Table of Contents

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

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

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.

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

[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.

RPython language and its meta-tracing JIT. Based on these 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-

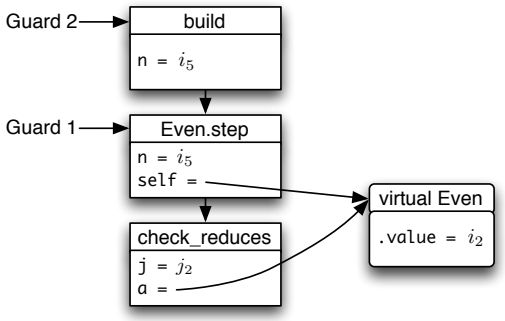


Figure 4. The resume data for Figure 3

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/`

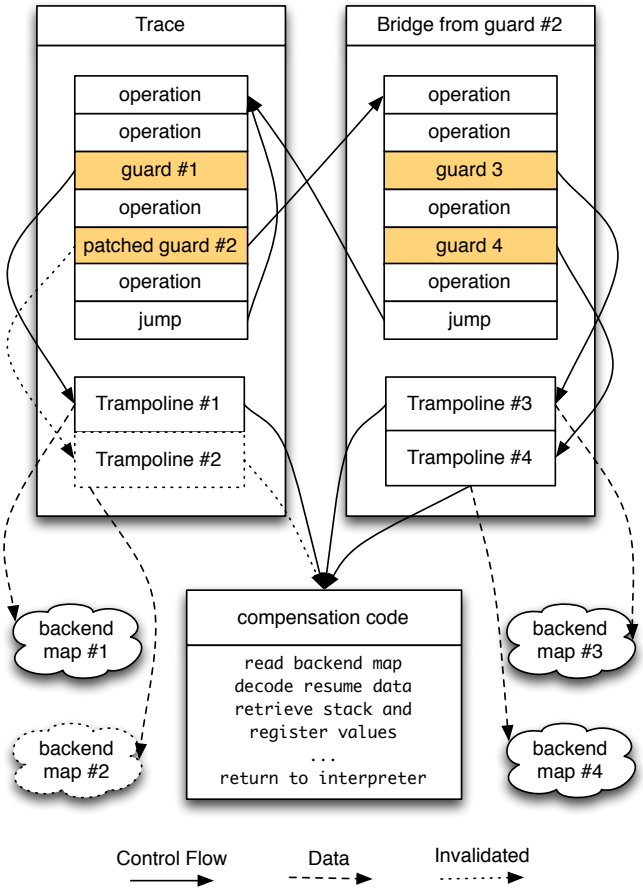


Figure 6. Trace control flow in case of guard failures with and without bridges

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.

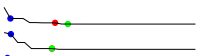
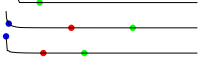
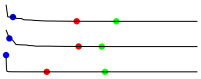
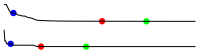
Benchmark	Sparkline	Failing	> 200 failures	50% of failures	99% of failures	99.9% of failures
chaos		3.5%	1.5%	2 ~ 0.225%	9 ~ 1.014%	11 ~ 1.239%
crypto_pyaes		3.0%	1.7%	2 ~ 0.209%	8 ~ 0.837%	8 ~ 0.837%
django		5.4%	1.8%	2 ~ 0.185%	4 ~ 0.369%	11 ~ 1.015%
go		4.0%	2.7%	18 ~ 0.060%	410 ~ 1.367%	795 ~ 2.651%
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.

S-RVM: a Secure Design for a High-Performance Java Virtual Machine

Yuval Yarom Katrina Falkner David S. Munro
The University of Adelaide
{yval,katrina,dave}@cs.adelaide.edu.au

Abstract

Reference protection mechanisms, which control the propagation of references, are commonly used to isolate and to provide protection for components that execute within a shared runtime. These mechanisms often incur an overhead for maintaining the isolation or introduce inefficiencies in the communication between the components.

This paper proposes a novel approach for component isolation that avoids runtime overheads by controlling references at compile time. We use the proposed approach to build S-RVM, a Java Virtual Machine based on JikesRVM, which enhances JikesRVM's security by isolating the VM from the application. Our experiments show that on the average S-RVM incurs no performance overhead when executing optimised code.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Design, Languages, Measurement, Performance, Security

1. Introduction

Virtual Machines (VMs) frequently need to execute several components of varying levels of trust. Examples includes mobile code e.g. applets downloaded to the browser, third party extensions to software systems and tasks in multi-tasking VMs.

VMs rely on software-based protection mechanisms to isolate trusted components from the untrusted ones. Software protection mechanisms can operate at two levels: reference protection[1] and type protection [21]. Reference protection is the ability to declare objects that are limited to a scope such that only code within that scope can name or hold a reference to these objects. Type protection, most often implemented as protection scopes, refers to the ability to allow access to members of an object based on the scope of the accessing code.

While reference protection is not as ubiquitous as type protection, extensive research into it has been done within the context of component isolation [2, 4, 10, 18] and in alias control [9, 25].

This paper focuses on isolating the application from the VM in metacircular VMs such as Singularity [2], JikesRVM [3] and JNode [1]. Metacircular VMs are implemented in the same language they target and execute within the same runtime environment they provide. As the application and the VM share the runtime environment, isolating the two is paramount for maintaining the VM's integrity. Nevertheless, the existing mechanisms for isolation are considered insufficient for metacircular VMs due to the performance overhead they introduce [8]. We argue that this overhead is avoidable and that with the right design component isolation can be provided at virtually no cost to the steady state performance.

To gain a better understanding of systems providing reference protection we introduce the concept of *zones* which are groupings of objects based on the references allowed to and from these objects. All objects in a zone share the same permissions.

In this paper we present a classification of zones based on the permitted references and show that a handful of zone types are sufficient to describe the reference protection properties of component isolation. This classification captures the salient features of the reference protection in the system and is, therefore, useful for comparing the properties of reference protection schemes.

We also identify the *sealed zone*—a zone type that can prevent external references to the components' internal data while supporting a public interface that can be accessed by other components. While sealed zones promise efficient communication through direct access, existing systems, such as the J-Kernel [18], are implemented as libraries and rely on proxy objects for en-

[1] The original term [21] use for this level is *Memory Protection*. As this term has since been overloaded we elect to use the less confusing term *reference protection*.

forcing the isolation. Consequently the J-Kernel incurs a 10% performance overhead.

We have built S-RVM—a Java VM based on JikesRVM—that uses sealed zones to isolate the VM from the application. In S-RVM the VM and the application code execute within separate *tasks*, each with its own type name space. The application can only name its own types and types that the VM exports. As types of internal VM objects are not assignment compatible with any application type and as the application cannot name these types, references from the application to VM internal objects cannot be created. Reference protection in S-RVM is, therefore, enforced at compile time and does not incur any runtime cost.

Restricting the references gives S-RVM its main advantage over JikesRVM. In JikesRVM an application has free access to the VM. A malicious application can use that access to breach the Java language security e.g. by bypassing the type safety. In S-RVM, the application only has access to the interface exposed by the VM, reducing the VM attack surface area. Hence S-RVM provides a more secure environment for running untrusted code than JikesRVM.

We have tested the performance of our system with the DaCapo benchmark suite [7]. Due to it's design, S-RVM takes slightly longer than JikesRVM to achieve optimal performance. When achieving steady-state, the mean performance over all the benchmarks in the suite is the same for both the original JikesRVM and for S-RVM.

2. Related Work

In this section we review the existing work on reference protection. Most of the research into reference protection has been done within the context of Multi-tasking VMs where it is used to provide some degree of task isolation, which can be restrictive, relaxed or anywhere along that spectrum. As most of the multi-tasking VMs are written in Java or languages very similar to it, we focus on Java in our discussion. We begin by reviewing the tools the Java language provides for component isolation and discuss their limitations. We proceed with a description of systems that provide reference protection ordered by decreasing level of isolation.

2.1 The Java Security Architecture

The Java programming language relies on three tools to create protection domains [16]. The first is the Java class loader concept [19]. One of the functionalities of the class loader is to partition the type name space. In addition to resolving the problem of name conflicts, this partitioning prevents code that has no access to a class loader from naming and using type information of types loaded by that class loader.

The second tool is the security manager. The security manager is a runtime authorisation mechanism that gets invoked whenever security sensitive operations are executed. The security manager verifies that the code has the required privilege level to execute the operation.

The third tool is the Java language type safety that ensures that malicious code cannot bypass the other two tools.

While these tools can and have been used to create a multi-tasking VM [6], three aspects of the Java security architecture render it less than ideal for isolating components. First, the isolation properties of the class loader mechanism are weak. Class loaders only protect the type information. They do not control the propagation of references to objects and do not control the use of objects through any of their supertypes (except for those loaded by the class loader). Second, the dynamic nature of the security manager introduces a runtime overhead. The reliance on examining the runtime stack for finding the calling context exacerbates the overhead. Third, the Java security manager is notoriously complex and designing a comprehensive security policy is hard.

In other words, the Java security architecture puts the onus of controlling object reference propagation on the programmer. To control propagation, the programmer needs to have an intimate knowledge of the system libraries and any other potential sharing sources as well as a good understanding of the extent of sharing incurred by disclosing each object reference. At the same time, the tools that Java provides to assist the programmer are both complex and expensive to use.

Many techniques for improving the Java security architecture have been suggested. These are divided into two main approaches: implement reference protection for complete or partial isolation of components; and provide the programmer with better control over either the propagation or the use of references. These techniques are described below.

2.2 Complete Isolation

MVM [10, 11] and JNode [1] isolate applications based on the observation that the only initially shared values in classes are the static fields, the associated `Class` object and `String` literals. By providing each application with its own set of values these systems completely isolate the applications. The exception is the sharing of strings between applications, which is supported by JNode and by early versions of MVM.

Isolation does not come without cost. In complete isolation the only way to share objects is through communication mechanisms, introducing the cost of marshalling and copying of data.

The Microsoft .Net framework [13] also provides complete isolation. Applications in .Net run within application domains, which are completely isolated from

each other. .Net supports the concept of remote types which allows applications to communicate across the application domain boundaries. Remote types are implemented as proxies that transparently marshall and send data across the communication channels. As such, they provide simpler communication semantics but do not reduce the communication overhead.

2.3 Object Sharing

Isolation with object sharing relaxes the requirements of complete isolation by allowing references from the component objects to shared objects. Component isolation is maintained by prohibiting external references to component objects.

KaffeOS [4, 5], Singularity [2, 14], XMem [26] and CoLoRS [27] provide object sharing. To enforce task isolation, KaffeOS and XMem use write barriers which introduce runtime checks for reference store operations. Singularity and CoLoRS avoid the runtime costs associated with write barriers by using a separate type hierarchy for shared objects. XMem and CoLoRS mostly rely on the OS process boundaries for protection. Reference protection in these system is used for maintaining referential integrity within shared memory sections.

Although sharing data reduces the overhead associated with marshalling objects across communication channels, it does not allow for remote method invocation. In those systems remote method invocation mechanisms are implemented on top of communication channels between applications.

2.4 Partial Isolation

Partial isolation makes a distinction between private and public objects within the component space. Cross-component references are permitted only to public objects.

Partial isolation is supported by the J-Kernel [18], where it is implemented using automatically generated wrapper objects, called capabilities, that can be shared between tasks. The advantage of this design is that it supports remote method invocation via the capabilities. However the extra processing required for creating a capabilities and converting data when invoking remote methods results in an overhead of about 10%.

2.5 No Isolation

Some multi-tasking VMs do not make guarantees in respect to isolation. Instead they provide the programmer with better tools for controlling propagation of references and for limiting the use of cross-task references.

I-JVM [15] provides initial isolation of components using the techniques described by MVM. It also provides components with access to a shared name service that can be used for publishing and accessing remote objects. While components are initially isolated, cross-component references are passed using the shared name service or other remote objects. The initial isolation provides the programmer with better controls of propagation. I-JVM, however, does not prohibit any cross-component references.

Secure Java [24] and Luna [17] do not control the propagation of references. Instead, they control the use of remote references. Secure Java uses hardware protection mechanism to restrict access to remote objects. Luna adds a remote protection scope to objects types.

2.6 Reference Protection for Alias Control

Confined Types [25] and Ownership Types [9] are both techniques for alias control that use reference protection to protect internal data structures.

Confined types are an extension of package scoped classes in Java. The type information of packaged scope classes is only accessible to code in the package. That is, code outside the package cannot extend these classes and cannot invoke methods or access fields defined in a package scoped class. Confined types extend the restrictions of package scoped classes by ensuring also that references to objects of a confined type cannot escape the package scope.

Ownership types are a method of protecting the internal representation of compound data structures. With ownership types, references can be tagged as “owned” by an object, limiting their propagation to a scope defined by the owner object.

Like the J-Kernel, confined types and ownership types allow indirect access to protected objects. Confined types are accessible through public classes in the package and owned types are accessible via the owner. Unlike the J-Kernel both confined types and ownership types rely on the static nature of the type system to avoid runtime overhead.

While both designs offer reference protection at no performance cost, the inherent limitations of the designs preclude their use for component isolation.

2.7 Summary

The different methods of providing component isolation vary in the level of isolation and the methods of providing it, but they do share one thing in common. They all report a performance overhead.

For some benchmarks in some systems the overhead can be as small as 1% [11]. For other benchmarks it can be over 20% [4]. However, the mean overhead of component isolation is always several percents.

While a few percents overhead may be an acceptable price for the better security of reference protection, we argue that this price is avoidable. We argue that the combination of a static type system and indirect access to protected objects is the key for providing reference protection at no performance cost.

3. Reference Protection

As seen in the previous section, many different approaches for reference protection have been suggested. To better understand the landscape it is useful to introduce the concept of *zones*, where a zone is a group of objects that share the same permissions for referring to and being referred to from objects outside the zone.

As it is much easier to make decisions based on groups of objects than to track permissions for each and every object, the use of zones is implicit in the design of reference protection systems. Hence, the classification we provide in this section adds clarity and allows comparing systems from across the domain. The classification identifies four types of zones that are used as building blocks for constructing the reference protection policy.

A *privileged* zone is a zone that can hold references to any other zone, regardless of the restrictions otherwise imposed on these zones. Privileged zones typically implement system functionality.

Objects in *isolated* zones can only be referenced from within the zone. External references into isolated zones are prohibited.

Shared zones are those that can have incoming references from multiple (non privileged) zones.

A *sealed* zone is a zone that can only have incoming references from a corresponding *interface* zone. References to the interface zone are typically always allowed.

Figure 1 shows the zone types and demonstrates possible allowed cross-zone references. For clarity we do not display privileged zones in this diagram.

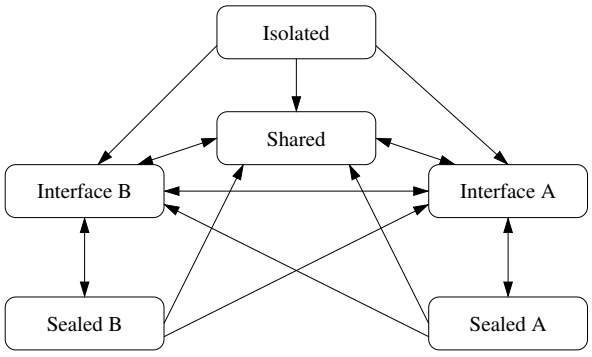


Figure 1. Non-privileged Zone Types

We apply our classification method to the systems described in Section 2 and show the results in Table 1, identifying the system, the system's name for each zone and our classification for that zone. For example, processes in Singularity [14] maintain independent heaps and do not share memory with each other. Hence, these heaps are isolated zones. Singularity processes communicate using channels, which are bi-directional message conduits. A channel can have an associated exchange heap that holds data shared by the processes at the ends of the channel, and is, therefore, a shared zone.

System	Zone	Zone Type
MVM	Isolates	Isolated
JNode	Root Isolate	Isolated (Priv.)
	App Isolates	Isolated
.Net	App. Domain	Isolated
J-Kernel	Domain Capabilities	Interface
	Domain objects	Sealed
KaffeOS	Kernel Heap	Shared (Priv.)
	Shared Heaps	Shared
	Process Heaps	Isolated
Singularity	Process	Isolated
	Exchange heap	Shared

Table 1. Zones in Multi-Tasking VMs

Sealed zones and their corresponding interface zones map naturally into a component system model where each component consists of private data, which is not accessible from other components, and a Remote Procedure Call (RPC) interface, which is used for communication between components. Under this mapping, a sealed zone encloses the private data of each component and the corresponding interface zone forms the RPC interface. The reference protection protects the private data, while RPCs are nothing more than method invocation on objects within the interface.

4. S-RVM

S-RVM is a metacircular Java Virtual Machine based on JikesRVM and designed to use reference protection with the sealed zones model to isolate the VM from the application.

The basic architecture of JikesRVM is depicted in Figure 2. The VM executes at the bottom layer. The VM provides the basic services for the Java libraries but, as most of it is written in Java, it uses some small part of the Java libraries (shown as *Core Libraries* in the diagram). The application itself executes on top of the Java Libraries.

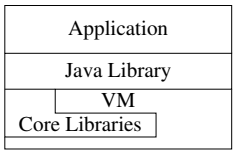


Figure 2. JikesRVM

Using the same environment for executing both the application and itself allows JikesRVM to achieve a high performance. This performance, however, comes at the price that there is no reliable way of separating VM objects from application objects [20].

The lack of clear boundaries between the VM and the application implicitly means that the application code has direct access to VM objects. As the VM maintains the type safety of the language, direct access to the VM can be used to bypass type safety. Java language security depends on type safety, hence direct access to VM objects allows applications to bypass the Java security.

Blurred boundaries between the application and the VM also cause difficulties for debugging and performance measurements. For example, without clear boundaries it is hard to collect performance data for application code only or to perform an object graph analysis for the application.

S-RVM reinstates a clear boundary between the application and the VM by treating the VM and the application as separate tasks. An interface layer is inserted between the VM and the application task. This layer provides the system services required for running a Java library. Reference protection ensures that the only VM objects accessible to the application are those defined to be in the interface layer. Figure 3 shows the main components of S-RVM.

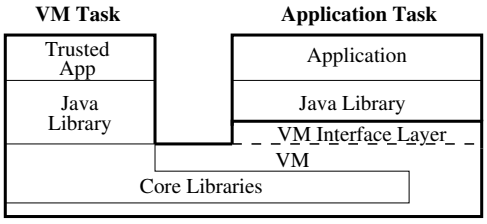


Figure 3. S-RVM

In addition to providing a clear boundary between the VM and the application, the interface layer also forms a trust boundary. The VM does not trust any code running within the application task, including the application task's copy of the Java library. Not trusting the application's library code reduces the VM attack surface area. It also decouples the Java library from the VM allowing the use of different implementations of the Java library for the VM and the application.

The rest of this section describes the implementation of reference protection and the interface layer and discusses some issues specific to securing the VM. These include the asymmetric privileges required, issues related to the trust boundary introduced in S-RVM and exception handling.

4.1 Reference Protection

The crux of separating the application and the VM lies in using a separate base class loader for each task. The base class loader in Java is responsible for loading the Java library classes. Using separate base class loaders effectively creates a separate class name space for each task.

With separate class name spaces, the type hierarchy as viewed by application code is completely distinct from the hierarchy seen by the VM code. Consequently, objects of application types are not assignment compatible with any VM type and vice versa. Thus, the separate name spaces, together with the type safety, isolate the VM from the application and lay the basis for reference protection.

In addition to the tasks name spaces, S-RVM creates a name space for shared types which is used for VM interface types. The `@Export` class annotation marks the classes that belong in the interface layer. When these classes are loaded, their names are added to the shared name space making them available to the application task.

Listing 1 demonstrates a snippet of the class `RVMType` the access to which is required for the implementation of Java reflection. The `@Remote` member annotation introduces a new protection scope by indicating which members of the exported class can be accessed from the application.

```
import org.vmmagic.mu.Export;
import org.vmmagic.mu.Remote;
@Export
public class RVMType {
  @Remote
  public RVMClassLoader getClassLoader() {
    // Implementation of getClassLoader
  }
}
```

Listing 1. An Interface Class

Importing a type is done by loading a stub corresponding to the type. Listing 2 shows the stub code corresponding to the snippet of the `RVMType` class. As can be seen in Listing 2 the stub code includes the class definition with the annotation `@Import`. As the VM only requires the member signatures when importing types, only minimal declarations of remote members are required. Stub classes are automatically generated from the exported class's class file.

```
import org.vmmagic.mu.Import;
@Import
public class RVMType {
  public RVMClassLoader getClassLoader()
        { return NULL; }
}
```

Listing 2. A Stub of an Interface Class

The presented design supports reference protection with the sealed zones model. The application cannot hold references to VM objects unless they are of types in the interface layer. Hence, the interface layer forms the interface zone in the model and other VM objects are in the sealed zone.

As presented, this design is not specific to isolating the VM from the application. The protection it provides is symmetric and the same design could be used for providing reference protection in other environments such as multi-tasking VMs or for isolating third party plugins. One of the challenges of the VM environment is the asymmetric privilege levels. This challenge is discussed below.

4.2 Privileged Access for the VM Task

The VM requires privileged access to application objects for a few special purposes, including garbage collection, passing references between application code and native code, exception throwing and array copying.

To allow passing of references between the VM and the application we add the type `MuObject` which is an unboxed reference to any object in the system. `MuObject` supports a single generic constructor that creates a `MuObject` reference from any object reference and a generic `get` method which returns the referenced object. JikesRVM uses `Object` references to refer to application objects. To avoid rewriting unnecessary sections of JikesRVM we also allow assigning any object to VM `Object` references.

References that can refer to all the objects in the system present the risk of breaching the reference protection. S-RVM does not, currently, verify that VM objects are not passed to the application using `MuObject` references. Faults in the VM may, therefore result in the application getting a reference to internal VM objects. Nevertheless, the generic nature of `MuObject.get()` introduces a dynamic type check before the referenced object can be used for any purpose, providing some level of protection against such VM faults.

Unboxed wrapper types are used to wrap arrays when these are passed to the VM `arraycopy` methods. The use of a different wrapper type for each array type allows us to keep the array element type information across the VM/application boundary and to avoid an otherwise required dynamic type check.

4.3 Creating a Trust Boundary

As discussed above, the interface layer is also a trust boundary. Unlike most Java Virtual Machines, S-RVM does not trust the Java library code used by the application. Instead, the interface layer is designed to provide a security barrier and to protect the VM from malicious applications.

Lack of trust is manifested in extra tests in interface methods. For example, to prevent the application code from using reflection on VM types the VM method `getObjectType()` which returns the type of an object is replaced with a secure version which, when invoked by the application, ensures that the object is an application object.

Lack of trust also implies that arrays cannot be shared between the application and the VM. Instead, S-RVM uses wrapper classes to provide the application with read-only access to VM arrays.

A slightly more involved consequence of the lack of trust is the handling of string objects. JikesRVM uses the `String` implementation from the GNU Classpath library. A `String` object uses an array of characters as a backing store. It also records the offset into the array and the length of the string. The contents of the backing store is considered to be constant and is only shared with code trusted to maintain this property. The backing store can, therefore, be shared between `String` objects.

Strings are frequently transferred across the boundary between the VM and the application. Common operations that manipulate `String` objects are JNI functions and creation of `String` literals during class loading. Copying `String` objects when transferring them between the application and the VM would introduce a significant overhead. On the other hand, sharing `String` objects or their backing stores would reduce isolation and require the VM to trust the application not to modify the contents of `String` objects.

To avoid copying yet maintain the safety of `String` objects, S-RVM introduces two unboxed wrappers for VM character arrays. The first, `MuCharArray`, provides read only access to the character array and is used as the backing store for `String` objects. The second, `MuWriteableCharArray`, provides write access to an underlying character array for the purpose of creating a `String` object. When a `MuWriteableCharArray` is converted to a `MuCharArray`, it is *sealed* and write permission to it is revoked, ensuring that `String` objects remain constant once created.

The S-RVM interface includes some utility methods that allow copying to unsealed `MuWriteableCharArrays` and from `MuCharArrays` using the VM implementation of the array copy functions. It also includes the `MuString` class which, like `String`, contains a `MuCharArray`, an offset and a length. `MuString` is used for packing the information about `String` objects when these are transferred between the application and the VM.

4.4 Exceptions

When Java code encounters an exceptional situation it can abort execution and *throw* an object of type `Throwable` or any of its subtypes. The VM is required

to generate and throw exceptions when certain conditions, e.g. when the application references a `null` pointer or when running out of memory, occur.

Due to the separate type hierarchies, exceptions generated in the VM task in S-RVM are not compatible with exceptions in the application task and vice versa. S-RVM combines two methods for ensuring exceptions are signaled and handled as expected.

When the exception is the result of a hardware trap, such as when it is the result of a `null` pointer reference or a division by zero, S-RVM uses an upcall to the task to generate the required exception object. For other exceptions, S-RVM wraps each remote method with an exception handler that converts the exception to the invoking task.

When S-RVM converts an exception it may lose some type information. This loss occurs because the application can declare exception types that are not recognised by the VM. S-RVM, therefore, converts application exceptions to the most specific supertype defined in the VM.

This loss of information cannot affect the VM response to the exception because the VM can only handle the exception types it recognises. If, however, the VM does not catch the exception before returning to the application or if the VM re-throws the exception, the loss of information may affect the application As all the exception conversions that occur in our benchmark do not lose type information, this is a theoretical rather than an actual problem.

To rectify the information loss, the conversion code can keep a reference to the original exception and use it instead of converting the exception back to the application. As the number of exceptions that are actually converted is very low (less than 2,000 conversions in eclipse and less than 1,000 in any of the other DaCapo benchmarks), and as these conversions only occur in the initial loading of classes, we expect changes in the conversion algorithm will only have a negligible effect on performance.

4.5 Summary

S-RVM is a proof-of-implementation of our design of component isolation using sealed zones. S-RVM represents a substantial change in JikesRVM's design and as such requires significant modifications to the software system of which we have described the main architectural changes required to implement sealed zones. However, retrofitting an existing system, enables us to make performance comparison with the underlying single-tasked VM. The results of this comparison are described in the next section.

5. Results

We have tested the performance of S-RVM on an IBM x3500 server with two quad-core Xeon E5345 processors and 24GB of RAM; running Fedora release 16. The VM was compiled using the production configuration, with edge count profiling information collected from running the DaCapo `fop` benchmark. We have retrofitted both S-RVM and JikesRVM with the `Double.toString()` implementation from the OpenJDK [22].

To measure the VM start up time we ran the time-honoured "Hello World!" program on both S-RVM and on JikesRVM 3.1.1. The JikesRVM image contains significant parts of the Java library precompiled at a high optimisation level. By contrast, the application task in S-RVM has no classes preloaded. In addition to loading the code for the application, the application task needs to load and compile those classes of the Java library that the application uses. S-RVM, therefore, takes much longer to start than JikesRVM. Running "Hello World!" in JikesRVM 3.1.1 takes 68ms. Running the same program in the S-RVM application task takes 245ms—almost four times longer.

To measure the performance of optimised code in the VM we use the DaCapo benchmark suite release 2006-10-MR2. The DaCapo suite test harness runs each test multiple times to allow adaptive compilers time to learn and adjust to the program patterns. Table 2 shows the performance of the DaCapo benchmarks in the first, third, tenth and twentieth iteration on both JikesRVM 3.1.1 and on S-RVM. For each benchmark we report the geometric mean of the results from 40 runs of the test, rounded to the nearest millisecond. For a measure of the relative overall performance, we also report the geometric mean of the running times of all benchmarks for each configuration.

Figure 4 shows the performance results for S-RVM relative to JikesRVM. The value of 100 represents the performance of JikesRVM for each benchmark scenario. As can be seen in the diagram, in the first iteration S-RVM consistently underperforms, with a mean performance indicating an overhead of over 10%. (The actual figure is around 12.54%.) However, as the number of repetitions increases, the gap closes until it disappears at 10 iterations.

We have also tested the performance of the S-RVM code without the added security of a separate task. (That is, the application code was executed within the context of the VM, as in JikesRVM.) The results, summarised in Figure 5 demonstrate a much more consistent behaviour, with an overall overhead of around 1.5%.

As S-RVM adds complexity to JikesRVM, a small overhead is expected. The results of running S-RVM without a separate task seem to match this expectation.

Benchmark	1 Iteration		3 Iterations		10 Iterations		20 Iterations	
	JikesRVM	S-RVM	JikesRVM	S-RVM	JikesRVM	S-RVM	JikesRVM	S-RVM
antlr	2,129	2,541	1,629	1,762	1,548	1,584	1,475	1,506
bloat	6,302	7,259	5,301	5,551	5,012	5,323	4,919	5,257
chart	9,373	10,083	6,444	6,353	6,383	6,214	6,403	6,217
eclipse	30,752	32,023	24,787	25,181	23,861	24,044	23,494	23,623
fop	2,202	2,600	1,636	1,694	1,546	1,533	1,501	1,498
hsqldb	2,909	3,273	1,959	2,011	1,881	1,836	1,775	1,677
jython	6,899	7,597	4,488	4,503	4,002	4,017	3,774	3,756
luindex	8,001	8,907	7,054	7,218	6,959	6,985	6,896	6,893
lusearch	2,583	2,980	1,657	1,605	1,459	1,454	1,438	1,443
pmd	5,122	5,793	4,085	4,094	3,907	3,828	3,774	3,711
xalan	2,598	2,910	1,845	1,905	1,623	1,626	1,598	1,595
mean	4,983	5,607	3,694	3,767	3,472	3,478	3,379	3,375

Table 2. Performance of JikesRVM and S-RVM

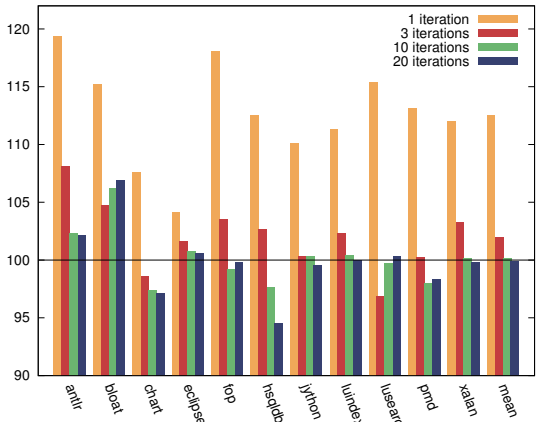


Figure 4. Relative Performance of S-RVM

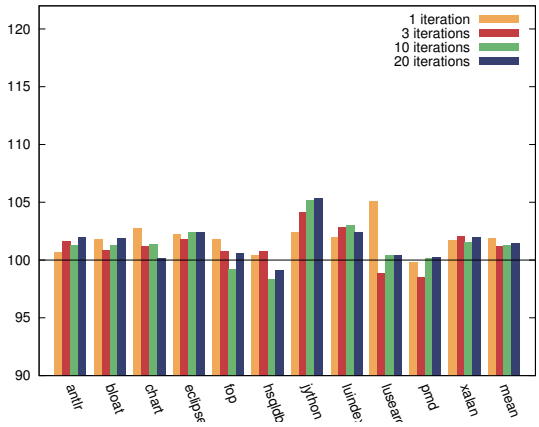


Figure 5. Performance of S-RVM Without Task Separation

The improved performance seen in most of the benchmarks when running the application in a separate task conflicts with our expectations.

We believe this improved performance is the result of having two separate copies of the Java library. With two copies of the Java library, each copy is optimised for a different workload. The library in the VM is optimised for the workload the VM generates while the library in the application is optimised for the workload of the application.

This hypothesis is supported by two observations. The first is that edge count information for some "hot" methods show significant differences between the VM and the application. E.g. DaCapo `fop` tends to invoke `AbstractMap.equals` with identical objects whereas the VM tends to use different objects (68% identical for the application vs. 28% for the VM). Edge count information improves optimisation. Consequently, significant differences in edge count profiling data are likely to translate to noticeable changes in performance.

The second observation is that the optimising compiler eliminates some code when compiling some application task methods. An example is the search of the `Atom` table in `String.intern()`, which is only required when executing within the VM.

In JikesRVM and when S-RVM is run without task separation, both these workloads share a single copy of the library. The optimising compiler cannot optimise the code to either workloads, resulting in a less than optimal performance of the Java library code.

In most benchmarks, the added performance of the specialised Java libraries is enough to more than offset the overhead introduced by the added complexity of S-RVM, resulting in a better performance for these benchmarks. The most notable exception is the `bloat` benchmark which shows a significant slowdown and the gap widens as the number of iterations increases.

Table 3 shows the minimum heap size required for executing the DaCapo benchmarks on JikesRVM and on S-RVM. S-RVM incurs a fairly constant overhead of about 8MB. While the relative overhead is significant, especially for the smaller tests, its absolute value does not seem to change much between the tests. With current memory sizes an overhead of 8MB is not expected to be significant except for the most extreme scenarios.

Benchmark	JikesRVM	S-RVM	Overhead
antlr	22MB	30MB	8MB (36%)
bloat	40MB	45MB	5MB (13%)
chart	36MB	45 MB	9MB (25%)
eclipse	60MB	69MB	9MB (15%)
fop	31MB	40MB	9MB (29%)
hsqldb	102MB	110MB	8MB (8%)
jython	35MB	44MB	9MB (26%)
luindex	24MB	31MB	7MB (29%)
lusearch	45MB	53MB	8MB (18%)
pmd	37MB	45MB	8MB (22%)
xalan	47MB	54MB	7MB (15%)
Average			7.9MB

Table 3. Memory Footprint of S-RVM

6. Conclusion

This paper addresses the issue of the overhead incurred by controlling reference propagation for isolating components. We present a study of zone types used in reference protection and propose the use of sealed zones for achieving isolation. We validate the proposed type system design by using it to implement S-RVM.

S-RVM is a proof-of-implementation Java VM which provides better security properties than JikesRVM, on which it is based. S-RVM segregates application objects and controls references between them and VM objects, thereby restricting application access to internal VM data structures. We measure the performance of S-RVM with the DaCapo benchmark suite and demonstrate that on the average S-RVM incurs no performance overhead over JikesRVM. These performance results confirm that the design works and that it works well.

7. Future Work

Rather than replicating library code used by the VM and the application, it may be possible to share the classes metadata, including both the byte code and the compiled code. Techniques for class sharing have been investigated in the past [11, 12]. Applying these and similar techniques to S-RVM can reduce the memory footprint and can provide pre-compiled library code to the application, reducing the application startup time. To provide the steady-state performance, "hot" methods would still need to be specialised and their code cannot be shared.

S-RVM runs two separate components—the VM and the application. Two natural extensions to this model are supporting multiple components and multi-tasking.

At this stage, a rudimentary support for multi-tasking is available. Initial performance evaluation indicates that While some parallel benchmarks perform reasonably well, in most cases multi-tasking in S-RVM is less efficient than executing multiple VMs. As the scheduling and memory management subsystems in JikesRVM are tuned for executing a single application, this result is to be expected. Past research [23] suggested algorithms for multi-tasking memory management which may alleviate the problem.

Resource accounting and task termination, are not addressed by the current implementation. Accounting for I/O and time resources is orthogonal to memory isolation. As the type information of objects in S-RVM indicate the task the object belongs to, we believe that the work we have done will facilitate the accounting of memory resources. Further research is required to validate this.

Task termination is a more difficult issue in the framework we have chosen. The main limitation is that JikesRVM does not, currently, support class unloading. Each task loads a significant number of classes. Without class unloading a terminating task would leave these classes in the system. Each of these classes consumes resources. Hence the total number of tasks that could, potentially, run on JikesRVM and derivative systems is limited.

Another issue that will need addressing is the conversion of legacy software. The level of effort required will depend to a large extent on the design of the specific legacy software. Monolithic software which lacks clear boundaries will, probably, require a major reengineering effort. On the other side of the spectrum, software designed for a distributed environment and which uses well defined explicit communication mechanisms might only require replacing the implementation of these mechanisms.

It would be interesting to see how much effort is required for converting applications built with the OSGi framework. Some reengineering would, probably, be required, but conversion may be facilitated by the use of automated tools.

8. Acknowledgements

This work was sponsored in part by the Defence Science and Technology Organisation, Australia.

References

[1] Jnode. `http://www.jnode.org/`.

[2] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *MSPC '06: Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 1–10, New York, New York, United States, 2006. ACM.

[3] B. Alpern, J. J. Barton, S. Flynn-Hummel, T. Ngo, J. C. Shepherd, C. R. Attanasio, A. Cocchi, D. Lieber, M. Mergen, and S. Smith. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 314–328, Denver, Colorado, United States, Nov. 1999.

[4] G. Back and W. C. Hsieh. The KaffeOS Java runtime system. *ACM Transactions on Programming Languages and Systems*, 27(4):583–630, July 2005.

[5] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, United States, October 2000.

[6] D. Balfanz and L. Gong. Experience with secure multiprocessing in java. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS '98)*, pages 398–405, Amsterdam, Netherlands, May 1998.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 169–190, Portland, Oregon, USA, Oct. 2006. ACM Press. ISBN 1-59593-348-4.

[8] S. M. Blackburn, S. I. Salishev, M. Danilov, O. A. Mokhovikov, A. A. Nashatyrev, P. A. Novodvorsky, V. I. Bogdanov, X. F. Li, and D. Ushakov. The Moxie JVM experience. Technical Report TR-CS-08-01, Australian National University, Department of Computer Science, May 2008.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *ACM SIGPLAN Notices*, 33 (10):48–64, 1998. ISSN 0362-1340.

[10] G. Czajkowski. Application isolation in the Java Virtual Machine. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 354–366, Minneapolis, Minnesota, United States, Oct. 2000.

[11] G. Czajkowski and L. Daynés. Multitasking without comprimise: a virtual machine evolution. *SIGPLAN Not.*, 36(11):125–138, 2001. ISSN 0362-1340.

[12] G. Czajkowski, L. Daynés, and N. Nystrom. Code sharing among virtual machines. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP02)*, pages 155–177, Malaga, Spain, June 2002.

[13] *Standard ECMA-335: Common Language Infrastructure (CLI)*. ECMA, fourth edition, June 2006.

[14] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. *SIGOPS Operearing Systems Review*, 40 (4):177–190, 2006. ISSN 0163-5980.

[15] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a Java virtual machine for component isolation in OSGi. In *the 39th International Conference on Dependable Systems and Networks (DSN 2009)*, pages 544–553, Estoril, Portugal, June 2009. IEEE Computer Society.

[16] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. pages 103–112.

[17] C. Hawblitzel and T. von Eicken. Luna: A flexible Java protection system. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 391–401, Boston, Massachusetts, United States, December 2002.

[18] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 259–270, June 1998.

[19] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 36–44, Vancouver, British Columbia, Canada, 1998. ACM. ISBN 1-58113-005-8.

[20] Y. Lin, S. M. Blackburn, and D. Frampton. Unpicking the knot: Teasing apart vm/application interdependencies. In *VEE '12: Proceedings of the 2012 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 181–190, London, United Kingdom, 2012. ACM. ISBN 978-1-60558-375-4.

[21] J. H. Morris, Jr. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, January 1973.

[22] Oracle Corporation. OpenJDK. `http://openjdk.java.net/`.

[23] S. Soman, C. Krintz, and L. Daynès. Mtm2: Scalable memory management for multi-tasking managed runtime environments. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP08)*, pages 335–361, Paphos, Cypress, July 2008. Springer-Verlag.

[24] L. van Doorn. A secure Java virtual machine. In *Proceedings of the USENIX Security Symposium*, pages 21–35. USENIX Association, 2000.

[25] J. Vitek and B. Bokowski. Confined types in Java. *Software–Practice and Experience*, 31(6):507–532, May 2001.

[26] M. Wegiel and C. Krintz. XMem: Type-safe, transparent, shared memory for cross-runtime communication and coordination. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 327–338, New York, NY, USA, June 2008. ACM Press.

[27] M. Wegiel and C. Krintz. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'10)*, pages 223–240, Reno/Tahoe, Nevada, USA, Oct. 2010.

Bypassing Portability Pitfalls of High-level Low-level Programming

Yi Lin, Stephen M. Blackburn
Australian National University
Yi.Lin@anu.edu.au, Steve.Blackburn@anu.edu.au

Abstract

Program portability is an important software engineering consideration. However, when high-level languages are extended to effectively implement system projects for software engineering gain and safety, portability is compromised—high-level code for low-level programming cannot execute on a stock runtime, and, conversely, a runtime with special support implemented will not be portable across different platforms.

We explore the portability pitfall of high-level low-level programming in the context of virtual machine implementation tasks. Our approach is designing a restricted high-level language called RJava, with a flexible restriction model and effective low-level extensions, which is suitable for different scopes of virtual machine implementation, and also suitable for a low-level language bypass for improved portability. Apart from designing such a language, another major outcome from this work is clearing up and sharpening the philosophy around language restriction in virtual machine design. In combination, our approach to solving portability pitfalls with RJava favors virtual machine design and implementation in terms of portability and robustness.

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

General Terms Design, Languages

Keywords Virtual machine, Restricted language, Portability, High-level low-level programming

1. Introduction

Current hardware trends are increasingly exposing software developers to hardware complexity. Novel techniques such as multicore and heterogeneous architectures increase hardware capacity, but also leave programmers a list of challenges if they wish to fulfill the hardware's potential. Dealing with complex hardware increases the difficulty of systems programming. In the meantime, the complexity of system software grows in pace with hardware evolution. With the increasing software complexity, it is even harder to retain correctness, security and productivity.

Modern high-level languages are widely used for application programming for the assurance of correctness and security as well as boosting productivity. High-level languages provide type-safety, memory-safety, encapsulation, and strong abstraction over hardware [12], which are desirable goals for system programming as well. Thus, high-level languages are potential candidates for system programming.

Prior research has focused on the feasibility and performance of applying high-level languages to system programming [1, 7, 10, 15, 16, 21, 22, 26–28]. The results showed that, with proper extension and restriction, high-level languages are able to undertake the task of low-level programming, while preserving type-safety, memory-safety, encapsulation and abstraction. Notwithstanding the cost for dynamic compilation and garbage collection, the performance of high-level languages when used to implement a virtual machine is still competitive with using a low-level language [2].

Using high-level languages to architect large systems is beneficial because of their merits in software engineering and safety. However, high-level languages are not a perfect fit for system programming. In order to effectively undertake system programming tasks, *extensions* and *restrictions* are two essentials of high-level low-level programming — both leave unsolved challenges.

Extensions cause portability pitfalls. Portability pitfalls of high-level low-level programming include *poor hardware portability* (i.e., low-level code being unable to run on different processors) and *poor portability of programs between different runtimes*. High-level languages (HLLs) are designed to abstract over low-level details, and most of them do not provide necessary semantics for low-level operations, which is a key requirement in system programming projects. In order to undertake a low-level programming task, high-level languages need to be extended and require special support from the runtime for those extensions. This leads to the fact that VM components written in the extended HLL cannot execute on a stock runtime, and, conversely, a runtime with special support implemented will not be portable across different platforms. Both break portability.

These portability pitfalls limit code reusability of high-level low-level programming. Efficient implementation of modern language runtimes requires experts from different areas, such as memory management, concurrency, scheduling, JIT compilation. This leads to a trade-off between the high cost of hiring a group of specialists and the risk of failure for lacking expertise. One possible solution to this tension is to encourage reusability. However, when the implementing language cannot execute with a proper hosting runtime on the target platform, reusability is difficult to achieve — any given runtime that wishes to host high-level code for low-level programming needs to be modified to support the new semantics, otherwise the newly introduced low-level semantics need to be carefully dealt with in other ways [13, 14]. Both involve non-trivial work for each porting.

Low-level languages (LL languages) do not have such issues. Low-level languages such as C/C++ usually have available compilers across most target platforms. Thus, a bypass approach of

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

Reprinted from VMIL'12, [Unknown Proceedings], October 21, 2012, Tucson, Arizona, USA, pp. 1–8.

translating a HLL into a LL language could be one possible way to solve this pitfall. However, generally a LL language bypass is not easy to achieve — a HLL's precise exceptions and dependence on its standard libraries are hard to map into a LL language, and efficient dynamic dispatch requires non-trivial work when targeting a non-OO LL language such as C.

Restrictions lack of definitions. System programming with a HLL also relies on restrictions for performance-critical operations or avoidance of possible program failure. We observed that some VM components are written by following strict restrictions. Restrictions include omitting high-level language features that are unnecessary or problematic for certain contexts of low-level programming. These restrictions bring the HLL closer to a LL language. This justifies the possibility of translating from the restricted HLL to a lower-level language to solve portability pitfalls. Currently such restrictions do not have clear definitions in the program, and are achieved by careful ad hoc hand coding. Explicit definitions of the restrictions and automatic checking are more principled and more robust.

Thus, this paper focuses on two topics: *a*) high-level language restriction in system programming, and *b*) translation from restricted high-level language to LL language. These two topics are independent but quite coherent in our context: the natural existence of language restriction leads to the possibility of our HLL-to-LL bypassing, which further provides a solution to portability pitfalls of high-level low-level programming.

In this paper, we first discuss the important concerns in our design of RJava and the proper position of restriction within a whole system programming project, based on some observations we made on an existing high-level low-level programming project, Jikes RVM [1]. Then we propose an explicitly-restricted language called RJava with proper low-level extensions, and use MMTk [5] as an example to show how the elements of RJava are used in practice. We finish by discussing key elements of a low-level language bypass for RJava which is currently under development.

The contributions in this paper are three-fold: 1) identifying the motivation and requirement for a well-defined restricted high-level language for virtual machine implementation use, 2) sharpening the philosophy around language restriction in virtual machine implementation, and 3) designing a restricted language which inherits benefits from high-level languages, but supports flexible restriction model and also allows low-level language bypass for improved portability.

2. Design Concerns of RJava

In this section, we tidy up our approach to language restriction and the proper position of restriction within a whole system programming project, which needs to be thoroughly thought through.

2.1 HLL Restriction in System Programming

Examining the rationale for HLL restrictions in system programming helps proper definition of such languages. Our approach is based on some important observations. We made those observations on Jikes RVM as an example of system programming with a HLL. These observations further justify that language restriction naturally exists in high-level low-level programming and our approach of formalizing and exploiting existing restrictions to favor a low-level bypass for improved portability is reasonable and will not be a regression in the term of language benefits.

Restrictions exist for performance and correctness. A general understanding of programming language restriction is that languages are restricted by omitting features because they are too complex or because some programmers have used them to write bad code [8]. *Language restrictions for system programming exist for correctness and performance.*

Correctness is one challenge for engineering complex system projects. The use of a HLL introduces much better software engineering, such as abstraction, a strong type system and automatic memory management, which promotes correctness to a more manageable level. However, the correctness of a VM implemented using a HLL still needs careful consideration, especially in metacircular cases when using a HLL. When implementing a language in the same language, one pitfall threatening correctness is infinite regress. The code to support language feature X needs to avoid using the feature X itself, otherwise that code would recursively invoke itself and be unable to finish. For example, the scheduler code needs to generally avoid any language feature regarding threading and scheduling, but instead use more basic primitives such as locking to fulfill its function. Another example is the memory manager, whichh provided the imputus for RJava. The memory manager has to avoid triggering object allocation during allocation code, or triggering another garbage collection during garbage collection, so primitives to operate on raw memory have to be added to the HLL and used to implement the memory manager. The lessons here are that a HLL for VM implementation has different restrictions when applied to different scopes. Our initial focus for RJava was the scope of implementing a portable memory manager, however, we generalize our approach to be as flexible as possible so that RJava can be adapted and used in other scopes with trivial effort.

Performance is critical in systems programming. This is probably the most powerful argument as to why people stick with lower-level languages for such tasks. However, using a HLL to implement a VM can achieve a very compatible performance with proper restrictions. For example, dynamic dispatch is one important feature of object-oriented languages that incurs considerable overhead, and its cost is measured to be as high as 13% of instructions in extreme circumstances [9]. Thus, dynamic dispatch is carefully avoided by restricting the syntax in the fast path of MMTk, the most frequently executed code and the most performance-critical region.

Understanding the reasons why restrictions exist helps correctly define the restrictions. Both reasons suggest that in VM implementation, language restrictions depend on the context and scope where the restricted variant is applied. Clear language restriction relies on clear scope definition, as will be discussed later in this section.

Restrictions reduce benefits from HLLs. Generally, high-level languages provide type-safety, memory-safety, encapsulation, and abstraction. However, some of the benefits disappear when the level of restriction increases. For example, in the most restricted part of Jikes RVM, which is MMTk, garbage collection is forbidden. In this situation, the runtime is no longer able to ensure the memory safety of MMTk, but leaves it to the programmer and static analysis. Similarly, in MMTk, type safety is limited to static checks – dynamic loading is forbidden, virtual dispatch and type casting are carefully avoided in the fast path and its correctness can only be statically checked. This suggests that, under performance-critical and correctness-critical circumstances where very strict restrictions have to be applied, the benefits of a HLL are reduced, and the benefits are principally static, i.e. type-safety and memory-safety at the source code level, and encapsulation and abstraction as software engineering tools. A restricted HLL still has clear advantages over low-level languages.

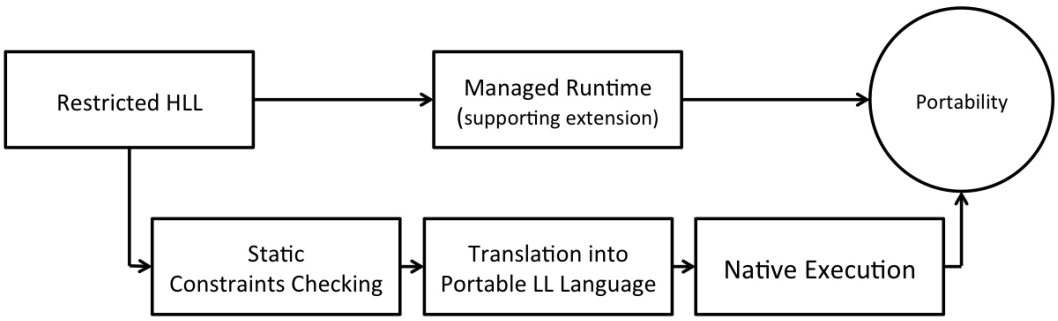


Figure 1: An illustration showing our bypass approach for portability issues.

Occurrences of	MMTk	Baseline Compiler	Rest of Jikes RVM	Eclipse (comparison)
'new' statements	0.59%	0.86%	2.40%	4.47%
'throws' declarations	0	0	0.21%	1.33%
Library imports	0	0.03%	0.40%	0.82%
Lines of Code	29933	17762	113359	-
Level of Restriction	From most restricted to not restricted.			

Table 1: Language restrictions in different scope of Jikes RVM (occurrences per LOC).

Restrictions are only applied to a limited scope. Restriction is essential for correctness and performance in system programming with HLLs, but different levels of restriction degrade high-level languages to different extents. Thus, one principle for system programming with HLLs is to minimize the scope where very strict restrictions are needed so to maximize the benefits from HLLs [11]. Table 1 reflects good design within Jikes RVM: the most restricted Java variant is used in a relatively small scope while the majority of the project is loosely restricted. Thus, heavier restrictions affect a small part of the system and do not detract the benefits of a HLL in other parts of the system. However, the strictly restricted scope (including MMTk and the baseline compiler) still has 47K LOC, which is important and large enough that is worth careful consideration.

2.2 Expressiveness vs. Restrictions

Higher-level languages are more expressive than low-level languages. Java, for example, is considered to have a 2.5× 'statement ratio' compared to the C language (i.e., on average, one Java statement needs 2.5 C statements to achieve the same function [17]). Restrictions to HLL syntax reduce expressiveness. In the limit, a restricted form of HLL that discards all features that C does not support will have a trivial mapping to C syntax. Such extreme restriction would favor our LL language bypass, but this is definitely not desirable. In contrast, if the language is minimally restricted, the expressiveness is maximally conserved, but the LL language bypass would be more difficult to achieve — Java's precise exceptions and dependence on its standard libraries are hard to map into a LL language, and efficient dynamic dispatch requires non-trivial work when targeting a non-OO LL language such as C. Thus, there is always a trade-off between expressiveness and how restricted the language is.

We resolve the trade-off with a simple principle: *we do not introduce more restrictions than necessary*. In addition to the two necessary reasons that restrict languages in VM implementation — correctness and performance — we have to put mappability to LL languages into consideration in our bypass approach. The language has to be restricted due to existing requirements on correctness and performance, and it also needs to be further restricted to adapt to LL language translation. We confine this set of restrictions to the minimum.

2.3 RJava in Different Scope

Ideally we want RJava to be a fixed language with constant restrictions so that we can use RJava to implement VM components where restrictions and portability are desired while we are able to use normal Java to implement the rest of a VM. However, this is not the case. 1) Restrictions are different among different VM components. This is mainly for a metacircular VM implementation. For example, a metacircular implementation of memory manager disallows object allocation during allocation and reclamation so that any language syntax that would introduce an object allocation is forbidden, including the use of libraries and the creation of exceptions. A scheduler, on the other hand, does not necessarily have any restriction regarding object allocation, but needs careful restriction around threading and synchronization. Thus, different components require different restrictions, and attempts to generalize restrictions among components would suppress expressiveness. 2) Restrictions are still different within one VM component. As explained before, a performance-critical scope needs more strict restrictions to remove any possible performance degradation.

As a result, instead of trying to define a universal set of restrictions that would be adaptable for general VM components, we define RJava with a set of fixed restrictions that favors easy mapping to a LL language which our frontend is able to translate to allow bypass. Also, RJava is designed to include a set of optional restriction rules that programmers can choose from to shape their own restriction ruleset for certain components. The RJava constraint checking tool processes each restriction rule in the ruleset and ensures code compliance.

3. Concrete RJava Language

The previous section discussed important concerns that affect our design of RJava. In this section, we present this restricted language

with its key elements and a concrete example of how RJava, the restricted language motivated by MMTk code, is re-adapted to MMTk and helps its robustness and portability.

3.1 Key Language Elements

RJava is a restricted subset from the Java language. It inherits the Java language syntax except that which is restricted. Extensions and restrictions are two major parts of high-level low-level programming, thus they are naturally two key elements in the RJava language.

The org.vmmagic Extension

The `org.vmmagic` extension is described in the paper "Demystifying Magic: High-level Low-level Programming" [12]. RJava takes the advantages of the existing `org.vmmagic` package for low-level semantics.

Most elements and ideas in `org.vmmagic` remain untainted when adopted by RJava. These include unboxed types and related intrinsic operations. However, some compiler 'pragmas' that are referred to as 'semantic regimes' are reconsidered and reconciled into RJava's restriction model. One example is `@Uninterruptible`. All the MMTk classes used to be described as `@Uninterruptible` to disallow garbage collection and thread switching. In RJava, `@Uninterruptible` is considered as a restriction rule, and can be integrated with other restriction rules to form a ruleset for MMTk. Another example is that some compiler intrinsics such as `@NoBoundsCheck` are categorized as restriction rules to coordinate with the content of this paper, since they restrict language run-time features. We now introduce the idea of restriction rules and rulesets.

Restriction Rule and Ruleset

Restriction rules and rulesets are fundamentals for RJava. We define restriction using Java's annotation syntax. Each restriction becomes a restriction rule, and is marked with the `@RestrictionRule` annotation for documentation. A restriction ruleset consists of different restriction rules or rulesets. This model brings some rigor to the definition to the restrictions and allows automatic checking. Figure 2 shows those elements with code examples.

`@RJavaCore` is a predefined ruleset that all RJava code should obey. The core ruleset contains restrictions to some language features that are infrequently used in VM implementation and also cannot be easily mapped to low-level languages. The ruleset suggests a minimum restriction to enable a feasible low-level language bypass while preserving expressiveness of the HLL. `@RJavaCore` also indicates language features that our frontend translator does not support, thus it must be contained by any user-defined ruleset for RJava.

We also provide different restriction rules with RJava. They are not included in `@RJavaCore` and their semantics are acceptable by the RJava frontend. Those restrictions can be used to aggregate user-defined rulesets and are essential to ensure correctness and performance for specific scopes. They can also be used solely to mark any code to indicate restrictions and also indicate a requirement for static constraint checks. However, defining a restriction ruleset specific to a certain scope is preferred than using scattered restrictions. It is best to have a 1-to-1 mapping between ruleset and scope wherever restrictions are needed. This design favors flexibility and allows clear definition of restricted scopes with certain rules.

In the next subsection, we give an example of how RJava restrictions are adapted in MMTk.

```
1 @RestrictionRule
2 public @interface NoDynamicLoading {
3 }
```

(a) An example of restriction rule.

```
1 @RestrictionRuleset
2
3 @NoDynamicLoading
4 @NoReflection
5 @NoException
6 @NoCastOnMagicType
7 ...
8 public @interface RJavaCore {
9 }
```

(b) RJava core restriction ruleset.

```
1 @RJavaCore
2 public class AnRJavaProgram {
3 ..
4 }
```

(c) RJavaCore clearly defines restrictions in a scope.

Figure 2: RJava restriction rules and rulesets.

3.2 A Concrete Example: MMTk in RJava

Though restrictions in RJava are inspired and motivated by the restricted coding patterns in MMTk, we design RJava to be a more flexible and general restricted language for implementing VM components. In this subsection, we show how RJava is applied to a specific scope (MMTk) to restrict its syntax and help with software engineering.

One important principle when coding with RJava is to map restriction rulesets to scopes. MMTk itself is a well-contained scope, thus we need a corresponding ruleset `@MMTk` to clarify the restrictions. Besides `@RJavaCore`, other restriction rules have to be carefully identified.

The memory manager fulfills two main tasks: object allocation and object reclamation. One obvious restriction for a metacircular implementation of a memory manager is to disallow object allocation in its own code during execution—otherwise, triggering object allocation in an allocating procedure would invoke another allocating procedure and triggering object allocation in a garbage collection would fail and invoke another garbage collection. We use the rule `@NoRunTimeAllocation` to describe this restriction. Object allocation in class static initializers and constructors (including methods used only by them) is allowed, since they can only be executed during the VM build process where object allocation is safe. The `@NoRunTimeAllocation` rule ensures that no **`new`** statements appear in places outside static initializers and constructors. This rule also implies two other restriction rules, `@NoException` (which is already included in `@RJavaCore`) and `@NoClassLibrary`. Using class libraries may introduce unexpected object allocation at run-time, since their implementation varies. It was possible to implement MMTk without using any class library classes[1], so we retain this restriction in the `@MMTk` ruleset.

Another restriction is `@Uninterruptible`. This annotation is inherited from the `org.vmmagic` package and we consider it to be a restriction rule. It informs the runtime to avoid triggering thread switching and garbage collection in certain scopes, and also to omit

[1] Use of Java's built in `String` and `Array` types is not restricted. However, the `@NoRunTimeAllocation` rule prohibits dynamic allocation of `Arrays` and `Strings`. This also implies a prohibition of `String` concatenation.

```
@RestrictionRuleset

@RJavaCore
@NoClassLibrary
@NoRunTimeAllocation
@Uninterruptible
public @interface MMTk {
}
```

(a) @MMTk restriction ruleset to map MMTk scope.

```
@RestrictionRuleset

@MMTk
@NoVirtualMethods
public @interface MMTkFastpath {
}
```

(b) @MMTkFastpath restriction ruleset to map fast path subscope of MMTk.

```
@RestrictionRuleset

@MMTkFastpath
@NoPutfield
@NoPutstatic
public @interface WriteBarrier {
}
```

(c) @WriteBarrier restriction ruleset to map write barrier code in the fast path.

```
@MMTkFastpath
public class GenMutator {
  ...

  public Address alloc() { ... }
  // no runtime alloc
  // virtual methods, has to be overridden

  @WriteBarrier
  public final void objectReferenceWriteBarrier() { ... }
  // no putfield, no putstatic on its own fields
  // non-virtual methods, thus no dynamic dispatching
}
```

(d) Restrictions ensure correctness and performance of the fast path.

Figure 3: MMTk with RJava.

emitting any code during code generation that would trigger thread switching or garbage collection.

The restriction ruleset for MMTk is showed in Figure 3a.

Subscope: MMTk Fast Path

The design of MMTk makes heavy use of the fast/slow path idiom. A fast/slow path idiom is a diamond-shaped control flow graph where the expected case is to do quick checks or operations to confirm a result. When some portion of the fast path fails, control transfers to the slow path that covers all remaining cases [19]. Take allocation in MMTk for example: The allocator's fast path tries to allocate space from its thread-local buffer. When the thread-local buffer is consumed, control is handed to the slow path where the allocator will acquire space from global memory and synchronization is needed. If the slow path still fails, a garbage collection will be triggered. The ratio that control falls into the slow path is typically 0.1% in MMTk allocation [6]. Thus, the fast path is the most performance-critical subscope in MMTk. MMTk forces all the fast path code to be inlined into its context to eliminate method invocation overhead, but also restricts syntax for performance improvement.

In the coding of the MMTk fast path, another restriction rule is carefully applied to minimize the performance overhead. The code avoids the possibility of dynamic dispatch by declaring all of its methods as non-virtual methods. In the fast path, all non-static non-private methods are either overridden or declared as 'final', thus there are no virtual methods and no dynamic dispatch in the fast path. We use `@NoVirtualMethods` to describe this restriction. We build the `@MMTkFastpath` ruleset based on `@MMTk`. Figure 3b shows the `@MMTkFastpath` ruleset.

Besides performance, correctness restrictions need to be reconsidered for the fast path. MMTk's fast paths include write/read barrier code. Barriers are a powerful tool to monitor mutator actions by tracking operations on objects. Take the write barrier for example: Because of metacircularity, the write barrier itself needs to avoid using `putfield` or `putstatic` on its own object fields, otherwise it leads to an infinite regress. We use `@NoPutfield` and `@NoPutstatic` to describe these restrictions. To avoid being overly restrictive, we form the `@WriteBarrier` ruleset that will be used only on write barrier code in the fast path. `@WriteBarrier` contains the `@MMTkFastpath` ruleset, and the two specific restriction rules stated above. Figure 3c shows this restriction ruleset.

Figure 3d gives an outline of `GenMutator` as parent of all mutators for generational garbage collection algorithms to show how these rulesets are used to properly restrict language semantics in MMTk, and help ensure its correctness and performance.

4. Current Work: An RJava to LL Language Bypass

Formalizing the restriction rules is one significant aspect for designing the RJava language. In this section, we introduce our current work, the RJava to LL language translation toolchain that materializes the bypass approach (see Figure 1).

The toolchain to enable RJava to LL language bypass includes a static constraint checking tool that ensures compliance to the declared restriction rules, a frontend that takes RJava as source and produces code in LL language, such as C/C++, and a backend that compiles LL language to native code.

4.1 Static Constraint Checking Tool

The static constraint checking tool examines the compliance of code with restriction rules declared on them. It can be used as one part of our LL language bypass toolchain, and can also be used as an independent tool to check original RJava code to detect any violation of restrictions.

There are existing tools for Java syntax checking, such as PMD [20]. These tools parse Java syntax and perform rule-based style checking. But they do not fit our requirements. To be able to precisely examine restrictions defined in RJava, our static constraint checking tool needs to be able to process not only at the syntactic level but also at the more complex semantic level. For example, `@NoRunTimeAllocation` requires that no object allocation appear outside static initializers, constructors or any methods only called by them. Thus, this implies requirements at a semantic level, such as the relationship between methods (call graph) that existing syntax checking tools are not able to deal with.

We are building our static constraint checking tool based on the Soot framework [24]. Soot is a Java optimization and static analysis framework, and provides various forms of analyses. We have built a prototype that is able to validate the `@NoRunTimeAllocation` restriction. What remains to be done is expanding the rule/ruleset checking to cover all of the other RJava restrictions.

4.2 Frontend: RJava to LL Language

The frontend is the most critical part in our toolchain to translate RJava into a low-level language. There are several important tasks that the frontend has to complete, besides simple syntax mapping:

Implementing compiler intrinsics. Intrinsic methods such as `Address.loadByte()` and compiler pragmas such as the `@Inline` annotation do not have concrete implementations in RJava, but rely on support from the managed runtime. Since our bypass approach removes the existence of the runtime, compiler intrinsics need to be implemented in the frontend. We expect that the generated code is plain low-level language. For example, `loadByte()` would become a pointer dereference and `@Inline` would become an inline keyword in the target language.

Unboxing magic types. The `org.vmmagic` package we use in RJava introduces unboxed types, such as `Address` and `Offset`. Java types are by default 'boxed' with additional information such as header, virtual method table, etc. However, this package makes the assumption that those magic types are specially treated as unboxed types by the runtime, thus they are not real objects at the run-time. This assumption prevents a memory manager creating objects when it operates on addresses and object references during object allocation requests. It also makes retrieving actual values of such types significantly more efficient. This assumption is equally important when RJava is translated into a LL language for the same reasons. Unboxing is needed during translation to convert such magic types into pointers that the target language supports.

Removing dependencies on the Java class library. Even without explicit use of the Java class library, Java syntax is related to its class library, such as implicit support from `String`, `Array` and the common superclass `Object`. We require that the generated LL language has no dependency on the Java runtime. Thus, the frontend needs to remove all dependencies on the Java class library, and replace implicit uses with syntax and features from the target language.

Converting object-oriented syntax (optional). This task is only essential when our frontend targets a non-object-oriented LL language. In such cases, the OO syntax needs to be removed during the translation. Generally this is possible since RJava is restricted to forbid some dynamic features of object-oriented languages. But this still needs careful consideration regarding performance.

Those tasks are sensitive and specific to the source language, i.e. RJava, and the target LL language. Thus, we do not aim for our frontend to be a flexible framework that could produce code for different targets. The C language is a suitable LL language to target. It is the dominant language in system programming, and it is also portable. However, our first implementation (under development) does not target C. This is for two main reasons. First, we are not aware of any existing Java-to-C translator for general use that we can base our implementation on. Existing translators are too fragile and too specific to their own projects. If C were our target, we would have to build such a translator from scratch. Second, C is not object-oriented: translation from RJava to C would require more development effort, and naive object-oriented syntax conversion may result in inefficiency in the target performance.

We choose C++ as our target. C++ is portable and has a similar syntax to Java, thus mapping from Java into C++ is easier. We implement our frontend by modifying J2C [25]. J2C is a translator that converts Java code into C++. The tasks listed above need to be implemented in J2C, so it can properly handle semantics specific to RJava. This part of the work is our focus, and is under development.

4.3 Backend: LL Language to Executable

Our frontend translates RJava into plain C++ syntax. RJava's bypass approach does not make any particular assumption about the backend. A general compiler that takes our frontend target (i.e., C++) as the source language can fit well in our bypass toolchain.

5. Future Work: Bootstrapping Java VM with RJava

The flexible design of RJava encourages its use for different components in VM design. Besides the memory manager, our first chosen component, the interpreter could be another candidate for implementation in RJava. Table 1 showed that the interpreter/baseline compiler has a similar restriction pattern in Jikes RVM. This suggests that it should be straightforward to adapt the baseline compiler into RJava, and is therefore suitable for LL language bypass.

Being able to implement a Java compiler/interpreter in RJava could introduce a full bootstrapping model to metacircular Java VMs. Most current metacircular VMs use a half bootstrapping model, i.e. the metacircular VM A requires another available Java VM B on the target machine so A's compiler can be executed on B and the executable will further execute its own code and the whole VM code. Half bootstrapping is blamed for bad portability, since it relies on the availability of another Java VM/compiler B. However, a Java compiler/interpreter written in RJava can execute as native code without the need of another available Java VM/compiler on the target platform. This would greatly enhance the portability of a metacircular VM.

There are difficulties lying in this direction that we will have to resolve in the future. One obvious point is whether we can implement an interpreter with RJava's restricted syntax. Though results showed that the baseline compiler in Jikes RVM uses a very similar pattern, more investigation is needed to ensure that, with acceptable refactoring/rework, an interpreter can strictly follow RJava restrictions. Furthermore, Java interpreter/baseline compiler is not an isolated component that can easily be decoupled from the rest of the VM. It requires support from other parts of the VM. This suggests code from other parts of the VM may be involved and have to be restricted with RJava syntax. The amount of code that has to be restricted is another concern. However, we believe that these difficulties can be overcome (the interpreter in the PyPy VM is written in RPython, which proves this is a feasible bootstrapping option for high-level language metacircular VMs) and RJava can be also used to benefit bootstrapping of metacircular Java VMs.

6. Related Work

Our design of the restricted language RJava is related to prior work in two main aspects: extending high-level languages for low-level programming, and the need for language restrictions in VM implementation.

Extending High-level Languages for Low-level Programming

The work [2009] by Frampton et al. (referred as the vmmagic work in the following discussion) described general concepts around extending high-level languages for low-level programming and the `org.vmmagic` framework. The `org.vmmagic` framework is solidly grounded in real world experience, including three Java-in-Java virtual machines [1, 7, 18], a Java operating system [21], and a C/C++ JVM [4]. This concrete framework introduced type-system extensions (raw storage and unboxed types) and semantic extensions (intrinsic functions and semantic regimes), and it was well

designed to resolve the tension between efficient low-level access and the encapsulation of low-level semantics. Our RJava also takes advantages of `org.vmmagic`. However, our work differs. The vmmagic work aimed at an efficient high-level low-level approach, and focused on extensions that would enable such an approach. As we explained earlier, extensions cause portability issues. Our work aims for a bypassing approach to solve portability issues of high-level low-level programming and along its way, we also examine, clarify and enforce language restrictions in VM implementation. There are three principal advances we make on the vmmagic work: 1) formalizing the restricted language RJava with clear extensions and scope-specific restrictions, 2) introducing a flexible design of restriction rules/rulesets and their compliance checking tool, and 3) implementing a translation toolchain that produces portable low-level language code from RJava and enables our bypass approach.

Language Restriction in VM Implementation

Our work is highly inspired by the work of RPython [3]. RPython is a restricted subset of Python and is used to write an interpreter in the PyPy virtual machine [22]. However, it is also an independent language that can be used for general use. RPython inherits most features from Python. It is restricted: for example, it is statically typed and does not allow dynamic modification of class or method definitions. The RPython backend supports code generation for different languages, such as LLVM code, C code, and even JVM and CLI code (work in progress). We have learned from RPython, from its success and also its imperfections. The design of RPython does not support flexible restrictions for different scopes (in Section 2.3, we explained its necessity in VM implementation). Besides, its restrictions are not clearly defined [23], so the restriction compliance cannot be automatically checked and programmers may not be able to realize restriction rule violations unless their code meets a translation error or a run-time error. We took those considerations into RJava's design, and addressed them with a flexible restriction rule/ruleset model that can be automatically checked. Currently RPython has a more reliable supporting framework than what RJava has, such as support for accurate garbage collection and precise exceptions, and RPython's backend can target several different languages. But we believe that with future development, RJava could be equally reliable, while being more flexible for different scopes of VM implementation.

7. Conclusion

We see a trend of applying high-level languages to systems programming to cope with the growing complexity of hardware and software. The security, productivity boost and software engineering advantages introduced by using a high-level language has benefited virtual machine's design and implementation. However, portability pitfalls of high-level low-level programming limit the reusability of VM components and the portability of VMs written in a high-level language.

Our approach is to formalize language restriction to define RJava, a restricted high-level language, for the implementation of virtual machine components. The design of RJava allows a low-level language bypass for improved portability, which promotes the reusability of VM components written in a high-level language, and provides better integration with legacy code.

We argue that restrictions are prevalent in virtual machine implementations for performance and correctness reasons, however, they are typically implicit, unprincipled, and ad hoc. The flexible and explicit restriction model of RJava requires virtual machine designers to consider scopes in a virtual machine along with restrictions to different scopes at an early stage. This explicit declaration not only benefits the design of virtual machines, but also favors automatic restriction compliance checks to enhance the robustness of the virtual machine and ease the implementors' work.

We hope that our insights and ideas will draw attention to the principled use of language restriction, and further encourage the implementation of virtual machines in high-level languages.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 314–324. ACM, 1999.

[2] B. Alpern, M. Butrico, A. Cocchi, J. Dolby, S. Fink, D. Grove, and T. Ngo. Experiences Porting the Jikes RVM to Linux/IA32. In *Proceedings of the 2nd Java(TM) Virtual Machine Research and Technology Symposium*, JVM '02, pages 51–64. USENIX Association, 2002.

[3] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8. doi: `10.1145/1297081.1297091`. URL `http://doi.acm.org/10.1145/1297081.1297091`.

[4] Apache. DRLVM – Dynamic Runtime Layer Virtual Machine. `http://harmony.apache.org/subcomponents/drlvm/`, 2009.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146. IEEE Computer Society, 2004.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and Realities: the Performance Impact of Garbage Collection. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '04/Performance '04, pages 25–36, New York, NY, USA, 2004. ACM. ISBN 1-58113-873-3. doi: `10.1145/1005686.1005693`. URL `http://doi.acm.org/10.1145/1005686.1005693`.

[7] S. M. Blackburn, S. I. Salishev, M. Danilov, O. A. Mokhovikov, A. A. Nashatyrev, P. A. Novodvorsky, V. I. Bogdanov, X. F. Li, and D. Ushakov. The Moxie JVM experience. Technical Report TR-CS-08-01, Australian National University, Department of Computer Science, May 2008.

[8] C2 Wiki. Restricted Programming Language. `http://c2.com/cgi/wiki?RestrictedProgrammingLanguage`.

[9] K. Driesen and U. Hölzle. The Direct Cost of Virtual Function Calls in C++. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, pages 306–323, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: `10.1145/236337.236369`. URL `http://doi.acm.org/10.1145/236337.236369`.

[10] C. Flack, T. Hosking, and J. Vitek. Idioms in Ovm. Technical Report CSD-TR-03-017, Purdue University, 2003.

[11] D. Frampton. *Garbage Collection and the Case for High-level Low-level Programming*. PhD thesis, Australian National University, 2010.

[12] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying Magic: High-level Low-level Programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 81–90. ACM, 2009.

[13] R. Garner. *JMTK: A Portabl Memory Management Toolkit*. PhD thesis, Australian National University, 2003.

[14] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit: a Substrate for Managed Runtime Environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, pages 51–62. ACM, 2010.

[15] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A Principled Approach to Operating System Construction in Haskell. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 116–128, New York, NY, USA, 2005.

ACM. ISBN 1-59593-064-7. doi: 10.1145/1086365.1086380. URL http://doi.acm.org/10.1145/1086365.1086380.

[16] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[17] S. McConnell. *Code Complete*. Microsoft Press, 1993.

[18] K. Palacz, J. Baker, C. Flack, C. Grothoff, H. Yamauchi, and J. Vitek. Engineering a Common Intermediate Representation for the Ovm Framework. *Science of Computer Programming*, 57:357–378, September 2005.

[19] M. Paleczny, C. Vick, and C. Click. The Java HotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267847.1267848.

[20] PMD Project. Pmd. http://pmd.sourceforge.net/pmd-5.0.0/.

[21] E. Prangsma. Why Java is practical for modern operating systems, 2005. Presentation only. See http://www.jnode.org.

[22] A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953. ACM, 2006.

[23] RPython Coding Guide. Rpython. http://doc.pypy.org/en/latest/coding-guide.html#id1.

[24] Sable Research Group, McGill. Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[25] J. Sieka. J2c project. http://code.google.com/a/eclipselabs.org/p/j2c/, 2012.

[26] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the Bare Metal of Wireless Sensor Devices: The Squawk Java Virtual Machine. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 78–88. ACM, 2006.

[27] D. Ungar, A. Spitz, and A. Ausch. Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment. In *Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 11–20. ACM, 2005.

[28] C. Wimmer, M. Haupt, M. L. V. D. Vanter, M. Jordan, L. Daynes, and D. Simon. Maxine: An Approachable Virtual Machine For, and In, Java. Technical report, Oracle Labs, 2012.

The JVM is Not Observable Enough (and What To Do About It)

Stephen Kell Danilo Ansaloni Walter Binder
University of Lugano
firstname.lastname@usi.ch

Lukáš Marek
Charles University
lukas.marek@d3s.mff.cuni.cz

Abstract

Bytecode instrumentation is a preferred technique for building profiling, debugging and monitoring tools targeting the Java Virtual Machine (JVM), yet is fundamentally dangerous. We illustrate its dangers with several examples gathered while building the DiSL instrumentation framework. We argue that no Java platform mechanism provides simultaneously adequate performance, reliability and expressiveness, but that this weakness is fixable. To elaborate, we contrast *internal* with *external* observation, and sketch some approaches and requirements for a hybrid mechanism.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—run-time environments

General Terms Measurement, Reliability, Performance

Keywords bytecode, instrumentation, DiSL, dynamic analysis, JPDA, JVMTI, profilers, debuggers

1. Introduction

Developers working with a virtual machine (VM) depend critically on its *observability*, meaning the ability to monitor and analyse the guest program's execution. Debuggers, profilers and other dynamic analysis tools are the programmer's interface to observability. In turn, the authors of these tools rely on VM-level mechanisms to build these tools; essentially every virtual machine provides some such facilities.

The Java Virtual Machine (JVM) is the target of many tools, developed by product engineers and researchers alike. It provides two basic observation facilities: the Java Platform Debug Architecture, a set of interfaces for interrogating a debug server running inside the VM; and the JVM Tool Interface, an interface for interposing an "agent" library which is commonly used to instrument bytecode at load time.

These mechanisms are inadequate. JPDA[1] is a usable basis for debuggers, but for dynamic analyses offers inherently limited performance and expressiveness. Meanwhile, our experiences building the DiSL instrumentation framework [11] using JVMTI-supported bytecode instrumentation show that common use cases cannot be realised without risking the introduction of show-stopping bugs, including deadlock and VM crashes. These problems can be worked around only by reducing the scope of observation.

We do not claim to be the first to observe these difficulties. In this paper our intention is to highlight them as a deeper issue. They are not simply quirks or "gotchas" for tool authors to be aware of; they are real obstacles to expanding the range and quality of tools available to programmers. By collecting the problems, underlining their severity, and characterising the requirements and design space for an eventual solution, we hope to advance the agenda of high-quality tool construction for managed runtimes. In summary, our contributions are:

- to identify seven design problems with instrumentation-based tools, illustrated with practical examples gathered during the development of DiSL;
- to survey the spectrum between *internal* and *external* observation, considering the JVM's relative strengths in these two modes;
- to sketch a set of requirements and possible approaches, motivated by existing literature, for a safe, efficient observability mechanism combining the benefits of internal and external observation.

Our latent position is that the JVM is not sacred. Considerable effort among researchers—ourselves included—is expended on building tools which exhibit good properties using only standard JVM interfaces. Much of this effort is wasted, because it ignores the real problem: the design of general, high-performance observability mechanisms is an open research challenge.

We begin by reviewing the JVM's observability facilities.

[1] Our canonical references for JPDA and JVMTI are the guides supplied by Oracle, as retrieved on 2012/8/16 from http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/. Note that strictly speaking, JVMTI is *part of* JPDA; when we refer to JPDA, we are more precisely referring to its other two constituent interfaces: JDI and JDWP.

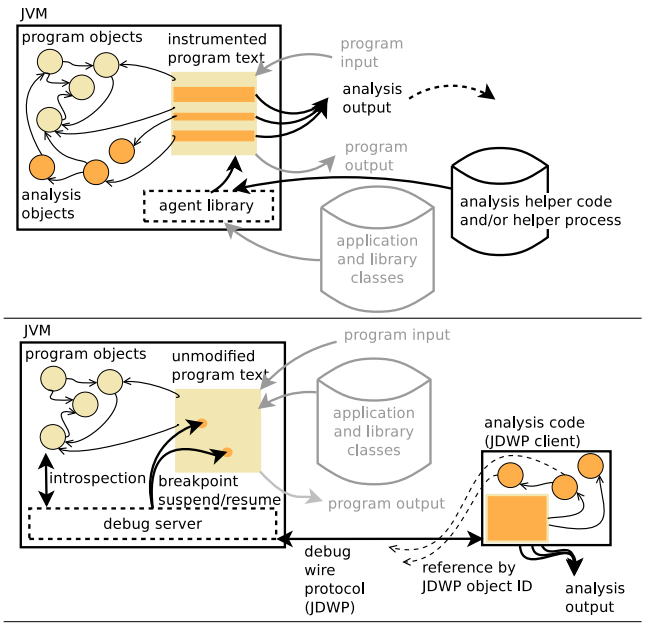


Figure 1. Internal observation by instrumentation (top) versus JPDA-based external observation (bottom)

2. Observability on the JVM

Like physical systems, software systems exhibit a tension between *observation* and *perturbation*: one cannot observe a system without affecting it somehow. This informs the design of VM-level observation mechanisms. The two mechanisms offered by the JVM platform—JVMTI and JPDA (contrasted in Fig. 1)—approach perturbation differently.

JVMTI allows construction of tools by linking a native library, called an "agent", into the VM. This library interposes on various VM events. Significant among these is class loading, where replacement code may be supplied by the agent. JVMTI's design deliberately emphasises tool construction by bytecode instrumentation: its documentation[2] notes that "this interface does not include some events that one might expect... [but] instead provides support for bytecode instrumentation, the ability to alter [the] bytecode instructions which comprise the target program". To minimise perturbation, the same document also suggests that agents should be "controlled by a separate process which implements the bulk of a tool's function without interfering with the target application's normal execution". Avoiding perturbation therefore becomes the tool author's problem.

Meanwhile, JDPA[3] "goes to great pains to avoid the execution of any code in the debuggee virtual machine" because in-process analysis "interferes with the behavior being analyzed... for example: ... competition for resources [means that] deadlock can occur" and that "many operations can only be reliably performed in a suspended virtual machine". As a result, the interface is relatively constrained in both expressiveness and performance: the wire protocol supports only a fixed set of queries, many of which execute only on suspended threads or a suspended VM. Although arbitrary analysis computations could be performed externally in the debugging process (and effectively this is what debugger-based Java expression evaluators do), implementing such an evaluator is a nontrivial undertaking, and the continual need to suspend and resume parts of the VM severely reduces overall performance. Most JVMs fall back to unoptimised or deoptimised execution of code observed by a debug client.

We first discuss various practicalities of bytecode instrumentation; JPDA is discussed subsequently.

3. Current practice

Most dynamic analysis tools for the JVM work by bytecode instrumentation, using JVMTI (or, rarely, performing the instrumentation offline). We call this *internal observation*: a single process contains both program and analysis.[4] Although JVMTI's documentation endorses a separate-process approach to minimise perturbation, to our knowledge only a small minority of instrumentation-based tools actually follow such a design. Simplicity and performance are likely explanations; certainly, these motivated DiSL's initial single-process design. Remote processes require marshalling and copying code, with its associated development and runtime overheads. In contrast, processing within local instrumentation does not incur these overheads, and benefits from JIT optimisations. In this section we review a series of problems encountered while building and using DiSL, which we believe are inherent to internal observation on today's JVM.

3.1 An example analysis

Consider a simple tool for identifying fields that are immutable (or likely to be) in a Java program, suggesting to the programmer that they could be made final. Clearly, we should instrument bytecode performing field writes, and record them as a set of per-field per-class "mutable" flags. Fields whose flag remains unset are likely to have immutable semantics, so could be made final. Such a tool was constructed in DiSL and has been used in published work [16]. Unfortunately, even simple instrumentations exhibit subtle problems; in the remainder of this section we describe several problems that this example and/or comparably simple instrumentation-based analyses easily encounter.

3.2 Deadlock of non-wait-free analyses

Fig. 2 shows a simplified set of stack traces that we have observed while using our immutability analysis. The analysis data structure, a WeakKeyIdentityHashMap, is protected

[2] Retrieved on 2012/8/16 from http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html.

[3] Quotations are from Sun's Frequently Asked Questions on the Java Platform Debugger Architecture, retrieved on 2012/8/16 from http://java.sun.com/javase/technologies/core/toolsapis/jpda/faqs.jsp.

[4] We note that often, as with DiSL, the instrumentation itself is done in a separate process spawned by the agent.

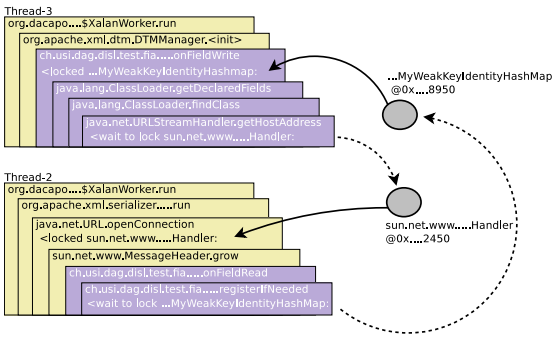


Figure 2. Deadlock between instrumentation and program. Purple (darker) boxes represent calls resulting from instrumentation; beige (lighter) boxes are in the base program.

by a lock, for which at least two threads are contending. Thread-3 is running user code that performs a field access, so the instrumentation locks the analysis structure. It then queries the target object's java.lang.Class to allocate some per-class state. In turn, this queries the class loader., which makes calls to the I/O library, requiring a lock on a Handler object. Unfortunately, Thread-2 has acquired a lock on the same Handler object and, owing to a field access in the same critical section, has itself called into the analysis and is waiting on the analysis lock. This is a classic deadlock: the two threads are contending for the same pair of locks but in opposite orders. Under instrumentation there is no way to enforce a global locking order, because the emergent ordering of locking operations depends on the implementation details of the instrumented code—which the tool author cannot reasonably know. One solution is to use only wait-free code in instrumentation, or to ensure that any lock taken out by instrumentation is a leaf lock (not held during any other locking operation). However, this requirement is even stronger than it first appears; we consider it shortly.

3.3 State corruption of non-reentrant code

Instrumentation can cause non-reentrant code to be invoked reentrantly, leading to state corruption. Consider a common requirement in instrumentation: to print out a message to the console. Normally a println() implementation avoids calling itself, so need not be reentrant. Suppose that it models a state machine, as in the sketch of Fig. 3. This code is perfectly correct; however, if we introduce some instrumentation to the definition of copySome() which prints out a message, the state machine will be advanced prematurely by a reentrant println() invocation, only to resume the first println() in the wrong state (causing an assertion failure at line 7). Note that the problem arises from interleaving *within a single thread*, so thread-safe code still exhibits this problem.

3.4 Calling methods

Our last two problems suggest that perhaps *any method* called on base program state from instrumentation is dangerous. We might therefore say: don't make any such calls!

```
1  /* inside a non−reentrant method,
2     perhaps java.io.PrintStream.println ()... */
3  try {
4    this.state = PENDING; // non−reentrant state machine
5    while (pos != len) pos = copySome(in, out, pos, len);
6  } finally {                    /*  ↖ now makes reentrant call ! */
7    assert this.state == PENDING; // fails following reentrant call
8    this.state = CLEAR;
9  }
```

Figure 3. Non-reentrant code corrupted by instrumentation

Indeed, to avoid deadlock (by wait-free instrumentation or "only leaf locking", as in §3.2) we must enforce this rule, because the waiting and locking behaviour of arbitrary methods is unknown. Unfortunately, enforcing this rule severely limits our expressiveness, because calling methods is indispensible in many circumstances. Suppose we want to analyse use of a library API making pervasive use of a user-defined type like Date or Currency as a method argument. Collecting contextual information about an event (for example, the month of the Date being passed) invariably means calling methods (on the Date object). Aggregating by such information also entails method calls—such as equals(), compareTo() or hashCode()—made by the aggregating container. All of these methods are entitled to perform locking. Although in these cases we could perhaps use JNI to make raw (private) field accesses instead of method calls, targeting private interfaces is little more desirable in instrumentation than in normal code. Method calls are also necessary to perform I/O, which invariably risks contention for per-VM data structures such as file handle tables.

3.5 Plausible instrumentation crashes the VM

It would be a convenient facility to instrument Object.<init>, because this captures all object initialization events. Unfortunately, doing so on at least one popular JVM (namely HotSpot) crashes the JVM.[5] Similarly, adding fields to Object might be an efficient way for an analysis to associate additional state with each object, but this also crashes HotSpot. The underlying problem is that there is no specification about what instrumentations are required to be supported by the VM; it is *undefined* whether this is a bug in HotSpot. (We emphasise that *all* problems described in this section, although inevitably triggered using specific JVM or library implementations, are not implementation-specific problems. Rather, sharing the JVM between instrumentation and user code creates unavoidable risk of bad interactions.)

3.6 Bytecode verification failure

Our immutability analysis needs to determine whether a given field write occurs during the execution of the target

[5] Actually, whether HotSpot crashes depends on precisely what instrumentation does, but without any obvious pattern. For example, we found that constructing a String with the += operator reliably crashed the VM, but constructing one from a literal did not.

object's constructor (meaning the field may be immutable) or afterwards (meaning it must be mutable). Unfortunately, to determine which field is being written, our analysis requires a reference to the containing object; if the constructor is still executing, this is an uninitialized object, and passing a references to it is conservatively forbidden by the bytecode verifier. We are forced to run this analysis with verification turned off. In general, objects which are not yet initialized may nevertheless be of interest to an analysis, but such analyses are not accommodated by the bytecode verifier.

3.7 Coverage underapproximations

DiSL supports instrumentation of the entire class library, not just user-supplied code. Indeed, instrumenting the sensitive code found deep in the libraries has helped expose the problems we have encountered. (However, all of them *could* arise purely in user code.) To allow the same library classes to be both *instrumented* and *used by* instrumentation without infinite recursion, a "bypass" is used: a thread-local flag records whether execution is currently at the base level (the program) or meta-level (the code inserted by instrumentation). Each method body is duplicated in both arms of an if–else construct testing this flag, with only the "false" (base level) copy being instrumented. In this way, helper calls made by the analysis into library code (such as containers) are not themselves analysed.[6] In general, we seek to avoid both this *over-analysis* (analysing the analysis, possibly causing infinite regress) and also *under-analysis* (loss of coverage, e.g. if we instead omitted to instrument the class library). Unfortunately, this thread-local bypass is only approximate; it avoids neither over- nor under-analysis. A simple example of under-analysis is the static initializer of a class which is used by the base program, but now also used *earlier* by instrumentation: its initializer will be run uninstrumented, when in fact it would later have been run by the base program and should therefore be analysed. The bypass is also active on all execution before the main() method, to avoid perturbing the load order of core classes (which is critical to VM bootstrapping), so these classes' static initializers are also not covered.

3.8 Reference handler over-analysis

The bypass flag avoids over-analysis within a single thread. However, when work is passed between threads, it cannot help. A notable example is reference handling. Many analyses make heavy use of WeakReferences, to track program objects without preventing their collection. Unfortunately, the task of appending cleared WeakReferences to their intended ReferenceQueue is usually implemented by a shared reference handler thread, implemented in class library code and therefore subject to instrumentation. All work done by this code is analysed, even though *some* of these references are due to the analysis rather than the base program. The problem is exacerbated when the analysis running in the reference handler thread itself allocates WeakReferences, hence creating yet more work for the reference handler, hence more allocations. This cycle is rate-limited by the lifetime of the WeakReference target, but can still exhaust memory. In affected applications of DiSL we have worked around this manually excluding the reference handler thread from analysis using an if-test in each inserted snippet. However, this turns overanalysis into underanalysis: reference processing for the base program is no longer analysed.

[6] This technique is called "polymorphic bytecode instrumentation" [12]. We previously believed it to offer adequate separation of meta-levels, until further experience uncovered the problems in §3.7 and §3.8.

4. External observation

Our immediate plans for improving DiSL rest on a new design strategy: "execute as little code as possible in the observed process". However, this statement begs two important questions: how little is possible, and will this really solve our problems? In this section we review the current options for external observation of JVMs, and also consider related designs not currently implemented by most JVMs.

4.1 Options for external observation of JVMs

Native code The closest vantage point "outside" a JVM is from native code in the same process. Some JVMTI-based systems such as hprof [10] perform analysis in native code to reduce perturbation. Unfortunately, most potential problems remain in some form. The biggest effect of shifting analysis to native code is to reduce coverage (since native code is not itself observed) and so reduce the likelihood of hitting a problem. However, native code is no safe haven; deadlock (§3.2) and reentrancy (§3.3) remain issues as presented earlier, and gathering contextual data (§3.4) will still generally require calls back into Java code.

Separate process The JVMTI documentation recommends doing most analysis from a separate process. However, in such a design, inserted bytecode must still be used to *collect* data and transmit it to a remote process using some IPC mechanism (such as a ring buffer in shared memory). So, while storage and computational processing are done in a separate process, the design does not fundamentally prevent any of the same problems from occurring. In particular, simply collecting data can easily require a method call; if so, then this call must be made within the observed process.

JVM debugging interfaces As outlined in §2, JPDA facilities designed primarily for the construction of interactive debuggers can also support a variety of dynamic analysis tools (including Caffeine [6] a trace collector and query engine, and the first version of the PROSE aspect weaver [14]). However, these tools run slowly: most JVM implementations run unoptimised code when a debugger is attached. (Even HotSpot's "full-speed debugging" works by dynamic deoptimisation of the debugged code.[7]) Furthermore, we

[7] Described in a HotSpot white paper, retrieved from http://www.oracle.com/technetwork/java/whitepaper-135217.html on 2012/8/17.

note that debug clients acquire the ability to query VM state thanks to the presence *within* the VM of a debug server (talking JDWP[8]). This is therefore arguably not external observation at all! *Pure* external observability requires that observing a program's execution involves adding *no* code in the target process. This is not supported by any Java platform specification. Significantly, JVMs need not publish their data representations or stack frame layouts, so cannot be observed from memory dumps or peek/poke-style interfaces.

4.2 Pure external observation

Some existing JVMs provide additional support for pure external observation of program state. We first discuss this support, then discuss a real-world use case.

4.2.1 Vendor extensions

Some JVM vendors have added limited forms of pure external observability using custom interfaces. HotSpot provides a set of "SA tools", for Serviceability Agent, a "Sun private component… developed by engineers… debugging HotSpot [who] then realized that SA could be used to craft serviceability tools for end users". In particular, SA "can expose Java objects as well as HotSpot data structures both in running processes and in core files".[9] SA tools include the `jmap` memory-map dumper, the `jstack` stack tracer, the `jhat` heap dump analyser, and others.

Specifying this kind of mechanism in the Java platform, ideally using compiler-generated descriptive debugging information (rather than "baked in" VM-specific knowledge used by the SA tools) would be a step forward, in allowing external observation without deoptimised execution and also in post-mortem cases. Fig. 4 illustrates. JVMs built on native compiler back-ends, such as gcj [2] for gcc or VMKit [5] with LLVM [9], already inherit this ability. However, just as method calls were preferable to digging for fields with JNI (§3.4), access to raw fields is less useful than the ability to isolate bytecode-based instrumentation would be.[10]

4.2.2 Use case: DTrace on Java

A cutting-edge application of external observation is found in DTrace [3], a dynamic tracing tool designed for safety, high coverage, and performance appropriate for use on production systems. DTrace primarily targets native code at both user and kernel levels. All analysis code runs in the kernel, sandboxed within an interpreted virtual machine subject to various load- and run-time checks.[11]

[8] http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html

[9] From "Serviceability in Hotspot", retrieved on 2012/8/15 from http://openjdk.java.net/groups/hotspot/docs/Serviceability.html.

[10] Pure external observers of Java will require notification when objects are moved by the collector; a portable protocol for this is also required.

[11] DTrace arguably then does *internal* observation of kernel code. It avoids deadlock and reentrancy problems (§3) using a wait-free analysis path which mutates only private state and may not itself be instrumented.

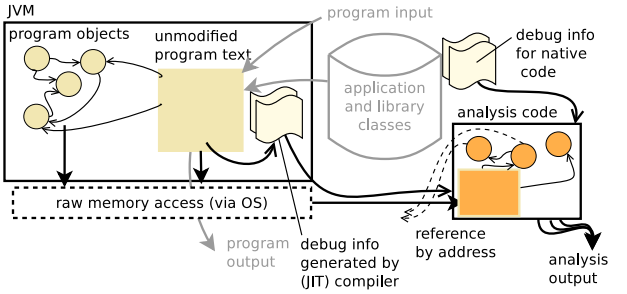


Figure 4. Pure external observation using descriptive debugging information

An essential design feature of DTrace is that little information is propagated proactively from the analysed program to DTrace. Rather, DTrace kernel code extracts state from the observed program (such as the stack trace, current function arguments, and data gathered from walking data structures), using memory access and debugging information much like a native debugger. In this way, probes can be enabled and disabled without the base program's involvement, and unwanted probe data can be discarded at source. This relies on the ability of DTrace kernel code to decode the data structures and stack frames of target code, so cannot be supported with existing JVM observability mechanisms. Existing portable solutions for running DTrace in Java code (including the JVMTI-based `dvmti`[12] provider, and the BTrace[13] bytecode instrumentation systems) are forced to proactively *marshal* data into predictable form, negating the "discard at source" feature and adding slowdown even for disabled probes (an overhead avoided by most DTrace providers). Recent versions of HotSpot now contain a built-in DTrace provider, which permits a more optimised but VM-specific approach (analogous with the "Serviceability Agent", §4.2.1), reaffirming our position that the specified observability mechanisms are not sufficient.

5. How to fix it

We remain committed to the approach of running analysis outside the JVM as far as possible. On the current JVM platform this dooms us to limitations. We believe that the design of VM-level mechanisms for fast, safe observation is an open challenge, and specifically that an optimal synthesis of internal and external observation is yet to be achieved. Here we sketch some ideas and requirements for such mechanisms.

Inlined guards and other "safe" instrumentation The dynamic compilation available in JVMs should allow us to achieve a *better* isolation/performance trade-off than in native code. Some code really is effect-free and can safely be inserted as instrumentation, where it can be optimised. This could, for example, avoid redundant traps in DTrace for false

[12] http://kenai.com/projects/dvm/

[13] http://kenai.com/projects/btrace/

predicates (§4.2.2). Useful guidance could come from purity analyses already performed by JIT compilers.

Isolated bytecode Executing analysis against snapshots of program state is a convenient abstraction. Object-level copy-on-write snapshots have already been demonstrated by work on asynchronous assertions [1]. The same approach could allow instrumentation bytecode to execute "as if" in the target process, but in an effect-free fashion. The resulting "sandboxed bytecode evaluator" could be a candidate for replacing JDWP, much as JVMTI uses bytecode instrumentation to replace various utility calls in its predecessor JVMPI [10].

Free association Maintaining *per-object* analysis state is currently done using associative mappings (e.g. keyed on WeakReferences). We noted (§3.5) that adding fields to Object would be a useful alternative. Meanwhile, adding fields to *every* object could be wasteful if only certain objects are of interest. Fast disjoint metadata implementations using virtual memory techniques have appeared in recent work [8, 13]. A useful addition to instrumentation libraries could be to specify the availability of an associative container keyed on object identity with a strong performance contract.

Meta-level separation as a VM service The concept of software-isolated processes or "isolates" is well developed [4, 7] and could be the basis of an isolated metalevel. A distinction from the normal case is that information flow *in one direction* must be permitted. Fitting a suitable design onto the JVM would at least require eliminating shared threads (cf. §3.8). (We note that DTrace's in-kernel virtual machine, described in §4.2.2, is another instance of software isolation, i.e. with respect to the wider kernel.)

Record/replay correctness It is currently a difficult task to actually test that an analysis does not unduly perturb the program it is observing. Some performance effect is always expected, and in the case of instrumentation, the path taken through the program will necessarily be modified too. An intuitive requirement is that we should be able to erase the analysis parts of the path and find the base program path otherwise unchanged. A useful approximation of this criterion might be available from record/replay systems such as that of Saito [15]: a replay log from an uninstrumented run should be replayable against the instrumented code without divergence, and producing the same output (as well as additional output from the analysis). This will likely require some support from the VM to tolerate execution differences without causing divergence, e.g. concerning garbage collection: the instrumented program will allocate more memory and collect more often.

6. Conclusions

We have shown, with examples, that bytecode instrumentation poses severe and unavoidable dangers as the basis of tool construction, yet no other Java platform mechanism is adequate. We have motivated the open challenge of designing a efficient, isolated observation mechanism suitable for JVMs, and have provided some initial design sketches.

Acknowledgments

The research presented here has been supported by the Swiss National Science Foundation (project CRSII2_136225), by the European Commission (Seventh Framework Programme grant 287746) and by the Czech Science Foundation (project GACR P202/10/J042). The authors thank their fellow DiSL authors and contributors: Aibek Sarimbekov, Petr Tůma, Yudi Zheng, Andreas Sewe.

References

[1] E. E. Aftandilian, S. Z. Guyer, M. Vechev, and E. Yahav. Asynchronous assertions. In *Proc. OOPSLA '11*. ACM.

[2] P. Bothner. Compiling Java with GCJ. *Linux Journal*, 2003.

[3] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proc. USENIX ATEC '04*. USENIX Association.

[4] G. Czajkowski and L. Daynés. Multitasking without compromise: a virtual machine evolution. In *Proc. OOPSLA '01*. ACM.

[5] N. Geoffray. *Fostering Systems Research with Managed Runtimes*. PhD thesis, Paris, France, September 2009.

[6] Y.-G. Gueheneuc, R. Douence, and N. Jussien. No Java without Caffeine: A tool for dynamic analysis of java programs. In *Proc. ASE '02*. IEEE, 2002.

[7] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007.

[8] S. Kell and C. Irwin. Virtual machines should be invisible. In *Proc. VMIL '11*, SPLASH '11 Workshops. ACM.

[9] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO '04*. IEEE Computer Society, IEEE.

[10] S. Liang and D. Viswanathan. Comprehensive profiling support in the Java virtual machine. In *Proc. COOTS '99*, COOTS'99, Berkeley, CA, USA. USENIX Association.

[11] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. In *Proc. AOSD '12*. ACM.

[12] P. Moret, W. Binder, and E. Tanter. Polymorphic bytecode instrumentation. In *Proc. AOSD '10*. ACM.

[13] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *Proc. PLDI '09*. ACM.

[14] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proc. AOSD '02*. ACM.

[15] Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proc. AADEBUG '05*. ACM.

[16] A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, and S. Z. Guyer. new Scala() instance of Java: a comparison of the memory behaviour of Java and Scala programs. In *Proc. ISMM '12*. ACM.

Some New Approaches to Partial Inlining

Bowen Alpern
Lehman College, CUNY
Bowen.Alpern@lehman.cuny.edu

Anthony Cocchi
Lehman College, CUNY
Anthony.Cocchi@lehman.cuny.edu

David Grove
IBM Research
groved@us.ibm.com

Abstract

This paper proposes two novel techniques for partial inlining. *Context-driven partial inlining* uses information available to the compiler at a call site to prune the callee prior to assessing whether the (pruned) body of the callee should be inlined. *Guarded partial inlining* seeks to inline the frequently taken fast path through the callee along with a test and a call to the original method to handle those instances where the fast path is *not* taken.

A fragile implementation of guarded partial inlining is described. An example, very loosely based on a simplified web server, is fabricated. Experimental evidence establishes the superiority of partial inlining over no inlining and over complete inlining *on this contrived example*. We show that these approaches to partial inlining are applicable in situations where previous approaches are not.

Potential effects of the widespread availability of partial inlining on software development are considered.

Categories and Subject Descriptors Software and its engineering [*Compilers; Procedures,functions and subroutines; Parallel programming languages*]

General Terms compiler optimization, inlining

Keywords partial inlining, X10

1. Introduction

Compilers and virtual machines can do more to help programmers achieve high performance without distorting the natural structure of their programs.

Inlining is a compiler optimization that replaces a call to a method with the (suitably mangled) body of the method. It is well known that inlining frequently called methods can significantly improve the performance of a program. The *direct* benefit of inlining a call is the elimination of a method call overhead. Additional *indirect* benefits are often realized in the form of opportunities to further optimize the combined code of the caller and inlined call. (The genesis of this work lies in a desire to exploit the existence of such opportunities to help drive the choice of which calls to inline.)

The cost of inlining is *code bloat*: usually inlining a call increases the size of the caller method.[1] During compilation, bloat is felt in different ways. Typically, a compiler is given a space-budget that limits the final size of a method being compiled. If inlining a method increases the size of the caller, it limits the compiler's ability to inline other methods. It also makes the caller itself a less attractive candidate for inlining into its callers. During execution, code bloat can degrade performance by increasing instruction cache misses or otherwise overflowing fixed-size hardware resources such as branch prediction tables.

Partial inlining is a technique for realizing (much of) the benefit of complete inlining with less bloat. Here, only part of the called method is inlined. Accessing the uninlined portion still requires a method call, but the call is not made if the execution stays within the inlined portion. Previous work on partial inlining (see section 5) has been done in a context (such as a JIT compiler) where the compiler has profile information that allows it to conclude that certain paths through a method will be *hot*, often taken, while others will be *cold*, seldom taken.

The long-term goal of this research is to enable partial inlining in cases where such information is either unavailable or given only in terms of hints (annotations) supplied by the programmer. In particular, our ultimate aim is to exploit information at the call site to prune the body of the callee with a view to evaluating the pruned body as a candidate for inlining.

The *lazy initialization* idiom, which appears in a wide variety of guises in programs, provides a prime example illustrating the *context-driven partial inlining* approach. Consider the Java method in Figure 1 for adding an edge to a directed graph.

If the compiler could establish at the call site that the `neighborhood` map contains a non-null entry for `v` then the `addEdge` method could be inlined *without the if-block*. For

[1] If the inlined size of a method body is no bigger than the call to the method, there is no downside to inlining the call. Such methods should always be inlined. This paper will not consider them further.

```
final void addEdge(Vertex v, Vertex w) {
  Set<Vertex> neighbors = neighborhoods.get(v);
  if (null == neighbors)  {
    neighbors = new HashSet<Vertex>();
    neighborhoods.put(v, neighbors);
  }
  neighbors.add(w);
}
```

Figure 1. A motivating example of the Lazy Initialization Pattern

```
final void addEdge$$(Vertex v, Vertex w) {
  Set<Vertex> neighbors = neighborhoods.get(v) {
  if (null == neighbors)  {
    addEdge(v, w);
  } else {
    neighbors.add(w);
  }
}
```

Figure 2. Synthetic addEdge method created for use by Guarded Partial Inlining

instance, suppose the caller was an input routine reading a graph for a file in a format where each vertex was followed by a list of its neighbors. The compiler might emit code with a call to `addEdge` for a vertex's first neighbor followed by a loop of partially inlined calls for the remaining neighbors.

In other circumstances, the compiler might be unable to determine that the `if-statement` is always unnecessary but be able infer (perhaps with the help of a hint from the programmer) that it is likely to be infrequently taken. In this case, a related technique, *guarded partial inlining*, would cause the synthetic `addEdge$$` method, shown in Figure 2, to be inlined in place of the call to `addEdge`. This synthetic method calls the original `addEdge` if its slow path is required. Otherwise, it performs the fast path in-line.

Guarded and context-driven partial inlining are closely related but different techniques. Guarded partial inlining inlines not only the fast path but the test and a call to the original code as well. If the compiler can deduce that the test will always hold at a call site (this is precisely the precondition for context-driven partial inlining), then other optimizations (e.g., constant propagation) will eliminate the test and the call, leaving code identical to that produced by context-driven partial inlining. However, the compiler can perform guarded partial inlining where context-driven partial inlining is *not* warranted. This is both a blessing and a curse. The compiler might be tempted to perform guarded partial inlining under circumstances where the fast path is very rarely taken. Thus, reliance on programmer hints seems to be required for guarded partial inlining, but not for context-driven partial inlining. For the purposes of this paper, the pertinent difference between the two techniques is that guarded partial inlining admits an easy prototypical implementation in the X10 compiler while supporting context-driven partial inlining would require implementing infrastructure in that compiler for keeping track of state information that it does not currently maintain (see sections 2 and 7). Consequently, a prototype of guarded partial inlining will be used in this preliminary exploration of the efficacy of both techniques.

The next section describes our prototypical implementation of guarded partial inlining. Section 3 explains an example loosely based on a simple web server. Section 4 establishes the efficacy of guarded partial inlining on this example. Section 5 relates the two approaches presented here to other approaches to partial inlining. Section 6 outlines paths to robust implementations of both guarded and context-driven partial inlining. Section 7 speculates that general availability of partial inlining would have a positive impact on software development. Section 8 concludes.

2. A Prototypical Implementation

To implement our prototype, we decided to build upon the existing inlining infrastructure in the X10 compiler. This choice may seem somewhat quixotic because of the relative immaturity of the X10 optimizer (the primary focus of the X10 language [6, 12] being on parallel and distributed computation) and the lack of extensive suites of X10 benchmarks. However, the compiler is open source and its internals are well-known to two of the authors. Furthermore, it is based on the Polyglot [11] extensible compiler framework, which makes it relatively easily to add new optimization passes. It also has mechanisms in place for processing program annotations and for annotation-directed inlining, both of which we will exploit.

The X10 compiler is a source-to-source compiler: it translates X10 programs to either Java or C++. The resulting programs are then "post-compiled" to Java bytecodes or binary code by invoking a Java or C++ compiler respectively. The option to compile X10 to C++ enabled us to simplify our experimental evaluation by avoiding the possibility of the JVM's just-in-time compiler doing additional profile-directed inlining, thus obscuring the impact of the inlining being done by the X10 compiler.

This prototype is intended as a mechanism for establishing the efficacy of guarded partial inlining. It does not purport to be robust. It relies heavily on the programmer to know what she or he is doing and to tell the compiler what to inline. Annotations are required both on the definition of a method to be partially inlined and on the calls to these methods. (This allows us to control exactly which calls are partially inlined.)

Two passes have been added to the X10 compiler. The first of these walks the abstract syntax tree representations of X10 classes looking for **method definitions** with the appropriate annotation. When such a method is found, a new method (with a suitably mangled name) is created and added to the class. The new method is an exact copy of the old one except that the else branch of the first conditional statement

```
1 def caller {
2     prelude;
3     @PartialInline callee();
4     postlude;
5 }
6 private @PartialInline def callee() {
7     prologue;
8     if (test) {
9        fastpath;
10    } else {
11       slowpath;
12    }
13    epilogue;
14 }
```

(a) The caller and callee before inlining.

```
1 def caller () {
2     prelude;
3     prologue;
4     if (test) {
5         fastpath;
6     } else {
7         callee();
8     }
9     epilogue;
10    postlude;
11 }
```

(b) The caller after inlining.

Figure 3. Guarded partial inlining pseudocode.

in the method body is replaced by a call to the original method.[2] The new method is marked with an annotation that will force the method to be inlined in a later phase of compilation.

The second new pass looks for appropriately annotated **method calls**. Such calls are rewritten to call the corresponding partial inline method with the mangled method name that was created in the former pass and are annotated to be inlined unconditionally. The standard inlining pass of the X10 optimizer does the actual inlining.

Given the skeletal X10-like pseudo-code[3] in figure 3(a), the X10 compiler will emit something like the pseudocode in figure 3(b) for the caller. This transformation is *invalid* unless `prologue`, `test`, and `epilogue` are idempotent. It is *inexpedient* unless all three and `fastpath` are small (or could be optimized away).

3. A Contrived Example

The purpose of this example is to present a situation in which guarded partial inlining is in some way better than both no inlining and complete inlining. To demonstrate that guarded partial inlining is better than complete inlining it suffices to show that the former produces less code bloat. Ideally, performance measurements would establish the superiority of the partially inlined (over uninlined) code. However, a myriad of factors contribute to the performance of parallel programs. It is difficult to tease out the component attributable to partial inlining.

The X10 compiler currently has two back-ends. It can either produce Java or C++ source code as its target. The Java code will run on a JVM with its own JIT compiler. As previously stated, it is difficult to determine whether to attribute any difference in performance to the transformation being measured, or to the JIT compiler. For this reason, our measurements are restricted to the C++ back-end.

The example is an abstraction of a simplified client-server architecture. Various *clients* generate work and put it on an *order queue*. A single *manager* takes work from the order queue and a *worker* from a *worker queue* and dispatches the worker on the work.

Figure 4 shows the initial version of the manager's X10 code. The `async` statement at line 6 creates a new X10 *activity*, a very lightweight thread for worker `w` to process order `p`. The dequeue method starting at line 9 shows the code for dequeuing work from the order queue (dequeuing from the worker queue is similar). The dequeue code exploits the knowledge that, while many activities may add items to the order queue, only one activity removes them. Thus, no synchronization is required unless there is only one item in the queue.

In the expected case, when there is more than one order on the order queue, removing an item entails a test, two assignments, and a return. The *unexpected* case (the `else` clause) is much worse than it looks because `when` and `atomic` are somewhat expensive synchronization operations in X10 generating many lines of C++ (or Java) code. Guarded partial inlining inlines the test and the two assignments. The return is naturalized into an already existing assignment of the result of the method to a local variable.[4]

[2] This is a minimal mechanism sufficient to allow us to explore the consequences of partial inlining. It is *unsafe*, an unwitting (or unscrupulous) programmer could use it to annotate a program in such a way as to change the program's semantics. To be absolutely clear, the authors do not advocate the inclusion of unsafe annotations in programming languages. We do feel that such annotations may be legitimately used in a research prototype.

[3] This pseudocode ignores method receivers and parameter passing to and returning values from methods. Also ignored is the problem of alpha renaming of local variables and formal parameters in the inlined method body. Such issues are important to get right in any implementation of inlining, but present no special problems for partial inlining. This paper does not consider them further.

[4] A call to the original method is also inlined as the else branch of the test. In the current implementation, this results in a redundant test (once at the

```
1  public def manage(count : int) {
2    for (n in 1..count) {
3      val p = @PartialInline orderQ.dequeue();
4      val w = @PartialInline workerQ.dequeue();
5      async w.work(p);
6    }
7  }
8
9  @PartialInline public def dequeue () : T {
10   if (null!=head && head!=tail) {
11     val item = head.item;
12     head = head.next;
13     return item;
14   } else {
15     if (null==head)
16       when (null!=head);
17     atomic {
18       val item = head.item;
19       head = head.next;
20       if (null == head)
21         tail = null;
22       return item;
23     }
24   }
25 }
```

Figure 4. Initial X10 code for the manage and dequeue methods from the example program.

Thus, if the expected case predominates, guarded partial inlining will eliminate two method call overheads from each iteration of the manager's inner loop.

Unfortunately, inlined or not, this code does not perform as well as it should.

One problem has to do with returning a value from inside an `atomic` block. In the current X10 C++ back-end, this entails creating a very large `Finalization` object, which is expensive to create and which could force a very expensive garbage collection.[5] This return is on the slow path through `dequeue` so it shouldn't cause any problems, but it is easy to eliminate by moving the return outside the block.

Another problem is that `head!=tail`, the pointer inequality test, is more expensive than it appears (or should be). It entails a virtual function call.[6] For the purposes of our experiments we eliminated this call by using X10's `@Native` annotation to replace the inequality test with a call to an inlined C++ function that just compares the pointers directly.

A final problem concerns the `async` statement in the manager inner loop. *Activity* (thread) creation is supposed to be extremely lightweight in X10. It is significantly less

call site and once in the call). Section 6 discusses how the redundant test could be eliminated.

[5] This is a performance bug in the X10 2.2.3 compiler and runtime that was identified as a result of this work and has been fixed for the upcoming X10 2.3.0 release.

[6] The inequality of boxed structs in X10 requires checking the fields of the structs. This should not matter here since `head` and `tail` are known to be instances of a class (and thus not structs), but the current X10 compiler misses this optimization. As a result a virtual call is made whose body is cheap, but the function call overhead remains.

```
1  public def manage(val numOrders : int) {
2    for (n in 0..(numOrders/buf.size-1)) {
3      for (i in 0..(buf.size-1)) {
4        buf(i) = @PartialInline orderQ.dequeue();
5      }
6      val w = workerQ.dequeue();
7      async w.work(buf);
8    }
9  }
10
11 @PartialInline public def dequeue () : T {
12   val item:T;
13   if (null!=head && differ(head, tail)) {
14     item = head.item;
15     head = head.next;
16     return item;
17   } else {
18     if (null==head)
19       when (null!=head);
20     atomic {
21       item = head.item;
22       head = head.next;
23       if (null == head)
24         tail = null;
25     }
26     return item;
27   }
28 }
```

Figure 5. Optimized X10 code for the manage and dequeue methods from the example program.

expensive than a normal thread creation and is implemented entirely at user level. However, it does entail several method calls encompassing hundreds of instructions. In short, its costs dwarf the savings we are hoping to measure. The remedy here is to batch up orders processed by a worker amortizing `async` (and worker) overhead across multiple order queue dequeues.[7] Figure 5 shows the example code of Figure 4 after it has been adjusted to avoid these problems.

4. Experimental Results

The example in the previous section is a concurrent application, and the benefits of guarded partial inlining should be realized during parallel execution. However, parallel execution makes it exceedingly difficult to accurately *measure* such benefits. Moreover, the benefit itself in this case, the call overhead for a non-virtual method, should not be expected to be very big, merely a handful of machine instructions.

In order to try to obtain accurate measurements, we tried to eliminate as much interference with the execution of the manager inner loop as we could. A single client was constrained to run to completion before the manager started. Limiting the number of X10 threads to one should guarantee that the manager will complete before any of the workers begin. (In addition, we constrain the workers to ignore the

[7] A similar strategy is used by the clients to enqueue multiple orders in order to amortize enqueue synchronization costs.

work they are given and return immediately.) To minimize the possibility of garbage collection interfering with timing of the manager (and thus, to assure more accurate and repeatable results), a garbage collection is forced just before this timing starts.

We measure the size and performance of three different compilations of the application: **vanilla** has no inlining of calls in the manager's manage() method to the `dequeue` method on the order queue; **complete** inlines all such calls completely; and **partial** uses guarded partial inlining on them. Otherwise, the compilations are identical and the X10 compiler was invoked with the recommended [16] -O and -NO_CHECKS flags.[8]

Minimum, average, and maximum accumulated inner-loop times of the manage() method for fifteen trials of each flavor (after two discarded "warm up" trials) on $20,000,000$ orders are reported in Table 1. The final column of the table shows the size of the x86 object code for these methods. The experiments were run on a machine with two Intel Xeon E7530 Nehalem processors. Each processor has six 2-way SMT cores running at 1.87 GHz sharing a 12 MB L3 cache. The machine is configured with 16 GB of memory. The machine was running Red Hat Enterprise Linux 6.3.

	min	avg	max	size
vanilla	0.294	0.298	0.299	351
partial	0.284	0.287	0.289	399
complete	0.287	0.288	0.290	1007

Table 1. Accumulated time (in seconds) of the manage() method of manager's inner-loop with $20,000,000$ orders and the size (in bytes) of the method.

Both partial and complete inlining show a modest but measurable (about 3%) improvement in performance. Partial inlining achieves this benefit at considerably less cost in code bloat.

5. Related Work

Inlining is a fundamental optimization technique applied in almost all modern optimizing compilers, therefore there is an extensive body of previous research and practical experience. We refer the interested reader to surveys such as Arnold et al. [1] for broader coverage; here we will restrict our attention to previous work in partial inlining and in exploiting static information in the caller to obtain better cost-benefit estimates for the inlined callee.

Partial Inlining Previous work in partial inlining has primarily relied on profile information or static heuristics to distinguish between the frequently executed portions of the callee method, which should be inlined, and the infrequently executed portions, which should not be inlined. In the Self-91 system, the compiler heuristically applied *deferred compilation* to avoid generating code for uncommon branches [5]. In the presence of inlining, deferred compilation has a similar effect as partial inlining: the inlined code is smaller since code is not generated for the unexpected control flow paths. Whaley [15] applied the ideas of deferred compilation in a JVM using dynamic profile data to identify rarely executed basic blocks that should not be inlined. Zhao and Amaral [17] approached partial inlining by implementing a procedure outlining transformation[9] that was run before the normal inlining pass in the ORC compiler. Recently Lee et al. [10] proposed a new algorithm for identifying the portion of a callee method to be partially inlined by using control flow edges leading to frequently executed procedure returns as seeds to the sub-graph construction algorithm.

Partial inlining can also be viewed as a mechanism for allowing traditional procedure-based compilation systems to achieve many of the benefits of more flexible trace-based compilation. The Dynamo system [2] pioneered many of the key concepts in dynamic, trace-based compilation. Inspired by Dynamo, Bruening and Duesterwald [4] did an early study on the applicability of trace-based compilation to Java and confirmed that confining optimization decisions to procedure boundaries does result in sub-optimal selection of compilation units. A number of subsequent systems explored region-based and trace-based just-in-time compilation [8, 9, 14].

Inlined Size Estimation A critical step in the cost-benefit decision of inlining is constructing an estimate of the cost: how big will the callee method be when inlined at a particular call site? Many compilers do this estimation based purely on characteristics of the callee method: they do not take into account how any information they may have available about the actual parameters at a specific call site could influence the optimization of the callee method when it is inlined in a specific context.

One notable exception is the inlining trials system of Dean and Chambers [7]. The key innovation of their system was to enable the compiler to learn how static information available at a call site would impact a callee method, thus yielding a cost-benefit decision tailored to the call site and allowing much larger callee methods to be considered as potential inlining candidates. In practice, their system would select larger than normally allowed callee methods to be inlined at exactly those callsites where the static information available about the actual parameters would enable the elimination of significant portions of the inlined callee method. This is the same effect we want to achieve in context-driven partial inlining, but we propose to approach it in the opposite direction by analyzing callee methods to determine what static information would be needed in the caller to profitably enable partial inlining of the callee.

[8] -O enables the X10 optimizer and also causes the g++ compiler to be invoked with -O2 -finline-functions. -NO_CHECKS disables runtime checking for null pointers, divide by zero, and out-of-bounds array accesses

[9] *Outlining* is the dual of inlining: it replaces a block of code with a procedure call to a new method that contains the original code.

```
def caller () {
    prelude;
    prologue;
    if (test) {
        fastpath;
        epilogue;
        postlude;
    } else {
        newCallee();
    }
}
def newCallee () {
    slowpath;
    epilogue;
    postlude;
}
```

Figure 6. Potential results of aggressive guarded partial inlining of the pseudocode from Figure 3(a).

The Jikes RVM optimizing compiler also heuristically adjusts its estimate of the inlined size of the callee method based on static information known about the actual parameters of the call. Sewe et al. [13] describe the original heuristics used by Jikes RVM and how they can be improved by using the dynamic call graph.

6. Future Work

Our prototype implementation of guarded partial inlining is both unsafe and unnecessarily limited.

The requirement that the call to a guarded partial inline method be annotated could be made optional. The X10 compiler can determine whether the declaration of the called method has the requisite annotation.

The prototype assumes that the evaluation of the test (and any prologue or epilogue code) in the method to be inlined will be idempotent. At a bare minimum, this assumption should be checked.

It would be better to avoid the redundant test altogether by crafting a new method to be called instead of the original in the event that the test is false. This new method would omit the test along with any prologue and/or epilogue code that would be inlined at the call site.

Guarded partial inlining achieves the direct benefit of inlining (in the expected case) but only gets some of the indirect benefits because the inlined call is wrapped in a context that handles the possibility that the test might fail. Although it is able to optimize the callee for the specific context it is inlined in the caller, it is unable to optimize subsquent code in the caller based on information discovered from the inlined callee because the non-inlined call introduces a merge in the dataflow information. To achieve these downstream indirect benefits, it would be necessary to move the code following the call site into both arms of the `if` statement evaluating the test. This approach would be particularly attractive if the code along the else branch could be encapsulated in a call to a synthesized out-of-line method.

Figure 6 shows what might be achieved by such aggressive guarded partial inlining. The indirect benefits of partial inlining are realized when the compiler optimizes

```
fastpath;  epilogue;  postlude;
```

under the assumption that test must hold. If such benefits are significant enough, the caller could be come a candidate for guarded partial inlining.

The prototype also assumes there is a unique fast path through the method to be inlined, that first (top-level) `if` statement distinguishes that path from the slow path(s), and that the fast path is the `then` branch of that `if` statement. These assumptions are unnecessarily restrictive. The `@PartialInline` annotation of a method declaration could take a path as an argument. The path could be encoded as a sequence of boolean values indicating the values to be assumed for successive boolean expressions encountered by the compiler on the method.[10]

The brute force approach to achieving context-driven partial inlining is surely unworkable. In principle, one could at each call site re-compile the callee in the light of information available at the call site. A scheme that cached the results of such re-compilations and reused them for call sites where the available information was "similar" would cut down on the cost, but probably not enough to make this approach practical.

A different approach would preprocess the callee, identifying information that could substantially simplify it. Such information could be structured as a predicate on its formal parameters (including its `this` parameter) along with any statically available information. Once obtained, such predicates could be used for either context-driven or guarded partial inlining. How to derive such predicates is the major open research question of our work.[11]

To achieve context-driven partial inlining, when the compiler deduces that one of the predicates associated with the callee at a specific call site must be true of the arguments to the call, a version of the callee tailored under the assumption of this predicate would be inlined. A lazy cache could be used to map predicates to the version of the method tailored under the assumption that the predicate holds.

Heretofore, this discussion has blithely assumed that the compiler is able to determine whether a predicate must hold at a particular call site. This is, of course, an undecidable problem. A compiler can only approximate its solution. How good an approximation an optimizer can provide depends in part on the infrastructure it provides for maintaining information about the program under compilation.

[10] Note that multiple annotations could associate multiple partial inline paths with a method. If this were the case, some annotation mechanism at the call site would be needed to determine which path(s) to partially inline.

[11] As an interim measure, interesting predicates could be supplied as boolean expression parameters to annotations. Multiple annotations could associate different partial inline predicates with the same method.

The X10 compiler has potential for providing a powerful infrastructure to explore these ideas, but there are significant missing pieces that must be completed to fulfill this potential. On one hand, the X10 type system maintains useful compile-time information about variables (and constants), such as the rank of an array and whether a reference is known to be non null. On the other, there is currently no mechanism in the X10 compiler to keep track of state information that must hold at a particular point in a program. For instance, at the first position in the `then` branch of an `if` statement, the condition of the `if` must hold, but the X10 compiler currently does not keep track of this information. As a practical matter, implementing context-driven partial inlining in X10 would entail developing a subsystem to keep track of such information.

The primary downside to doing compiler optimization research in X10 is the lack of competitive benchmarks. A clear picture of the potential benefits of context-driven or guarded partial inlining will not emerge until these techniques have been implemented in either a C/C++ or Java compiler for which more extensive benchmark suites, such as SPECInt or DaCapo [3], are available. While working on inlining in Jikes RVM, we observed that a few of the DaCapo benchmarks do have frequently called methods that are too large to be selected as inlining candidates, but that would be profitable to partially inline. It would be interesting to investigate whether or not our approach to partial inlining would be able to successfully optimize these call sites and what impact that would have on overall performance.

7. Discussion

Why should anyone care about partial inlining? After all, the performance improvement reported in section 4 was not very impressive. Furthermore, a programmer interested in obtaining the benefits of inlining at the reduced costs of partial inlining can easily achieve the same result by manually refactoring the source code.

In the first place, we only measured the direct benefits of guarded partial inlining. Indirect inlining benefits can be at least as substantial [7]. As indicated in the previous section, achieving such benefits from guarded partial inlining should be difficult but not impossible. However, achieving indirect benefits from context-driven partial inlining should be straightforward.

The more important point concerns refactoring source code to obtain better performance.

Within living memory, there were programmers who prided themselves on their ability to write more efficient assembly code than could be produced by an optimizing compiler. Such programmers have largely gone the way of the punch card and the rotary dial telephone. Modern performance programmers rely on their ability to trick a compiler into producing the machine code they desire. Refactoring source code to obtain better performance is their stock in trade.

We contend that the concern for performance continues to exert a deleterious influence on the clarity and maintainability of commercial software. We further contend that it ought to be a goal of compiler writers (and language designers) to reduce that influence *without compromising the performance programmer's ability to write high-performance source code*.

Performance programmers recognize the costs in clarity and maintainability in refactoring code to obtain better performance. When the expected benefit is great enough, they will put up with these costs. However, if the expected benefit is less great programmers will be tempted to forgo it. Easy access to partial inlining, using annotations say, would allow programmers to realize the benefit without incurring the cost.

For example, arrays in X10 are implemented in the standard library. Since arrays are considered to be performance critical, the code implementing them has been carefully crafted to match the capabilities and limitations of the X10 optimizer. In X10, arrays are more general than in C or Java. The underlying index space for X10 arrays can be multidimensional. Along each dimension, it can start (and end) at arbitrary integer values. It may also have gaps in it. In the general case, a method call verifies that an array index is valid on every array access. A `Rail` is an X10 array whose index space is one-dimensional, zero-based, and contiguous. An array index operation in X10 gets translated to an inlined call to a method that checks to see if the array is a `Rail`. If it is, the necessary bound checks are performed in-line. Otherwise, a general index verification method is called. The array index method (which bears an `@Inline` annotation) *is* the guarded partial inlining of the general index verification method. The code implementing the array class would be a little clearer if the latter method bore an `@PartialInline` annotation and the former did not exist.

Ubiquitously available partial inlining might, or might not, significantly improve the performance of existing code. It would certainly allow performance-obsessed programmers to obtain high-performance programs with less distortion to the natural shape of their source code.

8. Conclusion

This paper presents two new approaches to partial inlining. These approaches differ from previous ones in that they do not assume that the compiler has a priori runtime (or profile) information as to which paths through the method to be partially inlined are *hot*. *Guarded partial inlining* relies on programmer hints to identify candidate methods (and their hot paths). It causes a test and the purported hot path to be inlined along with a call to the original method in the case that the test fails. *Context-driven partial inlining* uses information available at a call site to customize a version of

the callee to be inlined. Guarded partial inlining becomes equivalent to context-driven partial inlining precisely when the compiler can deduce that the guarded partial inlining test must hold.

An admittedly fragile prototype of guarded partial inlining is described along with a somewhat contrived example. Completely and partially inlined versions of this example are observed to run slightly faster than an uninlined version. The partially inlined version is slightly bigger than the uninlined version but dramatically smaller than the completely inlined version. Plans for a more robust implementation of guarded partial inlining along with context-driven partial inlining are presented.

Finally, it is argued that pervasive availability of partial inlining would have a positive impact on how real world programs are written.

References

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349303. URL `http://doi.acm.org/10.1145/349299.349303`.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167488. URL `http://doi.acm.org/10.1145/1167473.1167488`.

[4] D. Bruening and E. Duesterwald. Exploring optimal compilation until shapes for an embedded just-in-time compiler. In *Proceedings of the 2000 ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*, Dec. 2000.

[5] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 1–15, New York, NY, USA, 1991. ACM. ISBN 0-201-55417-8. doi: 10.1145/117954.117955. URL `http://doi.acm.org/10.1145/117954.117955`.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094852.

[7] J. Dean and C. Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, LFP '94, pages 273–282, New York, NY, USA, 1994. ACM. ISBN 0-89791-643-3. doi: 10.1145/182409.182489. URL `http://doi.acm.org/10.1145/182409.182489`.

[8] A. Gal, C. W. Probst, and M. Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8. doi: 10.1145/1134760.1134780. URL `http://doi.acm.org/10.1145/1134760.1134780`.

[9] 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, pages 246–256, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL `http://dl.acm.org/citation.cfm?id=2190025.2190071`.

[10] J.-P. Lee, J.-J. Kim, S.-M. Moon, and S. Kim. Aggressive function splitting for partial inlining. In *Proceedings of the 2011 15th Workshop on Interaction between Compilers and Computer Architectures*, INTERACT '11, pages 80–86, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4441-0. doi: 10.1109/INTERACT.2011.14. URL `http://dx.doi.org/10.1109/INTERACT.2011.14`.

[11] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for java. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 138–152, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00904-3. URL `http://dl.acm.org/citation.cfm?id=1765931.1765947`.

[12] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification. `http://x10.sourceforge.net/documentation/languagespec/x10-223.pdf`, 2012.

[13] A. Sewe, J. Jochem, and M. Mezini. Next in line, please!: exploiting the indirect benefits of inlining by accurately predicting further inlining. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 317–328, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1183-0. doi: 10.1145/2095050.2095102. URL `http://doi.acm.org/10.1145/2095050.2095102`.

[14] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a java just-in-time compiler. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 312–323, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781166. URL `http://doi.acm.org/10.1145/781131.781166`.

[15] J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, lan-*

guages, and applications, OOPSLA '01, pages 166–179, New York, NY, USA, 2001. ACM. ISBN 1-58113-335-9. doi: 10.1145/504282.504295. URL `http://doi.acm.org/10.1145/504282.504295`.

[16] X10 Perf. X10 performance tuning. `http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html`, 2012.

[17] P. Zhao and J. N. Amaral. Function outlining and partial inlining. In *Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, SBAC-PAD '05, pages 101–108, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2446-X. doi: 10.1109/CAHPC.2005.26. URL `http://dx.doi.org/10.1109/CAHPC.2005.26`.

Faster Work Stealing With Return Barriers

Vivek Kumar
Australian National University
vivek.kumar@anu.edu.au

Stephen M. Blackburn
Australian National University
steve.blackburn@anu.edu.au

Abstract

Work-stealing is a promising approach for effectively exploiting software parallelism on parallel hardware. A programmer who uses work-stealing explicitly identifies potential parallelism and the runtime then schedules work, keeping otherwise idle hardware busy while relieving overloaded hardware of its burden. However, work-stealing comes with substantial overheads. Our prior work demonstrates that using the exception handling mechanism of modern VMs and gathering the runtime information directly from the victim's execution stack can significantly reduce these overheads.

In this paper we identify the overhead associated with managing the work-stealing related information on a victim's execution stack. A return barrier is a mechanism for intercepting the popping of a stack frame, and thus is a powerful tool for optimizing mechanisms that involve scanning of stack state. We present the design and preliminary findings of using return barriers on a victim's execution stack to reduce these overheads. We evaluate our design using classical work-stealing benchmarks. On these benchmarks, compared to our prior design, we are able to reduce the overheads by as much as 58%. These preliminary findings give further hope to an already promising technique of harnessing rich features of a modern VM inside a work-stealing scheduler.

Categories and Subject Descriptors D1.3 [*Software*]: Concurrent Programming – Parallel programming; D3.4 [*Programming Languages*]: Processors – Code generation; Compilers; Optimization; Run-time environments.

General Terms Design, Performance.

Keywords Scheduling, Task Parallelism, Work-Stealing, X10, Managed Languages.

1. Introduction

Work-stealing [4, 6, 9, 13] is a widely used framework for allowing programmers to explicitly expose *potential* parallelism. A work-stealing scheduler within the underlying language runtime schedules work exposed by the programmer, exploiting idle processors and unburdening those that are overloaded. Work-stealing schedulers are used in various programming languages, such as Cilk [6] and X10 [4], and in application frameworks, such as the Java fork/join framework [9] and Intel Threading Building Blocks [13].

Although the specific details vary among the various implementations of work-stealing schedulers, they all incur some form of sequential overhead as a necessary side effect of enabling dynamic task parallelism. In our prior work [8] we analyzed the sources of sequential overhead in work-stealing schedulers and designed and implemented two optimized work-stealing runtimes that significantly reduce overheads by building upon existing runtime services of modern JVMs. Our results demonstrate that we can almost completely remove the sequential overhead from a work-stealing implementation and therefore obtain performance improvements over sequential code even at modest core counts.

The techniques that we use in our prior work are:

1. Using the victim's execution stack as an *implicit* deque.
2. Modifying the runtime to extract execution state directly from the victim's stack and registers.
3. Dynamically switching the victim to *slow* versions of code to reduce coordination overhead.

Once a thief finds a potential victim, it exploits the runtime's existing yieldpoint mechanism to force the victim to yield. The victim is stopped while the steal is performed. However, when steals are frequent, forcing the victim to yield at each steal becomes costly.

The contributions of this paper are: 1) a detailed study of the cost associated with stealing from a victim in our prior work; 2) a detailed design for reducing this overhead and 3) evaluation of our new design using classical work-stealing benchmarks.

The rest of the paper is structured as follows. Section 2 discusses the related work. Section 3 provides the relevant background. Section 4 discusses our evaluation methodology. Section 5 discusses the motivation for this work. Section 6 explains the design of our new system. Section 7 discusses the performance evaluation of our new design and finally section 8 concludes the paper.

2. Related Work

The ideas behind work-stealing have a long history which includes lazy task creation [12] and the MIT Cilk project [6], which offered both a theoretical and practical framework. In [8], we exploit rich features of modern virtual machines and build a new work-stealing framework for X10 language's finish-async programming model [16]. We steal only one task at a time, just as in Java's fork/join framework [9].

Though stealing one task at a time has been shown to be sufficient to optimize computation along the 'critical path' to within a constant factor [1, 3], several authors have argued that the scheme can be improved by allowing multiple tasks to be stolen at a time [2, 5, 7, 10, 14]. In this work, we explore a different approach to minimizing steal overheads. We exploit the return barrier mechanism to optimize the steal process. The return barrier mechanism is not a new, it is used in various garbage collectors [15, 17], however, to our knowledge, it has not been applied to work-stealing until now.

3. Background

This section provides a brief overview of work-stealing runtimes in the context of the X10 (Try-Catch) implementation from our prior work [8].

Abstractly, work-stealing is a simple concept. *Worker threads* maintain a local set of *tasks* and when local work runs out, they become a *thief* and seek out a *victim* thread from which to *steal* work.

The elements of a work-stealing runtime are often characterized in terms of the following aspects of the execution of a task-parallel program:

Fork A fork describes the creation of new, potentially parallel, work by a worker thread. The runtime makes new work items available to other worker threads.

Steal A steal occurs when a thief takes work from a victim. The runtime provides the thief with the execution context of the stolen work, including the execution entry point and sufficient program state for execution to proceed. The runtime updates the victim to ensure work is never executed twice.

Join A join is a point in execution where a worker waits for completion of a task. The runtime implements the synchronization semantics and ensures that the state of program reflects the contribution of all the workers.

Our try-catch work-stealing framework is currently designed for the X10 **finish**–**async** style programming model. In the section below, we will discuss this model briefly.

3.1 Work-Stealing in X10

X10 is a strongly-typed, imperative, class-based, object-oriented programming language. X10 includes specific features to support parallel and distributed programming. A computation in X10 consists of one or more asynchronous activities (light-weight tasks). A new activity is created by the statement **async** S. To synchronize activities, X10 provides the statement **finish** S. Control will not return from within a finish until all activities spawned within the scope of the finish have terminated.

```
1  def fib(n:Int):Int {
2    if (n < 2) return n;
3
4    val a:Int;
5    val b:Int;
6
7    finish {
8      async a = fib(n-1);
9      b = fib(n-2);
10   }
11
12   return a + b;
13 }
```

Figure 1. X10's **finish**–**async** style programming model

X10 restricts the use of a local mutable variable inside **async** statements. A mutable variable (var) can only be assigned to or read from within the **async** it was declared in. To mitigate this restriction, X10 permits the asynchronous initialization of final variables (**val**). A final variable may be initialized in a child **async** of the declaring **async**. A definite assignment analysis guarantees statically that only one such initialization will be executed on any code path, so there will never be two conflicting writes to the same variable. Figure 1 shows X10's **finish**–**async** style programming model.

We have modified the X10 compiler to compile to X10 (Try-Catch) and hence, X10 (Try-Catch) represents a new backend for work-stealing.

3.2 X10 (Try-Catch) Java implementation

3.2.1 Leveraging Exception Handling Support

Most JVMs, including Jikes RVM, have very efficient support for exception handling. Because exceptions are important and potentially expensive, JVM implementers have invested heavily in optimizing the mechanisms. We leveraged these optimized mechanisms to efficiently implement the peculiar control flow requirements of work-stealing. The X10 (Try-Catch) system annotates **async** and **finish** blocks by wrapping them with **try/catch** blocks with special work-stealing exceptions. These allow the X10 (Try-Catch) runtime to walk the stack and identify all **async** and **finish** contexts within which a thread is executing.

The work-stealing implementation consists of following basic phases, each of which require special support from the runtime or library:

1. Initiation. (Allow tasks to be created and stolen atomically).
2. State management. (Provide sufficient context for the thief to be able to execute stolen execution).
3. Termination. (Join tasks and ensure correct termination).

3.2.2 Initiation

X10 (Try-Catch) avoids maintaining an explicit deque for workers. Instead, marker try/catch blocks are used to communicate the current deque state to the work-stealing runtime. The execution stack of a thread is used as an implicit deque.

A thief identifies its potential victim by checking the *steal flag* maintained by each worker thread. The steal flag is set as the first action within an **`async`**. This flag is cleared when the worker or a thief determines that there is no more work to steal. Once a thief finds a potential victim, it uses the runtime's yieldpoint mechanism to force the victim to yield — the victim is stopped while the steal is performed. The yieldpoint mechanism is used extensively within the runtime to support key features, including exact garbage collection, biased locking, and adaptive optimization. Only one thief is allow to steal from one victim at any given time. Different thieves can steal from different victims concurrently.

The head of the task deque corresponds to the top of the execution stack. The list of continuations (from newest to oldest) is established by walking the set of **`catch`** blocks that wrap the current execution state. Each worker has a `stealToken` that acts as a *tail* for the deque. None of the continuations found after this point is stolen. This `stealToken` also helps in guaranteeing atomicity. It acts as a *roadblock* for the worker and thieves to prevent either running or stealing continuations past that point.

3.2.3 State Management

When a task is stolen, the thief must: 1) acquire all state required to execute that task, and 2) provide an entrypoint to begin execution of the task, and 3) be able to return or combine return state with other tasks. Work-stealing implementations typically meet requirements 1) and 3) through the use of *state objects* that capture the required information about the task, and provide a location for data to be stored and shared across multiple tasks. Requirement 2) is handled differently depending on the execution model. This aspect of our X10 (Try-Catch) implementation is discussed in more detail below.

3.2.4 Termination

Control must only return from a **`finish`** context when all tasks within the context have terminated. To support this, a singly linked list is maintained. A node is lazily created for each **`finish`** context in which a task is stolen. This node maintains an atomic count of the number of active tasks in the **`finish`** context, and provides a location for the partial results to be passed between threads. When a thread decrements the atomic count to zero, it becomes responsible for running the continuation of the **`finish`** context. The X10 (Try-Catch) runtime will deliver a special exception at the appropriate point, allowing the thread to extract partial results and continue out from the **`finish`**.

3.3 Return Barriers

A return barrier, like a write barrier, allows the runtime to intercept a common event, and (conditionally) interpose special semantics. In the case of a write barrier, a runtime typically interposes itself on pointer field updates, conditionally remembering updates of pointers in certain conditions. On the other hand, a return barrier [15, 17], interposes special semantics upon the return from a method (which corresponds to the popping of a stack frame). One use for a return barrier is to keep track of a 'low water mark' for each stack since some particular event, such as the last garbage collection. In a language where pointers into the stack are not permitted, there is a guarantee that no part of the stack below the low water mark has been changed since the low water mark was set. This information can be used to reduce the overhead of stack scanning. In our work, we use a return barrier to 'protect' the victim from stumbling upon an active thief. We do this by installing a return barrier above the stealable frames, allowing the victim to ignore all steal activty that occurs below the low water mark. Only when the frame above the return barrier is popped does the victim need to consider the activity of thief.

A naive implementation of a return barrier would require some (modest) code to be executed upon every return, just as a write barrier is typically executed upon every pointer update. In our implementation we insert a trampoline that hijacks the return of a particular frame (the return is redirected to our trampoline). The trampoline executes the return barrier semantics (which may include re-installing the return barrier at a lower frame), before returning to the correct frame (whose address was remembered in a side data structure). This barrier has absolutely no overhead in the common case, and only incurs a modest cost when the frame targeted by the return barrier is popped.

We now motivate the work presented in the remainder of the paper with an analysis of the cost of stealing in a well-tuned work-stealing runtime.

4. Methodology

4.1 Benchmarks

Because the primary goal of our work is to reduce the cost of steal operations, we have intentionally selected benchmarks with high steal rates We have used a collection of three benchmarks, which are briefly described below. In each case we ported the benchmark to plain Java (for the sequential case), as well as to our JavaWS (Try-Catch) system, described below.

The three benchmarks we have are:

Jacobi Iterative mesh relaxation with barriers: 100 steps of nearest neighbor averaging on 1024×1024 matrices of doubles (based on an algorithm taken from Fork-Join [9]).

LUD Decomposition of 1024×1024 matrices of doubles (based on algorithm from Cilk-5.4.6 [11]). Block size of 16 is used to control the granularity.

Heat Diffusion Heat diffusion simulation across a mesh of size 4096×1024 (based on algorithm from Cilk-5.4.6). Leaf column size of 10 is the granularity parameter. Timestep used is 200.

4.2 Hardware Platform

All experiments were run on a machine with two Intel Xeon E7530 Nehalem processors. Each processor has six cores running at 1.87 GHz sharing a 12 MB L3 cache. The machine was configured with 16 GB of memory.

4.3 Software Platform

Jikes RVM Version 3.1.2. We used the production build.

4.4 Measurements

For each benchmark, we ran six invocations, with three iterations per invocation, where each iteration performed the kernel of the benchmark five times. We report the mean of the final iteration, along with a 95% confidence interval based on a Student t-test. For each invocation of the benchmark, the total number of garbage collector threads is kept the same as application threads. A heap size of 921 MB is used across all experiments. Other than this, we preserve the default settings of Jikes RVM.

All of our benchmarks make extensive use of arrays. The sequential versions of the benchmarks use Java arrays directly. However, the X10 compiler is not currently able to optimize X10 array operations directly into Java array operations, but does so through a wrapper with get/set routines. To avoid the overhead associated with this indirect array accesses, we use a system that we call JavaWS (Try-Catch), which uses try-catch work-stealing but operates directly on Java arrays without X10.

5. Motivating Analysis

As we noted in our prior work, although work-stealing is a very promising mechanism for exploiting software parallelism, it can bring with it formidable overheads to the simple sequential case. In our prior work, we exploited rich features that pre-exist within the JVM implementation to significantly reduce these overheads. We heavily rely on the yieldpoint mechanism of the JVM to stop the victim so that the thieves can extract the required information directly from the execution stack. However, when the steals become frequent, stopping the victim inside a yieldpoint at each steal may amount to a significant overhead. In our prior work, we performed a study to understand steal ratios across a range of benchmarks. That analysis shows that steals are generally infrequent, ranging from $1/10$ to $1/10^5$ steals/task (see Figure 6(a)).

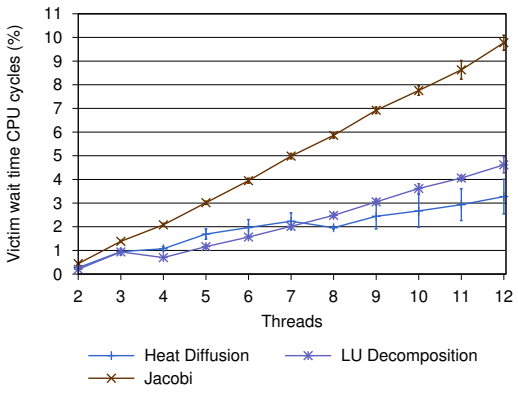


Figure 2. Percentage of CPU cycles lost by victims waiting for the steals to finish in default JavaWS (Try-Catch).

To further motivate our designs, we now measure: 1) the steal rate (steals/sec); 2) the overhead imposed by the steal mechanism upon the victims.

5.1 Steal Rate

The steal ratio is only one dimension of the steal overheads. We now measure the steal rate (steals/sec), which is shown in Figure 7(a). Steal rate is calculated by dividing the total number of steals by the benchmark execution time. This gives an indication of how frequently we are forcing the victim to execute the yieldpoint. The results obtained for the Jacobi in both these studies clearly indicates that a benchmark with a low steal *ratio* may have a high steal *rate*.

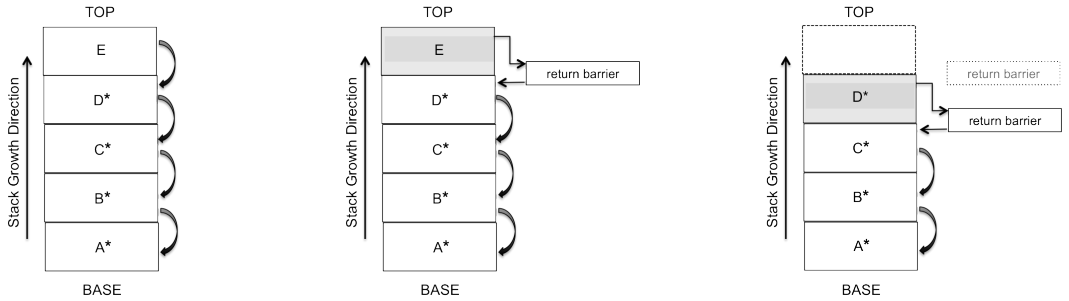
5.2 Stealing Overhead

Next we measured the cost of a steal as imposed upon the victim by the thief. We did this by measuring the percentage of CPU cycles lost by the victim while waiting for the thief to finish accessing its execution stack. We measure the CPU cycles required for the entire execution of the benchmark using hardware performance counters. We used Jikes RVM's existing mechanisms for measuring the number of cycles spent waiting for the thief to access the victim's stack. We start a timer when a victim is forced to execute yieldpoint. The timer is stopped when the thief finishes accessing the victim's execution stack and unlocks the victim from the yieldpoint. These cycles are summed for all the steals over the benchmark execution. At the end of execution we get the total CPU cycles lost to the thief. These are cycles which the victim could have utilized for carrying out its execution, had it not been forced to yield to the thief. We calculate this percentage of lost cycles and plot in Figure 2.

These numbers show that the steal overhead is as much as 10% for Jacobi.

6. Implementation

This section discusses the modifications made inside the JavaWS (Try-Catch) runtime [8] to use return barriers for



(a) Initial state of stack.

(b) Thief installs the return barrier on victim's execution stack.

(c) Thief installs the return barrier on victim's execution stack.

Figure 3. The victim's stack, installation, and movement of the return barrier.

assisting the steal process from a victim. We use the JavaWS (Try-Catch) runtime because it is the best performing work-stealing runtime.

6.1 Installing the First Return Barrier.

Figure 3(a) depicts a typical snapshot of a victim's execution stack. The stack frames having a stealable continuation are marked with a * in this figure. The newly executed methods occupy the stack frame slots on the top of execution stack. Each stack frame is recognized with the help of a frame pointer. The value stored inside this pointer is the frame pointer of the last executed method. The other important information, which is of interest to us, is the return address, which normally forms part of a stack frame. It holds the address where the control should be transferred after unwinding to the caller frame.

Once a thief has decided to rob this victim and discovers that the victim does not yet have a return barrier installed, the thief halts the victim by forcing it to execute the yieldpoint mechanism. After the victim has stopped, the thief starts scanning the stack frames to identify the oldest stealable continuation. In this case it is the frame *A*. However, before the thief unwinds down to the frame pointer for *A*, it notices that the first (newest) available continuation is *D*. It installs a return barrier to intercept the return from method *E*. The return address stored inside *E* is modified and this new address is now that of the return barrier trampoline method. Thus upon returning from *E*, the victim will find itself inside the return barrier trampoline. The trampoline maintains the address of frame *D*, and will return execution to *D* one the trampoline has been executed. The return address from the barrier is now the old value from frame *E*. Figure 3(b) depicts the victim's modified execution stack.

The victim holds a private boolean field *stealInProgress*, which is now marked as *true* by the thief. The thief then creates a clone of the entire stack of the victim and only then allows its victim to continue. The victim continues the rest of its computation, oblivious to the activity of the thief, while the thief proceeds with the steal process using the cloned stack. The cloned stack is also used for offline manipulation of the callee save registers.

6.2 Synchronization Between Thief and Victim During Steal Process.

When the victim finishing executing method *E*, it will return via the trampoline method of the return barrier. Before unwinding to *D*, the return barrier code checks the *stealInProgress* flag to determine whether a steal is in progress and whether the continuation being stolen is *D*. If *D* is not being stolen, the victim will reinstall its return barrier at the next available continuation after *D*. This is the method *C* in this case. The victim then returns to *D* and continues computation until it hits the return barrier again, at which point it repeats the process. Figure 3(c) shows this newly modified stack frame of the victim.

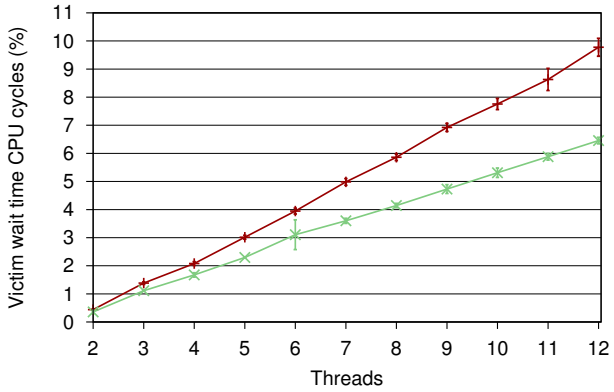
In the case where the victim discovers that the frame below the return barrier is being currently being stolen, it will wait on a condition variable. Once the steal is complete, the thief resets the victim's field *stealInProgress* to false and signals the victim. The victim now unwinds to the stolen frame and becomes a thief.

6.3 Stealing From a Victim with Return Barrier Pre-installed.

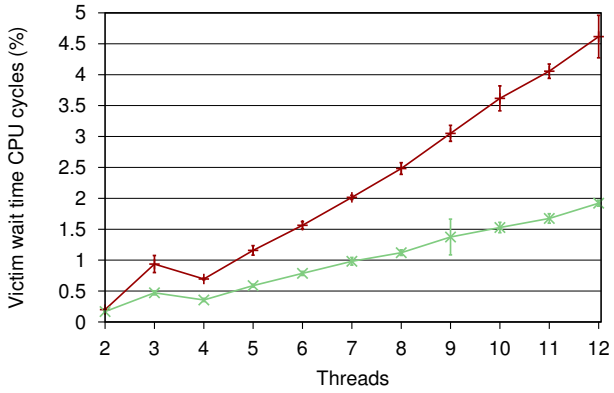
If, upon identifying a potential victim, the thief finds that there is already a return barrier installed on the victim's execution stack, it does not force the victim to execute the yieldpoint, but marks the victim's field *stealInProgress* as *true*. The steal and the victim's computation then happen concurrently, with the victim oblivious to the steal. The previously cloned stack of the victim is reused for the steal process and offline manipulation of the callee save registers.

7. Performance Evaluation

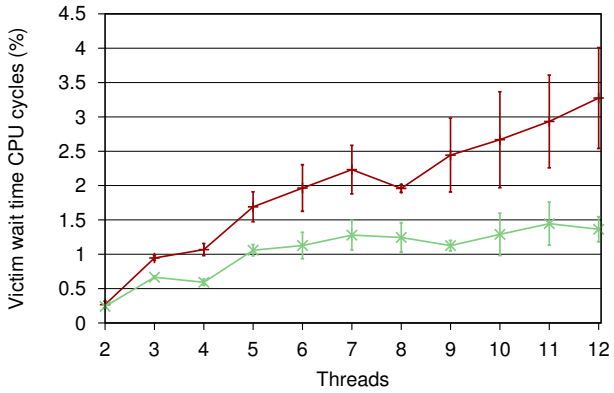
We start with measuring the percentage of CPU cycles lost by victims while waiting for the steal to finish. We then measure the speedup on both the modified and unmodified



(a) Jacobi

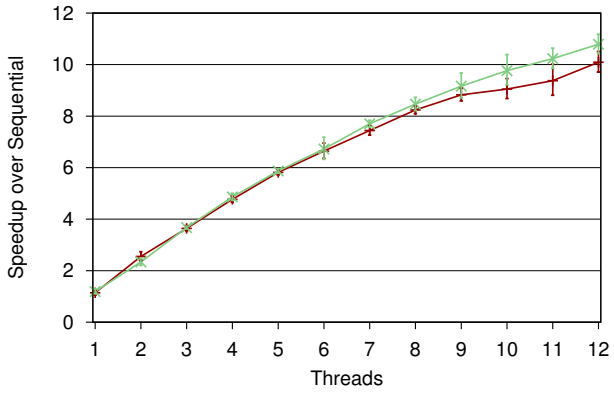


(b) LUD

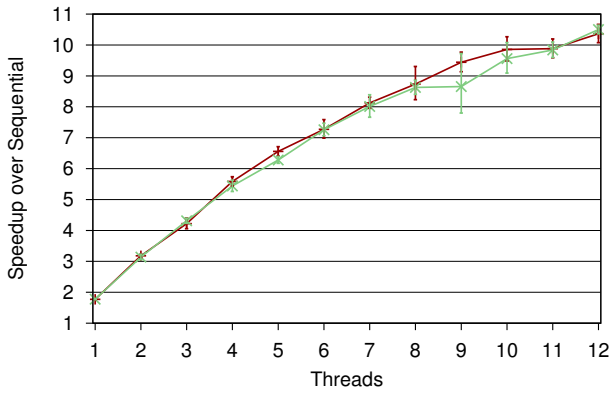


(c) Heat Diffusion

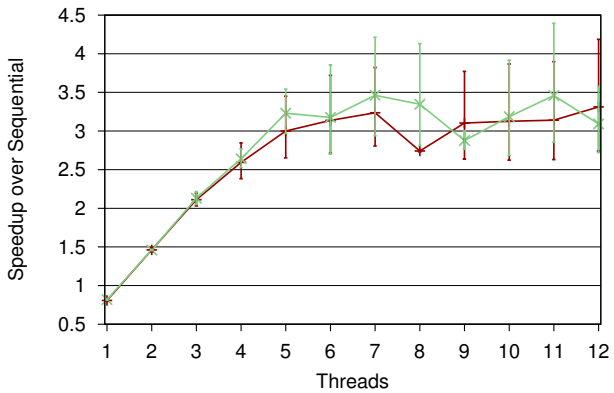
Figure 4. Percentage of CPU cycles lost by victims waiting for the steals to finish.



(a) Jacobi



(b) LUD



(c) Heat Diffusion

Figure 5. Speedup relative to sequential Java.

JavaWS (Try-Catch). We finish by examining the effect of the return barrier on the steal ratio and steal rate.

7.1 Cost of Stealing

We measure the percentage of CPU cycles lost by the victims while waiting for the thieves to gather the required information from its execution stack. Figure 4 shows this percentage for both the modified and unmodified systems. These results illustrate that using return barrier on a victim's execution stack can reduce the stealing overhead significantly, and by as much as 58%.

7.2 Work-Stealing Performance

Figure 5 shows the scalability of the benchmarks both on the default JavaWS (Try-Catch) and JavaWS (Try-Catch) with return barriers. The results show that scalability is not statistically significantly affected by the use of return barriers.

7.3 Steal Ratio and Steal Rate

To ensure that our design did not change the steal ratio and the steal rate, we measured them on our new design. The results are the figure 6 and figure 7. The results exactly match with the results on the unmodified system.

7.4 Summary

These results demonstrate that our approach is extremely effective at reducing the overhead associated with managing the work-stealing related information on a victim's execution stack. However, we do not notice increased speedup or increased steal rate even though our new design almost halved the stealing overhead. This is because the actual overhead is not large and hence there is no noticeable increase in performance of the system. However, for the cases where the cost of steals is significant, our new design will be helpful.

8. Conclusion

We believe that work-stealing will be an increasingly important approach for effectively exploiting software parallelism on parallel hardware. As an extension to our prior work, here we analyzed the overhead associated with stealing from the victim's execution stack. We designed a return barrier-based victim execution stack and evaluated it using a set of classical work-stealing benchmarks. Our preliminary results demonstrate that we can significantly reduce the overhead of the stealing process.

As future work, we plan to evaluate the steal-*N* strategy, which steals more than one frame at a time, and integrate it with our current work. We also plan to continue exploring ways in which JVM runtime mechanisms can be adapted to further improve work-stealing.

References

[1] N. Arora, R. Bolumofe, and C. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129. ACM, 1998.

[2] P. Berenbrink, T. Friedetzky, and L. Goldberg. The natural work-stealing algorithm is stable. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 178–187. IEEE, 2001.

[3] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46 (5):720–748, 1999. ISSN 0004-5411.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0.

[5] J. Dinan, D. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 53. ACM, 2009.

[6] M. Frigo, H. Prokop, M. Frigo, C. Leiserson, H. Prokop, S. Ramachandran, D. Dailey, C. Leiserson, I. Lyubashevskiy, N. Kushman, et al. The Cilk project. *Algorithms*, 1998.

[7] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 280–289. ACM, 2002.

[8] V. Kumar, D. Frampton, S. Blackburn, D. Grove, and O. Tardieu. Work–stealing without the baggage. In *Proceedings of the 2012 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, Tucson, Arizona, USA, Oct. 2012. ACM.

[9] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3.

[10] R. Lüling and B. Monien. A dynamic distributed load balancing algorithm with provable good performance. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 164–172. ACM, 1993.

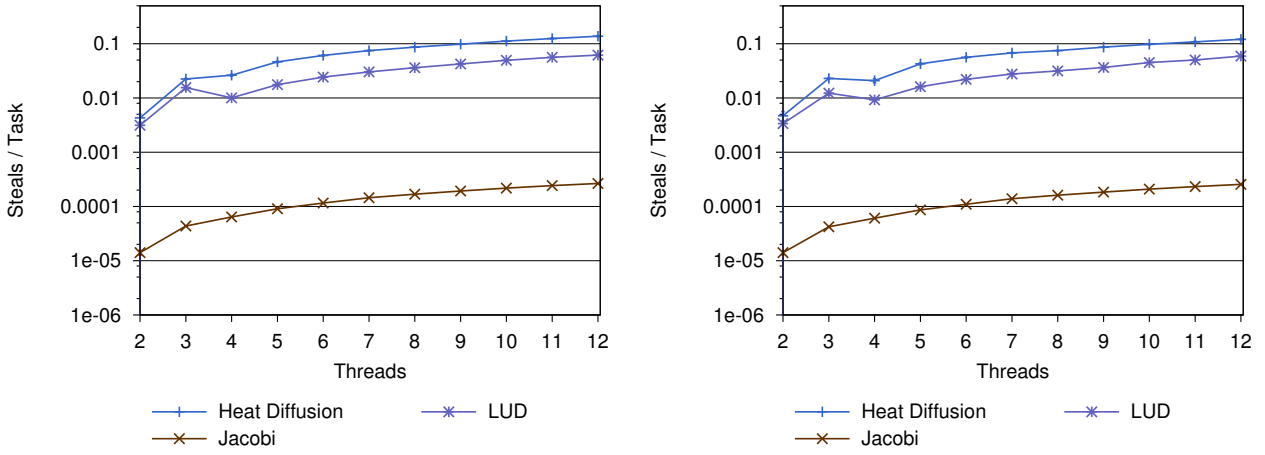
[11] MIT. The Cilk project. URL `http://supertech.csail.mit.edu/cilk/index.html`.

[12] E. Mohr, D. Kranz, and R. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *Parallel and Distributed Systems, IEEE Transactions on*, 2(3): 264–280, 1991.

[13] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[14] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 237–245. ACM, 1991.

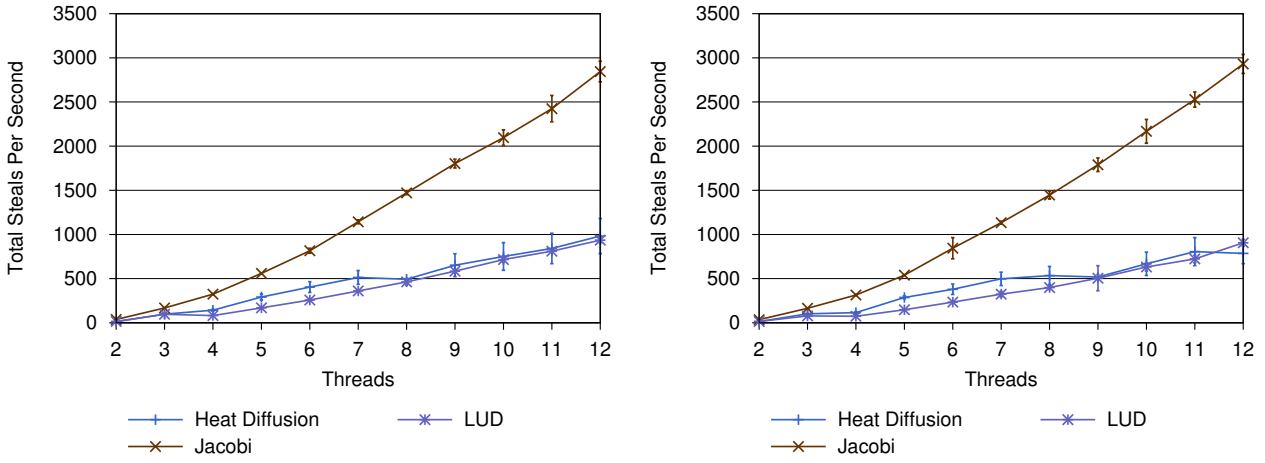
[15] H. Saiki, Y. Konaka, T. Komiya, M. Yasugi, and T. Yuasa. Real-time gc in jerty vm using the return-barrier method. In *Object-Oriented Real-Time Distributed Computing, 2005.*



(a) Steal ratio on default JavaWS (Try-Catch)

(b) Steal ratio on JavaWS (Try-Catch) with return barrier

Figure 6. Steal ratio comparison with our new implementation.



(a) Steal rate on default JavaWS (Try-Catch)

(b) Steal rate on JavaWS (Try-Catch) with return barrier

Figure 7. Steal rate comparison with our new implementation.

ISORC 2005. Eighth IEEE International Symposium on, pages 140–148. IEEE, 2005.

[16] O. Tardieu, H. Wang, and H. Lin. A work-stealing scheduler for X10's task parallelism with suspension. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 267–276, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1.

[17] T. Yuasa. Real–time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

Compilation Queuing and Graph Caching for Dynamic Compilers *

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

Thomas Würthinger
Oracle Labs
thomas.wuerthinger@oracle.com

Abstract

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

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

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

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

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

General Terms Algorithms, Languages, Performance

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

1. Introduction

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

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

In this paper, we present techniques for reducing those two metrics. The compilation of a method uses significant CPU resources such that it does not pay off to compile rarely executed methods [11]. Therefore, it is not sufficient to just reduce the number of invocations before a method gets compiled. Instead the runtime must make sure that the hottest methods are compiled earlier, because their compilation results in the biggest improvement on the program's execution speed. The aggressive inlining of dynamic compilers results in many methods being parsed several times (see Figure 4). This makes caching strategies for the method's intermediate representation an interesting target for optimizing the compilation time without major compiler modifications. This paper contributes the following:

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

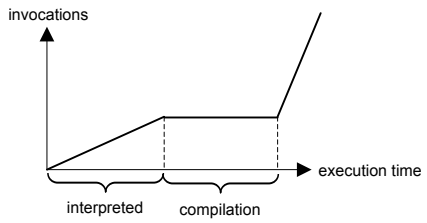


Figure 1. Executing a method in a managed runtime.

* This work was supported by Oracle.

VMIL '12, Tucson, Arizona

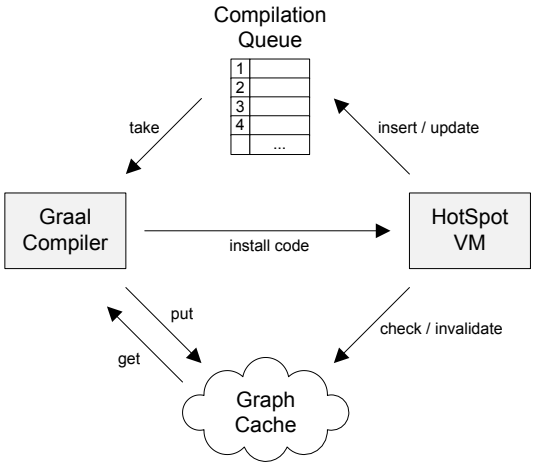


Figure 2. System architecture of the Graal VM.

2. System Overview

We implemented our system in the context of the Graal OpenJDK project [13]. The Graal VM is a modification of the Java HotSpot™ VM where the compiler and the compilation queue are replaced with an implementation in Java. Figure 2 gives a schematic view of the system components that are relevant for the techniques described in this paper. The HotSpot™ VM starts executing the Java program in the interpreter. When a method gets hot, the VM inserts it into the compilation queue with a priority value that determines the hotness. If the hotness changes over time, the VM has the ability to update the priority value.

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

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

The cached graphs can be built using optimistic assumptions about the current state of the application. Such assumptions can be invalidated by subsequent changes in application behavior. Therefore, the runtime must be capable of removing graphs from the cache if one of their assumptions no longer holds. This invalidation can also be necessary for installed machine code. If machine code associated with an invalid assumption is executed, it is *deoptimized* [9], and execution continues in the interpreter.

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

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

3. Compilation Queuing

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

In our system the compiler needs to make two choices when compiling methods: *When* a method should be considered for compilation, and *which priority* the method should have as soon as it is ready to be compiled.

3.1 Detecting Hot Methods

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

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

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

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

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

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

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

3.2 Method Hotness Model

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

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

Size-Based. Smaller methods are considered to be more important than larger methods, because their compilation finishes faster and so their execution in compiled form will sooner improve the overall performance. With aggressive inlining this does not hold any more - the method size does not correlate with compilation times enough to make this algorithm beneficial.

Kulkarni's Method. Kulkarni [11] introduces an algorithm that scales the method invocation count by a global counter which

is incremented for every method invocation:

$$\text{Priority} = \frac{\text{method invocation count}}{\text{global count} - \text{global count at first invocation}}$$

This technique favors methods that have recently been invoked very frequently.

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

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

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

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

This approach provides a good initial approximation of method importance and can react quickly if a method suddenly becomes more important. This happens, for example, when an application enters a new phase [12].

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

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

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

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

3.3 Compilation Queue with Priorities

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

- The compilation queue itself is an efficient thread-safe priority queue, along with worker threads that take elements from the queue and process them. The number of worker threads defaults to the number of available cores in the system.

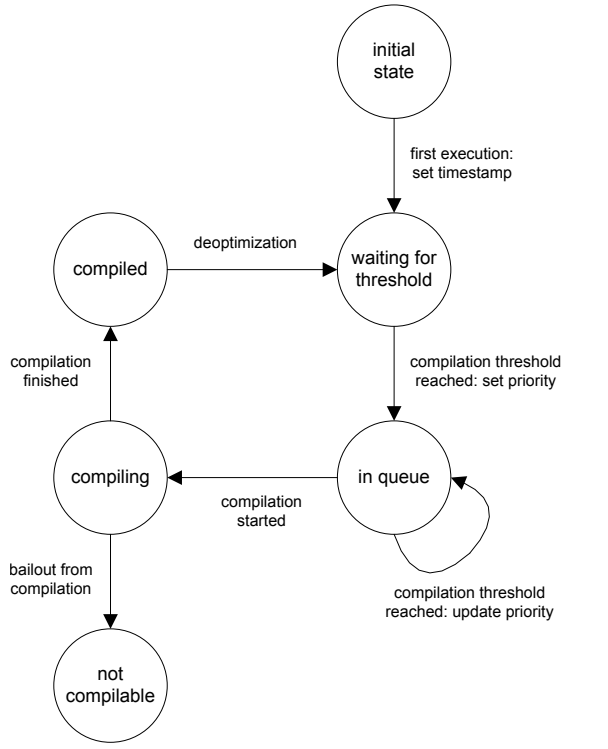


Figure 3. Overview of the states and transitions of methods.

- Each time a method is ready to be compiled a *compilation task* is created and put into the compilation queue. The compilation task is associated with the method in such a way that it is possible to look up the current compilation task for a method.
- The compilation tasks are ordered according to the priorities (i.e., the invocation counter speeds) of their methods.
- If two methods happen to have the same priority then they will be ordered on a FIFO basis, so that the oldest compilation task is executed first. This rarely happens in practice since the priorities are very fine grained.
- Compilation tasks may be reordered when the priority of a method improves. Since reordering within the compilation queue is a time-consuming task it is only performed if the priority changes significantly (by at least a factor of two). The original compilation task is canceled (by simply setting a flag on it) and a new one is inserted with the new priority. Not trying to remove the old compilation task from the queue helps avoid concurrency problems.

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

We also implemented two versions of compile queues that use thread priorities to communicate the importance of the current compilation task to the operating system:

These modifications, however, did not lead to a measurable performance increase under normal benchmarking conditions. Benchmark measurements in highly contended scenarios, which should theoretically benefit from these thread priority optimizations, also did not show any consistent improvement. The lower compilation thread priority does not play a large enough role given the multitude of factors that come into play in highly contended situations.

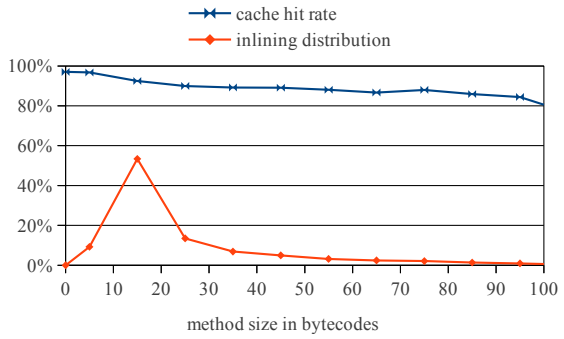


Figure 4. Cache hit rate and inlining distribution during typical benchmarks, in relation to the inline method's size.

4. Graph Caching

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

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

Given the right infrastructure the compiler could cache intermediate results (i.e., compiler graphs) for methods that are inlined multiple times. The high reuse will translate to the high cache hit rates shown in Figure 4. However, such a system needs to fulfill some specific requirements and it also needs to be aware of how some parts of the compiler interact with the cache. These requirements and interactions are explained in the following sections.

4.1 Late Inlining

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

- The compiler needs to make the inlining decision very early. This forces certain optimizations, such as global value numbering and canonicalization, to be done already during bytecode parsing, which makes parsing more complicated.

  Also, these optimizations often work better when they are performed as a separate step.

- Optimizations that happen later, like escape analysis [2], cannot perform inlining even if they see that it would be beneficial.

Graal, on the other hand, does *late inlining* (like, for example, JRockit [14]). It does not contain any facilities to perform inlining during the bytecode parsing step. This significantly decreases the complexity of Graal's bytecode parsing component, because it does not need to deal with multiple method scopes at once.

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

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

4.2 Garbage Collection

Most Java JIT compilers use some variant of explicit region memory allocation [7], also called zone allocation or arena allocation. This means that all temporary data structures allocated during compilation are freed en bloc at the end of the compilation process.

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

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

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

4.3 Assumptions

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

Static Assumptions deal with the state surrounding a method, e.g., the list of loaded classes. For example, they are used in the optimization of potentially polymorphic calls. If class hierarchy analysis guarantees that only one class has been loaded as the potential receiver type, the call can be replaced by a static call.

If other potential receiver classes are loaded later on, the assumption is violated and the code depending on the assumption needs to be invalidated.

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

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

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

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

When the behavior of a method that has been compiled changes, some dynamic assumptions might not hold any more. In case the execution reaches such an assumption, the only thing a system without a graph cache needs to do is to invalidate the compiled

version of the method. When using a graph cache, however, this also means that if the dynamic assumption originates from within a cached graph, this graph needs to be evicted from the cache. Subsequent compilations would otherwise reuse a cached graph that represents an outdated behavior, possibly leading to repeated deoptimizations.

It would be possible to immediately invalidate all compiled methods that contain the now invalid cached graph, but this is not necessary since the compiled methods will either not expose the changed behavior or eventually deoptimize on their own. The caching system only needs to be aware that it might receive requests to evict graphs from the cache that have already been evicted.

4.4 Graph Caching Implementation

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

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

5. Evaluation

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

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

DaCapo 9.12-bach is split into 14 independent sub-benchmarks [6], each of which represents a usage pattern commonly found on Java Virtual Machines: avrora, batik, eclipse, fop, h2, jython, luindex, lusearch, pmd, sunflow, tomcat, tradebeans, tradesoap and xalan.

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

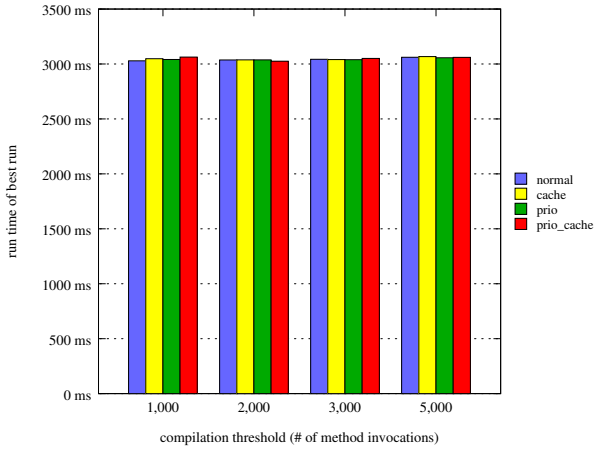


Figure 5. Geometric mean of the best runs of all benchmarks, for different VM configurations, clustered by compilation threshold.

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

The evaluation contains results for four different VM configurations:

normal Unmodified compilation queue policy, without graph caching.

cache Unmodified compilation queue policy, with graph caching.

prio Priority queue compilation policy, without graph caching.

prio_cache Priority queue compilation policy, with graph caching.

5.1 Peak Performance

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

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

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

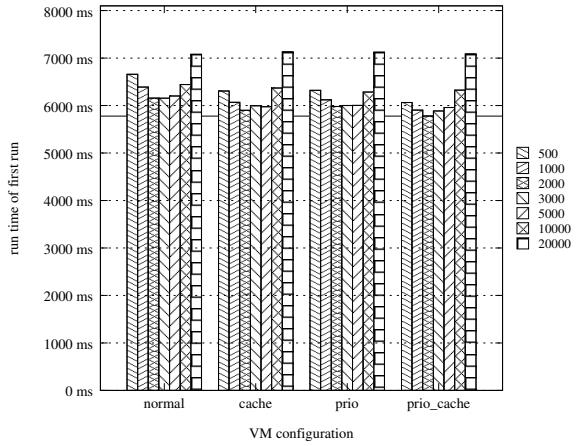


Figure 6. Geometric mean of the first run of all benchmarks, for different compilation thresholds, clustered by VM configuration.

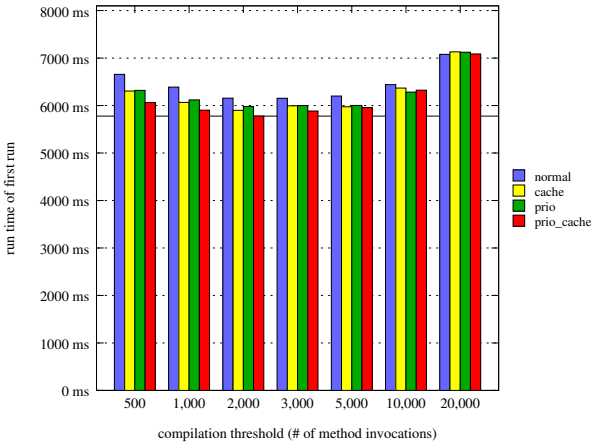


Figure 7. Geometric mean of the first run of all benchmarks, for different VM configurations, clustered by compilation thresholds.

inlined the first time, it is mature enough so that caching the graph does not have a negative effect on peak performance.

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

Some DaCapo sub-benchmarks have a high variation (e.g., h2 and pmd), while others produce very consistent results (e.g., batik and eclipse). While some sub-benchmarks show a slight increase or decrease in peak performance from the compilation queue and graph caching optimizations, the overall average (geometric mean of all sub-benchmarks), as shown in Figure 5, is not influenced by the VM configuration.

5.2 First Run Performance

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

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

5.2.1 Overall Influence on First Run Performance

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

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

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

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

5.2.2 Detailed Influence on First Run Performance

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

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

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

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

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

5.2.3 Compilation Queue Optimizations

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

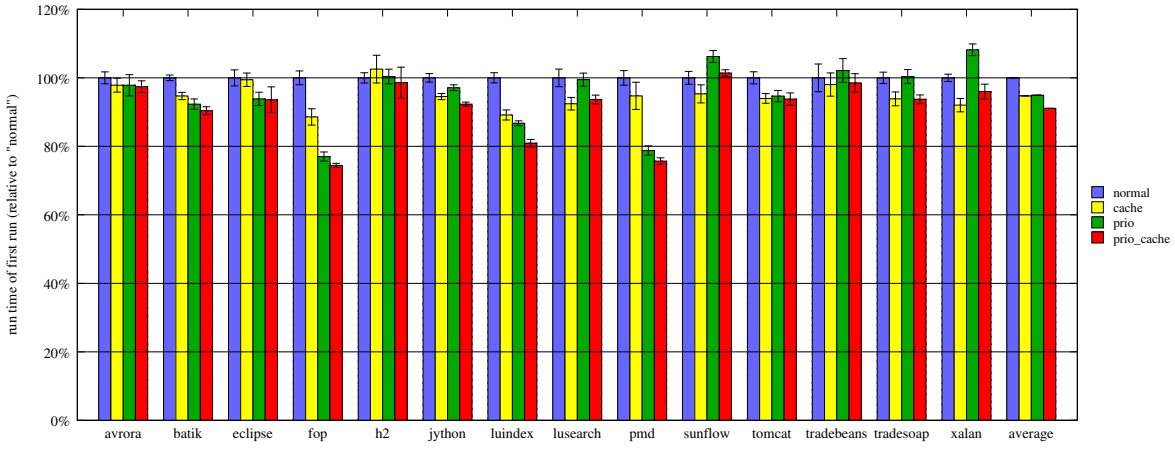


Figure 8. Benchmark results for the first run with a compilation threshold of 500 invocations.

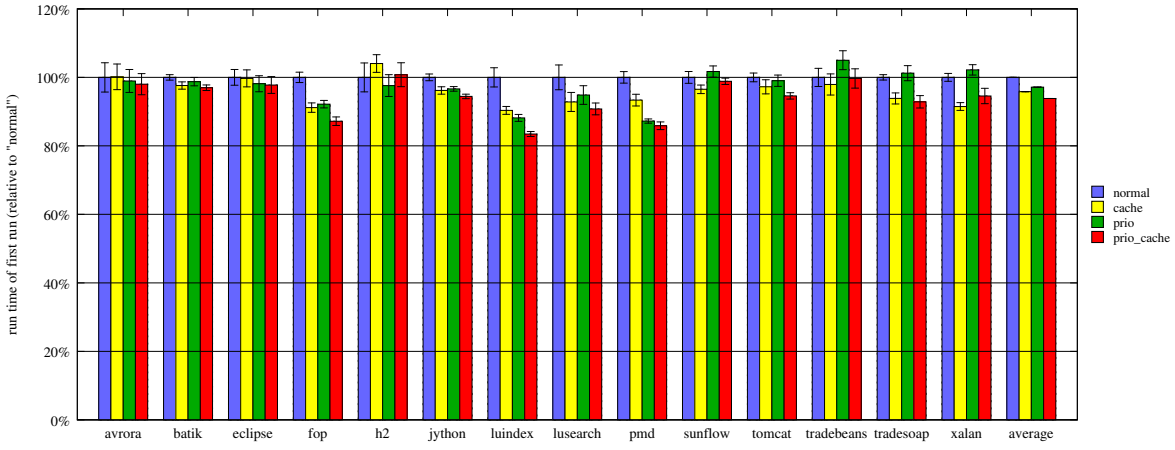


Figure 9. Benchmark results for the first run with a compilation threshold of 2,000 invocations.

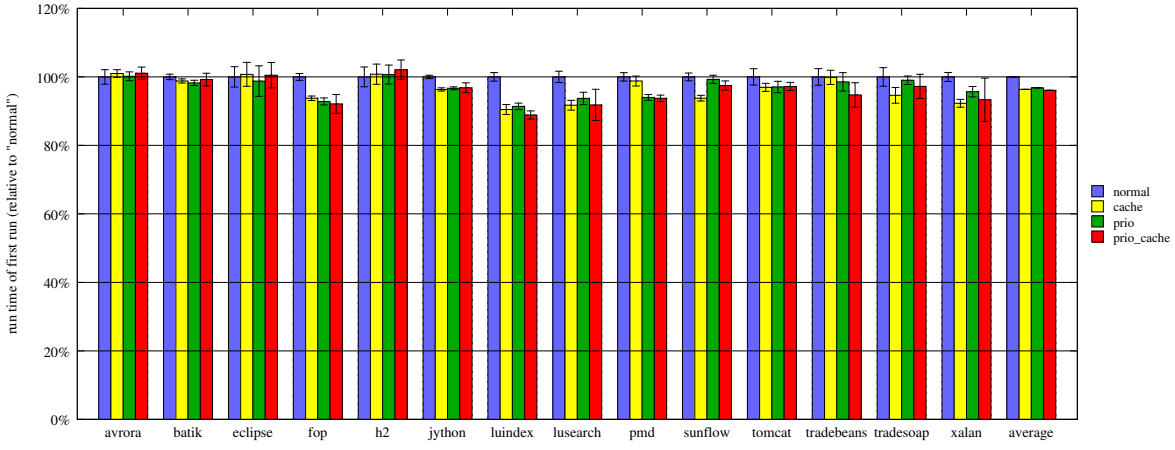


Figure 10. Benchmark results for the first run with a compilation threshold of 5,000 invocations.

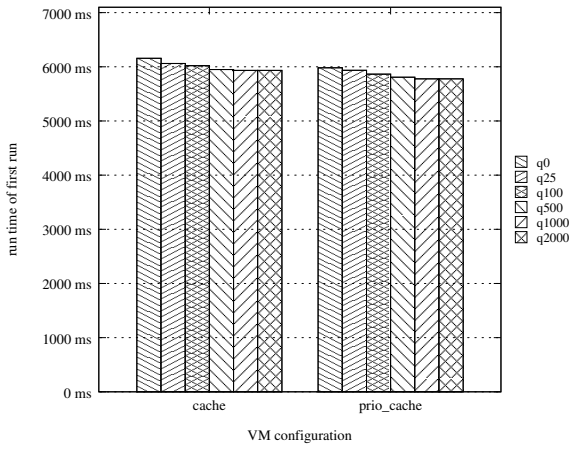


Figure 11. Geometric mean of the first run of all benchmarks, for different graph cache sizes, clustered by VM configuration.

5.2.4 Graph Caching

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

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

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

We measured the total time spent compiling methods during all DaCapo sub-benchmarks. The total time without graph caching is 225 seconds, while the total time with graph caching is 185 seconds, which amounts to an 18% reduction in compilation time.

6. Related Work

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

It is generally assumed that there at least 2 modes of execution: a baseline mode which requires very little time to start but runs rather slowly and one or more optimized modes which take more time to start, but run faster afterwards. Such a system can be efficient because applications usually spend most of their time running only a small portion of their code. For example, SELF-93 [8] uses a fast baseline compiler and sends only a selected number of methods to a better, but slower, optimizing compiler. This technique is used in almost all JIT systems when both performance and interactivity are required.

Such systems can be further improved by running the compiler on a separate thread, in order to avoid pauses in the application code an thus make the startup phase of the application even less noticeable. This background compilation has been proposed by Plezbert and Cytron [16], is used in the HotSpot JVM [15] and has also been studied in the Jalapeño JVM [10]. In the case of background compilation there is often more than one compilation thread, in order to drain the compilation queue faster. This is especially beneficial in a multi-processor environment where it is more likely that the compilation really happens in parallel to the application's execution.

The most common way of selecting methods for compilation is by looking at how much they are being used by the application at runtime. However, doing so means that when these method are compiled they have already been running a lot in a slow mode. Various systems have been proposed to predict which methods will be beneficial to compile before they become hot and thus further reduce the cost of startup. For example, Campanoni et al. [4] inspect the code of the methods and use analysis such as static branch prediction to chose the method which are the most likely to be needed. Kulkarni [11] also proposes to use profiling information from previous runs of the same application to be able to immediately start to compile the methods which were hot in previous runs.

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

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

Further stressing the importance of these studies, Nagpurkar et al. [12] has shown that bursts of compilations do not only happen at startup but also when the application enters a new phase of its execution.

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

7. Future Work

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

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

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

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

8. Conclusions

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

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

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

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

Acknowledgments

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

References

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press. doi: http://doi.acm.org/10.1145/1167473.1167488.

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

[3] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. *SIGPLAN Not.*, 46(6):74–85, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993508. URL http://doi.acm.org/10.1145/1993316.1993508.

[4] S. Campanoni, M. Sykora, G. Agosta, and S. Crespi Reghizzi. Dynamic look ahead compilation: A technique to hide jit compilation latencies in multicore environment. In O. de Moor and M. Schwartzbach, editors, *Compiler Construction*, volume 5501 of *Lecture Notes in Computer Science*, pages 220–235. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-00721-7.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL http://doi.acm.org/10.1145/115372.115320.

[6] DaCapo Project. *The DaCapo Benchmark Suite*, 2012. http://dacapobench.org/.

[7] D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI '98 Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 313–323. ACM, 1998. doi: http://dx.doi.org/10.1145/277652.277748.

[8] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, July 1996. ISSN 0164-0925. doi: 10.1145/233561.233562. URL http://doi.acm.org/10.1145/233561.233562.

[9] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143114. URL http://doi.acm.org/10.1145/143095.143114.

[10] C. Krintz, D. Grove, D. Lieber, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *SOFTWARE: PRACTICE AND EXPERIENCE*, 31:200–1, 2000.

[11] P. A. Kulkarni. JIT compilation policy for modern machines. In *OOPSLA '11 Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 773–788. ACM, 2011. doi: http://dx.doi.org/10.1145/2048066.2048126.

[12] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online Phase Detection Algorithms. In *CGO '06 Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–123. IEEE Computer Society, 2006. doi: http://dx.doi.org/10.1109/CGO.2006.26.

[13] OpenJDK Community. *Graal Project*, 2012. http://openjdk.java.net/projects/graal/.

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

[15] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™ server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, pages 1–12. USENIX, 2001.

[16] M. P. Plezbert and R. K. Cytron. Does "just in time" = "better late than never"? In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 120–131, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. doi: 10.1145/263699.263713. URL http://doi.acm.org/10.1145/263699.263713.

Author Index