An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler

Gilles Duboscq* Thomas Würthinger† Lukas Stadler* Christian Wimmer† Doug Simon†
Hanspeter Mössenböck*

*Institute for System Software, Johannes Kepler University Linz, Austria †Oracle Labs
{duboscq, stadler, moessenboeck}@ssw.jku.at
{thomas.wuerthinger, christian.wimmer, doug.simon}@oracle.com

Abstract

We present a compiler intermediate representation (IR) that allows dynamic speculative optimizations for high-level languages. The IR is graph-based and contains nodes fixed to control flow as well as floating nodes. Side-effecting nodes include a framestate that maps values back to the original program. Guard nodes dynamically check assumptions and, on failure, deoptimize to the interpreter that continues execution. Guards implicitly use the framestate and program position of the last side-effecting node. Therefore, they can be represented as freely floating nodes in the IR. Exception edges are modeled as explicit control flow and are subject to full optimization. We use profiling and deoptimization to speculatively reduce the number of such edges. The IR is the core of a just-in-time compiler that is integrated with the Java HotSpot VM. We evaluate the design decisions of the IR using major Java benchmark suites.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Algorithms, Languages, Performance

Keywords Java, virtual machine, just-in-time compilation, intermediate representation, speculative optimization

1. Introduction

Speculative optimizations in a just-in-time compiler rely on profiling feedback that characterizes program behavior. The compiler can focus on the most likely paths taken through the program and cut off cold branches that are highly unlikely to be taken. This reduces code size and opens additional optimization opportunities, because the cold branch and its influence on program state need not be taken into account when compiling a method. In high level languages such as Java, a single operation can include an implicit control-flow split. For example, a field access includes a null check on the receiver that can throw an exception. This control-flow path is not visible in the original source program, but the compiler still has to handle it. In this context, the speculative reduction of those control-flow paths is important. Dynamic languages can profit even more from this reduction, because an operation in such a language typically has a more complex control-flow structure.

When one of the cold branches is still taken, program execution must continue in the interpreter. This mechanism to jump from the optimized machine code back to the interpreter is called *deoptimization* [17]. It requires bookkeeping in the compiled code that allows the reconstruction of the interpreter state at deoptimization points. This state includes the values of local variables and the operand stack. Due to escape analysis, some of those values may reference virtual objects that need to be allocated during deoptimization.

The design of an IR largely influences whether a compiler writer can express optimizations in a simple way [6]. In the context of speculative optimizations, it also decides whether such optimizations are possible at all and how much additional footprint is required for enabling deoptimization. This paper contributes the description and evaluation of a novel IR with the following properties:

- Speculative optimizations using deoptimization points which map optimized program state back to interpreter state.
- Insertion, movement, and coalescing of deoptimization points without any constraints.
- Explicit representation of exception edges as normal control flow and omits them speculatively based on profiling feedback.

VMIL '13 October 28, Indianapolis, Indiana, USA

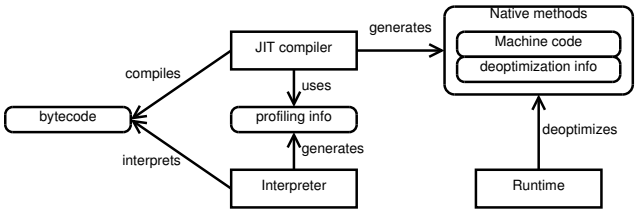

Figure 1. System overview.

2. System Overview

Our IR is part of Graal OpenJDK project [23] and its Graal Virtual Machine (VM). The Graal VM is a modification of the Java HotSpot VM. The Java HotSpot VM uses mixed-mode execution: all methods are initially interpreted, and frequently executed methods are scheduled for just-in-time (JIT) compilation. Thus execution starts in the interpreter, which is slow but has low startup costs. It also generates profiling information during interpretation. When the VM decides that a method should be compiled, it makes a request to the compiler, which can optimize Java bytecodes better with the help of the profiling information. Figure 1 shows a schematic overview of this system.

The Java HotSpot VM has two JIT compilers: the client compiler and the server compiler. The client compiler [20] aims at fast compilation speed, while the server compiler [24] aims at better optimization at the expense of slower compilation speed. Both use speculative optimizations and deoptimization. In contrast to our IR, however, their IR is not centered around the principle of speculative optimizations; deoptimization is an add-on feature expressed by a dedicated instruction, not a first-class concept as described in this paper.

The Graal compiler can be used to replace the standard compilers of the Java HotSpot VM. It is written in Java and aims to produce highly optimized code through extensive use of speculative optimizations. An optimization is speculative when the compiler makes an assumption that it cannot guarantee during compilation. Instead, it requires the runtime system to monitor the assumption and discard the machine code when the assumption no longer holds. For example, the compiler can replace a virtual method call with a static call and then inline a method if there is currently only one implementation of the method available. If later class loading adds another implementation, the assumption no longer holds and the code is deoptimized.

In order to do so, the runtime uses deoptimization information generated by the compiler to reconstruct the state of the interpreter (i.e., the state of the virtual machine) from the state of the physical machine. In the Java VM, this state consists of all local variables and operand stack slots. In the context of escape analysis, where allocations of method-local objects are eliminated, the deoptimization information also contains the mapping necessary to reconstruct such objects

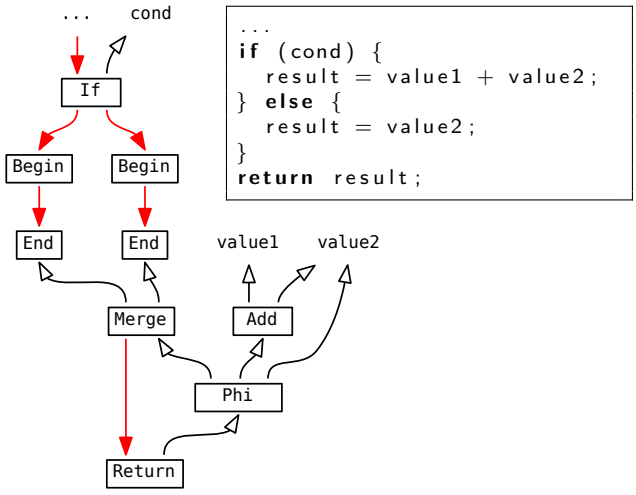

Figure 2. Example graph with control-flow and data-flow edges.

on the heap. Figure 1 shows that this data is directly associated with the machine code of a compiled method.

3. Intermediate Representation

Our IR is based on a directed graph structure. Each node produces at most one value and it is in static single assignment (SSA) form [8]. To represent data flow, a node has *input* edges (also called use-def edges) pointing "upwards" to the nodes that produce its operands. To represent control flow, a node has *successor* edges pointing "downwards" to its different possible successors. Note that the two kinds of edges point in opposite directions. In the example in Figure 2, the If node has one input edge pointing "upwards" for the condition and two successor edges pointing "downwards" for the true and false branches. This mirrors the edge direction usually found in a data-flow graph and a control-flow graph, respectively. In summary, the IR graph is a superposition of two directed graphs: the data-flow graph and the control-flow graph.

The IR automatically maintains reverse edges for all node connections. Therefore *usage* edges (also called def-use edges) and *predecessor* edges can also be used to traverse the graph. Unlike the direct edges, these reverse edges are implicitly maintained and not ordered, which means that they can only be observed as an unordered set of usages or predecessors.

Each block of the control-flow graph begins with a Begin node. Two or more lines of control flow can merge at a Merge node, which is a subclass of the Begin node. These Merge nodes need to know the order of their predecessors, so that an SSA Phi node can select the correct input value for each of the merge predecessors. As predecessor edges are not ordered, Merge nodes are connected to their prede-

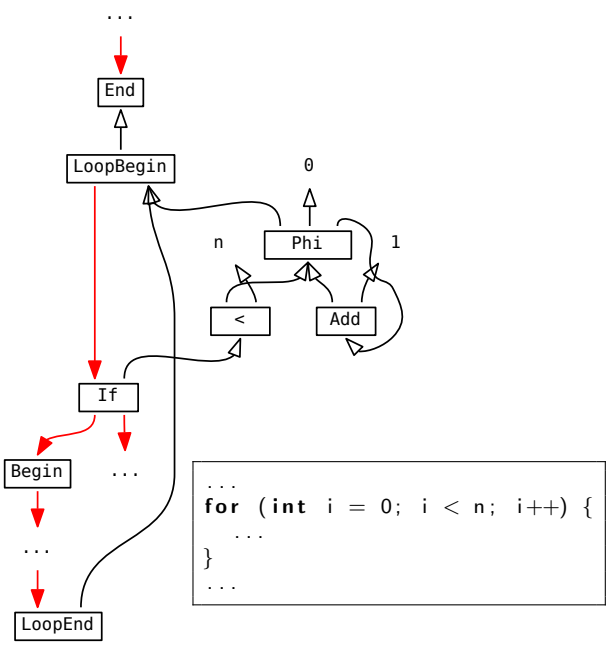

Figure 3. Example graph for a loop.

cessors using input edges pointing to End nodes. These End nodes are at the end of the control flow of the merge's predecessors. The Phi nodes are attached to their Merge node through a special input edge. This structure is illustrated in Figure 2. The filled arrows show the control flow, the empty arrows show the data flow.

For simplicity, the IR only represents reducible loops. Methods containing irreducible loops are not compiled and are left for the interpreter to execute. In Java bytecodes produced from a Java source program without obfuscation, there can never be any method with irreducible loops. The IR models loops explicitly: the loop header is a LoopBegin node. The back-edges of a loop are represented with LoopEnd nodes, which are attached to their LoopBegin through an input edge. Since the LoopBegin node merges the control flow of the loop pre-header and backward edges, Phi nodes can be attached to LoopBegin nodes. This structure is illustrated in Figure 3.

For the transparent management of LoopBegin nodes merging control flow, LoopBegin node is a subclass of Merge node and LoopEnd node is a subclass of End node.

3.1 Fixed and Floating Nodes

Nodes are not necessarily fixed to a specific point in the control flow. The control-flow splits and merges provide a backbone around which most other nodes are *floating*. For example, the Add node in Figure 2 is floating. These floating nodes are only constrained by their data-flow edges and through additional dependencies such as memory dependencies. The constraints maintain the program semantics but allow more freedom of movement for operations. When a node needs to express a dependency to a specific branch, it can have an input edge pointing to a specific Begin node.

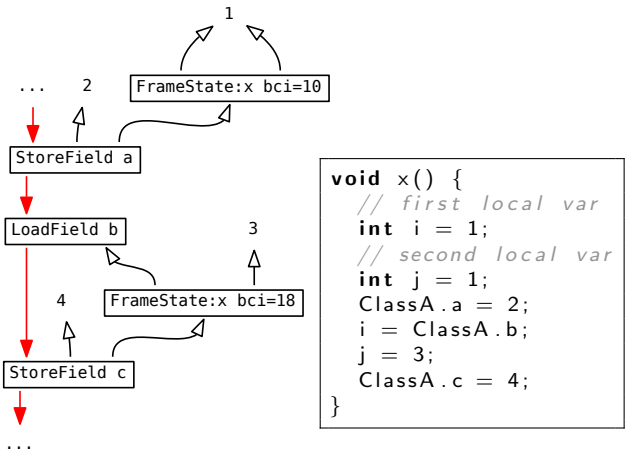

Figure 4. FrameState nodes. The two local variable slots are expressed as inputs to the FrameState nodes.

As explained by Click [5], this representation simplifies a number of optimizations by removing the burden of maintaining a valid schedule or performing code motion. When emitting code, the IR is fully scheduled by assigning each node to a block in the control-flow graph and by ordering the nodes inside each block. During scheduling, simple heuristics are used to apply some code motion optimization such as hoisting code out of loops.

4. Deoptimization

4.1 Framestates

We enable speculative optimization through the use of deoptimization, which transfers execution from optimized code to the interpreter. To be able to do this, we need to know

- where we want to continue interpreting the code, and
- how to reconstruct the VM state at the continuation point from the physical machine state of the optimized code.

To know where we want to continue, we keep a reference to the method and a bytecode index (bci). For the VM state, we keep a mapping of the local variables and operand stack slots to their values in the IR. When emitting deoptimization information we can then map them to their physical location.

In our IR we keep track of this data for nodes that can have side effects on the global state of the virtual machine such as memory writes, method invocations and monitor acquires or releases. We call those nodes *state split* nodes. For these nodes, we keep the information about the state the virtual machine would be in *after* they execute.

The mapping from the VM state to IR nodes is not part of the state split nodes themselves, but it is expressed as

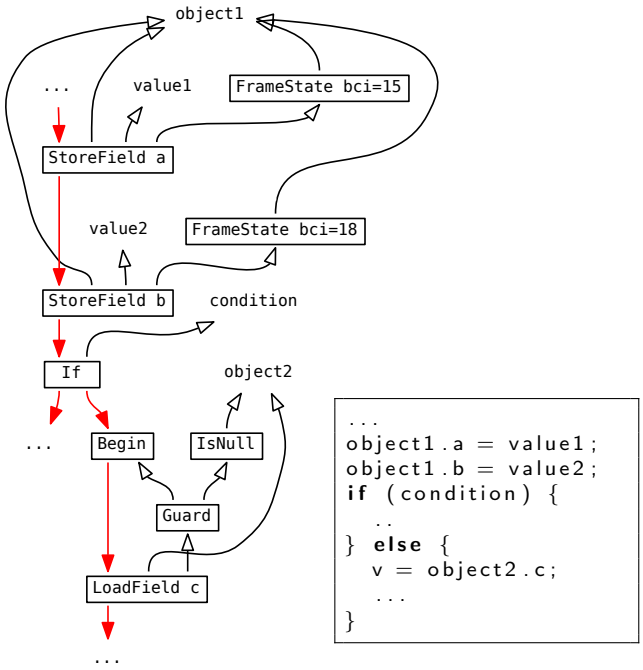

Figure 5. Framestates and guards. The LoadField node is guarded by a null check. FrameState nodes are kept for the state after two StoreField nodes.

FrameState nodes. These nodes contain the method and bci of the continuation point and they have inputs for the local variables and operand stack slots. State splits have a FrameState node as an input that describes the VM state after they have been executed. An example of this is shown in Figure 4: In this example the two stores to static fields have side effects and therefore need references to FrameState nodes. These nodes describe the VM state after they have been executed. In contrast, the load of the static field does not have side effects[1], so it does not need a FrameState.

Before emitting code we need to associate the deoptimization information with the nodes that can trigger deoptimization. To do so, we use the deoptimization information of the last dominating state split node. This means that all of the instructions between two state split nodes deoptimize to the same point, which corresponds to the deoptimization information of the first state split node. Some instructions that have already been executed by the compiled version are thus re-executed by the interpreter after deoptimization. This re-execution is sound because none of the re-executed instructions modify the global state (otherwise they would be state split nodes themselves).

For example, the Guard node of Figure 5 performs a null check before the object is accessed. In the unexpected case that the object is indeed null, it triggers deoptimization,

[1] we assume ClassA has already been initialized.

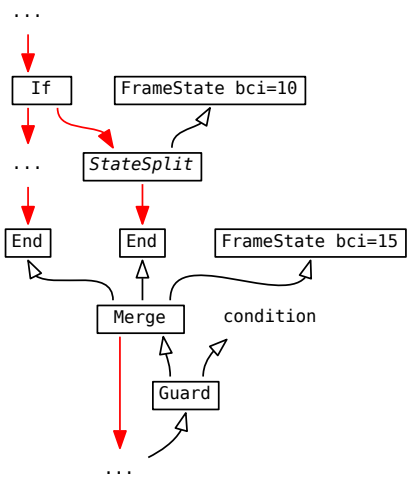

Figure 6. Example where a Merge node requires a FrameState node.

so it needs deoptimization information. In this example, we select the deoptimization information contained in the FrameState bci=18 node attached to the second StoreField node (StoreField b).

Using the last dominating state split requires keeping the deoptimization data at a merge point, if any of the merging branches contains a state split node (as illustrated in Figure 6). In that case, any deoptimization triggered after the Merge node deoptimizes to this Merge node. Otherwise, a deoptimization triggered by a node after the Merge node could deoptimize to the dominator of the Merge node and thus could re-execute the side effects of the branch containing the state split node.

This model gives us a complete mapping, so that, once the graph is scheduled, any node which may cause deoptimization can be associated with deoptimization information.

4.1.1 Hierarchical Framestates

A FrameState node can express the VM state of exactly one method activation. Method inlining, however, introduces multiple nested activations into one IR instance, so that the deoptimization information consists of multiple nested VM states. This is encoded in the IR by letting a FrameState node reference another FrameState node that describes the surrounding activation. This reference to the surrounding activation is called the *outer* FrameState. An example of such an outer FrameState is shown in Figure 7.

It is important to note that multiple inner FrameStates can refer to the same outer FrameState, which leads to trees of FrameStates in the IR.

4.1.2 Virtual Objects

Some advanced optimizations require even more information for mapping to the VM state. For example, escape analysis [3] leads to scalar replacements of objects, which means

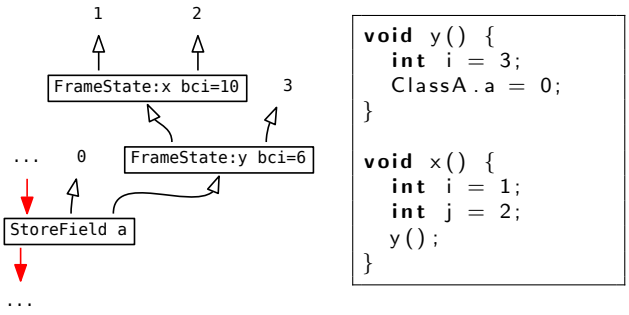

Figure 7. Nested FrameState nodes when compiling x() with an inlined instance of y(). The local variables in x() have the values 1 and 2, while the local variable in y() has the value 3.

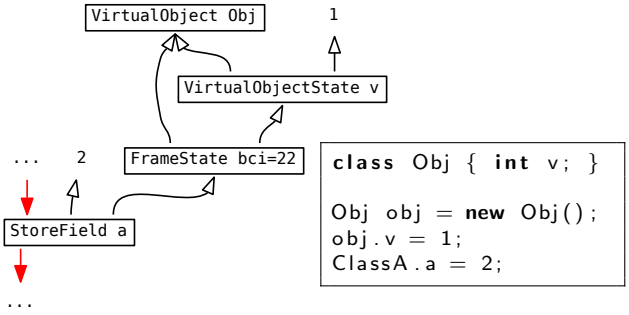

Figure 8. Representation of virtual objects and their current state via FrameState nodes.

that objects that would be allocated according to source level semantics are not actually allocated. In this case the mapping to VM state includes information about which object allocations were eliminated and what the current contents of these objects is. With this information, the objects can be reconstructed during deoptimization as explained by Kotzmann and Mössenböck [19].

Figure 8 shows an example of how virtual objects are expressed in the IR. Each virtual object is represented as a VirtualObject node, and every local variable or stack slot that refers to the eliminated object points to this node. The actual contents of the virtual object are expressed as VirtualObjectState nodes, which are added to the leaf FrameState nodes. The reason for this split representation is that multiple inner FrameStates can refer to the same outer FrameState, and a virtual object that is part of the outer local variables or stack slots can have different field values for inner FrameStates.

4.2 Guards

The Graal IR uses Guard nodes to check the assumptions taken for speculative optimizations. They take as input a boolean condition that needs to be checked. If the check fails, a deoptimization is triggered. Any node that depends on the assumption checked by a guard takes the guard as an input, thereby ensuring that the node and this guard are always ordered correctly during scheduling.

For example, we can decide to inline a method at a virtual call site based on profiling data showing that only one of the possible target methods was ever called. We then need to check this assumption at run time by inserting a Guard node which checks the type of the receiver. The entry of the inlined call takes this guard as an input so that the assumption is checked before entering the inlined code.

Once a guard is associated with deoptimization information, which specifies where it should deoptimize to and in which state, it becomes difficult to move it without breaking the ordering of side effects. Thanks to the complete mapping provided by FrameState nodes, the Guard nodes do not need to be associated with specific deoptimization information before code emission and can thus be moved freely in the IR.

Guard nodes are floating nodes which are constrained by their usages, by their input condition, and by an additional input which points to a Begin node. This additional input ensures that the Guard node stays in the correct branch. We refer to this as a Guard node that is *anchored* to a branch. For example, the LoadField node in Figure 5 takes the assumption that object2 is not null using a Guard node. This guard should only be checked in the correct branch of the if statement because the condition could be related to object2 being null or not. If the guard was to be scheduled above the If node, the program would still execute correctly, but it could deoptimize too often and deteriorate program performance.

When Guard nodes are inserted, they are anchored to the furthest dominator of the node that they need to guard which still post-dominates this node. This is the first dominating Begin node whose predecessor is a control-split. This ensures that the guard is tested only if it is needed but also that guards cluster below Begin nodes. This is important because Guard nodes, like most floating nodes, are subject to global value numbering so multiple Guard nodes with the same condition and anchor are coalesced automatically.

After their insertion, we can still change the node a guard is anchored to. For example, we can replace the two Guard nodes in Figure 9 with one Guard node anchored at a different position, without worrying about any ordering with respect to other guards or nodes with side effects.

5. Exception Handling

When compiling a language that has an exception mechanism, exception edges have to be added to control-flow graph to account for the control-flow transitions that can happen when an exception is raised. In a language such as Java, a

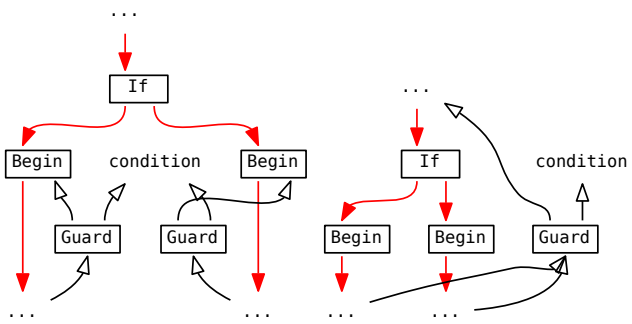

Figure 9. Hoisting and coalescing two Guard nodes above an If node.

significant number of instructions may throw an exception and, as a result, a large number of exception edges is needed.

In our IR, we can take the optimistic assumption that exceptions are not thrown and thus completely eliminate most of the exception edges. This simplifies further analyses and optimizations by reducing the size and the complexity of the IR graph. Also, if an exception handler is not reached by any exception edge then this exception handler does not need to be parsed and compiled, which improves compilation time and reduce the size of the compiled code.

When an exception is thrown at a position where the compiled code does not expect it, the runtime uses deoptimization to go back to the interpreter. The interpreter then re-executes the code and throws the exception. To implement this in our IR, we use Guard nodes checking the assumption that no exception needs to be thrown. Any further analysis can make the assumption without taking any special care.

For example, when an operation needs to check that one of its operands is not null, we insert a null check guard. When a null value is encountered, deoptimization is triggered and the interpreter throws the NullPointerException

Since deoptimization is costly, we need to handle exceptions efficiently in programs that use exception handling as normal control flow. If an exception is thrown frequently for a certain node, we insert explicit exception checks. An explicit check can simply be the insertion of the runtime check using an If node followed by an explicit exception edge in the case the check fails. On this edge the exception object can be a pre-allocated object, eliminating the allocation cost.

In our IR, two different nodes are used for method invocation: one (Invoke node) that does not expect an exception to be raised and another (InvokeWithException node) that acts as a control-flow split between normal control flow and exceptional control flow. On its exception edge, an exception-aware invoke is followed by a special ExceptionObject node that represents the exception raised during the method call. This structure can be seen in Figure 10.

When the runtime is unwinding the stack because of an exception, it directly maps the program counter of the

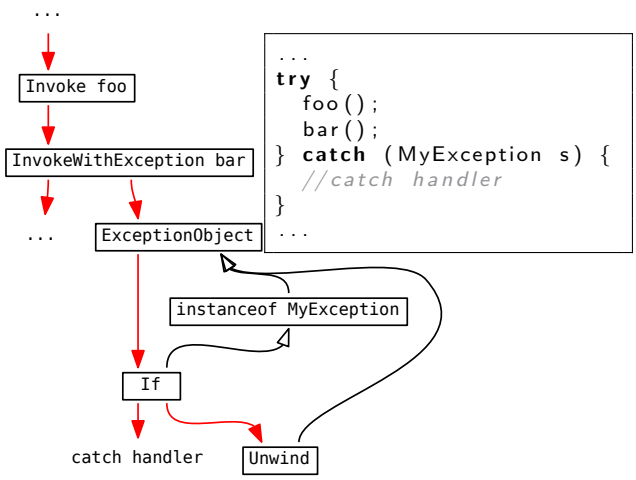

Figure 10. Exception edges handling with invoke nodes and a simple exception dispatch chain.

invoke throwing the exception to the program counter of the exception branch. If such a mapping does not exist, it means the invoke did not have an exception edge and a mapping to the proper deoptimization information allows the runtime to handle the exception in the interpreter. In this case the exception has already been thrown and is in a "pending" state, so that the interpreter restarts execution at the bytecode of the invoke and immediately handles the exception.

The explicit exception edges lead to a chain of If nodes that check the type of the exception object to find the right exception handler. At the end of this chain, if the type of the exception does not match any of the exception handlers, control flow goes to an Unwind node that forwards the exception to the next method on the call stack. A simple example is illustrated in Figure 10.

Explicitly modeling the exception handling chains for points where exceptions actually happen allows us to apply the same optimizations to the exception handling code than to the rest of the code. For example, if the exact type of an exception object is discovered through inlining or other transformations, the exception handling chain can be optimized to jump directly to the correct handler.

Using Guard nodes to check for exceptions gives us the ability to re-order those checks without constraints since it is a property of our Guard nodes.

6. Evaluation

6.1 Deoptimization

We maintain deoptimization information at state splits rather than at instructions that may cause deoptimization. Deoptimization information adds a large number of edges and constraints to the IR graph. We measure the number of state split nodes and the number of nodes that need deoptimiza-

tion information after all optimizations have been applied. We found that, in the DaCapo benchmarks, there are on average 29% more nodes that need deoptimization information than state split nodes. This shows that our model does not increase the number of FrameState nodes that are necessary, but actually reduces it.

We also measure the number of distinct FrameState nodes that are actually assigned to nodes which need deoptimization information. The results show that on average only 32% of the FrameStates are actually assigned in the DaCapo benchmarks.

To evaluate the importance of FrameState nodes, we measure the number of inputs they take. They have on average 10.5 inputs which means that they add a lot of edges to the graph. Those edges become constraints for scheduling and need to be processed during data-flow analysis. This means that reducing the number of FrameState nodes helps the compiler.

When inserting Guard nodes, we measure how many are actually created and how many are just immediately coalesced with an existing one. On average, 49.7% of the Guard nodes in the DaCapo benchmarks are actually created. This shows that even before any other analysis half of the runtime checks have already been eliminated thanks to automatic coalescing and to the careful selection of the anchoring nodes.

6.2 Exception Handling

To evaluate the impact of our design for exception handling, we measure, during parsing, the number of places where exception edges are needed according to the JVM specifications [21] and the number of edges that we actually insert. The results are shown in Figure 11 and Figure 12. These results indicate that very few exception edges are actually needed and confirm the motivation of our design.

The breakdown per type of exception edge in Figure 11 also shows that 40.0% of the exception edges come from invokes, which shows the importance of being able to exclude those edges. On the other hand, throw statements (also in Figure 11) always throw an exception, so there is no interest in eliminating them. A number of explicit exception edges are also necessary for bounds checks. These mainly come from the eclipse benchmark which uses bounds checks exceptions in some parts of its logic.

To assert the importance of eliminating exception edges, we measure the number of nodes just after parsing. In the DaCapo benchmarks, there are 0.73 potential exception edges per nodes. This means that if we included all exception edges the control-flow graph would be extremely fragmented. This would have an adverse effect on the quality of further optimizations. It would also increase the code size.

We also measure the number of exception handlers that are present in methods parsed by the compiler and the number of these handlers that are actually parsed and included in the compiled code. The results can be seen in Figure 12.

	Potential	Explicit	
Bounds check	166,770	498	(0.30%)
Invoke	1,296,646	14,454	(1.11%)
Null check	1,525,061	686	(0.04%)
Throw	2,241	2,241	(100.00%)
Checkcast	99,192	0	(0.00%)
Div/Rem	6,082	0	(0.00%)
Allocation	110,078	0	(0.00%)
Monitor null check	33,631	0	(0.00%)
Total	3,239,701	17,879	(0.55%)

Figure 11. Number of potential exception edges required by Java specification vs. number of explicit exception edges actually inserted in the IR aggregated for all DaCapo benchmarks. Breakdown by kind of exception edge.

	Exception edges		Exception handlers	
Benchmark	Potential	Explicit	Potential	Parsed
avrora	147,020	629 (0.43%)	952	39 (4.10%)
batik	211,087	745 (0.35%)	1,507	28 (1.86%)
eclipse	519,079	1,958 (0.38%)	10,029	292 (2.91%)
fop	208,616	776 (0.37%)	1,111	21 (1.89%)
h2	181,582	1,022 (0.56%)	1,831	80 (4.37%)
jython	257,057	2,197 (0.85%)	2,346	50 (2.13%)
luindex	143,630	611 (0.43%)	960	21 (2.19%)
lusearch	136,506	753 (0.55%)	945	25 (2.65%)
pmd	214,393	3,354 (1.56%)	1,629	136 (8.35%)
sunflow	147,146	675 (0.46%)	685	23 (3.36%)
tomcat	292,531	1,175 (0.40%)	4,296	113 (2.63%)
tradebeans	238,901	1,114 (0.47%)	2,826	99 (3.50%)
tradesoap	368,987	2,220 (0.60%)	7,253	191 (2.63%)
xalan	173,166	650 (0.38%)	1,949	24 (1.23%)
Total	3,239,701	17,879 (0.55%)	38,319	1,142 (2.98%)

Figure 12. Number of exception edges and exception handlers found in the input code vs. the number kept for compilation.

On average, only 2.98% of the handlers are parsed. This is a significant reduction which further simplifies the IR of methods containing exception handlers. This has also the positive effect of reducing the size of the emitted code.

6.3 Performance

We measure the performance of the Graal compiler (at revision `5a9d68c3a7d7` [1]) and compared it to the performance of the Java HotSpot client and server compilers using HotSpot 25.0-b43 [2]. The benchmarks are run on a Xeon E5-2690 running Ubuntu 12.04. We measure peak performance for DaCapo 9.12 [9], SPECjvm2008 [28] and SPECjbb2005 [27]. The results in Figure 13 show that the Graal compiler and its IR can run a variety of benchmarks and performs well compared to industry-leading compilers. These three compilers do not implement the same optimizations. For example, our compiler does not implement advanced loop transformations or array bounds check elimination which are implemented in the server compiler. The server compiler also implements a wider set of compiler intrinsics than our compiler.

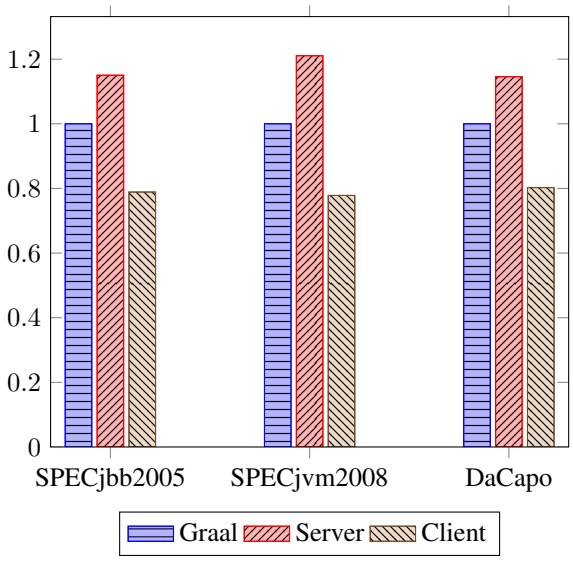

Figure 13. Performance of the Java HotSpot VM using Graal and client or server compiler for SPECjvm2008, SPECjbb2005 and DaCapo. Operations per minute (higher is better), normalized to Graal's score.

7. Future Work

Currently, we assign deoptimization information to deoptimization instruction just before code emission. Thus after this point no more optimization is performed and the scheduling is fixed. It would be interesting to do this assignment earlier since the evaluation shows that only a small number of deoptimization information is assigned. We could then eliminate some code that became dead because it was only referenced from unused FrameState nodes. Since FrameState nodes are eliminated, some constraints are removed from the graph, which could also lead to a better schedule. To do this, only the deoptimizing instructions need to be fixed to the control flow so that they can be assigned deoptimization information.

This framework for speculative optimization can be used for many purposes beyond exception handling. We could use it for example for aliasing checks to implement better scalar replacement.

8. Related Work

8.1 Intermediate Representation

The Graal IR is close to the graph IR presented by Click [6, 7] where nodes are not necessarily fixed to a specific point in the control flow. This kind of IR retains ideas from the Program Dependence Graph (PDG) of Ferrante et al. [13] where only the dependencies necessary to express the program semantics are kept.

A description of the design and implementation of our IR in the Graal compiler is given by the authors in [12].

8.2 Deoptimization

Deoptimization has first been proposed for the implementation of the SELF language [17]. It is implemented in the HotSpot VM and used in both the client [20] and server [24] compilers. In HotSpot, deoptimization transfers the execution to the interpreter. In the Jikes RVM, there is no interpreter, so deoptimization is done by transferring the execution to a different version of the compiled code. This is called On-Stack Replacement (OSR) and was proposed for the Jikes RVM by Fink and Qian [14]. In the HotSpot VM terminology, the OSR name is only used for the transfer of execution from the interpreter to compiled code [20]. The RPython tracing JIT [25] also support guards and handles them in a similar fashion to the HotSpot VM.

In these implementations, instructions that can trigger deoptimization maintain their own deoptimization information throughout the complete compilation process. This adds strong ordering constraints for these instructions. This difficulty is well illustrated by Odaira and Hiraki [22] and Sundaresan et al. [30] who want to move instructions that may throw exceptions. Using our IR, we can move and insert Guard nodes without having to employ dynamic code patching or other techniques to overcome this re-ordering.

The Crankshaft compiler of the JavaScript v8 VM [15] uses a model similar to ours for handling deoptimization. While its IR does not have floating nodes, JavaScript-specific nodes that can cause deoptimization are manually moved during the compilation process. We did not find published work about the Crankshaft compiler, but the source code is available [16].

Binary translator such as Transmeta's Code Morphing software (CMS) [10] also have support for speculative optimizations. In CMS this is done using a rollback mechanism which triggers when an assumption is not verified.

Similar goals to deoptimization can be achieved via code duplication and modification of control flow. A predicate can be checked to dispatch the code between one version that makes an assumption and another that does not. This has been done for example by Sias et al. [26]. This method can make the global control very complex if a large number of assumptions need to be taken. Thus it would be impractical for assumptions about exceptions not occurring. In the context of nested loops, this issue has been studied by Djoudi et al. [11] but their method can not be applied more generally.

Another technique for speculative optimization has been proposed by Kelsey et al. [18]. In their model targets sequential code, where blocks of speculatively optimized code and safe code run in parallel. The execution of the safe code is used to verify the execution of the speculatively optimized code. The speedup comes from the fact that the verification of block can start as soon as the speculatively optimized execution of the previous block is finished. This allows overlap in the execution of the safe code. Even if reasonable speedup

is achieved, this technique only works for a sequential program and requires the developer to annotate the source code. Also this technique only work when you can allocate multiple cores to run one sequential program.

8.3 Exception Handling

To mitigate the high number of exception edges, some compilers such as Jikes RVM [4], and HotSpot's client compiler [20], use a *factored control-flow graph* where exception edges are implicit and summarized for each control-flow block. But this still requires the compiler's analyses to take into account the instructions that may throw an exception while iterating over instructions in a block for an optimization. Our implementation using guards, simply makes the assumption that exceptions do not happen and can then optimize as if this assumption was verified.

This follows the general principle that if some code inside a method has never been executed during profiling, it can be speculatively left out during compilation. This has been studied for general control flow by Whaley [31].

The HotSpot server compiler [24] also uses deoptimization to handle unlikely exceptions. It also uses profiling information to include explicit exception edges when necessary. But it can not exclude exception edges for invokes. Our evaluation has shown that those account for a large number of the exception edges. Also, even if the invokes have an exception edge, the runtime has no direct mapping of the program counter of the invoke to the program counter of the exception edge.

An other approach to reducing the impact of exceptions is shown by Su and Lipasti [29]. They compile regions of code under the assumption that exceptions are never triggered, then during execution, this assumption is checked using special hardware support. If this assumption is invalidated, the changes that happened in the failing region are rolled back using hardware support. The region is then recompiled without assumptions and re-executed. This approach is similar to our use of deoptimization in that no time is spent compiling exception handlers. Our technique has a small runtime overhead that does not exist when assumptions are monitored by hardware. But the requirement of hardware support can be prohibitive since it requires features that do not exist in common platforms. Also adding a new kind of assumption could require new hardware support which is costly while in our model, a new kind of assumption can be introduced very easily.

9. Conclusions

Our IR is suited for dynamic speculative optimization thanks to its deoptimization framework. It allows the compiler to easily take assumptions at any time and then transparently optimize the code using these assumptions. One key feature and contribution of our framework is its ability to re-order and move the assumption checks freely. This is important to simplify the implementation of optimizations dealing with code motion.

The power of this feature becomes visible in our implementation of exception handling in which we can overcome the precise semantics of the Java exception. We can move the checks for exception and we take advantage of this to remove redundant checks by coalescing them.

This framework can now be used as the basis for further speculative optimizations in a dynamic compiler environment. We believe that it is simple and powerful enough to be able to concentrate on the speculative aspect of optimizations in a dynamic compiler rather than how to accommodate any kind of constraints imposed in other frameworks.

References

[1] Graal, revision `5a9d68c3a7d7`. URL `http://hg.openjdk.java.net/graal/graal/file/5a9d68c3a7d7`.

[2] HotSpot Express 25, build 43. URL `http://hg.openjdk.java.net/hsx/hsx25/hotspot/file/hs25-b43`.

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

[4] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pages 21–31. ACM Press, 1999. ISBN 1-58113-137-2. doi: 10.1145/316158.316171.

[5] C. Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–257. ACM Press, 1995. ISBN 0-89791-697-2. doi: 10.1145/207110.207154.

[6] C. Click and M. Paleczny. A simple graph-based intermediate representation. In *Papers from the ACM SIGPLAN workshop on Intermediate representations*, IR '95, pages 35–49. ACM Press, 1995. ISBN 0-89791-754-5. doi: 10.1145/202529.202534.

[7] C. N. Click, Jr. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, 1995.

[8] 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.

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

[10] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing™ software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–24. IEEE Computer Society, 2003. ISBN 0-7695-1913-X.

[11] L. Djoudi, J.-T. Acquaviva, and D. Barthou. Compositional approach applied to loop specialization. *Concurrency and Computation: Practice and Experience*, 21(1):71–84, Jan. 2009. ISSN 1532-0626. doi: 10.1002/cpe.v21:1.

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

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987. ISSN 0164-0925. doi: 10.1145/24039.24041.

[14] S. J. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 241–252. IEEE Computer Society, 2003. ISBN 0-7695-1913-X.

[15] Google. V8 JavaScript Engine, . URL `http://code.google.com/p/v8/`.

[16] Google. V8 SVN repository, . URL `http://v8.googlecode.com/svn/`.

[17] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992. ISBN 0-89791-475-9. doi: 10.1145/143095.143114.

[18] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast track: A software system for speculative program optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 157–168. IEEE Computer Society, 2009. ISBN 978-0-7695-3576-0. doi: 10.1109/CGO.2009.18.

[19] T. Kotzmann and H. Mössenböck. Run-time support for optimizations based on escape analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 49–60. IEEE Computer Society, 2007. ISBN 0-7695-2764-7. doi: 10.1109/CGO.2007.34.

[20] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–7:32, May 2008. ISSN 1544-3566. doi: 10.1145/1369396.1370017.

[21] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[22] R. Odaira and K. Hiraki. Sentinel pre: Hoisting beyond exception dependency with dynamic deoptimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 328–338. IEEE Computer Society, 2005. ISBN 0-7695-2298-X. doi: 10.1109/CGO.2005.32.

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

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

[25] D. Schneider and C. F. Bolz. The efficient handling of guards in the design of RPython's tracing JIT. In *Proceedings of the ACM workshop on Virtual Machines and Intermediate Languages*, pages 3–12. ACM Press, 2012. ISBN 978-1-4503-1633-0. doi: 10.1145/2414740.2414743.

[26] J. W. Sias, S.-z. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W.-m. W. Hwu. Field-testing impact epic research results in itanium 2. In *Proceedings of the International Symposium on Computer architecture*, pages 26–. IEEE Computer Society, 2004. ISBN 0-7695-2143-6.

[27] Standard Performance Evaluation Corporation. SPECjbb2005, . URL `http://www.spec.org/jbb2005/`.

[28] Standard Performance Evaluation Corporation. SPECjvm2008, . URL `http://www.spec.org/jvm2008/`.

[29] L. Su and M. H. Lipasti. Speculative optimization using hardware-monitored guarded regions for java virtual machines. pages 22–32. ACM Press, 2007. ISBN 978-1-59593-630-1. doi: 10.1145/1254810.1254814.

[30] V. Sundaresan, M. Stoodley, and P. Ramarao. Removing redundancy via exception check motion. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 134–143. ACM Press, 2008. ISBN 978-1-59593-978-4. doi: 10.1145/1356058.1356077.

[31] J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 166–179. ACM Press, 2001. ISBN 1-58113-335-9. doi: 10.1145/504282.504295.