The current and future optimizations performed by the Java HotSpot Compiler

Huib van den Brink
Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
hjbrink@cs.uu.nl

Abstract

Over the years Java became quite popular and currently is used a lot. The reasons for this are its structure like objects and garbage collection, but also its platform independency. Since the existence of Java, a lot of development and improvement took place over time. For instance great effort is made to execute the Java applications as efficient as possible. Where performance was lost due to the platform independency, they had to make that up by being smart and taking counter measurements. This paper will discuss the optimization techniques used in the Java HotSpot Compiler, in order to execute the program code as fast and efficient as possible.

1 Introduction and background

HotSpot is Sun's Java Virtual Machine implementation. It is a mixed mode system, interpreting Java bytecodes until 'hotspots' are detected. It then invokes an optimizing compiler to generate native code to implement the bytecode semantics. In order to understand the optimizations performed by the Java HotSpot compiler some background knowledge is required. Therefore I will explain what the Java Virtual Machine is all about in this section.

1.1 From source to execution

In order to come from a definition of something to perform, to a real execution, certain steps are involved. First correct Java sourcecode is created and fed into a compiler. This compiler takes the Java sourcecode and transforms it into bytecode, which is somewhat less complicated, consisting out of more basic blocks. Loops for instance are replaced by a block of code containing conditional jumps. However, this representation still is machine and operating system independent. To execute the bytecode a so called Java Virtual Machine is invoked. The JVM reads in the bytecode and performs the operations specified by that bytecode. So in reality the JVM transforms the bytecode into something specific that the OS and underlying machinery can understand.

1.2 Executing bytecode

When transforming bytecode into something that is executable by the underlying systems, some choices can be made. One can either compile or interprete the bytecode. The big difference here lies in what is done with the result.

Interpretation takes a piece of the program, analyses it and checks it for errors before executing it. So the analysis is done time and time again, which can be quite inefficient. Whereas the compilation consists of checking and analyzing the source but then producing code for a more low-level and perhaps a more specific language, eliminating a lot of checks for the future. So compiling is all about pre-computing and pre-analyses of whatever can be done before execution. Thus after compilation, all further invocations of the compiled pieces can be done by just only executing it. It should come as no surprise that compiled code executes fast. However, compiling pieces that will only be executed once is more costly, due to the overhead, than the performance gained by the transformation.

So why not compile it all at startup one asks. This can be very costly. The classloader loads the classes on demand and when needed. So existing but unused classes are never loaded into the memory. Analysis and optimization transformations would have to be performed runtime. Doing the full compilation when loading classes can lead to unpredictable stalling moments in the created application. This undesirable side effect forces us to come with smarter solutions. A tradeoff is required and the solution Sun has taken will be discussed later on in this paper.

The true meaning of a Just-In-Time compiler is the ability to translate java bytecodes into native machine code on the fly. A JIT running on the end user's machine actually executes the bytecodes and compiles the methods when more efficient.

1.3 Modes

Characteristics of applications can be divided and assigned to two usages. The server and the client. A client such as a normal user wants to have a small startup time and fast execution in short term accepting the performance penalty involved in the long term, whereas server applications are often more than willingly to offer some time at startup and some performance during execution in order to achieve a overall fast execution in the long term. The result is the possibility of more or less analysis on the bytecode in order to perform some optimizations. It is not more than logical that the JVM nowadays provide both modes enabling the user to choose what option it prefers. The default however is the Client, forcing the server applications to specifically supply the request of analysis as the application is running.

The Java HotSpot Client Compiler is a three-phase compiler. First a high-level intermediate representation (HIR) is constructed out of the bytecode. The second phase involves a so called low-level intermediate representation (LIR) out of the HIR. The last phase performs only very limited optimizations and generates machine code. The optimizations here don't go much further than the method inlining, which is explained later on in this paper, of functions without exception handlers or synchronizations constraints.

The Java HotSpot Server Compiler is tuned for the performance profile of typical server applications.

1.4 Performance compared to C and C++

Even in the Server mode when time is provided in order to perform a full optimization, the performance-win, compared to for instance optimizations performed on C and C++ code, can't be achieved because of the following reasons.

Bytecode is not known to be from a compiler that can be trusted. So in order to avoid malicious bytecode, the JVM itself has to check the integrity. This is what makes Java dynamically safe, but this of course also comes with a price. To determine if programs violate the language semantics or try to directly access unstructured memory, dynamic type-tests must be performed frequently. For instance when casting and when storing into object arrays.

Java allocates all objects on a heap. This however is not the case in C++ where stacks are used often. As a result object allocation rates are much higher for Java than for C++. Also the influence of the garbage collector changes the type of memory allocation overhead[8].

Virtual invocations are more common and used more frequently in Java than that it is in C++. Virtual indicates the fact that Java programs are dynamically linked. References to methods initially are only symbolic. The invoke instructions refer to a constant pool entry containing symbolic references. A symbolic reference contains information such as class name, method name and method descriptor (containing return type and the number as well as the types of it's arguments), uniquely defining the method. The first time the JVM encounters the invoke instruction, the symbolic reference must be resolved and replaced for a direct reference. The `invokevirtual` instruction is used for non-private, non-static, non-interface, initialized instance methods that don't override methods of a superclass. All this indirection, needed to stay platform independent, results in a loss of performance.

So the benefits of being flexible and platform independent, while staying safe, comes with a price.

2 Supporting techniques

Some of the analysis and transformations performed don't immediately improve the performance. The need for those techniques is that they enable and simplify other optimizations. In this section some those techniques are discussed, while the subsequent sections will refer to these analysis.

2.1 Use-define chain

Use-Definition chains model the relationship between the definitions of variables and their uses in a sequence of assignments. Basically it exists of a list of all the instructions that are required to set the variable for each variable reference. It's counterpart, the definition-use chains links definitions with all their uses.

Concider for instance the following example.

$[inty = 0;]^1$
$[intx = 0;]^2$
$[x = x + y;]^3$
$[y = x + 1;]^4$
$[x = 1 - x;]^5$
$[storeToDb(x);]^6$
$[x = 3;]^7$

The following chains could then be constructed.

		x	y		x	y
1		$\emptyset$	$\emptyset$		$\emptyset$	$\{3\}$
2		$\emptyset$	$\emptyset$		$\{3\}$	$\emptyset$
3		$\{2\}$	$\{1\}$		$\{4,5\}$	$\emptyset$
4		$\{3\}$	$\emptyset$		$\emptyset$	$\emptyset$
5		$\{3\}$	$\emptyset$		$\{6\}$	$\emptyset$
6		$\{5\}$	$\emptyset$		$\emptyset$	$\emptyset$
7		$\emptyset$	$\emptyset$		$\emptyset$	$\emptyset$

Use-Definition Definition-Use

The use-definition maps for each basic block the point where a value was assigned to the variables used. For instance, the third block uses both the variables declared and assigned in the blocks `1` and `2`. The definition-use however constructs a mapping for each block by looking what variables are declared or values assigned. Then the points are gathered, where that variable is used, while taking any shadowing into regard. So the first `x` in the third block is used in both block `4` and `5`, but not in `6` since it is being reassigned in `5`.

2.2 Data-flow analysis

The dataflow analysis (DFA) visualizes sourcecode as a graph. Nodes are elementary blocks and edges describe how control might pass on from one block to another. It should come to you as no surprise, that for constructing this information, a control flow graph is used.

A control flow graph (CFG) shows all possible paths present in a program. An if statement for instance has two paths, the path for `true` and one for the `false` condition. Each node in the graph represents a basic block, ie. a straight-line piece of code without any jumps. Directed edges are used to represent jumps in the control flow.

One implementation of the DFA is the reaching definition analysis, as discussed in the next section, since this embodies the technique discussed.

2.3 Reaching Definition analysis

The reaching definition analysis determines the scope of the visibility of variables.

In the following example each line represents a basic block.

```
int a = 1;
int b = a;
int c = a;
```

In here the `a` is a reaching definition in the second and third block

In the following example however the `a` in the first block isn't a reaching definition in the other blocks, for it is being shadowed in the second block.

```
int a = 1;
    a = 2;
int b = a;
```

The `a` of the second block however still is a reaching definition at the third block.

2.4 Static Single Assignment form

Static Single Assignment form (SSA) isn't an optimization on its own, but many of the actual optimizations relay on this technique. The SSA enables optimizations as 'sparse conditional constant propagation', 'dead code elimination', 'global value numbering', which all will be discussed later on in this paper, and a few other optimization techniques. The internal representation (IR) of the HotSpot compiler is in the SSA form.

The SSA transforms the IR so that every variable is assigned a value only once. Existing variables in the original IR are renamed when necessary, mostly to the original name with a number or such, in order to make assignments unique.

```
int a = 1;
    a = 2;
int b = a;
```

The `b` becomes the value the second `a` has, making the first `a` useless. So this can be written as:

```
int a_1 = 1;
    a_2 = 2;
int b_1 = a_2;
```

In order to rewrite this, the Reaching definition analysis as described above is used. So now several versions of

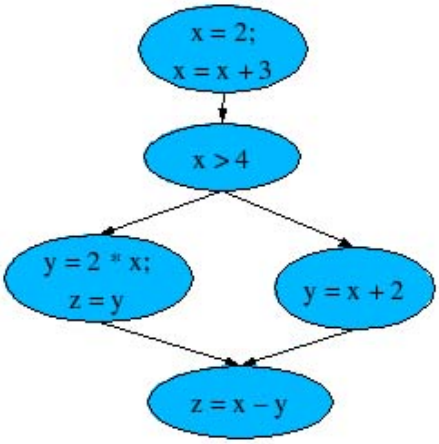

Figure 1: A control flow example

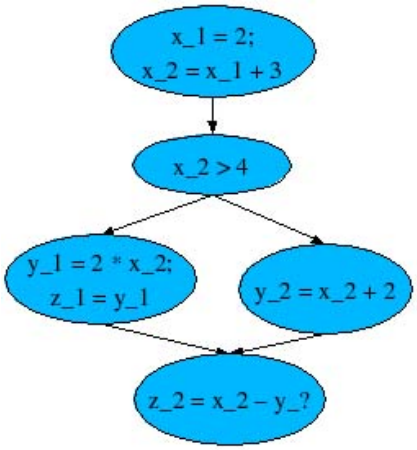

Figure 2: Assigning unique variable names

the `a` variable have arisen. This is what simplifies and improves the optimizations mentioned above.

In SSA form the use-define chains (the relation and dependency between definitions of variables and their uses in a sequence of assignments) are explicit because each chain contains only a single element.

The SSA analysis can't be done on abstract syntax trees, instead the Three address code IR is used. In simple this type changes $'1 + 2 * 3'$ into $'t1 \leftarrow 2 * 3'$ and $'t2 \leftarrow 1 + t1'$.

Control flow however make things a bit more complex. As shown in figure 1 an if statement creates several execution paths. In the first block the `x` is shadowed by another `x`, but after the `if` condition both paths create their own `y`. And in the last block the `z` from the `true` branche is shadowed.

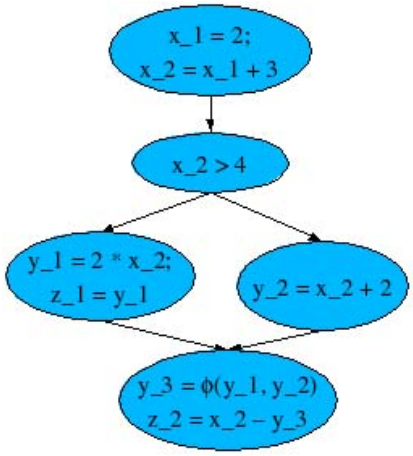

Figure 3: The solution for control flow

In figure 2 the numbering is shown, making each variable unique with only one assignment to them. In this forward analysis the `x` is renumbered when it is reassigned, and where the `x` is used, it is substituted with the `x` of the latest number. For instance the `if` uses the `x_2` but doesn't affect the `x_2` itself. The `y` in both branches, both get their own number, but this creates a problem in the latest block. In that block it is unknown what `y` should be used, because this is determined by the conditional block.

The solution to this is shown in figure 3. A ϕ (Phi) statement is introduced in the beginning of the last block. This function generates a `y_3` by "choosing" out of the `y_1` or `y_2`, depending on which path it arrived from. Now it's possible to just use the `y_3` in the rest of the statements and still be sure that the correct value is used.

The ϕ functions however aren't actually implemented. They are just markers in order to place the resulting value in a certain location being used from that point on.

Notice that the `x` doesn't need a ϕ, for there is no ambiguity in it. So in order to determine where the ϕ functions should be inserted, a concept called 'dominance frontiers' is used. Simply said, if node B cannot be reached before walking through A, than A strictly dominates B. So when the branches of for instance an `if` statement merge, the nodes in the branches of that `if` don't strictly dominate the nodes after the `if`, because the path taken is not fixed. That are exactly the locations where the ϕ functions could be inserted, depending on the variables defined or shadowed in the nodes leading towards the ϕ spot.

Whereas the Java Hotspot compiler uses SSA, functional languages (e.g. Haskell) typically will use the Continuation Passing Style. Continuation Passing Style (CPS) has the structure that the events in the future, ie. the rest of the computation, are passed as a parameter. The value passed as this parameter is the continuation. A continuation is a procedure that takes the value of the current expression and computes the rest of the computation. Procedures do not return values, but instead they invoke the continuation with the result.

3 Traditional optimizations

Many of the optimizations used by the Java HotSpot compiler are quite common in compiler techniques. This section discusses the general analysis and transformations used to improve the performance.

3.1 Method inlining

With method inlining, the body of the method invoked is inserted into the location where it is invoked. Simply said, every call is replaced by the body of that method. This of course only applies to situations where this is valid, e.g. not in the cases where recursion is used.

Take for example, the following part.

```
int counter = 0;
public void main()
{
        increment();
        increment();
}

public void increment()
{
        counter++;
}
```

This could, without changing the semantics, be replaced with the following.

```
int counter = 0;
public void main()
{
        counter++;
        counter++;
}
```

This measurement can save space on the stack, as well as eliminating the overhead of the call, caused by checks, redirection and internal registrations. However, duplicating large pieces of code many times could pollute the memory.

Also note that inheritance, by defining interfaces and overriding methods, makes this all less straightforward and even complex.

3.2 Constant propagation

Constant propagation takes the values of the variables, where they are known, and replace the usage of such variable by their values. The following example illustrates the benefit of this.

```
public static final int id = 3;

int incrId()
{
    return id + 1;
}
```

The expression `return id + 1;` now easily could be replaced with `return 4;`. With non-final variables however, reassignments should be taken into account.

The Java HotSpot compiler uses optimistic sparse conditional constant propagation[9] with SSA, which is both fast as well as accurate.

3.3 Dead code elimination

Dead code elimination is about detecting pieces of codes that are unreachable. By looking at scopes and all further information available, which for instance could be obtained by applying constant propagation, pieces of code could be detected that never are invoked. While this is common for libraries, to have methods that never actually are invoked, it is the task of the runtime system to eliminate as much as unused code as possible.

One point taken into regard is the paths present in the control flow. The following example is showing this.

```
boolean ascending = true;
if(ascending)
    counter++;
else
    counter--;
```

Combined with constant propagation, it is decidable that only one path is ever taken, and it is also known which path that is. Therefore the `else` branch could be removed by the dead code elimination algorithm.

3.4 Loop invariant hoisting

Sometimes statements can be moved out of an loop into just before the loop. By doing this parts are only executed once, instead of needlessly executing it at each round. Take for instance the following example.

```
while(i < n-1)
{
    j += (n+1) * array[i] * (pi+2)
    i++;
}
```

Some part of the calculation now can be situated outside the loop, without affecting the result. This gives the following code.

```
int max = n-1;
int tmp = (n+1) * (pi+2)
while(i < max)
{
    j += tmp * array[i]
    i++;
}
```

So if a loop contains $i \leftarrow a \bigoplus b$ and `a` and `b` don't change, there is an opportunity to relocate this part. Still care must be taken then, like there may only be one definition of i in the loop and the statement hoisted should dominate all loop exits.

3.5 Common Subexpression elimination

This optimization basically identifies instances of identical expressions, ie. pieces that evaluate to the same value. Another thing that is decided is whether or not the replacement into a newly introduced single variable is worthwhile. This variable then of course holds the computed value of the redundant expression.

The following example for instance shows a commonality among the two expressions.

```
int x = v * w + z;
int y = y * v * w;
```

When the performance benefit is greater then the overhead, which is caused by the stack allocation and the extra use of memory space, the following transformation could be performed, for it is semantically the same:

```
int tmp = v * w;
int x = tmp + z;
int y = y * tmp;
```

So the question in here is, how expensive is a multiplication compared to an extra variable. There is always a trade-of that has to be made. Creating an excessive number of temporary values could take longer than recalculation. In practice situations like array indexing calculations however do benefit from this technique.

Finally it should be mentioned that the local common subexpressions elimination only works on basic blocks whereas the more complex global variant works on entire procedures. The global CSE relies on dataflow analysis in order to determine which expressions are available at which points in a procedure.

3.6 Global Value Numbering

Global Value Numbering (GVN) can optimize the use of variables, values and calculations, as well as it is used to improve the effect of constant propagation, which will be explained later on. The optimization performed involves the elimination of redundant code by looking at the values stored in the different variables.

While common subexpression evaluation can eliminate certain pieces of redundant code, the GVN can detect cases the CSE wouldn't have noticed. In the same way CSE optimizes pieces the GVN would not cover, creating the need for both. The word Global means that value-number mappings hold across basic block boundaries as well.

Global value numbering works by assigning a value number to variables and expressions. When it can be proven that certain expressions are equivalent, the same number is assigned to them. Take for example the following piece of code:

```
int v = 2;
int w = 2;
int x = v + 3;
int y = w + 3;
```

The resulting mapping would be: $[v \mapsto 1, w \mapsto 1, x \mapsto 2, y \mapsto 2]$. As can be seen, an correct GVN algorithm detects that `v` and `w` are the same, but also that `x` and `y` are equivalent. With this information and observation the following transformation could be applied:

```
int v = 2;
int w = v;
int x = v + 3;
int y = x;
```

Now the constant propagation, by substituting the uses of `w` and `y`, followed by the dead code elimina-

tion algorithm, easily can remove the `w` and `y` when not reassigned anywhere else (and don't forget the SSA).

The difference between CSE and GVN lies in the fact that CSE matches lexically identical expressions whereas GVN tries to determine an underlying equivalence. An example where CSE wouldn't give an optimal result is:

```
int x = v * w;
int y = v;
int z = y * w;
```

The mapping now would be: $[v \mapsto 1, w \mapsto 2, x \mapsto 3, y \mapsto 1, z \mapsto 3]$. CSE wouldn't be able to eliminate the recomputation of `z`, whereas GVN doesn't experience any trouble at all.

GVN is based on the SSA intermediate representation, so that false variable name-value mappings are prevented.

3.7 Null-check & range-check elimination

The JVM always performs checks to ensure that data outside an array cannot be read. This makes software reliable and safe. The drawback however are the runtime checks performed over and over again.

If a method accesses `a[i]` in more than one statement, while the `i`'s are the same and the `a` hasn't been reassigned (think SSA), only the first access needs a bound check. To detect the locations `i` is used, a data-flow analysis is performed.
Another helpful optimization is the loop unrolling, duplicating the body as often as needed. Not only simplifies this the array bounds check only to be performed twice (at the beginning and end), but due to the expanded body the resulting optimizations can have more effect, for there are more instructions to analyze in relation to their cohesion.

In the same way a null check can be made obsolete by finding the locations where the variable becomes not null and null. Anywhere in between null checks can be omitted.

4 Phases of the Java HotSpot Compiler

The activities performed by the Java Hotspot compiler can be divided in several phases. This section describes those phases and their relation to optimizations performed.

4.1 Parser

The first pass over the bytecode, performed by the parser, identifies basic blocks and their predecessors. The second pass then visits each basic block, consisting of bytecode, in order to translate it into the compiler's IR. After the parsing has been completed, def-use edges are constructed in a batch pass to eliminate any useless code that was produced by the parser. These def-use edges are preserved and updated during optimization until a machine specific representation is created.

The optimizations applied during parsing are `Ideal`, `Value`, `Identity` and `GVN`, which actually are a subset of those applied during post-parsing, for the def-use information is not available yet.
Ideal optimizes the local and upwards structure of the nodes along the use-def edges. It may reorder the inputs of a commutative operation. For instance `(constant + (constant + variable))` could be changed into `(variable + (constant + constant))`.
Value produces a type for the result of the node. For instance it could transform something of the format `(constant + constant)` into just `(constant)`.
Identity detects when the result is equivalent to one of its input and returns that input then instead. So `(constant + 0)` results in `(constant)`.
And finally the Global Value Numbering was discussed earlier on in this paper.

4.2 Machine independent optimizations

When methods are placed for compilation, in most cases the necessary class loading and initialization already has been performed by the interpreter. For that reason the generated code doesn't handle class initialization. When it is discovered that initialization should be performed for a class to be compiled, an uncommon trap is generated, which is nothing more than a trampoline back to the interpreted mode. The compiled code then is deoptimized and is flagged as being unusable. Threads entering the method are interpreted until its recompilation is finished.

All `invoke` instructions are handled in a single function to keep related pieces of code together. First a check is performed that the destination method is loaded and initialized. Failure at this point would cause a uncommon trap, instead of the call. After checking these safety conditions, the callee is either being inlined or a calling mechanism is generated. Such calling mechanism can be either static or dynamic. Static calls,

used for calls to static methods or non-inlined virtual calls that only have one receiver, dispatch directly to the entry point of a method. Dynamic calls, used for the other occasions, dispatch to the unverified entry point of a method. Failure paths generate uncommon traps, resulting in deoptimization and recompilation of the method.

Virtual and interface calls are examined to determine if there is only one receiver. Firstly with a class hierarchy analysis (CHA) and secondly with profiling, done during interpretation. If a single receiver is identified, the callee will either be inlined or called as an optimized virtual call using the static call mechanism. The CHA is needed now, for a class lower in the hierarchy could override the method, by providing a different implementation, causing the compiled code to be deoptimized.

When the profiler detects a method that is only invoked fromout one location, without having the CHA also spotting that, runtime verification code is generated and the target method is inlined.

When deoptimization occurs, because for instance due to an uncommon trap, all dependent methods also are being deoptimized. Threads currently executing in the method are rolled forward to a safepoint, at which the native fame is converted into an interpreter one. So the trap is not visible to the executing thread until it has been brought to a safepoint. Execution of the method then continues in the interpreter. For this reason the VM must be able to regenerate the interpreter's JVM state at various points of the program. Luckily this is very much the same information required to debug optimized code. The exact JVM state is given as input to safepoints and procedure calls, making the JVM state thus 'live' in the safepoints. The optimizer and register allocator then are able to use this state information.

Next def-use edges are constructed and GVN is applied by performing a pessimistic sparse iterative algorithm. Then forward propagation of value and identity situations is performed.

Finally optimistic sparse conditional constant propagation and iterative GVN are re-applied until a fixedpoint is reached, in order to perform global dead code elimination. And when control-flow changes, the phi nodes also need to be updated. After an fixedpoint has been reached, nodes that have been identified as constants are replaced in a recursive use-def traversal that updates both use-def and def-use information. Then the worklist is initialized with nodes that use the newly discovered constants. While it also contains '`if`' and loop nodes plus conversions from integer to boolean and pointer to boolean. These nodes are likely to benefit from the optimistic type information generated during constant propagation. In particular, null checks benefit from improved null/not-null type information.

4.3 Instruction selection

Transforming the machine-independent instructions to machine instructions is performed in a bottom-up fashion. This is done before placing the instructions into basic blocks, so that block boundaries don't obstruct the selection process.

First the nodes are divided into possibly overlapping subtrees, and the root node of each subtree is being labeled. Candidates for forming a subtree are nodes with multiple users and nodes that may not be duplicated because of side-effects. Root nodes produce their results in a register enabling reuse without recomputation. Not all shared nodes however become a root, this is the case when the node is being reached by multiple paths, leading to duplicate computation otherwise. Nodes that aren't translated into machine instructions, e.g. PhiNodes, are situated in a dontcare array.

Then machine specific nodes are created. For each root node a postorder walk, so in a bottom up fashion, along the use-def edges is performed recursively. Each node visited then is passed to a DFA.

A Deterministic finite state machine or deterministic finite automaton (DFA) limits the legal states that can be reached out of a given other state. It consists of five elements. A set of finite states (S), a finite set called the alphabet (Σ), a transition function $(T : S \times \Sigma \rightarrow S)$, the start state $(s \in S)$ and finally the set of accept states $(A \subseteq S)$. For example:
$S = \{S_1, S_2\}$
$\Sigma = \{0, 1\}$

$T =$

	0	**1**
S_1	S_2	S_1
S_2	S_1	S_2

$s = S_1$
$A = \{S_1\}$
can be seen as the definition for figure 4, defining if an binary number is odd or even.

The DFA determines the lowest cost native instruction for each possible machine-independent instruction by producing a new state vector, that can be used when the node's parent is visited. This process provides optimal

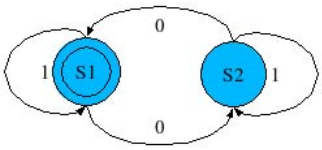

Figure 4: State diagram for binary numbers

instruction selection for each subtree. The result is a binary state tree defining the least cost instruction to be generated at each level. So when recursively traversing that state tree bottom up, the lowest execution cost machine instructions for that level could be retrieved.

4.4 Global code motion and scheduling

The machine instructions generated by the previous step do not necessarily appear in an optimal order for a particular hardware platform. Instruction scheduling is a compiler optimization that rearranges the generated machine instructions such that the execution speed is improved. Therefore instructions are placed into basic blocks in this step. In order to do so, a control flow graph skeleton of basic blocks is constructed. A virtual call for `is_block_start` means a node identifying the start of an block, whereas `is_block_proj` is a node representing the end of a block, so these nodes now are constructed wherever necessary. Next dominator information is constructed and block frequency estimates are generated. This requires two forward passes. The first to forward propagate and the second to propagate information around loops.

Assigning the instructions to the blocks is done in three stages. First the earliest legal block for each instruction is identified. This is done recursively by examining the earliest legal block for each input and finding the one which is deepest in the dominator tree. The base cases are determined by the nodes situated in the basic block skeleton, like `Start`, `Region`, `Phi`, `Goto` and `Return`.

Then the latest legal block for each instruction is identified by walking over the def-use edges in a depth-first way. Note that the def-use information used during optimization was lost when new machine specific nodes were generated, therefore it is being rebuild at the start of the scheduling process. When there is a dependency encountered, some measurements could be taken to limit the dependency. For instance when the node is a store to memory, we insert a anti-dependency edge, for instance an edge from the store to the load that must precede it. Control-flow and alias analysis are used to identify loads and stores that are independent, by finding out if they access distinct memory locations. However, when a load has been placed into a too late block, the store depending on it can't be scheduled.

The last stage selects a block between the earliest and the latest legal location by walking up the dominator tree and identifying the block with the least frequency.

And finally the local scheduler orders instructions internal to a basic-block, using a worklist and heuristics as delaying loads and preferring instructions with many inputs.

4.5 Register allocation

First the IR that still is in the SSA form is converted to a non-SSA form. A pass is performed over the control flow graph (CFG), which inserts virtual copies at phi node sites. The copies are virtual due to the fact that they are not inserted fully into the graph until later. Then a global live analysis on the annotated CFG is performed, gathering information about the legal set of registers for each live range. With that information a interference graph (IFG) is constructed for the method. The IFG represents the analysis of which live ranges interfere, ie. could compete for the same registers.

Then the IFG is used to perform a pass of copy merging. This pass is fairly aggressive in the sense that it could remove a copy which then later has to be reinserted again, while still assuming that copies are necessary and trying to prove that they are not. Finally all virtual copies remaining are then converted to actual copy instructions in the CFG. Now the CFG contains the non-SSA version of the method, while the normal IR version of the method is also retained with its SSA form and phi nodes. This dual representation allows SSA based analysis and optimizations while ensuring that all the copies necessary are allocated, in order to restore normal naming. A last pass then inserts extra uses of object pointers, serving as the base of a derived pointer. These are sited at safepoints to ensure that the base object pointer values are still live and available at the safepoint location, in order to enable the garbage collector to inspect and relocate them.

Next an iterative loop is used to perform live analysis, to gather the legal register sets for the live ranges, and build an interference graph (IFG) for the method. Then we merge where it can be proven that removing the copy will not force another interfering live range to fail. Finally standard simplification[3] and the color

selection phases of a graph coloring allocator[2] are performed. In addition, each basic block is tagged with a value which indicates that the block has either high or low register pressure, and where in the block a transition from low to high pressure takes place. This information is used during the live range splitting pass, which performs a reaching definition data flow propagation. In this way fewer copies are inserted than splitting at every use and definition, speeding up the process of live analysis.

The last stage involves the cleanup and bookkeeping activity. Unnecessary copies are removed, by using a variant of register tracking to determine that a value that was split can actually live in a single location given the actual coloring assigned to the method. Further the location of all object pointers are gathered which is needed for the garbage collection. This process produces the final version of the CFG which is used for output in the final phase of the compilation.

4.6 Peephole optimization

The peephole optimization inspects each sequence of adjacent instructions to determine if the instructions may be reorganized in a better sequence. The machine independent part visits every instruction and invokes its machine dependent peephole optimization. An example of such optimization is changing a `MOV dest_reg src_reg` followed by a `INC dest_reg` into the single instruction `LEAL dest_reg dest_reg+1`, which is valid for the IA32 platform and due to the fact that the destination of the move and increment are equal.

4.7 Code generation

Besides the generation of executable machine code, the code generator produces debug info, exception tables, relocation information and an implicit-null check table for use by the runtime system. Debug information for the runtime is associated with the offset to their safepoint. Safepoints at which a deoptimization may occur, also records debug info describing either the constant value or native storage location for monitors, locals and expression stack entries. Such locations stored can be either a register or a stack frame offset. An exception handler table at each call site maps the bytecode index for each handler to its handler's offset. This is done in order to enable the runtime system to redirect exceptions to the correct handler in the generated code, in those cases where the transition can not be determined compile time. Generated code then is moved from the buffer, in which it is generated, to its installed location in the `CodeCache`. The method's implicit null check table contains each offset at which an implicit null check may occur, combined with the offset of the corresponding handler.

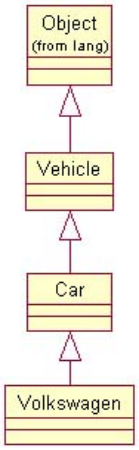

Figure 5: Inheritance scheme of a Volkswagen

Preceding of the code generation and the creation of the runtime system table, the size of the executable is calculated in a straightforward way by sizing each instruction. With all sizes and offsets computed, branch instructions are given their offsets. When the target architecture provides a shorter branch instruction format for local displacements, a replacement is generated. The final code, ie. raw machine object code, is emitted into a buffer which then is returned to the runtime system that registers it.

5 Beyond transforming sourcecode

Not all optimizations are purely about mutating the sourcecode into an optimized version with keeping the semantics the same. Some are about using the memory in a smart way or performing checks as optimal as possible, focusing on the most frequently scenario.

5.1 Fast subtype checking

Subtype checks[4] occur when a program wishes to know if class $S_{ubclass}$ implements class T_{ype}, while S or T is not known at compile time. This could be triggered by the keyword `instanceof` as well as storing an Object of some type into an object-array, causing a `checkcast` operation. Large applications use such checks a lot, making it essential to perform such checks very fast.

A simple example of this is:

```
Volkswagen vw = new Volkswagen();
Vehicle v = id(vw);
if(v instanceof Volkswagen)
        vw = (Volkswagen)v;

public Vehicle id(Vehicle v){
        return v; }
```

Though Java enables single inheritance, as shown in figure 5, multiple supertypes are allowed, using the interface mechanism. As a consequence, types aren't just a simple tree. `Object[][]` for instance is subtype of both `Clonable`, `Object[]` and `Object`. Since Java array references can be polymorphic, they can point various types of arrays, hence the store check is required between the object stored and the element type of the array.

Checks for array supertypes and array stores are handled using the same general mechanism. Class loading and unloading doesn't require modifying any data structures. In all cases the checking code is small enough to be completely inlined, so calls are never made out of the compiled Java code into the VM.

Every object has a display that contains the supertypes. The display for `Integer[][]` for instance is `{Object, Object[], Object[][], Number[][], Integer[][]}`. Now the type's depth is equal to the length of his display, so for every subtype this position is fixed. And if the depth of `T` now is greater then that of `S`, we for sure know that `S` is not a subtype of `T`. The depth of Vehicle never is greater then that of Car or Volkswagen. And now we know that the offset can be determined with indirection, a check simply can be made as shown in figure 6. The optimization is that every class has one field containing the offset for a lookup in subtypes. So the supertype contains the location it should be referenced at by its subtypes. In the figure all subtypes of `A` have the class `A` referenced at position 16.

Before we talked about the primary type, ie. objects and array's, but for the secondary type however, like interfaces and array's containing interfaces, some additional techniques are required. A linear search on the secondary supertype list is performed, but luckily that list typically is very short. And in order to improve this some more, a one element cache is used for frequently searched interface types.

Normally a general form is applied everywhere,

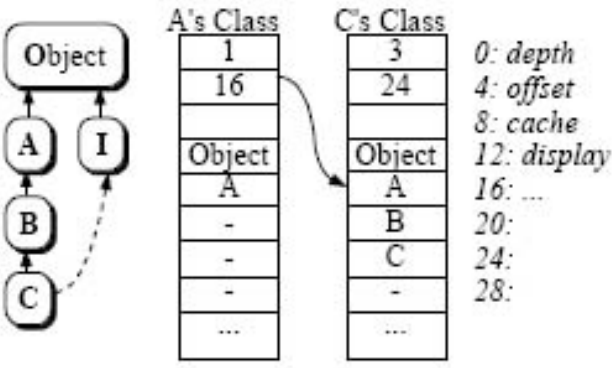

Figure 6: Unknown class C is a subtype of A

which is fine for the common cases, but sometimes special cases can be identified, enabling extra optimizations to be applied for the specific situation.

The `instanceof` and `checkcast` operations are partially optimized before insertion into the IR. A reflective type check, in principle fully polymorphic in both sub- and supertype, produces the same code as an `instanceof` bytecode if the supertype is a constant.

5.2 Efficiently using the heap

Unlike global variables, local variables of a primitive type, are allocated on a stack. When a method is entered the call stack is filled with the local variables and parameters. And when the method exits, the values are popped off the stack freeing memory. So a stack grows and shrinks in a specific order. Entering a method places data on the top of the stack while exiting removes all data, ensuring that the previous state was preserved as before the entering of the method. This system is called LIFO (Last In First Out) and handles memory very efficient.

A heap works differently. It's just an amount of reserved space where parts of data can be stored and any order of removement is allowed. Placing new data at the end of the previous block of data needs a check for if the reserved amount of space isn't full yet. If it is, then the gaps created by removing pieces of data in the meanwhile must be filled up. So shifting data around is necessary before the new data could be added. When this is done often performance will be decreased.

Objects however are situated at heaps. And as empirical studies show, in most of the programs the vast majority of the objects created, ie. often greater then 95 percent, are very short lived. One cause for instance is that they are used as temporary data structures, but

also other reasons appear. By placing newly created objects into an object nursery, the objects can be placed contiguous in a stack-like fashion. Allocation then becomes extremely fast, since it merely involves updating a single pointer and performing a single check for nursery overflow. But when the nursery overflows, most of the objects in that nursery already have deceased by then. Therefore it is efficient to move the few surviving objects elsewhere, while avoiding any reclamation work for dead objects in the nursery.

A nice sidestep however is that classes themselves also are placed onto the heap, enabling the garbage collector for the objects also to handle the classes themselves.

6 Mustang and beyond

The next Java version to be released by Sun is Mustang. The version management of Java changed a bit over time, so J2SE 1.6 now is called Java SE 6.0 and carries the name 'Mustang', while Java SE 7 will carry the codename Dolphin. This section discusses the newly added optimizations in Mustang and tries to say something about the long term improvement strategies.

6.1 Escape analysis

Unlike global variables, local variables, ie. variables declared in a method, constructor or block, are allocated on a stack. When a method is entered the call stack is filled with the local variables and parameters. And when the method exits, the values are popped off the stack freeing memory.

While class/static variables are stored in static memory, the instance variables, ie. non-static variables known in the whole class, use a heap to store the data in. But there is more to it. Variables referencing newly created or existing objects are situated on the heap for it is not known what side effecting the rest of application has on the object referenced, ie. an object may have several references to it out of anywhere else. Remember that it is perfectly legal to create an instance of an object in order to bind it to a local variable, and then pass the reference of it to the rest of the application.

Luckily it is possible to detect the situations where a stack allocation is more appropriate then using the heap. The content of isolated objects can be used by propagating data. This technique is called the escape analysis.

In order to determine if the usage of the stack allocation is possible, the analysis has to decide whether or not certain objects remain confined to a single thread for their entire lifetime, and that the lifetime is bounded by the lifetime of a given stack frame. Finally references could be replaced by hoisting object fields into registers.

The following code example clearly shows a situation in which optimizing the heap allocation could be applied by using the escape analysis. The class UserInfo contains all kinds of information about a user, whereas the User object contains the name and age. So UserInfo has a reference to a User object.

```
class User
{
  private String name;
  private int age;

  User(String name, int age)
  {
    this.name = name;
    this.age = age;
  }

  User(User u){
    this(u.name, u.age); }

  String getName(){
    return name; }

  int getAge(){
    return age; }
}
```

The User class is just a placeholder for name and age. Getters and setters are provided to access private information. The constructor with the User object as parameter just copies the data out of the given User instance.

The UserInformation object contains a reference to a User object. That User object can be retrieved out of an UserInformation object by invoking the getUser() method. This method does not just return the reference, but also constructs a new instance and copies the internal data into it to ensure that the referenced User object won't be mutated.

```
class UserInformation
{
 private User u;

 User getUser(){
   return new User(u); }
```

```
boolean isEqual(UserInformation other)
 {
  User otherUser = other.getUser();

  boolean sameAge =
    otherUser.getAge() == u.getAge();
  boolean sameNames =
    otherUser.getName().equals(
                          u.getName());

  return (sameAge && sameNames);
 }
}
```

For checking the equality of UserInformation a isEqual method is provided taking a UserInformation object as argument. First the User reference is abstracted out of the provided 'other' UserInformation. Then the name and age are requested out of both the 'other' and the local User objects in order to compare them.

The first step in the optimization is to inline the call to `getUser()`, as well as the call to `getAge()` and `getName()`. De result of this replacement is shown below.

```
boolean isEqual(UserInformation other)
{
 User otherUser =
  new User(other.u.name, other.u.age);

 boolean    sameAge =
   otherUser.age == u.age;
 boolean sameNames =
   otherUser.name .equals( u.name );

 return (sameAge && sameNames);
}
```

Now the escape analysis can determine that the User object allocated in the first line never escapes its scope. By escape is meant that a reference to some object is not stored into the heap or passed to unknown code that might keep a copy. Secondly it gets clear that the User object referenced never is modified.
Provided that the User object is truly thread-local and its lifetime is known to be bounded to the basic block in which it is allocated, it can be either stack-allocated or optimized away entirely.

By creating local variables, of a primitive type, and assigning the required data to them the need for heap allocation vanishes. Notice that this is legal because in earlier stage it is proven to not mutate the User object that was originally maintained.

```
boolean isEqual(UserInformation other)
{
   int tmpAge = other.u.age;
   String tmpName = other.u.name;

   boolean sameAge = tmpAge == u.age;
   boolean sameNames =
            tmpName .equals( u.name );

   return (sameAge && sameNames);
}
```

Above the finally total optimized code for the `isEqual` method is shown, using nothing more than the escape analysis. The original `getUser` invocation could only be optimized because of its use. Had it been in a way with side-effecting and sharing with other objects, such optimization wouldn't be applicable, considering the semantics and its correctness.

In Java SE 6 (Mustang) this optimization technique is introduced. It will convert heap allocation to stack allocation (or no allocation) where applicable and appropriate. The result is faster average allocation time, reduced memory usage, less cache misses and improved garbage collector behavior.

Escape analysis makes it convenient to use the object oriented way of thinking while using and transforming to constructions under the hood that would otherwise be very clumsy in the source itself.

6.2 Tiered Compilation

Mustang will be the first version of Java that provides tiered compilation. This means that both the client- and servermode are present in the same Virtual Machine. This enables code to be quickly compiled by the clientmode compiler, while further optimizations by the servermode compiler can be performed when needed.

As known, the client compiler is faster, while the server compiler generates better code. However, it is not made clear by Sun how the memory adjustments, caused by choosing the client or server compiler, are addressed by the tiered compilation.

6.3 The future

The very first versions of the JVM only were an interpreter. Later on template generated code was supported and finally the interpreter plus optimized code was

realized. So big changes can be seen in the evolution of the JVM. Improvements however still aren't brought to a hold, for every time we learn something more as we go along in realizing and implementing techniques designed to optimize the performance.

As explained in the beginning the optimizations are performed every time the application is run. The need for this lies in the fact that portability is required and security must be enforced at the client side, for it is unknown if the compiler creating the bytecode could be trusted or not. However there still lie some opportunities in resituating the optimizations from the HotSpot into the compiler generating the bytecode [7].

The bytecode instruction format is not very capable of transporting the results of program analyses and optimizations. An alternative mobile-code representation could be introduced [5]. So probably it's only a matter of time until the HotSpot VM will be using an alternative mobile-code transportation format, with as result a shift of workload to the bytecode producer, while still not jeopardizing the safety of the client.

References

[1] Michael Paleczny, Christopher Vick and Cliff Click. The Java HotSpot Server Compiler. Proceedings of the Java Virtual Machine Research and Technology Symposium.

[2] Michael D. Smith, Norman Ramsey and Glenn Holloway A Generalized Algorithm for Graph-Coloring Register Allocation Harvard University

[3] Antony L. Hosking Register allocation Purdue University

[4] Cliff Click and John Rose. Fast Subtype Checking in the HotSpot JVM.

[5] Jeffery von Ronne, Michael Franz, Niall Dalton and Wolfram Amme. Compile Time Elimination of Null- and Bound-Checks. University of California and Friedrich-Schiller-Universität Jena.

[6] Fridtjof Siebert and Andy Walter. Deterministic Execution of Java's Primitive Bytecode Operations. Proceedings of the Java Virtual Machine Research and Technology Symposium.

[7] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu and Toshio Nakatani. Design and Evaluation of Dynamic Optimizations for a Java Just-In-Time Compiler. IBM Tokyo Research Laboratory.

[8] Sun Microsystems. The Java HotSpot Virtual Machine. A Technical White Paper.

[9] Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. IBM T.J. Watson Research Center.

[10] Cliff Click and Keith D. Cooper. Combining Analyses, Combining Optimizations. Rice University.

[11] Erik Stenman. Advanced Compiler Techniques. Virtutech.

[12] Thomas van Drunen. Uniting Global Value Numbering and Partial Redundancy Elimination. Purdue University.

[13] Andrew Myers. Introduction to Compilers - Dataflow analysis Cornell University.

[14] Tim Lindholm and Frank Yellin from Sun Microsystems. The Java Virtual Machine Specification - Second Edition.