation in the traces. However, as will be shown, a large percentage of all operations are optimized away before code generation. Since there are a lot of guards the resume data needs to be stored in a very compact way. On the other hand, tracing should be as fast as possible, so the construction of resume data must not take too much time.

3.1 Capturing of Resume Data During Tracing

Every time a guard is recorded during tracing the tracer attaches preliminary resume data to it. The data is preliminary in that it is not particularly compact yet. The preliminary resume data takes the form of a stack of symbolic frames. The stack contains only those interpreter frames seen by the tracer. The frames are symbolic in that the local variables in the frames do not contain values. Instead, every local variable contains the SSA variable of the trace where the value would later come from, or a constant.

3.2 Compression of Resume Data

After tracing has been finished the trace is optimized. During optimization a large percentage of operations can be removed (Figure 7). In the process the resume data is transformed into its final, compressed form. The rationale for not compressing the resume data during tracing is that a lot of guards will be optimized away. For them, the compression effort would be lost.

The core idea of storing resume data as compactly as possible is to share parts of the data structure between subsequent guards. This is useful because the density of guards in traces is so high, that quite often not much changes between them. Since resume data is a linked list of symbolic frames, in many cases only the information in the top frame changes from one guard to the next. The other symbolic frames can often be reused. The reason for this is that, during trac-

ing only the variables of the currently executing frame can change. Therefore if two guards are generated from code in the same function the resume data of the rest of the frame stack can be reused.

In addition to sharing as much as possible between subsequent guards, a compact representation of the local variables of symbolic frames is used. Every variable in the symbolic frame is encoded using two bytes. Two bits are used as a tag to denote where the value of the variable comes from. The remaining 14 bits are a payload that depends on the tag bits. The possible sources of information are:

- For small integer constants the payload contains the value of the constant.
- For other constants the payload contains an index into a per-loop list of constants.
- For SSA variables, the payload is the number of the variable.
- For virtuals, the payload is an index into a list of virtuals, see next section.

3.3 Interaction With Optimization

Guards interact with optimizations in various ways. Using many classical compiler optimizations the JIT tries to remove as many operations, and therefore guards, as possible. In particular guards can be removed by subexpression elimination. If the same guard is encountered a second time in a trace, the second one can be removed. This also works if a later guard is weaker and hence implied by an earlier guard.

One of the techniques in the optimizer specific to tracing for removing guards is guard strengthening [3]. The idea of guard strengthening is that if a later guard is stronger than an earlier guard it makes sense to move the stronger guard to the point of the earlier, weaker guard and to remove the weaker guard. Moving a guard to an earlier point is always valid, it just means that the guard fails earlier during the trace execution (the other direction is clearly not valid).

The other important point of interaction between resume data and the optimizer is RPython's allocation removal optimization [5]. This optimization discovers allocations in the trace that create objects that do not survive long. An example is the instance of `Even` in Figure 2. Allocation removal makes resume data more complex. Since allocations are removed from the trace it becomes necessary to reconstruct the objects that were not allocated so far when a guard fails. Consequently the resume data needs to store enough information to make this reconstruction possible.

Storing this additional information is done as follows: So far, every variable in the symbolic frames contains a constant or an SSA variable. After allocation removal the variables in the symbolic frames can also contain "virtual" objects. These are objects that were not allocated so far, because the optimizer removed their allocation. The structure of the heap objects that have to be allocated on guard failure is described by the virtual objects stored in the symbolic frames. To this end, the content of every field of the virtual object is described in the same way that the local variables of symbolic frames are described. The fields of the virtual objects can therefore be SSA variables, constants or other virtual objects. They are encoded using the same compact two-byte representation as local variables.

During the storing of resume data virtual objects are also shared between subsequent guards as much as possible. The same observation as about frames applies: Quite often a virtual object does not change from one guard to the next, allowing the data structure to be shared.

A related optimization is the handling of heap stores by the optimizer. The optimizer tries to delay stores into the heap as long as possible. This is done because often heap stores become unnecessary due to another store to the same memory location later in the trace. This can make it necessary to perform these delayed stores when leaving the trace via a guard. Therefore the resume data needs to contain a description of the delayed stores to be able to perform them when the guard fails. So far no special compression is done with this information, compared to the other source of information delayed heap stores are quite rare.

Figure 3 shows the optimized version of the trace in Figure 2. Allocation removal has removed the `new` operation and other operations handling the instance. The operations handle unboxed numbers now.

Figure 4 sketches the symbolic frames of the first two guards in the trace. The frames for `check_reduces` and `Even.step` as well as the description of the allocation-removed virtual instance of `Even` are shared between the two guards.

```
label(j2, i2, descr=label1)                    -1
j3 = int_add(j2, 1)                            25
i5 = int_rshift(i2, 2)                         17
b3 = int_eq(i5, 1)                             18
guard_false(b3)                                18
i6 = int_and(i5, 1)                            6
b4 = int_is_zero(i6)                           6
guard_true(b4)                                 6
b5 = int_lt(j3, 100)                           24
guard_true(b5)                                 24
jump(j3, i5, descr=label1)                     -1
```

Figure 3. Optimized trace

4. Guards in the Backend

After the recorded trace has been optimized, it is handed over to the platform specific backend to be compiled to machine code. The compilation phase consists of two passes over the lists of instructions, a backwards pass to calculate live ranges of IR-level variables and a forward pass to emit the instructions. During the forward pass IR-level variables are assigned to registers and stack locations by the register allocator according to the requirements of the emitted instructions. Eviction/spilling is performed based on the live range information collected in the first pass. Each IR in-

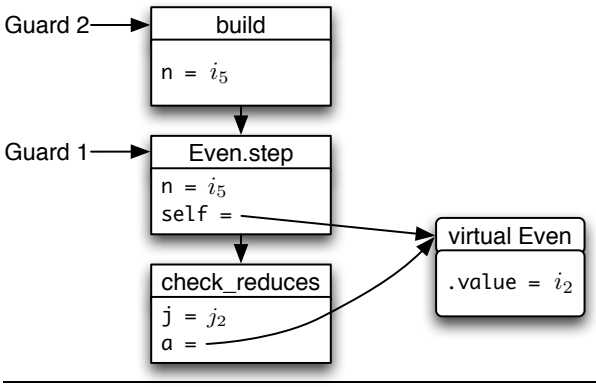


Figure 4. The resume data for Figure 3

struction is transformed into one or more machine level instructions that implement the required semantics. Operations without side effects whose result is not used are not emitted. Guard instructions are transformed into fast checks at the machine code level that verify the corresponding condition. In cases the value being checked by the guard is not used anywhere else the guard and the operation producing the value can merged, further reducing the overhead of the guard. Figure 5 shows how the `int_eq` operation followed by a `guard_false` from the trace in Figure 3 are compiled to pseudo-assembler if the operation and the guard are compiled separated or if they are merged.

```
b3 = int_eq(i5, 1)                    18
guard_false(b3)                       18
```

```
CMP r6, #1
MOVEQ r8, #1
MOVNE r8, #0
...
CMP r8, #0
BEQ <bailout>
```

```
CMP r6, #1
BNE <bailout>
...
...
...
...
```

Figure 5. Result of separated (left) and merged (right) compilation of one guard and the following operation (top).

Attached to each guard in the IR is a list of the IR-variables required to rebuild the execution state in case the trace is left through the guard. When a guard is compiled, in addition to the condition check two things are generated/-compiled.

First, a special data structure called *backend map* is created. This data structure encodes the mapping from IR-variables needed by the guard to rebuild the state to the low-level locations (registers and stack) where the corresponding values will be stored when the guard is executed. This data structure stores the values in a succinct manner. The encoding is efficient to create and provides a compact representation of the needed information in order to maintain an acceptable memory profile.

Second, for each guard a piece of code is generated that acts as a trampoline. Guards are implemented as a conditional jump to this trampoline in case the guard check fails. In the trampoline, the pointer to the backend map is loaded and after storing the current execution state (registers and stack) execution jumps to a generic bailout handler, also known as *compensation code*, that is used to leave the compiled trace.

Using the encoded location information the bailout handler reads from the stored execution state the values that the IR-variables had at the time of the guard failure and stores them in a location that can be read by the frontend. After saving the information the control is returned to the frontend signaling which guard failed so the frontend can read the stored information and rebuild the state corresponding to the point in the program.

As in previous sections, the underlying idea for the low-level design of guards is to have a fast on-trace profile and a potentially slow one in case the execution has to return to the interpreter. At the same time, the data stored in the backend, required to rebuild the state, should be as compact as possible to reduce the memory overhead produced by the large number of guards. The numbers in Figure 9 illustrate that the compressed encoding currently has about 15% to 25% of the size of of the generated instructions on x86.

As explained in previous sections, when a specific guard has failed often enough a bridge starting from this guard is recorded and compiled. Since the goal of compiling bridges is to improve execution speed on the diverged path (failing guard) they should not introduce additional overhead. In particular the failure of the guard should not lead to leaving the compiled code prior to execution the code of the bridge.

The process of compiling a bridge is very similar to compiling a loop. Instructions and guards are processed in the same way as described above. The main difference is the setup phase. When compiling a trace we start with a clean slate. The compilation of a bridge is started from a state (register and stack bindings) that corresponds to the state during the compilation of the original guard. To restore the state needed to compile the bridge we use the backend map created for the guard to rebuild the bindings from IR-variables to stack locations and registers. With this reconstruction all bindings are restored to the state as they were in the original loop up to the guard. This means that no register/stack reshuffling is needed before executing a bridge.

Once the bridge has been compiled the corresponding guard is patched to redirect control flow to the bridge in case the check fails. In the future, if the guard fails again it jumps to the code compiled for the bridge instead of bailing out. Once the guard has been compiled and attached to the loop the guard becomes just a point where control-flow can split. The guard becomes the branching point of two conditional paths with no additional overhead. Figure 6 shows a diagram of a compiled loop with two guards, Guard #1 jumps to the

trampoline, loads the backend map and then calls the bailout handler, whereas Guard #2 has already been patched and directly jumps to the corresponding bridge. The bridge also contains two guards that work based on the same principles.

5. Evaluation

The results presented in this section are based on numbers gathered by running a subset of the standard PyPy benchmarks. The PyPy benchmarks are used to measure the performance of PyPy and are composed of a series of micro-benchmarks and larger programs.[5] The benchmarks were taken from the PyPy benchmarks repository using revision `ff7b35837d0f`.[6] The benchmarks were run on a version of PyPy based on revision `0b77afaafdd0` and patched to collect additional data about guards in the machine code backends.[7] The tools used to run and evaluate the benchmarks including the patches applied to the PyPy sourcecode can be found in the repository for this paper.[8] All benchmark data was collected on a MacBook Pro 64 bit running Max OS 10.8 with the loop unrolling optimization disabled.[9]

We used the following benchmarks:

chaos: A Chaosgame implementation creating a fractal.

crypto_pyaes: An AES implementation.

django: The templating engine of the Django Web framework.[10]

go: A Monte-Carlo Go AI.[11]

pyflate_fast: A BZ2 decoder.

raytrace_simple: A ray tracer.

richards: The Richards benchmark.

spambayes: A Bayesian spam filter.[12]

simpy_expand: A computer algebra system.

telco: A Python version of the Telco decimal benchmark,[13] using a pure Python decimal floating point implementation.

twisted_names: A DNS server benchmark using the Twisted networking framework.[14]

[5] `http://speed.pypy.org/`

[6] `https://bitbucket.org/pypy/benchmarks/src/ff7b35837d0f`

[7] `https://bitbucket.org/pypy/pypy/src/0b77afaafdd0`

[8] `https://bitbucket.org/pypy/extradoc/src/tip/talk/vmil2012`

[9] Since loop unrolling duplicates the body of loops it would no longer be possible to meaningfully compare the number of operations before and after optimization. Loop unrolling is most effective for numeric kernels, so the benchmarks presented here are not affected much by its absence.

[10] `http://www.djangoproject.com/`

[11] `http://shed-skin.blogspot.com/2009/07/disco-elegant-python-go-player.html`

[12] `http://spambayes.sourceforge.net/`

[13] `http://speleotrove.com/decimal/telco.html`

[14] `http://twistedmatrix.com/`

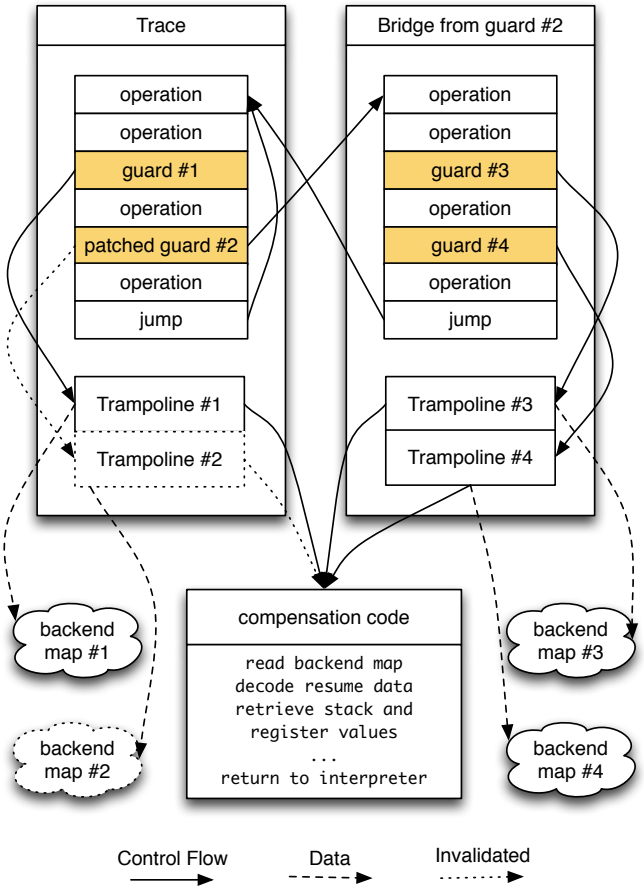


Figure 6. Trace control flow in case of guard failures with and without bridges

From the mentioned benchmarks we collected different datasets to evaluate the frequency, the overhead and overall behaviour of guards, the results are summarized in the remainder of this section. We want to point out three aspects of guards in particular:

- Guards are very common operations in traces.
- There is overhead associated with guards.
- Guard failures are local and rare.

All measurements presented in this section do not take garbage collection of resume data and machine code into account. Pieces of machine code can be globally invalidated or just become cold again. In both cases the generated machine code and the related data is garbage collected. The figures show the total amount of operations that are evaluated by the JIT and the total amount of code and resume data that is generated. The measurements and the evaluation focus on trace properties and memory consumption, and do not discuss the execution time of the benchmarks. These topics were covered in earlier work [5] and furthermore are not influenced that much by the techniques described in this paper.

Benchmark	# Traces	Ops. before	Guards before	Ops. after	Guards after	Opt. rate	Guard opt. rate
chaos	3213	21787	3954 ~ 18.1%	5168	888 ~ 17.2%	76.3%	77.5%
crypto_pyaes	3516	19675	2795 ~ 14.2%	6028	956 ~ 15.9%	69.4%	65.8%
django	4021	22740	5111 ~ 22.5%	5661	1137 ~ 20.1%	75.1%	77.8%
go	870805	785747	130499 ~ 16.6%	152966	29989 ~ 19.6%	80.5%	77.0%
pyflate-fast	147104	85886	13826 ~ 16.1%	21639	4019 ~ 18.6%	74.8%	70.9%
raytrace-simple	11585	89414	14174 ~ 15.9%	17526	2661 ~ 15.2%	80.4%	81.2%
richards	5138	32461	5503 ~ 17.0%	5552	1044 ~ 18.8%	82.9%	81.0%
spambayes	471321	242423	42053 ~ 17.3%	70962	12693 ~ 17.9%	70.7%	69.8%
sympy_expand	174113	92238	20333 ~ 22.0%	22417	4532 ~ 20.2%	75.7%	77.7%
telco	9364	97821	20356 ~ 20.8%	15794	2804 ~ 17.8%	83.9%	86.2%
twisted_names	250114	222535	47490 ~ 21.3%	49947	9561 ~ 19.1%	77.6%	79.9%

Figure 7. Number of operations and guards in the recorded traces before and after optimizations

5.1 Frequency of Guards

Figure 7 summarizes[15] the total number of operations that were recorded during tracing for each of the benchmarks and what percentage of these operations are guards. The static number of operations was counted on the unoptimized and optimized traces. The figure also shows the overall optimization rate for operations, which is between 69.4% and 83.89%, of the traced operations and the optimization rate of guards, which is between 65.8% and 86.2% of the operations. This indicates that the optimizer can remove most of the guards, but after the optimization pass these still account for 15.2% to 20.2% of the operations being compiled and later executed. The frequency of guard operations makes it important to store the associated information efficiently and also to make sure that guard checks are executed quickly.

5.2 Guard Failures

The next point in this discussion is the frequency of guard failures. Figure 8 presents for each benchmark a list of the relative amounts of guards that ever fail and of guards that fail often enough that a bridge is compiled.[16] It also contains sparklines depicting the failure rates for the failing guards in decreasing order, each normalized to the most failing guard. The numbers presented for guards that have a bridge represent the failures up to the compilation of the bridge and all executions of the then attached bridge.

From Figure 8 we can see that only a very small amount of all the guards in the compiled traces ever fail. This amount varies between 2.4% and 5.7% of all guards. As can be expected, even fewer, only 1.2% to 3.6% of all guards fail often enough that a bridge is compiled for them. Also, of all failing guards a few fail extremely often and most fail rarely. Reinforcing this notion the figure shows that, depending on the benchmark, between 0.008% and 0.225% of the guards are responsible for 50% of the total guards failures. Even considering 99.9% of guard failures the relative amount of guards does not rise above 3%. The colored dots in the sparklines correspond to 50%, 99% and 99.9%. These results emphasize that as most of the guards never fail it is important to make sure that the successful execution of a guard does not have unnecessary overhead.

This low guard failure rate is expected. Most guards do not come from actual control flow divergences in the user program, but from type checks needed for type specialization. Various prior work has shown [10, 15, 21] that most programs in dynamic languages only use a limited amount of runtime variability. Therefore many guards are needed for making the traces behave correctly in all cases but fail rarely.

5.3 Space Overhead of Guards

The overhead that is incurred by the JIT to manage the resume data, the backend map as well as the generated machine code is shown in Figure 9. It shows the total memory consumption of the code and of the data generated by the machine code backend and an approximation of the size of the resume data structures for the different benchmarks mentioned above. The machine code taken into account is composed of the compiled operations, the trampolines generated for the guards and a set of support functions that are generated when the JIT starts and which are shared by all compiled traces. The size of the backend map is the size of the compressed mapping from registers and stack to IR-level variables and finally the size of the resume data is the size of the compressed high-level resume data as described in Section 3.[17]

For the different benchmarks the backend map has about 15% to 20% of the size compared to the size of the generated machine code. On the other hand the generated machine code has only a size ranging from 20.5% to 37.98% of the size of the resume data and the backend map combined and being compressed as described before.

Tracing JIT compilers only compile the subset of the code executed in a program that occurs in a hot loop, for

[15] In all tables the minimum and maximum values for each column are highlighted in dark/light gray.

[16] The threshold used is 200 failures. This rather high threshold was picked experimentally to give good results for long-running programs.

[17] Due to technical reasons the size of the resume data is hard to measure directly at runtime. Therefore the size given in the table is reconstructed from debugging information stored in log files produced by the JIT.

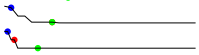
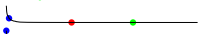
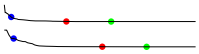
Benchmark	Sparkline	Failing	> 200 failures	50% of failures	99% of failures	99.9% of failures
chaos		3.5%	1.5%	2 ~ 0.225%	9 ~ 1.014%	11 ~ 1.239%
crypto_pyaes		3.0%	1.7%	2 ~ 0.209%	8 ~ 0.837%	8 ~ 0.837%
django		5.4%	1.8%	2 ~ 0.185%	4 ~ 0.369%	11 ~ 1.015%
go		4.0%	2.7%	18 ~ 0.060%	410 ~ 1.367%	795 ~ 2.651%
pyflate-fast		3.9%	2.6%	1 ~ 0.025%	31 ~ 0.771%	64 ~ 1.592%
raytrace-simple		4.2%	3.2%	5 ~ 0.188%	42 ~ 1.578%	65 ~ 2.443%
richards		5.7%	3.6%	2 ~ 0.192%	23 ~ 2.203%	30 ~ 2.874%
spambayes		4.0%	2.5%	1 ~ 0.008%	110 ~ 0.852%	266 ~ 2.060%
sympy_expand		4.9%	2.6%	9 ~ 0.199%	73 ~ 1.611%	125 ~ 2.758%
telco		3.0%	2.3%	5 ~ 0.178%	43 ~ 1.534%	62 ~ 2.211%
twisted_names		2.4%	1.2%	9 ~ 0.094%	46 ~ 0.481%	101 ~ 1.055%

Figure 8. Failing guards, guards with more than 200 failures and guards responsible for 50%, 99% and 99.9% of the failures relative to the total number of guards

Benchmark	Code	Resume data	Backend map
chaos	157.1 KiB	390.5 KiB	24.4 KiB
crypto_pyaes	170.4 KiB	493.2 KiB	24.1 KiB
django	233.5 KiB	577.2 KiB	51.0 KiB
go	4871.0 KiB	22877.6 KiB	888.1 KiB
pyflate-fast	729.3 KiB	2036.7 KiB	150.7 KiB
raytrace-simple	491.6 KiB	1427.7 KiB	74.0 KiB
richards	157.1 KiB	685.1 KiB	17.6 KiB
spambayes	2499.9 KiB	6601.5 KiB	331.7 KiB
sympy_expand	929.2 KiB	2231.1 KiB	214.0 KiB
telco	516.5 KiB	1514.1 KiB	77.6 KiB
twisted_names	1694.9 KiB	5486.0 KiB	228.4 KiB

Figure 9. Total size of generated machine code and resume data

Benchmark	Compressed	Naive	xz compressed
chaos	390.48 KiB	1312.44 KiB	82.27 KiB
crypto_pyaes	493.17 KiB	1685.70 KiB	90.00 KiB
django	577.23 KiB	2383.15 KiB	109.70 KiB
go	22877.60 KiB	91200.30 KiB	3753.16 KiB
pyflate-fast	2036.74 KiB	7422.01 KiB	380.38 KiB
raytrace-simple	1427.70 KiB	4591.58 KiB	270.48 KiB
richards	685.10 KiB	2579.73 KiB	116.98 KiB
spambayes	6601.51 KiB	36708.27 KiB	1248.16 KiB
sympy_expand	2231.07 KiB	10048.70 KiB	442.48 KiB
telco	1514.11 KiB	6352.27 KiB	285.35 KiB
twisted_names	5485.98 KiB	30032.90 KiB	1034.82 KiB

Figure 10. Resume data sizes

this reason the amount of generated machine code will be smaller than in other just-in-time compilation approaches. This creates a larger discrepancy between the size of the resume data when compared to the size of the generated machine code and illustrates why it is important to compress the resume data information.

Why the efficient storing of the resume data is a central concern in the design of guards is illustrated by Figure 10. This figure shows the size of the compressed resume data, the approximated size of storing the resume data without compression and an approximation of the best possible compression of the resume data by compressing the data using the *xz* compression tool, which is a "general-purpose data compression software with high compression ratio".[18]

The results show that the current approach of compression and data sharing only requires 18.3% to 31.1% of the space compared to a naive approach. This shows that large parts of the resume data are redundant and can be stored more efficiently using the techniques described earlier. On the other hand comparing the results to the xz compression which only needs between 17.1% and 21.1% of the space required by our compression shows that the compression is not optimal and could be improved taking into account the trade-off between the required space and the time needed to build a good, compressed representation of the resume data for the large amount of guards present in the traces.

[18] `http://tukaani.org/xz/`

6. Related Work

6.1 Guards in Other Tracing JITs

Guards, as described, are a concept associated with tracing just-in-time compilers to represent possible divergent control flow paths.

SPUR [3] is a tracing JIT compiler for a CIL virtual machine. It handles guards by always generating code for every one of them that transfers control back to the unoptimized code. Since the transfer code needs to reconstruct the stack frames of the unoptimized code, the transfer code is large.

Mike Pall, the author of LuaJIT describes in a post to the lua-users mailing list different technologies and techniques used in the implementation of LuaJIT [20]. Pall explains that guards in LuaJIT use a datastucture called snapshots, similar to RPython's resume data, to store the information about how to rebuild the state from a guard failure using the information in the snapshot and the machine execution state. According to Pall [20] snapshots for guards in LuaJIT are associated with a large memory footprint. The solution used

there is to store sparse snapshots, avoiding the creation of snapshots for every guard to reduce memory pressure. Snapshots are only created for guards after updates to the global state, after control flow points from the original program and for guards that are likely to fail. As an outlook Pall mentions plans to switch to compressed snapshots to further reduce redundancy.[19] It should be possible to combine the approaches of not creating snapshots at all for every guard and the resume data compression presented in this paper.

Linking side exits to pieces of later compiled machine code was described first in the context of Dynamo [2] under the name of fragment linking. Once a new hot trace is emitted into the fragment cache it is linked to the side exit that led to the compilation of the fragment. Fragment linking avoids the performance penalty involved in leaving the compiled code. Fragment linking also allows to remove compensation code associated to the linked fragments that would have been required to restored the execution state on the side exit.

Gal et. al [14] describe the HotpathVM, a JIT for a Java VM. They experimented with having one generic compensation code block, like the RPython JIT, that uses a register variable mapping to restore the interpreter state. Later this was replaced by generating compensation code for each guard which produced a lower overhead in their benchmarks. HotpathVM also records secondary traces starting from failing guards that are connected directly to the original trace. Secondary traces are compiled by first restoring the register allocator state to the state at the side exit. The information is a mapping stored in the guard between machine level registers and stack to Java level stack and variables.

For TraceMonkey, a tracing JIT for JavaScript, Gal et. al [13] illustrate how it uses a small off-trace set of instructions that is executed in case a guard failure to return a structure describing the reason for the exit along with the information needed to restored the interpreter state. TraceMonkey uses trace stitching to avoid the overhead of returning to the trace monitor and calling another trace when taking a side exit. In this approach it is required to write live values to an activation record before entering the new trace.

6.2 Deoptimization in Method-Based JITs

Deoptimization in method-based JITs is used if one of the assumptions of the code generated by a JIT changes. This is often the case when new code is added to the system, or when the programmer tries to debug the program.

Deutsch et. al. [12] use stack descriptions to make it possible to do source-level debugging of JIT-compiled code. Self uses deoptimization to reach the same goal [16]. When a function is to be debugged, the optimized code version is left and one compiled without inlining and other optimizations is entered. Self uses scope descriptors to describe the frames that need to be re-created when leaving the optimized code. The scope descriptors are between 0.42 and 1.09 times the size of the generated machine code. The information needed for debugging together is between 1.22 and 2.33 times the size of generated machine code, according to the paper.

Java Hotspot [19] contains a deoptimization framework that is used for debugging and when an uncommon trap is triggered. To be able to do this, Hotspot stores a mapping from optimized states back to the interpreter state at various deoptimization points. There is no discussion of the memory use of this information.

The deoptimization information of Hotspot is extended to support correct behaviour when scalar replacement of fields is done for non-escaping objects [17, 18]. The approach is extremely similar to how RPython's JIT handles virtual objects. For every object that is not allocated in the code, the deoptimization information contains a description of the content of the fields. When deoptimizing code, these objects are reallocated and their fields filled with the values described by the deoptimization information. The data structures for the deoptimization information are very similar to those used by RPython's tracing JIT. For every compiled Java method there is a *scope entry* for the stack and one for the local variables. The objects that are replaced by scalars are described by *object entries*, which are equivalent to RPython's virtual objects.

The papers does not describe any attempts to share the object entries and scope entries between different deoptimization safe points. This seems to not be needed in a method-based JIT compiler, because method-based JITs have fewer deoptimization points than tracing JITs. Indeed, in the evaluation presented in the second paper [17] the number of safe points is low for the benchmarks presented there, between 167 and 1512.[20] The size of the debugging information in the presented benchmarks is at most about half the size of the machine code generated.

7. Conclusion

In this paper we have concentrated on guards, an operation found in tracing just-in-time compilers and used to denote points of possible control flow divergence in recorded traces. Based on the observation that guards are a frequent operation in traces and that they do not fail often, we described how they have been implemented in the high- and low-level components of RPython's tracing JIT compiler.

Additionally we presented experimental data collected using the standard PyPy benchmark set to evaluate previous observations and assumptions about guards. Our experiments confirmed that guards are a very common operation in traces. At the same time guards are associated with a high overhead, because for all compiled guards information needs to be stored to restore the execution state in case of

[19] This optimization is now implemented in LuaJIT, at the time of writing it has not been fully documented in the LuaJIT Wiki: `http://wiki.luajit.org/Optimizations\#1-D-Snapshot-Compression`

[20] The fact that the density of safe points is low also means that the sharing approaches of this paper likely would not work well.

a bailout. The measurements showed that the compression techniques used in PyPy effectively reduce the overhead of guards, but they still produce a significant overhead. The results also showed that guard failure is a local event: there are few guards that fail at all, and even fewer that fail very often. These numbers validate the design decision of reducing the overhead of successful guard checks as much as possible while paying a higher price in the case of bailout due to having to decode a compressed state representation. The compressed state representation reduces the memory footprint of rarely used data.

Based on the observation that guard failure is rare it would be worth exploring if a more aggressive compression scheme for guards would be worth the memory saving in contrast to the increased decoding overhead. Based on the same observation we would like to explore the concept of LuaJIT's sparse snapshots and its applicability to RPython's JIT. There is an ongoing effort to replace the backend map in RPython's JIT with a simpler technique that does not require decoding the backend map on each guard failure.

Acknowledgements

We would like to thank David Edelsohn, Samuele Pedroni, Stephan Zalewski, Sven Hager, and the anonymous reviewers for their helpful feedback and valuable comments while writing this paper. We thank the PyPy and RPython community for their continuous support and work: Armin Rigo, Antonio Cuni, Maciej Fijałkowski, Samuele Pedroni, and countless others. Any remaining errors are our own.

References

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, Montreal, Quebec, Canada, 2007. ACM.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *PLDI 2000*.

[3] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: a trace-based JIT compiler for CIL. In *OOPSLA*, Reno/Tahoe, Nevada, USA, 2010. ACM.

[4] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 59–68, Vienna, Austria, 2010. ACM.

[5] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. In *PEPM*, Austin, Texas, USA, 2011.

[6] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. ICOOOLPS '11, page 9:1–9:8. ACM, 2011.

[7] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS*, pages 18–25, Genova, Italy, 2009. ACM.

[8] C. F. Bolz, A. Kuhn, A. Lienhard, N. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. Back to the future in one week — implementing a Smalltalk VM in PyPy. In *Self-Sustaining Systems*, pages 123–139. 2008.

[9] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a jitting VM for prolog execution. In *PPDP*, Hagenberg, Austria, 2010. ACM.

[10] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger. How developers use the dynamic features of programming languages: the case of Smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 23–32. ACM, 2011.

[11] 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.

[12] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL*, Salt Lake City, Utah, 1984. ACM.

[13] A. Gal, M. Franz, B. Eich, M. Shaver, and D. Anderson. Trace-based Just-in-Time Type Specialization for Dynamic Languages. In *PLDI 2009*.

[14] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. VEE 2006, pages 144–153. ACM, 2006.

[15] 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*, pages 19–28, Wellington, New Zealand, 2009. Australian Computer Society, Inc.

[16] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. PLDI '92, page 32–43. ACM, 1992.

[17] T. Kotzmann and H. Mossenbock. Run-time support for optimizations based on escape analysis. CGO '07, page 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

[18] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. VEE '05, page 111–120. ACM, 2005.

[19] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*, Monterey, California, 2001. USENIX Association.

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

[21] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI 2010*, pages 1–12, Toronto, Ontario, Canada, 2010. ACM.

[22] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *DLS*, Portland, Oregon, USA, 2006. ACM.