

# Understanding How Graal Works - a Java JIT Compiler Written in Java

Chris Seaton, 3 November 2017

*These are the notes for a talk I gave at JokerConf 2017, which is why they're written as if we were talking and going through some slides together.*

One of the reasons that I'm a programming language researcher is that almost everyone I meet in the wider computer technology community uses programming languages and many of them are interested in how they work. When I was first introduced to programming as a small child and shown a programming language, the first thing I wanted to know was how did the language work, and one of the first things I wanted to do was to implement my own language.

What I'm going to do today in this talk is show you something about how the programming language you all use, Java, works. What's different is that I'm going to use a project called Graal that implements Java in Java, so that I can show you how Java works, using Java.

Graal is just one part of how Java works though - it's a just-in-time compiler. This is the part of a JVM that converts Java bytecode to native machine code at runtime, and it's a big part of what makes Java as high performance as it is. It's also what many people I think would consider to be one of the most complex and opaque parts of the JVM that they may think is outside what they can understand, and this talk aims to help change that.

If you know what a JVM is, you know broadly what those two terms mean - *bytecode* and *machine code* - and you can read everyday Java code then you will have enough to follow me I hope.

I'm going to talk about why we would want to write a new JVM JIT-compiler in Java itself, and then I'm going to start showing you it's not as special as you may think it is, by diving into building it, using it, and starting to show you how the code is just as understandable as any other application.

I'll talk about just a little bit of the theory but then I'll show you how this is applied to compile a method all the way from the bytecode to the machine code. I'll show some more details, and then finally I'll talk about why this is a good thing to have beyond being interesting to have more of Java implemented in Java for the sake of it.

I'm going to use screenshots of Eclipse, rather than running it during the presentation, to avoid the inevitable embarrassments of trying to live code.

## What is a JIT compiler?

I'm sure many of you know what a JIT-compiler is, but I'll just cover the basics so nobody is sitting there afraid to ask that fundamental question.

When you run the `javac` command, or compile-on-save in your IDE, your Java program is compiled from Java source code into *JVM bytecode*. This is a binary representation of your Java program. It's more compact and simpler than the source code. However a conventional processor in your laptop or server can't actually execute JVM bytecode.

To run your Java program then the JVM interprets the bytecode. Interpreters are often a lot slower than native code running on a real processor, so the JVM at runtime can also run another compiler, this time compiling your bytecode to the machine code that your processor can actually run.

This compiler is usually a lot more sophisticated than the `javac` compiler, running complex optimisations to produce high quality machine code.

## Why write a JIT compiler in Java?

The OpenJDK implementation of the JVM contains two conventional JIT-compilers today. The *client* compiler, also called *C1*, is designed to run more quickly and produce less optimised code. The *server* compiler, also called *opto*, or *C2*, is designed to take a little more time to run but to produce better optimised code.

The idea was that the client compiler was better for desktop applications, where longer pauses for JIT-compilation are annoying, and the server compiler was better for long-running server applications which can spend more time on compilation.

Today they can be combined, so that code is compiled with C1 first and then C2 later if they are still being executed a lot and it looks worth spending the extra time. This is called tiered compilation.

Let's focus on C2 - that was the server compiler which optimises more.

We can clone OpenJDK from a GitHub mirror, or we can browse it directly on the website.

```
$ git clone https://github.com/dmlloyd/openjdk.git
```

The code for C2 is located at `openjdk/hotspot/src/share/vm/opto`.

```

Project
divnode.cpp — ~/Documents/jokerconf17/demo/openjdk

568 //-----Idealize-----
569 // Dividing by a power of 2 is a shift.
570 Node *DivLNode::Ideal( PhaseGVN *phase, bool can_reshape ) {
571     if (in(0) && remove_dead_region(phase, can_reshape)) return this;
572     // Don't bother trying to transform a dead node
573     if( in(0) && in(0)->is_top() ) return NULL;
574
575     const Type *t = phase->type( in(2) );
576     if( t == TypeLong::ONE )           // Identity?
577         return NULL;                 // Skip it
578
579     const TypeLong *tl = t->isa_long();
580     if( !tl ) return NULL;
581
582     // Check for useless control input
583     // Check for excluding div-zero case
584     if (in(0) && (tl->hi < 0 || tl->lo > 0)) {
585         set_req(0, NULL);           // Yank control input
586         return this;
587     }
588
589     if( !tl->is_con() ) return NULL;
590     jlong l = tl->get_con();      // Get divisor
591
592     if (l == 0) return NULL;        // Dividing by zero constant does not idealize
593
594     // Dividing by MINLONG does not optimize as a power-of-2 shift.
595     if( l == min_jlong ) return NULL;
596
597     return transform_long_divide( phase, in(1), l );
598
599

```

hotspot/src/share/vm/opto/divnode.cpp 571:60

LF UTF-8 C++ jdk9/jdk9 0 files

First of all, C2 is written in C++. There's of course nothing inherently wrong with C++, but it does have issues. C++ is an unsafe language - meaning that errors in C++ can cause the VM to crash. It's probably also just the age of the code, but the C++ code in C2 has become very hard to maintain and extend.

One of the key people behind the C2 compiler, Cliff Click, has said that he would never write a VM in C++ again, and we've just heard that Twitter's JVM team have said that they think that C2 have become stagnant and is past needing replacement because development is so difficult.



## Things I won't do again...

- Write a VM in C/C++
  - Java plenty fast now
  - Mixing OOPS in a non-GC language a total pain
  - Forgetting 'this' is an OOP
    - Across a GC-allowable call
  - Roll-your-own malloc pointless now

<https://www.youtube.com/watch?v=Hqw57GJSrac>

So, going back to the question, what is it about Java that helps solve these problems? Well, it's all the same things that mean you write your applications in Java rather than C++. So that's probably safety - exceptions rather than crashes, no real memory leaks or dangling pointers - good tooling, like being able to use debuggers, profilers, and tools like VisualVM - good IDE support, and more.

You may think *but how is it possible to write something like a JIT-compiler in Java?* You may think that this could only be possible in a low level systems language like C++, but I hope to convince you in this talk that this isn't true at all! In fact a JIT-compiler just needs to be able to accept JVM bytecode and produce machine code - you give it a `byte[]` in and you want a `byte[]` back. It will do a lot of complex things to work out how to do that, but they don't involve the actual system at all so they don't need a 'systems' language, for some definition of systems language that doesn't include Java, like C or C++.

## Setting up Graal

The first thing we need is Java 9. The interface that Graal uses, called JVMCI, was added to Java as JEP 243 *Java-Level JVM Compiler Interface*, and the first version to have this was Java 9. I'm using 9+181. There are backports to Java 8 available if people have special requirements.

```
$ export JAVA_HOME=`pwd`/jdk9
$ export PATH=$JAVA_HOME/bin:$PATH
$ java -version
java version "9"
Java(TM) SE Runtime Environment (build 9+181)
Java HotSpot(TM) 64-Bit Server VM (build 9+181, mixed mode)
```

The next thing we need is a build tool, called `mx`. This is a bit like Maven or Gradle, but it's probably not something you'd ever chose for your own applications. It has some special logic to support some complex use-cases, but we're just going to use it to do a simple build.

We can clone `mx` from GitHub. I'm using commit 7353064. Then just add the executable to our `PATH`.

```
$ git clone https://github.com/graalvm/mx.git
$ cd mx; git checkout 7353064
$ export PATH=`pwd`/mx:$PATH
```

Then we want to clone Graal itself. I'm using the version from a distribution called GraalVM, version 0.28.2.

```
$ git clone https://github.com/graalvm/graal.git --branch vm-enterprise-0.28.2
```

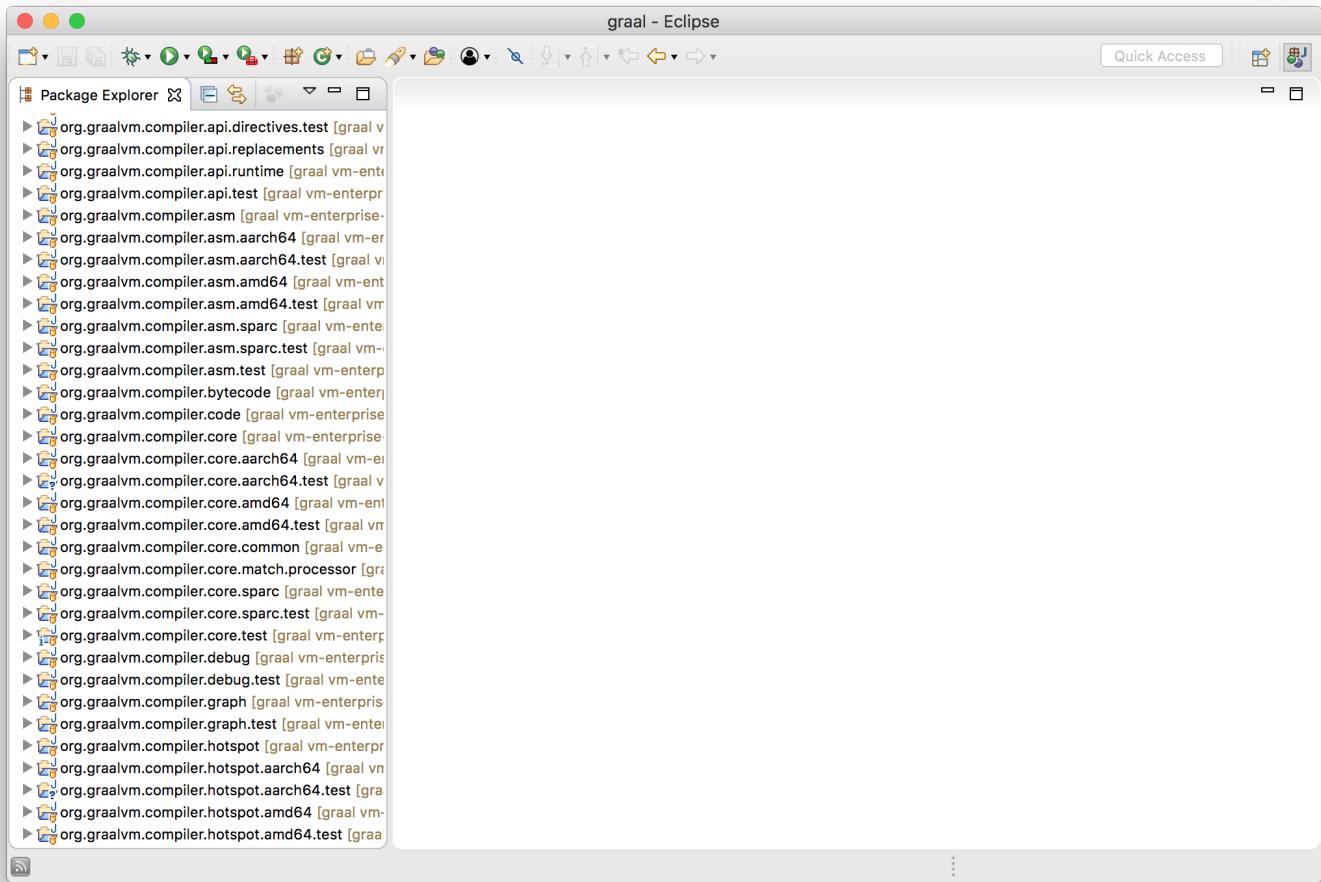
This repository contains some other projects we aren't interested in right now, so we just want to switch to the `compiler` sub-project, which is the Graal JIT-compiler itself, and build that using `mx`.

```
$ cd graal/compiler
$ mx build
```

Now I'm going to use the Eclipse IDE to open the Graal source code. I'm using Eclipse 4.7.1. `mx` can generate an Eclipse project definition for us.

```
$ mx eclipseinit
```

You will want to open the `graal` directory as workspace, then do *File, Import..., General, Existing projects* and select the `graal` directory again. You may then need to attach the JDK sources if you aren't running Eclipse itself with Java 9.



Ok now that we have everything ready, let's show this working. We'll use this very simple code.

```
class Demo {
    public static void main(String[] args) {
        while (true) {
            workload(14, 2);
        }
    }

    private static int workload(int a, int b) {
        return a + b;
    }
}
```

We'll compile that with `javac`, and then we'll run the JVM. First of all I'll show you the conventional C2 JIT-compiler working. To do this I'll turn on a couple of flags, `-XX:+PrintCompilation` which tells the JVM to log when a method is compiled, and I'll use `-XX:CompileOnly=Demo::workload` so that only that method is compiled. If we don't do this the volume of output is a bit overwhelming and the JVM will be more clever than we want and optimise away the code we want to look at.

```
$ javac Demo.java
$ java \
-XX:+PrintCompilation \
-XX:CompileOnly=Demo::workload \
Demo
...
     113      1      3      Demo::workload (4 bytes)
...
```

I won't explain that beyond saying that is the log entry which shows that the `workload` method is being compiled.

Now we're going to use the Graal that we just built as our JIT-compiler in our Java 9 JVM. We need to add some more complicated flags here.

--module-path=... and --upgrade-module-path=... add Graal to the module path. Remember that the module path is new in Java 9 as part of the Jigsaw module system, and you can think of it as being like the classpath for our purposes here.

We need -XX:+UnlockExperimentalVMOptions because JVMCI (the interface that Graal uses) is just experimental at this stage.

We then use -XX:+EnableJVMCI to say that we want to use JVMCI, and -XX:+UseJVMCICompiler to say that we actually want to use it and to install a new JIT compiler.

We use -XX:-TieredCompilation to disable tiered compilation to keep things simpler and to just have the one JVMCI compiler, rather than using C1 and then the JVMCI compiler in tiered compilation.

And we still use -XX:+PrintCompilation and -XX:CompileOnly=Demo::workload as before.

Just as before, we see that one method being compiled. We've now used the Graal that we just built to do the compilation though. For the moment you'll just have to take my word for that.

```
$ java \
--module-
path=graal/sdk/mxbuild/modules/org.graalvm.graal_sdk.jar:graal/truffle/mxbuild/modules/com.oracle.truffle.truffle_api.jar \
--upgrade-module-path=graal/compiler/mxbuild/modules/jdk.internal.vm.compiler.jar \
-XX:+UnlockExperimentalVMOptions \
-XX:+EnableJVMCI \
-XX:+UseJVMCICompiler \
-XX:-TieredCompilation \
-XX:+PrintCompilation \
-XX:CompileOnly=Demo::workload \
Demo
...
      583   25           Demo::workload (4 bytes)
...
```

## The JVM compiler interface

What we've done there is quite extraordinary isn't it? We had a JVM installed, and we replaced the JIT-compiler in it to a new one that we just compiled without having to change anything in the JVM. What makes this possible is a new interface in the JVM called the JVMCI - the JVM compiler interface - this is what I said was JEP 243 and had gone into Java 9.

The idea is analogous to some other existing JVM technologies.

You may have added custom source code processing to the `javac` compiler in the past using the Java annotation processing API. This lets you recognise Java annotations and a model of the Java source code they're attached to and produce new source files from them.

You may have also added custom Java bytecode processing to the JVM using Java agents. These allow you to intercept Java bytecode as it's loaded and modify it.

The JVMCI is similar in idea to these. It lets you plug in your own Java JIT-compiler, written in Java.

At this point I'll say something about the approach I've taken to showing code in this presentation. I'll show slightly simplified identifiers and logic as text on the slides to help you understand the idea at first, and then I'll switch to screenshots of Eclipse to show you the actual code, which may be a bit more complicated but the big idea will be the same. A big part of this talk is about showing that you really can dive into the real code, so I don't want to hide it even if it is a little bit complex.

This is where I want to start dispelling the idea that you may have that a JIT-compiler is going to be extremely complicated.

What does a JIT-compiler take in as input? It takes the bytecode of the method it is compiling, and bytecode is, as the name suggests, just an array of bytes.

What does a JIT-compiler produce as output? It produces machine code compiled from the method. Machine code is also just an array of bytes.

So the interface that you need to implement when you write a new JIT-compiler and plug it into the JVM will look something like this.

```
interface JVMCICompiler {
    byte[] compileMethod(byte[] bytecode);
}
```

So if you were thinking how can Java do something as low level as JIT-compile to machine code, it's not as low level as you thought is it? It's a pure function from `byte[]` to `byte[]`.

In reality it's actually a little more complex than that. Just the bytecode isn't quite enough - we also want some information like the number of local variables, the size of stack needed, and information collected from profiling in the interpreter so that we know how the code is running in practice. So we'll say instead that the input is a `CompilationRequest`, which tells us which `JavaMethod` we want to compile, and say that this can give us all the information we need.

```
interface JVmCICompiler {
    void compileMethod(CompilationRequest request);
}

interface CompilationRequest {
    JavaMethod getMethod();
}

interface JavaMethod {
    byte[] getCode();
    int getMaxLocals();
    int getMaxStackSize();
    ProfilingInfo getProfilingInfo();
    ...
}
```

Also, the interface doesn't have you return the compiled machine code - instead you call another API to tell the JVM that you want to install some machine code.

```
HotSpot.installCode(targetCode);
```

Now to write a new JIT-compiler for the JVM we just need to implement this interface. We get the information about the method that we want to compile, and it's then over to us to compile it to machine code and call `installCode`.

```
class GraalCompiler implements JVmCICompiler {
    void compileMethod(CompilationRequest request) {
        HotSpot.installCode(...);
    }
}
```

Let's switch to the Eclipse IDE with Graal in it to see what some of these interfaces and classes look like in practice. As I've said, they'll be a little more complicated, but not much more.

graal - jdk.vm.ci.runtime.JVMCICompiler - Eclipse

```
2 * Copyright (c) 2015, Oracle and/or its affiliates. All rights reserved.
3 package jdk.vm.ci.runtime;
4
5 import jdk.vm.ci.code.CompilationRequest;
6 import jdk.vm.ci.code.CompilationRequestResult;
7
8 public interface JVMCICompiler {
9     int INVOCATION_ENTRY_BCI = -1;
10
11     /**
12      * Services a compilation request. This object should compile the method to machine code and
13      * install it in the code cache if the compilation is successful.
14      */
15     CompilationRequestResult compileMethod(CompilationRequest request);
16 }
17
```

Read-Only Smart Insert 37 : 1

graal - org.graalvm.compiler.hotspot/src/org/graalvm/compiler/hotspot/HotSpotGraalCompiler.java - Eclipse

```
//
78 public class HotSpotGraalCompiler implements GraalJVMCICompiler {
79
80     private final HotSpotJVMCIRuntimeProvider jvmciRuntime;
81     private final HotSpotGraalRuntimeProvider graalRuntime;
82     private final CompilationCounters compilationCounters;
83     private final BootstrapWatchDog bootstrapWatchDog;
84     private List<DebugHandlersFactory> factories;
85
86     HotSpotGraalCompiler(HotSpotJVMCIRuntimeProvider jvmciRuntime, HotSpotGraalRuntimeProvider graalRuntime, OptionValues
87         this.jvmciRuntime = jvmciRuntime;
88         this.graalRuntime = graalRuntime;
89         // It is sufficient to have one compilation counter object per Graal compiler object.
90         this.compilationCounters = Options.CompilationCountLimit.getValue(options) > 0 ? new CompilationCounters(options)
91         this.bootstrapWatchDog = graalRuntime.isBootstrapping() && !DebugOptions.BootstrapInitializeOnly.getValue(options)
92     }
93
94     public List<DebugHandlersFactory> getDebugHandlersFactories() {
95         if (factories == null) {
96             factories = Collections.singletonList(new GraalDebugHandlersFactory(graalRuntime.getHostProviders().getSnippet
97         }
98         return factories;
99     }
100
101    @Override
102    public HotSpotGraalRuntimeProvider getGraalRuntime() {
103        return graalRuntime;
104    }
105
106    @Override
107    public CompilationRequestResult compileMethod(CompilationRequest request) {
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
948
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1088
1089
1090
1091
1092
1093
1094
1095
1096
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1287
1288
1289
1290
1291
1292
1293
1294
1295
1295
1296
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1387
1388
1389
1390
1391
1392
1393
1394
1394
1395
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1487
1488
1489
1490
1491
1492
1493
1494
1494
1495
1496
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1587
1588
1589
1590
1591
1592
1593
1594
1594
1595
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1687
1688
1689
1690
1691
1692
1693
1694
1694
1695
1696
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1787
1788
1789
1790
1791
1792
1793
1794
1794
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1887
1888
1889
1890
1891
1892
1893
1894
1894
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1987
1988
1989
1990
1991
1992
1993
1994
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2087
2088
2089
2090
2091
2092
2093
2094
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2187
2188
2189
2190
2191
2192
2193
2194
2194
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2248
2249
2250
2251
```

The screenshot shows the Eclipse IDE interface with the title bar "graal - org.graalvm.compiler.hotspot/src/org/graalvm/compiler/hotspot/HotSpotGraalCompiler.java - Eclipse". The code editor displays the HotSpotGraalCompiler.java file. The code implements the JVMBICompiler interface, specifically overriding the compileMethod method. It includes logic to handle shutdown, bootstrapping, and compilation requests, including a try-with-resources block for a CompilationWatchdog. The code is annotated with Javadoc and suppression annotations like @Override and @SuppressWarnings("try"). The status bar at the bottom shows "Writable", "Smart Insert", and the time "107 : 38".

```
103     return graalRuntime;
104 }
105
106 @Override
107 public CompilationRequestResult compileMethod(CompilationRequest request) {
108     return compileMethod(request, true);
109 }
110
111 @SuppressWarnings("try")
112 CompilationRequestResult compileMethod(CompilationRequest request, boolean installAsDefault) {
113     if (graalRuntime.isShutdown()) {
114         return HotSpotCompilationRequestResult.failure(String.format("Shutdown entered"), false);
115     }
116
117     ResolvedJavaMethod method = request.getMethod();
118     OptionValues options = graalRuntime.getOptions(method);
119
120     if (graalRuntime.isBootstrapping()) {
121         if (DebugOptions.BootstrapInitializeOnly.getValue(options)) {
122             return HotSpotCompilationRequestResult.failure(String.format("Skip compilation because %s is enabled", DebugOptions.BootstrapInitializeOnly.name));
123         }
124         if (bootstrapWatchDog != null) {
125             if (bootstrapWatchDog.hitCriticalCompilationRateOrTimeout()) {
126                 // Drain the compilation queue to expedite completion of the bootstrap
127                 return HotSpotCompilationRequestResult.failure("hit critical bootstrap compilation rate or timeout", true);
128             }
129         }
130     }
131     HotSpotCompilationRequest hsRequest = (HotSpotCompilationRequest) request;
132     try (CompilationWatchdog w1 = CompilationWatchdog.watch(method, hsRequest.getId(), options);
133          BootstrapWatchdog Watch w2 = bootstrapWatchdog == null ? null : bootstrapWatchdog.watch(request);
134          ) {
135         ...
136     }
137 }
```

What I want to do now is to show you that we can modify Graal and immediately use it in Java 9. I'll add my own logging message that just says when Graal is compiling a method, and I'll add this to the interface method that JVMCI calls.

```
class HotSpotGraalCompiler implements JVMBICompiler {
    CompilationRequestResult compileMethod(CompilationRequest request) {
        System.err.println("Going to compile " + request.getMethod().getName());
        ...
    }
}
```

```

103     return graalRuntime;
104 }
105
106 @Override
107 public CompilationRequestResult compileMethod(CompilationRequest request) {
108     System.out.println("Going to compile " + request.getMethod().getName());
109     return compileMethod(request, true);
110 }
111
112 @SuppressWarnings("try")
113 CompilationRequestResult compileMethod(CompilationRequest request, boolean installAsDefault) {
114     if (graalRuntime.isShutdown()) {
115         return HotSpotCompilationRequestResult.failure(String.format("Shutdown entered"), false);
116     }
117
118     ResolvedJavaMethod method = request.getMethod();
119     OptionValues options = graalRuntime.getOptions(method);
120
121     if (graalRuntime.isBootstrapping()) {
122         if (DebugOptions.BootstrapInitializeOnly.getValue(options)) {
123             return HotSpotCompilationRequestResult.failure("Skip compilation because %s is enabled", Det
124         }
125         if (bootstrapWatchDog != null) {
126             if (bootstrapWatchDog.hitCriticalCompilationRateOrTimeout()) {
127                 // Drain the compilation queue to expedite completion of the bootstrap
128                 return HotSpotCompilationRequestResult.failure("hit critical bootstrap compilation rate or timeout", t
129             }
130         }
131     }
132     HotSpotCompilationRequest hsRequest = (HotSpotCompilationRequest) request;
133     +new CompilationWatchDog(w1 - CompilationWatchDog.watchMethod(hsRequest.getId(), options));

```

I'm going to turn off the normal HotSpot compilation logging now. We can see our message printed from our modified version of the compiler.

```

$ java \
--module-
path=graal/sdk/mxbUILD/modules/org.graalvm.graal_sdk.jar:graal/truffle/mxbUILD/modules/com.oracle.truffle.truffle_api.jar \
 \
--upgrade-module-path=graal/compiler/mxbUILD/modules/jdk.internal.vm.compiler.jar \
-XX:+UnlockExperimentalVMOptions \
-XX:+EnableJVMCI \
-XX:+UseJVMCICompiler \
-XX:-TieredCompilation \
-XX:CompileOnly=Demo::workload \
Demo
Going to compile workload

```

If you try that edit on your own in Eclipse you'll notice that we don't even have to run our build system, `mx build`, there. The normal Eclipse compile on save is enough. And we certainly don't have to recompile the JVM itself. We can just plug our newly modified compiler into the existing JVM.

## The Graal graph

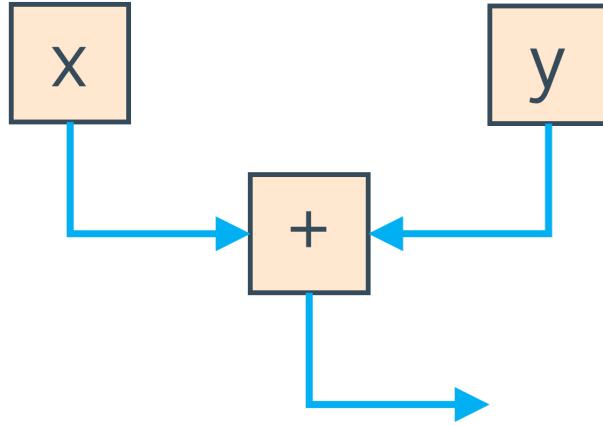
Ok, so we know that Graal converts one `byte[]` into another `byte[]`. Let's now talk about a little bit of the theory and data structures that it uses to do that, because it's a little unusual, even for a compiler.

Fundamentally what a compiler does is to work with and manipulate your program. To do this it needs to represent your program using some kind of data structure. Bytecode and similar lists of instructions is one way, but it's not very expressive.

Instead, Graal uses a graph to represent your program. If you have a simple addition operator that adds two local variables, the graph is one node to load each local variables, one node to add them, and two edges to say that the result of loading the local variables goes into the addition operation.

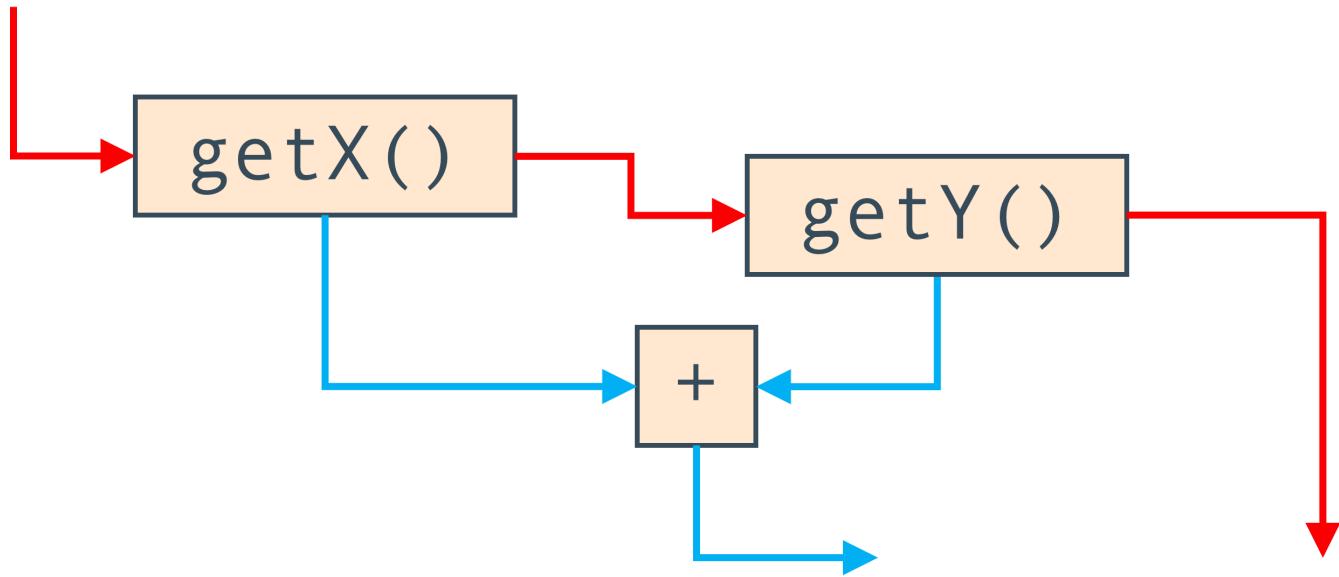
This is sometimes called a program-dependence-graph.

If we have an expression like `x + y`, we get nodes for the local variables `x` and `y`, and a node that adds them together.



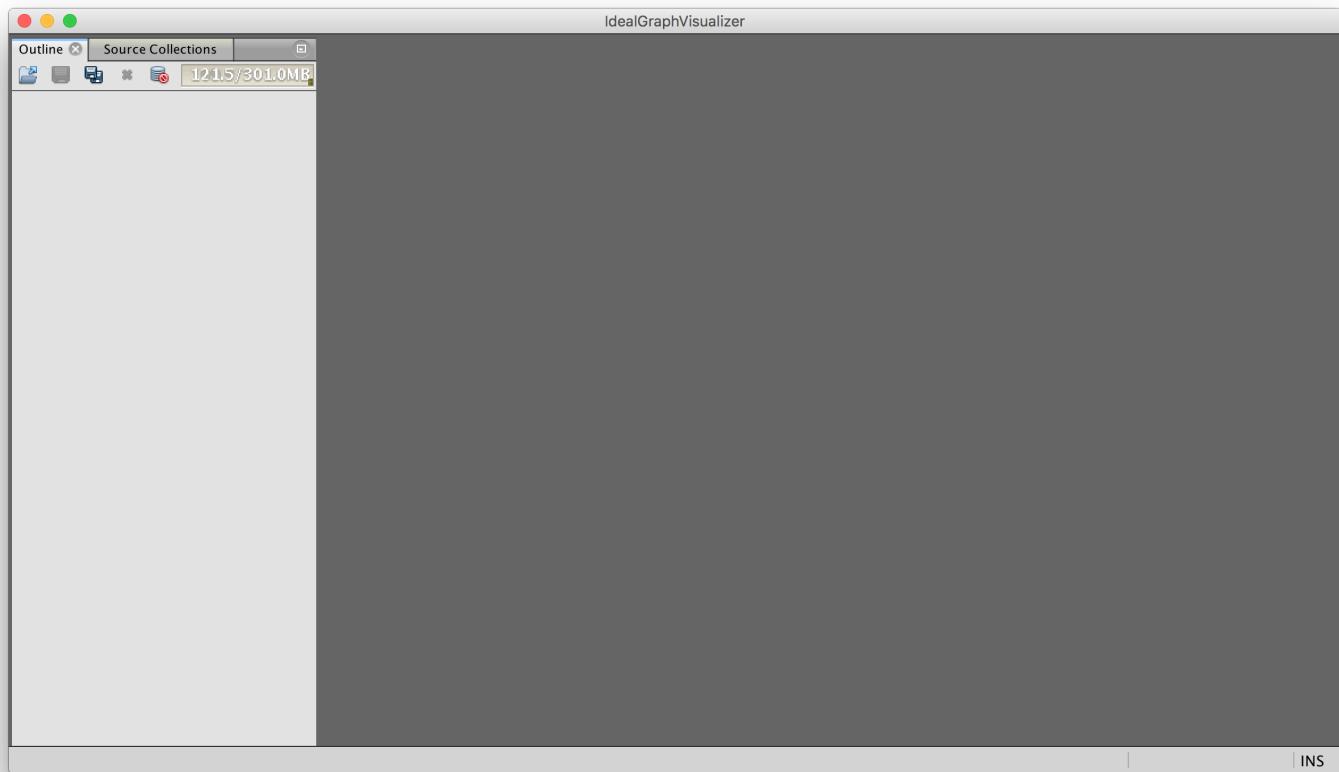
The blue edges in this graph represent how the data flows from reading the local variables, into doing the add.

We can also use edges to represent the order that the program has to run in. If we call methods instead of reading local variables, `getX() + getY()`, then we need to remember which order they were called in and we can't re-order them (without knowing what code is inside them). To do this we also have an edge which says do one call and then do the other, and this is shown in red.



So the Graal graph is in some ways two graphs at once composed on each other. The nodes are the same, but one set of edges says how the data flows through them, and the other says how control moves between them.

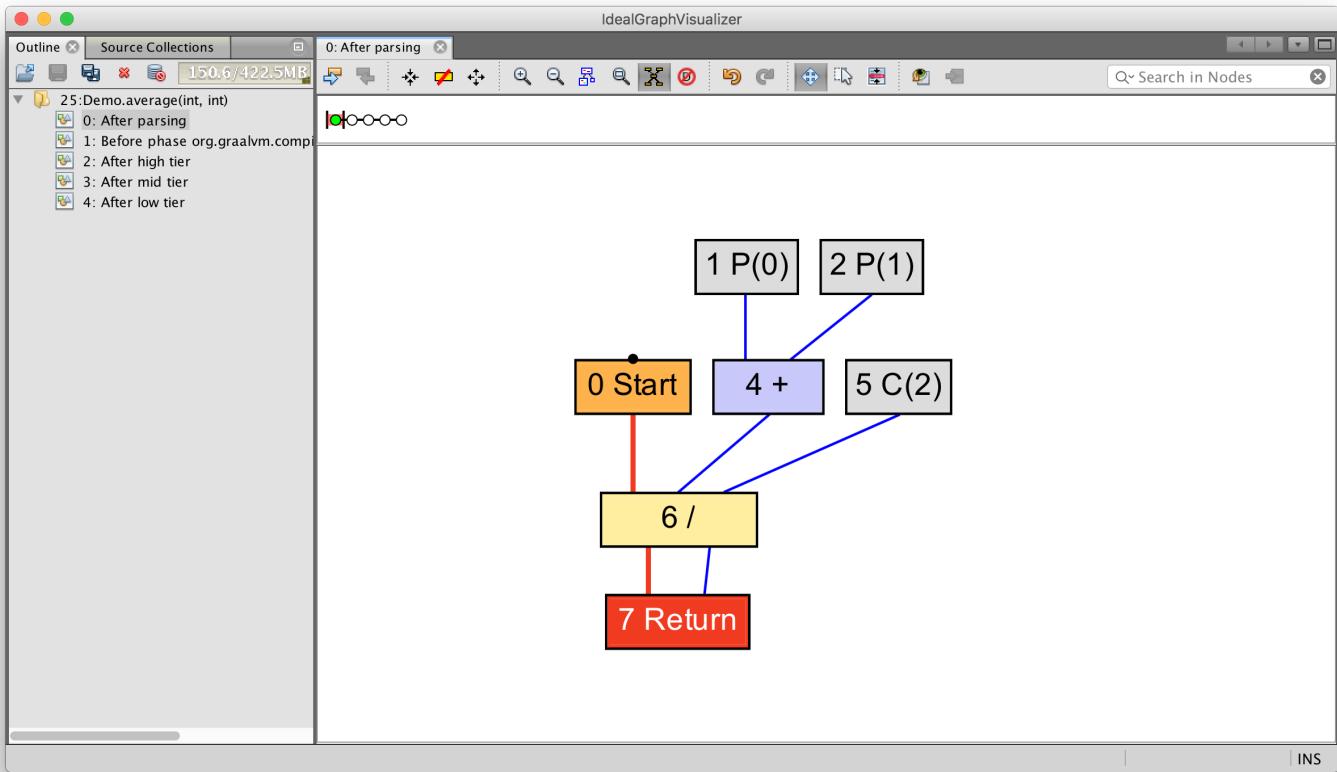
You can see real Graal graphs by using a tool called the IdealGraphVisualiser, or IGV. You can start this by running `mx igrv`.



Then run the JVM with `-Dgraal.Dump`.

We can see very simple dataflow only by writing a simple expression.

```
int average(int a, int b) {  
    return (a + b) / 2;  
}
```

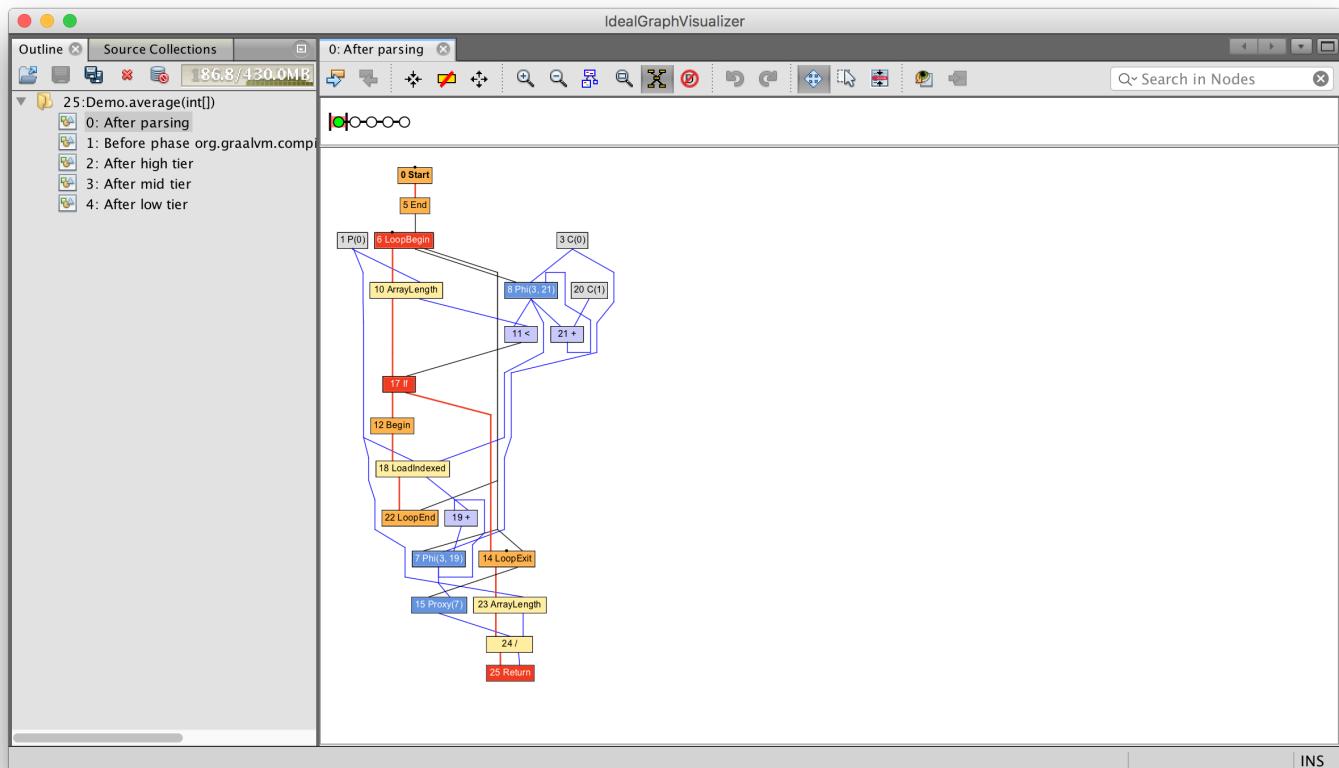


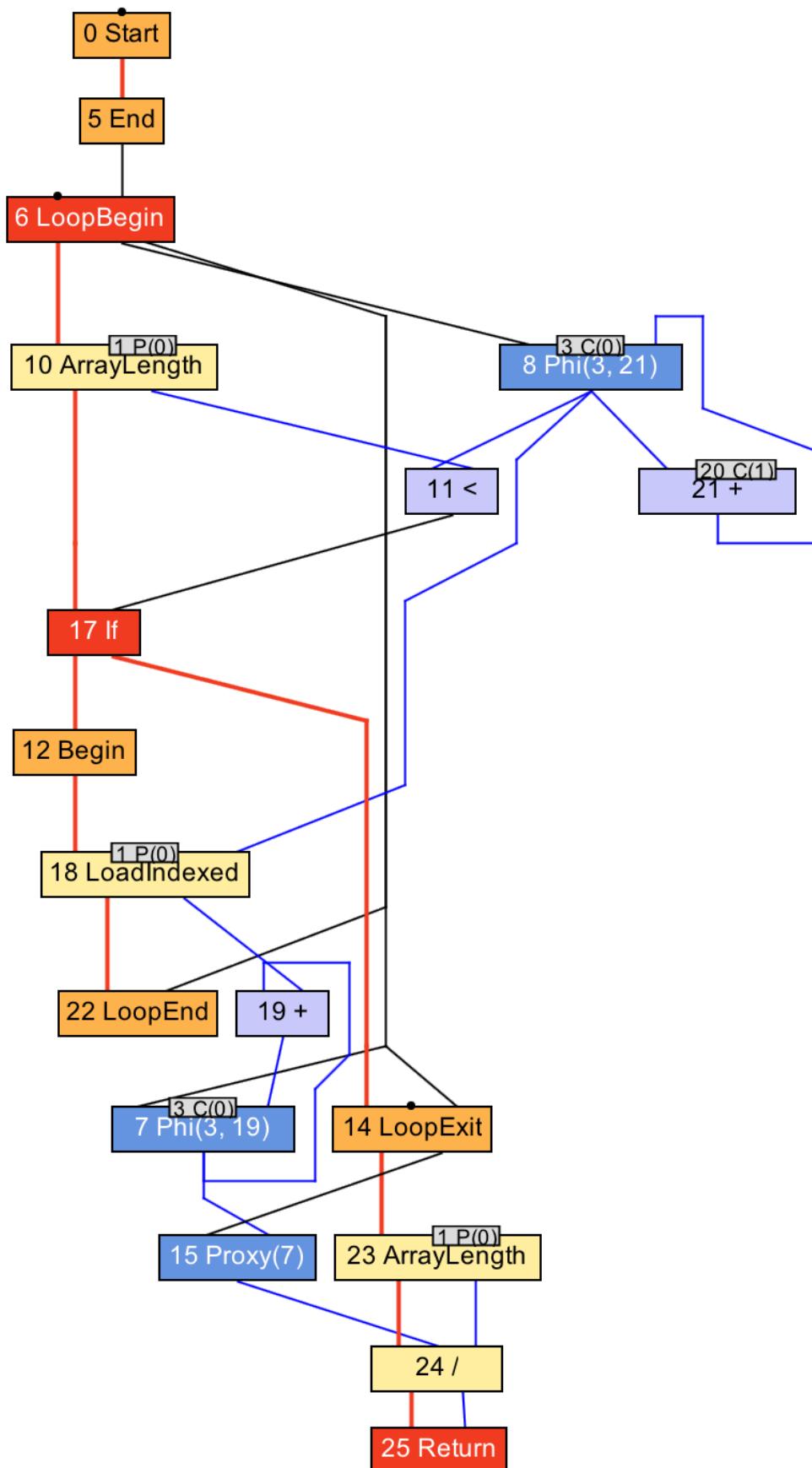
You can see how parameters **0** (written as  $P(0)$ ) and **1**(written as  $P(1)$ ) feed into an add operation, which feeds into a divide operation with the constant value **2** (written as  $C(2)$ ). This value is then returned.

We can see more complex data and control flow together if we introduce a loop.

```

int average(int[] values) {
    int sum = 0;
    for (int n = 0; n < values.length; n++) {
        sum += values[n];
    }
    return sum / values.length;
}
  
```





Now we have nodes to begin and end a loop, to read from the array, and to read the array length. As before, the blue lines show how data flows as before, and the red lines show how control moves.

You can start to see why this data structure is sometimes called a sea-of-nodes, or a soup-of-nodes.

I should say that C2 uses a very similar data structure, and it was really C2 that popularised the idea of a sea-of-nodes compiler, so that isn't the innovation in Graal.

I won't show the process by which this graph is created until the next section of the talk, but when Graal has your program in this format, it optimises and compiles it by modifying this data structure. And that's part of why writing JIT compilers in Java works well; Java is an object-orientated language, and a graph is a set of objects with references connecting them as edges.

## From bytecode to machine code

Let's see what these ideas look like in practice, and follow along some of the way through the compilation pipeline.

Bytecode in

Compilation starts with the bytecode. We'll go back to our tiny addition example.

```
int workload(int a, int b) {
    return a + b;
}
```

We'll show the bytecode that we get in by printing it out as the compiler starts.

```
class HotSpotGraalCompiler implements JVMMCICompiler {
    CompilationRequestResult compileMethod(CompilationRequest request) {
        System.err.println(request.getMethod().getName() + " bytecode: "
            + Arrays.toString(request.getMethod().getCode()));
        ...
    }
}

workload bytecode: [26, 27, 96, -84]
```

So that's the input to the compiler - the bytecode.

The bytecode parser and graph builder

This array of bytes is parsed as JVM bytecode into a Graal graph by the graph builder. This is a kind of abstract interpretation. It interprets the Java bytecode, but instead of passing around values it passes around loose ends of edges and connects them up as it goes.

Let's use the advantages of having Graal written in Java and see how this works using Eclipse's navigation tools. We know our example has an add node in it, so let's see where those are created.

graal - org.graalvm.compiler.nodes/src/org/graalvm/compiler/nodes/calc/AddNode.java - Eclipse

```
25+ import org.graalvm.compiler.core.common.type.ArithmeticOpTable;
40
41 @NodeInfo(shortName = "+")
42 public class AddNode| extends BinaryArithmeticNode<Add> implements NarrowableArithmeti
43
44     public static final NodeClass<AddNode> TYPE = NodeClass.create(AddNode.class);
45
46     public AddNode(ValueNode x, ValueNode y) {
47         this(TYPE, x, y);
48     }
49
50     protected AddNode(NodeClass<? extends AddNode> c, ValueNode x, ValueNode y) {
51         super(c, ArithmeticOpTable::getAdd, x, y);
52     }
53
54     public static ValueNode create(ValueNode x, ValueNode y) {
55         BinaryOp<Add> op = ArithmeticOpTable.forStamp(x.stamp()).getAdd();
56         Stamp stamp = op.foldStamp(x.stamp(), y.stamp());
57         ConstantNode tryConstantFold = tryConstantFold(op, x, y, stamp);
58         if (tryConstantFold != null) {
59             return tryConstantFold;
60         }
61         if (x.isConstant() && !y.isConstant()) {
62             return canonical(null, op, y, x);
63         } else {
64             return canonical(null, op, x, y);
65         }
66     }
67
68     private static ValueNode canonical(AddNode addNode, BinaryOp<Add> op, ValueNode forX, ValueNode forY) {
69         AddNode self = addNode;
```

Writable Smart Insert 42 : 21

graal - org.graalvm.compiler.nodes/src/org/graalvm/compiler/nodes/calc/AddNode.java - Eclipse

The screenshot shows the Eclipse IDE interface with the following details:

- Top Bar:** Shows the title "graal - org.graalvm.compiler.nodes/src/org/graalvm/compiler/nodes/calc/AddNode.java - Eclipse". Below it is the standard Eclipse toolbar.
- Left Side:** A tree view showing the project structure.
- Main Editor:** Displays the content of the `AddNode.java` file. The code implements a static method `create` that takes two `ValueNode` parameters and returns a new `ValueNode`. It uses a `BinaryOp<Add>` from the `ArithmeticOpTable` and performs various checks and returns a canonical form or the original node.
- Call Hierarchy View:** Located below the editor, this view lists methods that call the `create` method. The results are grouped by package:
  - `add(StructuredGraph, ValueNode, ValueNode) : ValueNode` - `org.graalvm.compiler.nodes.calc.BinaryArithmeticNode`
  - `add(ValueNode, ValueNode) : ValueNode` - `org.graalvm.compiler.nodes.calc.BinaryArithmeticNode` (marked with a red square)
  - `canonical(MulNode, BinaryOp<Mul>, Stamp, ValueNode, ValueNode) : ValueNode` - `org.graalvm.compiler.nodes.calc.MulNode` (2 matches)
  - `findSynonym(ValueNode, ValueNode) : LogicNode` - `org.graalvm.compiler.nodes.calc.IntegerLessThanNode.LessThanOp` (2 matches)
  - `genFloatAdd(ValueNode, ValueNode) : ValueNode` - `org.graalvm.compiler.java.BytecodeParser`
  - Selected Item:** `genIntegerAdd(ValueNode, ValueNode) : ValueNode` - `org.graalvm.compiler.java.BytecodeParser` (highlighted with a blue rectangle). This item has three associated methods:
    - `genArithmeticOp(JavaKind, int) : void` - `org.graalvm.compiler.java.BytecodeParser`
    - `processBytecode(int, int) : void` - `org.graalvm.compiler.java.BytecodeParser` (4 matches)
    - `genIncrement() : void` - `org.graalvm.compiler.java.BytecodeParser`
  - `org.graalvm.compiler.java.BytecodeParser.genIntegerAdd(ValueNode) : ValueNode` - `org.graalvm.compiler.java/src`

```

3416     genStoreIndexed(array, index, kind, value);
3417 }
3418
3419 private void genArithmeticOp(JavaKind kind, int opcode) {
3420     ValueNode y = frameState.pop(kind);
3421     ValueNode x = frameState.pop(kind);
3422     ValueNode v;
3423     switch (opcode) {
3424         case IADD:
3425         case LADD:
3426             v = genIntegerAdd(x, y);
3427             break;
3428         case FADD:
3429         case DADD:
3430             v = genFloatAdd(x, y);
3431             break;
3432         case ISUB:
3433         case LSUB:
3434             v = genIntegerSub(x, y);
3435             break;
3436         case FSUB:
3437         case DSUB:
3438             v = genFloatSub(x, y);
3439             break;
3440         case IMUL:
3441         case LMUL:
3442             v = genIntegerMul(x, y);
3443             break;
3444         case FMUL:
3445         case DMUL:
3446             v = genFloatMul(x, y);

```

We can see the bytecode parser creating them, and that leads us to the code where the `IADD` bytecode is parsed. (By the way, `IADD` is bytecode 96, which we saw in the input bytecode we printed.)

```

private void genArithmeticOp(JavaKind kind, int opcode) {
    ValueNode y = frameState.pop(kind);
    ValueNode x = frameState.pop(kind);
    ValueNode v;
    switch (opcode) {
        ...
        case LADD:
            v = genIntegerAdd(x, y);
            break;
        ...
    }
    frameState.push(kind, append(v));
}

```

I say that this is abstraction interpretation because this looks a lot like a bytecode interpreter. If this was an actual JVM interpreter, it would be popping two values from the stack, doing an addition, and then pushing the result. Here we pop two nodes from the stack which represent the computation when we actually run the program, add a new node for the addition, and then push that onto the stack representing the result of the addition when it is actually run.

This gives us our Graal graph.

Emitting assembly

When we want to turn the Graal graph into machine code, we need to generate machine code bytes for each node in the graph. This is done by asking each node to generate the machine code itself, in a `generate` method.

```

void generate(Generator gen) {
    gen.emitAdd(a, b);
}

```

Again, we're working at quite a high level of abstraction here. We have a class that lets us emit the machine code instruction without having to know here the details of how that works.

Now, the details of `emitAdd` are a bit complicated and abstracted because arithmetic operators have a lot of different combinations of operands to encode and the different operators can share most of their code, so I will change our program to do something a little more simple.

```
int workload(int a) {
    return a + 1;
}
```

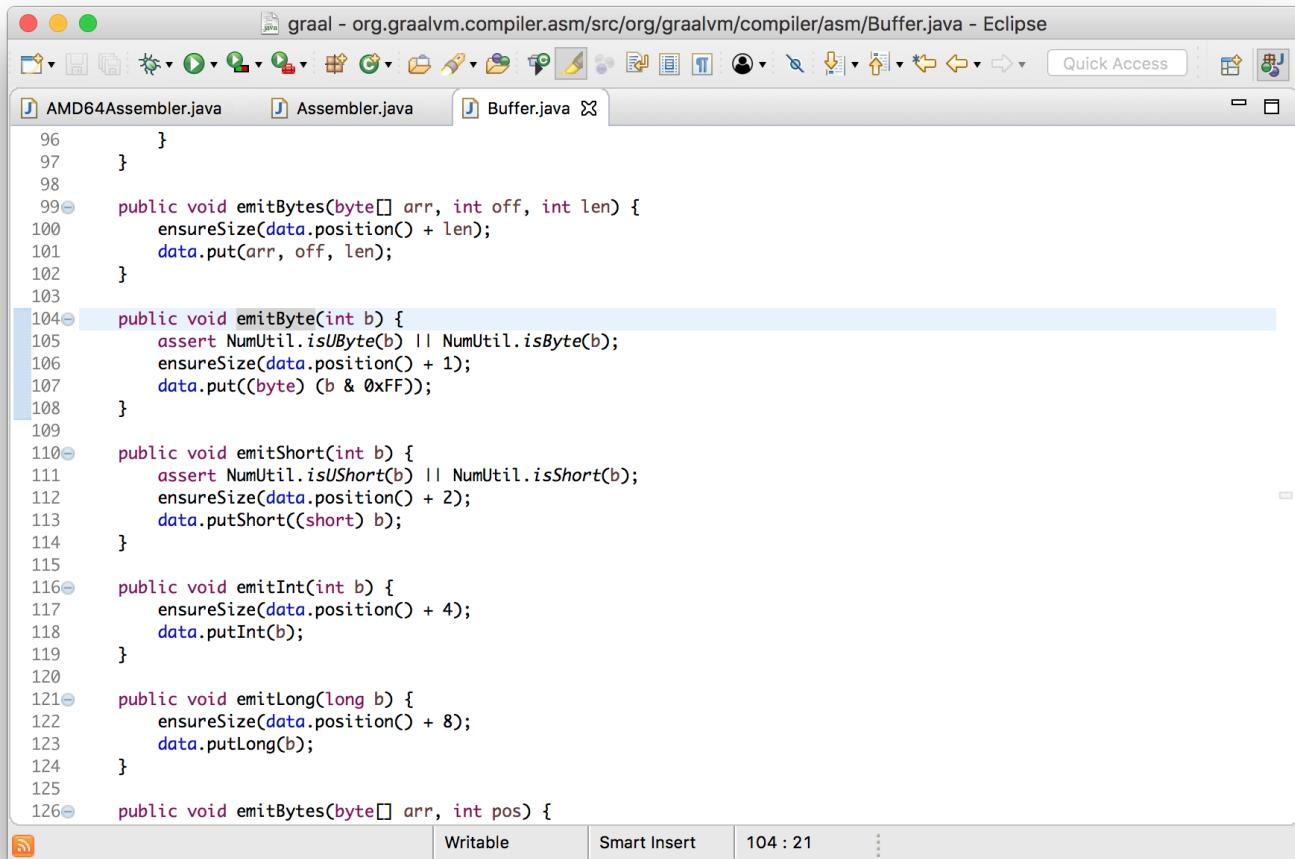
I'll tell you that this will use an increment instruction, and I'll show you what that looks like in the assembler.

```
void incl(Register dst) {
    int encode = prefixAndEncode(dst.encoding);
    emitByte(0xFF);
    emitByte(0xC0 | encode);
}

void emitByte(int b) {
    data.put((byte) (b & 0xFF));
}
```

The screenshot shows the Eclipse IDE interface with the file `AMD64Assembler.java` open. The code is part of the GraalVM compiler's assembly layer for AMD64 architecture. It includes methods for emitting the `incl` instruction (which increments a register) and the `jcc` (jump if condition) instruction. The `jcc` method handles both 8-bit and 32-bit displacement sizes, depending on the `forceDisp32` flag. The code uses various helper methods like `prefixAndEncode` and `isByte` to build the machine code bytes.

```
1831
1832     public final void imull(Register dst, Register src, int value) {
1833         if (isByte(value)) {
1834             AMD64RMIOp.IMUL_SX.emit(this, DWORD, dst, src, value);
1835         } else {
1836             AMD64RMIOp.IMUL.emit(this, DWORD, dst, src, value);
1837         }
1838     }
1839
1840     protected final void incl(AMD64Address dst) {
1841         prefix(dst);
1842         emitByte(0xFF);
1843         emitOperandHelper(0, dst, 0);
1844     }
1845
1846     public void jcc(ConditionFlag cc, int jumpTarget, boolean forceDisp32) {
1847         int shortSize = 2;
1848         int longSize = 6;
1849         long disp = jumpTarget - position();
1850         if (!forceDisp32 && isByte(disp - shortSize)) {
1851             // 0111 tttt #8-bit disp
1852             emitByte(0x70 | cc.getValue());
1853             emitByte((int) ((disp - shortSize) & 0xFF));
1854         } else {
1855             // 0000 1111 1000 tttt #32-bit disp
1856             assert isInt(disp - longSize) : "must be 32bit offset (call4)";
1857             emitByte(0x0F);
1858             emitByte(0x80 | cc.getValue());
1859             emitInt((int) (disp - longSize));
1860         }
1861     }
1862 }
```



```
graal - org.graalvm.compiler.asm/src/org/graalvm/compiler/asm/Buffer.java - Eclipse
AMD64Assembler.java Assembler.java Buffer.java

96     }
97 }
98
99     public void emitBytes(byte[] arr, int off, int len) {
100        ensureSize(data.position() + len);
101        data.put(arr, off, len);
102    }
103
104    public void emitByte(int b) {
105        assert NumUtil.isUByte(b) || NumUtil.isByte(b);
106        ensureSize(data.position() + 1);
107        data.put((byte) (b & 0xFF));
108    }
109
110    public void emitShort(int b) {
111        assert NumUtil.isUShort(b) || NumUtil.isShort(b);
112        ensureSize(data.position() + 2);
113        data.putShort((short) b);
114    }
115
116    public void emitInt(int b) {
117        ensureSize(data.position() + 4);
118        data.putInt(b);
119    }
120
121    public void emitLong(long b) {
122        ensureSize(data.position() + 8);
123        data.putLong(b);
124    }
125
126    public void emitBytes(byte[] arr, int pos) {
```

We can see here that it's emitting bytes to the output, and all this does it add the byte to a standard `ByteBuffer` - it's just building up an array of bytes.

#### Assembly out

In the same way as we reviewed the bytecode going in, let's view the machine code coming out. We'll modify where the machine code is installed to print the bytes.

```
class HotSpotGraalCompiler implements JVMCICompiler {
    CompilationResult compileHelper(...) {
        ...
        System.err.println(method.getName() + " machine code: "
            + Arrays.toString(result.getTargetCode()));
        ...
    }
}
```

```

174     Suites suites = getSuites(providers, options);
175     LIRSuites lirSuites = getLIRSuites(providers, options);
176     ProfilingInfo profilingInfo = useProfilingInfo ? method.getProfilingInfo(!isOSR, isOSR) : DefaultProfilingInfo.get
177     OptimisticOptimizations optimisticOpts = getOptimisticOpts(profilingInfo, options);
178
179     /*
180      * Cut off never executed code profiles if there is code, e.g. after the osr loop, that is never
181      * executed.
182     */
183     if (isOSR && !OnStackReplacementPhase.Options.DoptAfterOSR.getValue(options)) {
184         optimisticOpts.remove(Optimization.RemoveNeverExecutedCode);
185     }
186
187     result.setEntryBCI(entryBCI);
188     boolean shouldDebugNonSafepoints = providers.getCodeCache().shouldDebugNonSafepoints();
189     PhaseSuite<HighTierContext> graphBuilderSuite = configGraphBuilderSuite(providers.getSuites().getDefau
190     GraalCompiler.compileGraph(graph, method, providers, backend, graphBuilderSuite, optimisticOpts, profilingInfo, su
191
192     if (!isOSR && useProfilingInfo) {
193         ProfilingInfo profile = profilingInfo;
194         profile.setCompilerIRSize(StructuredGraph.class, graph.getNodeCount());
195     }
196
197     System.err.println(method.getName() + " machine code: " + Arrays.toString(result.getTargetCode()));
198
199     return result;
200 }
201
202 public CompilationResult compile(ResolvedJavaMethod method, int entryBCI, boolean useProfilingInfo, Compilatio
203     StructuredGraph graph = createGraph(method, entryBCI, useProfilingInfo, compilationId, options, debug);
204

```

I'll also use a tool that disassembles machine code as it's installed. This is standard for HotSpot - it's not a Graal thing. I'll show you how to build the tool - it lives in the OpenJDK repository but it isn't included in the JVM by default so we have to built it ourselves.

```

$ cd openjdk/hotspot/src/share/tools/hsdis
$ curl -O http://ftp.heanet.ie/mirrors/gnu/binutils/binutils-2.24.tar.gz
$ tar -xzf binutils-2.24.tar.gz
$ make BINUTILS=binutils-2.24 ARCH=amd64 CFLAGS=-Wno-error
$ cp build/macosx-amd64/hsdis-amd64.dylib ../../../../..

```

Now I'm going to add two new options, `-XX:+UnlockDiagnosticVMOptions` and `-XX:+PrintAssembly`.

```

$ java \
  --module-
  path=graal/sdk/mxbuild/modules/org.graalvm.graal_sdk.jar:graal/truffle/mxbuild/modules/com.oracle.truffle.truffle_api.jar \
  \
  --upgrade-module-path=graal/compiler/mxbuild/modules/jdk.internal.vm.compiler.jar \
  -XX:+UnlockExperimentalVMOptions \
  -XX:+EnableJVMCI \
  -XX:+UseJVMCICompiler \
  -XX:-TieredCompilation \
  -XX:+PrintCompilation \
  -XX:+UnlockDiagnosticVMOptions \
  -XX:+PrintAssembly \
  -XX:CompileOnly=Demo::workload \
  Demo

```

Now we can run our example and see our add instruction being emitted.

```

workload machine code: [15, 31, 68, 0, 0, 3, -14, -117, -58, -123, 5, ...]
...
0x0000000010f71cda0: nopl    0x0(%rax,%rax,1)
0x0000000010f71cda5: add     %edx,%esi          ;*iadd {reexecuted=0 rethrow=0 return_oop=0}
                                         ; - Demo::workload@2 (line 10)

0x0000000010f71cda7: mov     %esi,%eax         ;*ireturn {reexecuted=0 rethrow=0 return_oop=0}
                                         ; - Demo::workload@3 (line 10)

0x0000000010f71cda9: test    %eax,-0xcb8da9(%rip)   # 0x00000000102b74006
                                         ; {poll_return}

0x0000000010f71cdaf: vzeroupper
0x0000000010f71cdb2: retq

```

Ok, let's verify that we really are controlling all of this. Let's change the implementation of addition to make it actually subtraction instead. I'm going to edit the `generate` method of the addition node to emit a subtraction instruction instead of an addition instruction.

```

class AddNode {
    void generate(...) {
        ... gen.emitSub(op1, op2, false) ... // changed from emitAdd
    }
}

```

```

graal - org.graalvm.compiler.nodes/src/org/graalvm/compiler/nodes/calc/AddNode.java - Eclipse
File Edit View Insert Run Tools Window Help
AddNode.java
114     if (ret != this) {
115         return ret;
116     }
117
118     if (forX.isConstant() && !forY.isConstant()) {
119         // we try to swap and canonicalize
120         ValueNode improvement = canonical(tool, forY, forX);
121         if (improvement != this) {
122             return improvement;
123         }
124         // if this fails we only swap
125         return new AddNode(forY, forX);
126     }
127     BinaryOp<Add> op = getOp(forX, forY);
128     return canonical(this, op, forX, forY);
129 }
130
131 @Override
132 public void generate(NodeLIRBuilderTool nodeValueMap, ArithmeticLIRGeneratorTool gen) {
133     Value op1 = nodeValueMap.operand(getX());
134     assert op1 != null : getX() + ", this=" + this;
135     Value op2 = nodeValueMap.operand(getY());
136     if (shouldSwapInputs(nodeValueMap)) {
137         Value tmp = op1;
138         op1 = op2;
139         op2 = tmp;
140     }
141     nodeValueMap.setResult(this, gen.emitAdd(op1, op2, false));
142 }
143 }

```

If we run this we can see both the machine code bytes change and see the new instruction.

```

workload machine code: [15, 31, 68, 0, 0, 43, -14, -117, -58, -123, 5, ...]
0x00000000107f451a0: nopl    0x0(%rax,%rax,1)
0x00000000107f451a5: sub     %edx,%esi          ;*iadd {reexecuted=0 rethrow=0 return_oop=0}
                                         ; - Demo::workload@2 (line 10)

0x00000000107f451a7: mov     %esi,%eax         ;*ireturn {reexecuted=0 rethrow=0 return_oop=0}

```

```

; - Demo::workload@3 (line 10)

0x0000000107f451a9: test    %eax,-0x1db81a9(%rip)      # 0x000000010618d006
                           ; {poll_return}
0x0000000107f451af: vzeroupper
0x0000000107f451b2: retq

```

So what have we learned here? Graal really does take in a simple byte array of bytecode, we can see how the graph nodes are created from that, we can see how the nodes emit instructions, and how the instructions are encoded. We saw that we could change how Graal worked.

[26, 27, 96, -84] → [15, 31, 68, 0, 0, 43, -14, -117, -58, -123, 5, ...]

## Optimisations

Ok, we've shown how we get the graph, and how the nodes in the graph are output as machine code. Let's talk now about how Graal optimises the graph to make it more efficient.

An optimisation phase is just a method that has the opportunity to modify the graph. You write phases by implementing an interface.

```

interface Phase {
    void run(Graph graph);
}

```

## Canonicalisation

Canonicalisation means rearranging nodes into a uniform representation. This has some other purposes as well, but for the purposes of this presentation I'll say that it really means constant folding and simplifying the nodes.

Nodes are responsible for simplifying themselves - they have a method `canonical`.

```

interface Node {
    Node canonical();
}

```

Let's look at something like the negate node, which is the unary subtraction operator. The negate node will remove itself and its child if it's being applied to another negate node, leaving just the value behind. It simplifies  $- - x$  to just  $x$ .

```

class NegateNode implements Node {
    Node canonical() {
        if (value instanceof NegateNode) {
            return ((NegateNode) value).getValue();
        } else {
            return this;
        }
    }
}

```

```
58     }
59
60     @Override
61     public ValueNode canonical(CanonicalizerTool tool, ValueNode forValue) {
62         ValueNode synonym = findSynonym(forValue, getOp(forValue));
63         if (synonym != null) {
64             return synonym;
65         }
66         return this;
67     }
68
69     protected static ValueNode findSynonym(ValueNode forValue) {
70         ArithmeticOpTable.UnaryOp<Neg> negOp = ArithmeticOpTable.forStamp(forValue.stamp()).getNeg();
71         ValueNode synonym = UnaryArithmeticNode.findSynonym(forValue, negOp);
72         if (synonym != null) {
73             return synonym;
74         }
75         if (forValue instanceof NegateNode) {
76             return ((NegateNode) forValue).getValue();
77         }
78         if (forValue instanceof SubNode && !(forValue.stamp() instanceof FloatStamp)) {
79             SubNode sub = (SubNode) forValue;
80             return SubNode.create(sub.getY(), sub.getX());
81         }
82         return null;
83     }
84
85     @Override
86     public void generate(NodeLIRBuilderTool nodeValueMap, ArithmeticLIRGeneratorTool gen) {
87         nodeValueMap.setResult(this, gen.emitNegate(nodeValueMap.operand(getValue())));
88     }
}
```

This is a really great example of how understandable Graal is. The logic here is almost as simple as it could be.

If you have a clever idea for how to simplify a Java operation, you can modify the `canonical` method to do it.

### Global value numbering

Global value numbering is a technique to remove code that is redundant because it appears more than once. In this example,  $a + b$  could be calculated just once and the value then used twice.

```
int workload(int a, int b) {
    return (a + b) * (a + b);
}
```

Graal can compare nodes to see if they're equal. It's simple - they're equal if they have the same inputs. Graal's global value numbering phase looks to see if each node is equal to any other and replaces them with a single copy if they are. It does this efficiently by putting all the nodes into a hash map. It's a bit like a cache of nodes.

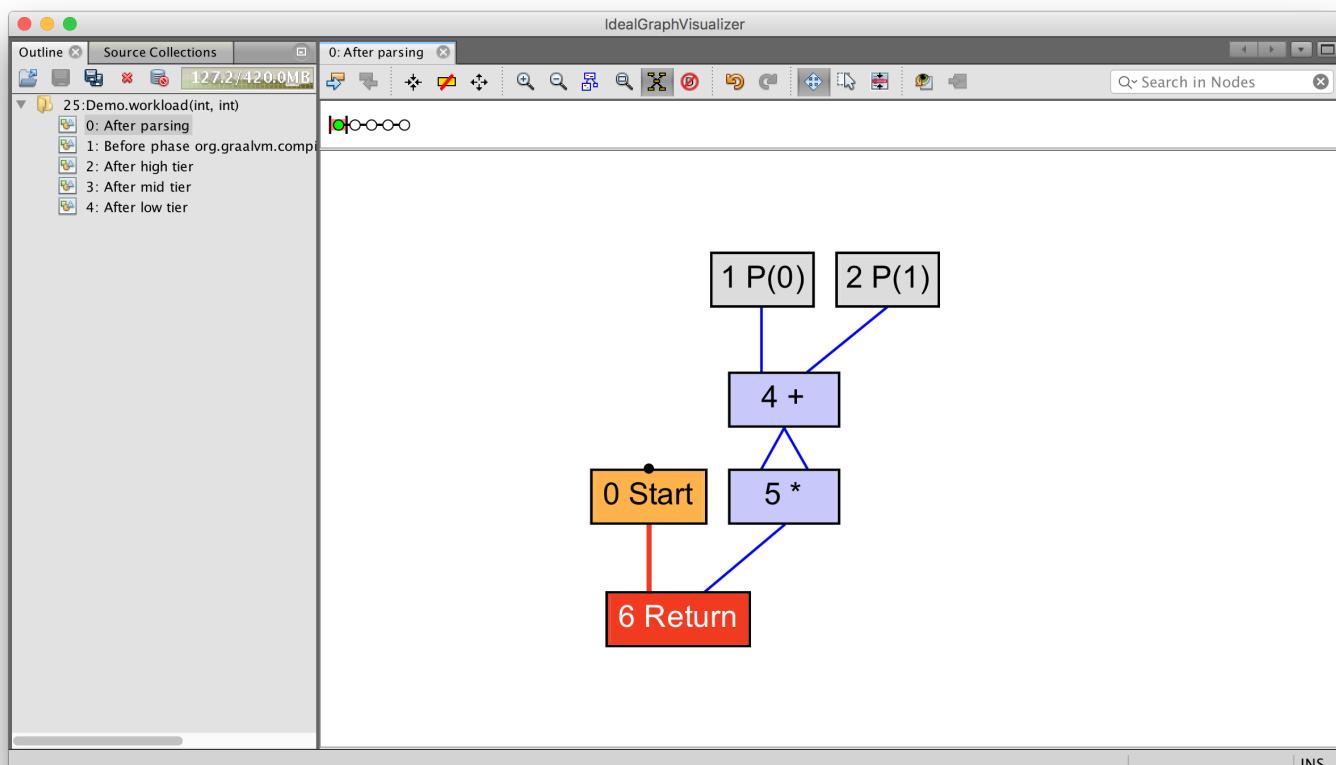
graal - org.graalvm.compiler.phases.common/src/org/graalvm/compiler/phases/common/CanonicalizerPhase.java - Eclipse

```

274         }
275         valueNode.usages().forEach(workList::add);
276     }
277     return false;
278 }
279
280 public boolean tryGlobalValueNumbering(Node node, NodeClass<?> nodeClass) {
281     if (nodeClass.valueNumberable()) {
282         Node newNode = node.graph().findDuplicate(node);
283         if (newNode != null) {
284             assert !(node instanceof FixedNode || newNode instanceof FixedNode);
285             node.replaceAtUsagesAndDelete(newNode);
286             COUNTER_GLOBAL_VALUE_NUMBERING_HITS.increment(debug);
287             debug.log("GVN applied and new node is %1s", newNode);
288             return true;
289         }
290     }
291     return false;
292 }
293
294 private AutoCloseable getCanonicalizableContractAssertion(Node node) {
295     boolean needsAssertion = false;
296     assert (needsAssertion = true) == true;
297     if (needsAssertion) {
298         Mark mark = node.graph().getMark();
299         return () -> {
300             assert mark.equals(node.graph().getMark()) : "new node created while canonicalizing " + node.getClass();
301             node.graph().getNewNodes(mark).snapshot();
302         };
303     } else {
304     }

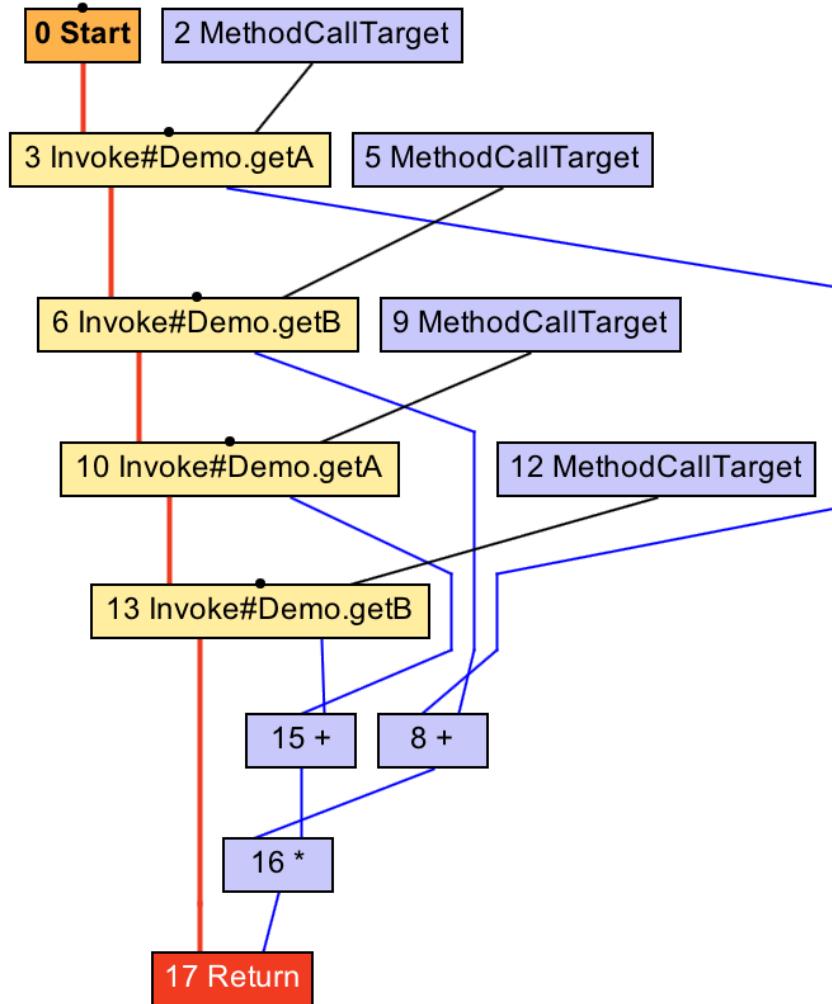
```

Writable Smart Insert 281 : 28



Note that test that the node isn't fixed - this means that the node doesn't possibly have a side effect so has to happen at a certain point of time. If we use a method call instead then the terms become fixed and not redundant so they aren't merged into one.

```
int workload() {
    return (getA() + getB()) * (getA() + getB());
}
```



### Lock coarsening

Let's look at one more complex example. Sometimes people write code that synchronises on the same monitor twice immediately after each other. They may not literally write this, but it may result from other optimisations such as inlining.

```
void workload() {
    synchronized (monitor) {
        counter++;
    }
    synchronized (monitor) {
        counter++;
    }
}
```

We'll de-sugar that and say that it's effectively doing this.

```
void workload() {
    monitor.enter();
    counter++;
}
```

```
    monitor.exit();
    monitor.enter();
    counter++;
    monitor.exit();
}
```

We could optimise this code to enter the monitor just once, instead of leaving it just to enter it again. This is lock coarsening.

```
void workload() {
    monitor.enter();
    counter++;
    counter++;
    monitor.exit();
}
```

In Graal this is implemented in a phase called `LockEliminationPhase`. Its `run` method looks for all monitor exit nodes and sees if they are immediately followed by another enter node. It then confirms that they use the same monitor, and if so will remove them both, leaving the outer enter and exit nodes.

```
void run(StructuredGraph graph) {
    for (monitorExitNode monitorExitNode : graph.getNodes(MonitorExitNode.class)) {
        FixedNode next = monitorExitNode.next();
        if (next instanceof monitorEnterNode) {
            AccessmonitorNode monitorEnterNode = (AccessmonitorNode) next;
            if (monitorEnterNode.object() == monitorExitNode.object()) {
                monitorExitNode.remove();
                monitorEnterNode.remove();
            }
        }
    }
}
```

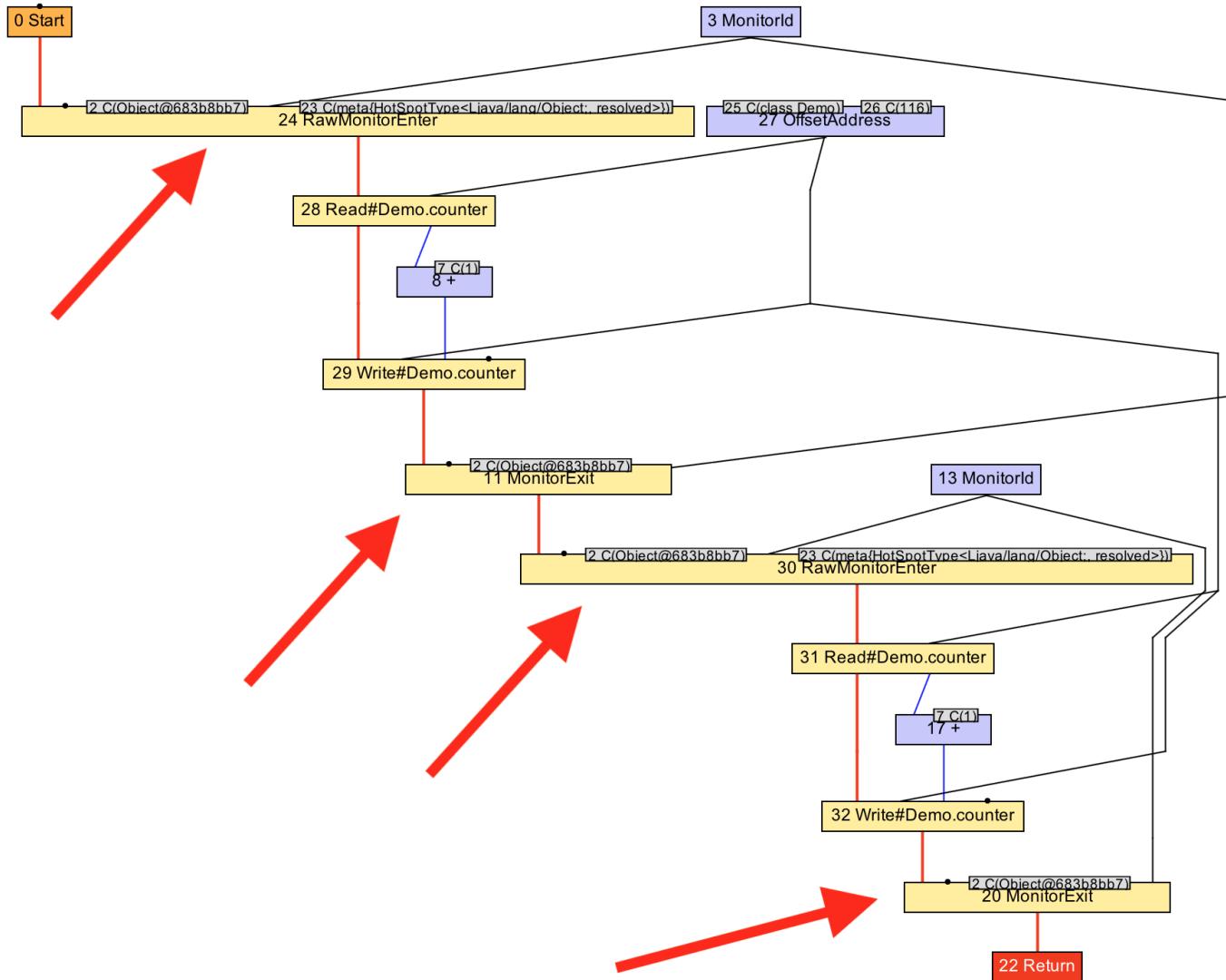
The screenshot shows the Eclipse IDE interface with the title bar "graal - org.graalvm.compiler.phases.common/src/org/graalvm/compiler/phases/common/LockEliminationPhase.java - Eclipse". The main window displays the Java code for the LockEliminationPhase class. The code is annotated with line numbers from 35 to 64. It includes comments explaining the logic of coarsening locks and combining monitor enter and exit operations. The Eclipse toolbar at the top and status bar at the bottom are visible.

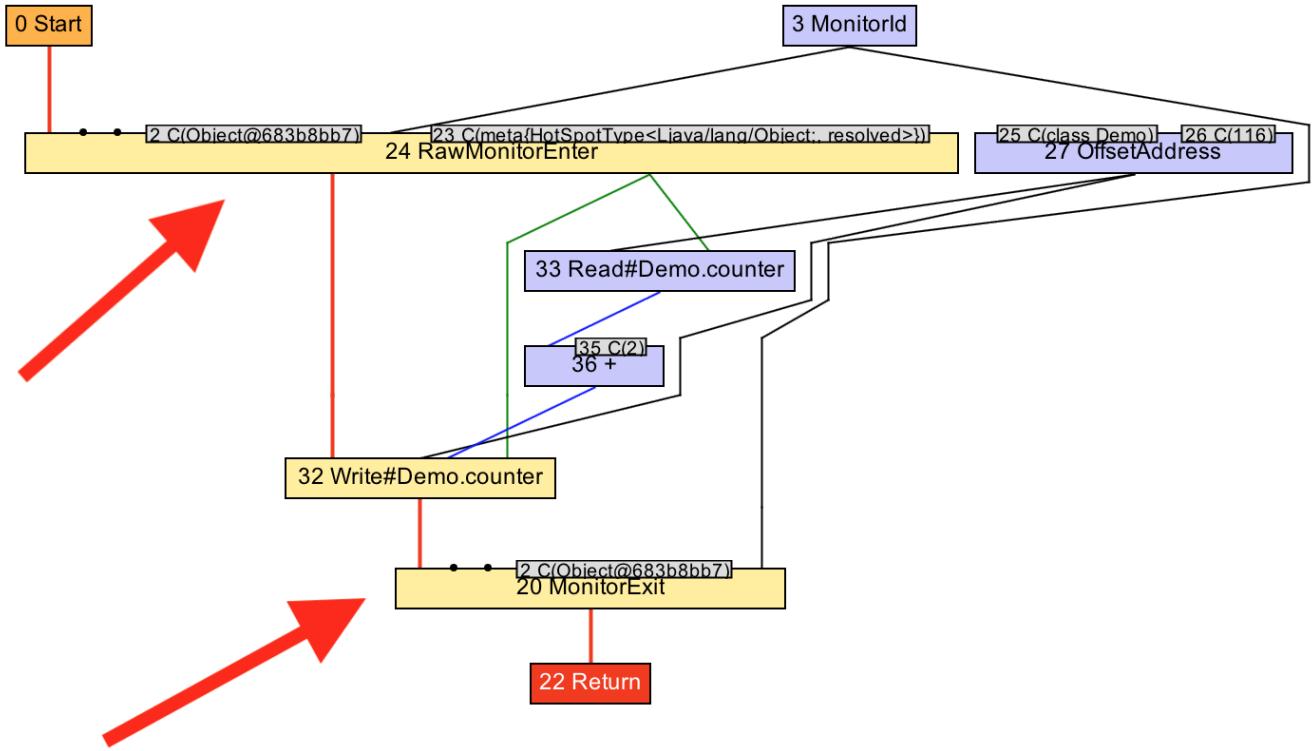
```
35
36 public class LockEliminationPhase extends Phase {
37
38     @Override
39     protected void run(StructuredGraph graph) {
40         for (MonitorExitNode monitorExitNode : graph.getNodes(MonitorExitNode.TYPE)) {
41             FixedNode next = monitorExitNode.next();
42             if ((next instanceof MonitorEnterNode || next instanceof RawMonitorEnterNode)) {
43                 // should never happen, osr monitor enters are always direct successors of the graph
44                 // start
45                 assert !(next instanceof OSRMonitorEnterNode);
46                 AccessMonitorNode monitorEnterNode = (AccessMonitorNode) next;
47                 if (GraphUtil.unproxy(monitorEnterNode.object()) == GraphUtil.unproxy(monitorExitNode.object())) {
48                     /*
49                     * We've coarsened the lock so use the same monitor id for the whole region,
50                     * otherwise the monitor operations appear to be unrelated.
51                     */
52                     MonitorIdNode enterId = monitorEnterNode.getMonitorId();
53                     MonitorIdNode exitId = monitorExitNode.getMonitorId();
54                     if (enterId != exitId) {
55                         enterId.replaceAndDelete(exitId);
56                     }
57                     GraphUtil.removeFixedWithUnusedInputs(monitorEnterNode);
58                     GraphUtil.removeFixedWithUnusedInputs(monitorExitNode);
59                 }
60             }
61         }
62     }
63 }
64
```

The reason that this is worth doing is that it means less code for the extra exit and enter, but it also allows us to apply more optimisations such as combining the two increments into a single add of 2.

```
void workload() {
    monitor.enter();
    counter += 2;
    monitor.exit();
}
```

Let's see this working with IGV. We can see the graph go from having the two pairs of monitor enters and exits, to just one pair, after the optimisation phase has run, and how the two increments become one addition of the number two.





## Some practicalities that I haven't talked about

In giving a high-level overview of how Graal works of course I'm skipping over a lot of important practical details for making it work well and making the code it produces efficient, and I'm actually also skipping some essential things that are needed to make it work at all.

I haven't talked about some of these parts of Graal because they aren't as simple conceptually to show like the code I've gone through already, but I'll still point you at where you can find them if you want.

### Register allocation

In the model of the Graal graph we have nodes passing values to each other over their edges. But what are these edges in practice? If a machine instruction needs data to be passed in or passed out what does it use?

The edges are eventually mapped to the processor's registers. Registers are like the processor's hardware local variables. They're the highest part of the system's memory hierarchy, above the various levels of processor cache, which are in turn above the system RAM. Machine instructions can read or write registers, and value can so be passed from one instruction to another by being written into a register by the first, and then read from a register by the second.

Deciding which registers to use for each edge is a problem called *register allocation*. Graal uses similar register allocation algorithms to other JIT-compilers (it's the *linear scan* algorithm).

### Scheduling

The other essential problem I haven't mentioned is that we have a graph of nodes without any clear order to run them in, but the processor needs a linear sequence of instructions in a definite order.

For example an add instruction could take two values as input to add together, and if there isn't a need to execute one before the other - if they don't have any side effects - then the graph doesn't tell us which to execute first. When we emit machine code though we do need to make a decision and put one of the inputs's code before the other.

This problem is called graph scheduling. The scheduler needs to decide what order to run all the nodes in. It decides when to run code based on the constraints that all values need to have been computed by the time that they're needed for a computation. You could produce a schedule which just works, but you can improve the performance of code for example by not running a computation until the value is definitely needed.

You can get even more sophisticated by applying your understanding of what resources your processor has and giving it work in an order which means those resources are used as efficiently as possible.

## What can you use Graal for?

I said at the start of the presentation in the legal slide that Graal is just a research project at the moment, rather than an Oracle product that we are supporting. What do we anticipate could be the applications of the research that is going into Graal?

## A final-tier compiler

Graal could be used as a final-tier in HotSpot, via JVMI, which is what I've been showing in my demos. As we develop new optimisations in Graal that HotSpot doesn't currently have, Graal could be the compiler that you use in order to get better performance.

Twitter have talked about how they're using Graal for this purpose, and with the release of Java 9 this is something practical that you can do today if you want to start experimenting. You just need `-XX:+UseJVMICompiler` and the other flags to get started.

The great thing about JVMI is that it lets you ship Graal separately from the JVM. You can deploy one version of the JVM, and then deploy new versions of Graal separately. In the same way that you don't need to recompile the JVM to use a Java agent, you don't have to recompile the JVM to update the compiler if you are using Graal.

*Metropolis* is an OpenJDK project to implement more of the JVM using the Java programming language. Graal could be one step towards doing this.

The screenshot shows a web browser window with the URL `cr.openjdk.java.net` in the address bar. The content of the page is an email message:

**From:** John Rose  
**To:** discuss@openjdk.java.net  
**Subject:** Call for Discussion: New Project: Metropolis

I would like to invite discussion on a proposal for a new OpenJDK Project[1], to be titled “Project Metropolis”, an incubator for experimenting with advanced JVM implementation techniques. Specifically, we wish to re-implement significant parts of Hotspot’s C++ runtime in Java itself, a move we call *Java-on-Java*. The key experiments will center around investigating Graal[2] as a code generator for the JVM in two modes: as an online compiler replacing one or more of Hotspot’s existing JITs, and as an offline compiler for Java code intended to replace existing C++ code in Hotspot. In the latter role, we will experiment with static compilation techniques (such as the Substrate VM[3]) to compile Java into statically restricted formats that can easily integrate with C++ as used in Hotspot.

The Project will be an experimental technology incubator, similar to the Lambda, Panama, Valhalla, and Amber projects. Such incubator projects absorb changes from the current Java release, but do not directly push to Java releases. Instead, they accumulate prototype changes which are sometimes discarded and sometimes merged by hand (after appropriate review) into a Java release.

(In this model, prototype changes accumulate quickly, since they are not subject to the relatively stringent rules governing JDK change-sets. These rules involving review, bug tracking, regression tests, and pre-integration builds. The Metropolis project will have similar rules, of course, but they are likely to be more relaxed.)

Implementing the Java runtime in the Java-on-Java style has **numerous advantages**, including:

- **Self optimization:** We obtain more complete control of optimization techniques

<http://cr.openjdk.java.net/~jrose/metropolis/Metropolis-Proposal.html>

## Your own specific optimisations

It's also possible to extend Graal with extra optimisations. In the same way that you can plug Graal into the JVM, you can plug new compilation phases into Graal. If you knew that you wanted to apply a specific optimisation to your application you could write a Graal phase to do that. Or if you had some specific set of machine code instructions that you wanted to run you could write a new intrinsic method rather than having to write a native method and call it via JNI.

Charles Nutter has already been proposing doing this for JRuby and has demonstrated it having a positive performance impact when he added a new phase to Graal to relax object identity for Ruby's boxed numbers. I'm sure he'll be doing some conference talks about this soon.

## Ahead-of-time compilation

Graal is just a Java library. JVMCI provides interfaces that Graal can use to do lower-level things such as installing machine code, but most of Graal is pretty-well isolated from them. That means you can use Graal for other applications beyond being a JIT-compiler.

There isn't really much difference between a JIT-compiler and a more conventional ahead-of-time compiler, and Graal can be used to build one of those as well. There are actually two projects doing this.

Java 9 includes a tool for compiling classes to machine code ahead-of-time to reduce time needed for JIT compilation, particularly during the startup phase of an application. You still need a full JVM to use this code - it just uses the ahead-of-time compiled code rather than running the compiler on demand.

The version of Graal that Java 9 AOT uses is a little older, and only included in the Linux builds, which is why I haven't been using it for this demo and instead showed how I've built a more recent version and the command line arguments needed to plug it in.

The second project is more ambitious. The SubstrateVM is an ahead-of-time compiler that compiles Java applications to native code which does not have a dependency on the JVM. You literally get a single statically linked executable out of it. You don't need a JVM and the binary can be as small as a few megabytes. The SubstrateVM uses Graal to do this compilation. In some configurations, the SubstrateVM can also compile Graal into itself so that it can compile code at runtime as well, just-in-time. So Graal is ahead-of-time compiling itself.

```
$ javac Hello.java

$ graalvm-0.28.2/bin/native-image Hello
  classlist: 966.44 ms
    (cap): 804.46 ms
    setup: 1,514.31 ms
  (typeflow): 2,580.70 ms
  (objects): 719.04 ms
  (features): 16.27 ms
    analysis: 3,422.58 ms
    universe: 262.09 ms
    (parse): 528.44 ms
    (inline): 1,259.94 ms
  (compile): 6,716.20 ms
    compile: 8,817.97 ms
    image: 1,070.29 ms
  debuginfo: 672.64 ms
    write: 1,797.45 ms
  [total]: 17,907.56 ms

$ ls -lh hello
-rwxr-xr-x 1 chrisseaton staff 6.6M 4 Oct 18:35 hello

$ file ./hello
./hellojava: Mach-O 64-bit executable x86_64

$ time ./hello
Hello!

real 0m0.010s
user 0m0.003s
sys 0m0.003s
```

## Truffle

Another project using Graal as a library is called Truffle. Truffle is a framework for creating interpreters for languages on top of the JVM.

Most other languages which are built on top of the JVM work by emitting JVM bytecode, which is then JIT-compiled as normal. (But as I described when I talked about how the JVM's JIT compilers were black boxes, it isn't always easy to control what happens to this bytecode.) The Truffle approach is different - you write a simple interpreter for your language, following some rules, and Truffle will automatically combine the program and the interpreter to produce optimised machine code, using a technique known as *partial evaluation*.

Partial evaluation has some interesting theory behind it, but practically we can talk about partial evaluation as being inlining and constant folding of a program with the data running through it. Graal has logic for inlining and constant folding, so Truffle can use it to build a partial evaluator.

This is how I was introduced to Graal - via Truffle. I work on an implementation of the Ruby programming language called TruffleRuby that uses the Truffle framework, and so also uses Graal. TruffleRuby is easily the fastest implementation of Ruby, often 10x faster than other implementations, while still implementing almost all of the language and standard library.

<https://github.com/graalvm/truffleruby>

## Summary

There's one key thing that I really wanted to get across in this talk - a Java JIT-compiler can be treated as any other code would be. JIT-compilation has a lot of complexity - mainly inherited from the complexity of the underlying architecture and also because we want the generated code to be as efficient as it can be without spending too much time in the JIT - but it's still a high-level task. The interface to a JIT compiler isn't much more than converting the `byte[]` of JVM bytecode to the `byte[]` of machine code.

This is a task which is well-suited to being done in Java. The actual compilation is not a low-level task needing a low-level and unsafe systems programming language like C++.

The Java code in Graal is not magic. I won't pretend it's always simple, but much of it an interested novice can read and understand.

I'd encourage you to go and look yourself using the pointers I've given you. If you start with the classes I've shown then you won't be lost when you first open Eclipse and see the long list of packages. From these starting points you can follow definitions, navigate to callers, and so on to learn more about the code base.

If you're already used to trying to control and tune the JIT, using tools for the existing JVM JIT-compilers, such as JITWatch, then you may find that reading the code will help further understand why Graal compiles your code as it does. And if you find things aren't working as you want, you can indeed change Graal, and just relaunch the JVM. You don't even need to leave your IDE to do it, as I showed with my hello-world example.

We're working on some amazing research projects using Graal, such as the SubstrateVM and Truffle, which really change what could be possible in Java in the future. These are made possible because all of Graal is written in plain Java. If we used something like the LLVM project to write our compiler, as some other companies propose, then it's not as easy to re-use the code in some many contexts.

And finally, it's now possible to use Graal with an unmodified JVM. With JVMBI now being part of Java 9, you can plug in Graal like you plug in an annotation processor or Java agent today.

Graal is a large project with a lot of people working on it. As I said I don't really work on Graal myself, I'm not much more than a user of it, as you could be!

- [More information about TruffleRuby](#)
- [Understanding How Graal Works - a Java JIT Compiler Written in Java](#)
- [Flip-Flops — the 1-in-10-million operator](#)
- [Deoptimizing Ruby](#)
- [Very High Performance C Extensions For JRuby+Truffle](#)
- [Optimising Small Data Structures in JRuby+Truffle](#)
- [Pushing Pixels with JRuby+Truffle](#)
- [Tracing With Zero Overhead in JRuby+Truffle](#)
- [How Method Dispatch Works in JRuby+Truffle](#)
- [A Truffle/Graal High Performance Backend for JRuby](#)