```
**********************************************************
   43950 Thu Aug 25 01:58:52 2011
new/src/share/vm/ci/ciEnv.cpp
**********************************************************
______unchanged_portion_omitted_
```

```
 886 // ------------------------------------------------------------------
 887 // ciEnv::validate_compile_task_dependencies
 888 //
 889 // Check for changes during compilation (e.g. class loads, evolution,
 890 // breakpoints, call site invalidation).
 891 void ciEnv::validate_compile_task_dependencies(ciMethod* target) {
 887 // ciEnv::check_for_system_dictionary_modification
 888 // Check for changes to the system dictionary during compilation
 889 // class loads, evolution, breakpoints
 890 void ciEnv::check_for_system_dictionary_modification(ciMethod* target) {
 892   if (failing())  return;  // no need for further checks

 894   // First, check non-klass dependencies as we might return early and
 895   // not check klass dependencies if the system dictionary
 896   // modification counter hasn't changed (see below).
 897   for (Dependencies::DepStream deps(dependencies()); deps.next(); ) {
 898     if (deps.is_klass_type())  continue;  // skip klass dependencies
 899     klassOop witness = deps.check_dependency();
 900     if (witness != NULL) {
 901       record_failure("invalid non-klass dependency");
 902       return;
 903     }
 904   }
 893   // Dependencies must be checked when the system dictionary changes.
 894   // If logging is enabled all violated dependences will be recorded in
 895   // the log.  In debug mode check dependencies even if the system
 896   // dictionary hasn't changed to verify that no invalid dependencies
 897   // were inserted.  Any violated dependences in this case are dumped to
 898   // the tty.

 906   // Klass dependencies must be checked when the system dictionary
 907   // changes.  If logging is enabled all violated dependences will be
 908   // recorded in the log.  In debug mode check dependencies even if
 909   // the system dictionary hasn't changed to verify that no invalid
 910   // dependencies were inserted.  Any violated dependences in this
 911   // case are dumped to the tty.
 912 #endif /* ! codereview */
 913   bool counter_changed = system_dictionary_modification_counter_changed();
 914   bool test_deps = counter_changed;
 915   DEBUG_ONLY(test_deps = true);
 916   if (!test_deps)  return;

 918   bool print_failures = false;
 919   DEBUG_ONLY(print_failures = !counter_changed);

 920   bool keep_going = (print_failures || xtty != NULL);
 921   int klass_violations = 0;

 903   int violated = 0;

 923   for (Dependencies::DepStream deps(dependencies()); deps.next(); ) {
 924     if (!deps.is_klass_type())  continue;  // skip non-klass dependencies
 925 #endif /* ! codereview */
 926     klassOop witness = deps.check_dependency();
 927     if (witness != NULL) {
 928       klass_violations++;
 906       ++violated;
 929       if (print_failures)  deps.print_dependency(witness, /*verbose=*/ true);
 930     }
 931 #endif /* ! codereview */
```



```
 932     // If there's no log and we're not sanity-checking, we're done.
 933     if (!keep_going)  break;
 934   }
 908   }

 936   if (klass_violations != 0) {
 910   if (violated != 0) {
 937     assert(counter_changed, "failed dependencies, but counter didn't change");
 938     record_failure("concurrent class loading");
 939   }
 940 }

 942 // ------------------------------------------------------------------
 943 // ciEnv::register_method
 944 void ciEnv::register_method(ciMethod* target,
 945                             int entry_bci,
 946                             CodeOffsets* offsets,
 947                             int orig_pc_offset,
 948                             CodeBuffer* code_buffer,
 949                             int frame_words,
 950                             OopMapSet* oop_map_set,
 951                             ExceptionHandlerTable* handler_table,
 952                             ImplicitExceptionTable* inc_table,
 953                             AbstractCompiler* compiler,
 954                             int comp_level,
 955                             bool has_debug_info,
 956                             bool has_unsafe_access) {
 957   VM_ENTRY_MARK;
 958   nmethod* nm = NULL;
 959   {
 960     // To prevent compile queue updates.
 961     MutexLocker locker(MethodCompileQueue_lock, THREAD);

 963     // Prevent SystemDictionary::add_to_hierarchy from running
 964     // and invalidating our dependencies until we install this method.
 965     MutexLocker ml(Compile_lock);

 967     // Change in Jvmti state may invalidate compilation.
 968     if (!failing() &&
 969         ( (!jvmti_can_hotswap_or_post_breakpoint() &&
 970           JvmtiExport::can_hotswap_or_post_breakpoint()) ||
 971          (!jvmti_can_access_local_variables() &&
 972           JvmtiExport::can_access_local_variables()) ||
 973          (!jvmti_can_post_on_exceptions() &&
 974           JvmtiExport::can_post_on_exceptions()) )) {
 975       record_failure("Jvmti state change invalidated dependencies");
 976     }

 978     // Change in DTrace flags may invalidate compilation.
 979     if (!failing() &&
 980         ( (!dtrace_extended_probes() && ExtendedDTraceProbes) ||
 981           (!dtrace_method_probes() && DTraceMethodProbes) ||
 982           (!dtrace_alloc_probes() && DTraceAllocProbes) )) {
 983       record_failure("DTrace flags change invalidated dependencies");
 984     }

 986     if (!failing()) {
 987       if (log() != NULL) {
 988         // Log the dependencies which this compilation declares.
 989         dependencies()->log_all_dependencies();
 990       }

 992       // Encode the dependencies now, so we can check them right away.
 993       dependencies()->encode_content_bytes();

 995       // Check for {class loads, evolution, breakpoints, ...} during compilation
```

```
 996       validate_compile_task_dependencies(target);
 969       // Check for {class loads, evolution, breakpoints} during compilation
 970       check_for_system_dictionary_modification(target);
 997     }

 999     methodHandle method(THREAD, target->get_methodOop());

1001     if (failing()) {
1002       // While not a true deoptimization, it is a preemptive decompile.
1003       methodDataOop mdo = method()->method_data();
1004       if (mdo != NULL) {
1005         mdo->inc_decompile_count();
1006       }

1008       // All buffers in the CodeBuffer are allocated in the CodeCache.
1009       // If the code buffer is created on each compile attempt
1010       // as in C2, then it must be freed.
1011       code_buffer->free_blob();
1012       return;
1013     }

1015     assert(offsets->value(CodeOffsets::Deopt) != -1, "must have deopt entry");
1016     assert(offsets->value(CodeOffsets::Exceptions) != -1, "must have exception e

1018     nm =  nmethod::new_nmethod(method,
1019                                compile_id(),
1020                                entry_bci,
1021                                offsets,
1022                                orig_pc_offset,
1023                                debug_info(), dependencies(), code_buffer,
1024                                frame_words, oop_map_set,
1025                                handler_table, inc_table,
1026                                compiler, comp_level);

1028     // Free codeBlobs
1029     code_buffer->free_blob();

1031     // stress test 6243940 by immediately making the method
1032     // non-entrant behind the system's back. This has serious
1033     // side effects on the code cache and is not meant for
1034     // general stress testing
1035     if (nm != NULL && StressNonEntrant) {
1036       MutexLockerEx pl(Patching_lock, Mutex::_no_safepoint_check_flag);
1037       NativeJump::patch_verified_entry(nm->entry_point(), nm->verified_entry_poi
1038                   SharedRuntime::get_handle_wrong_method_stub());
1039     }

1041     if (nm == NULL) {
1042       // The CodeCache is full.  Print out warning and disable compilation.
1043       record_failure("code cache is full");
1044       {
1045         MutexUnlocker ml(Compile_lock);
1046         MutexUnlocker locker(MethodCompileQueue_lock);
1047         CompileBroker::handle_full_code_cache();
1048       }
1049     } else {
1050       NOT_PRODUCT(nm->set_has_debug_info(has_debug_info); )
1051       nm->set_has_unsafe_access(has_unsafe_access);

1053       // Record successful registration.
1054       // (Put nm into the task handle *before* publishing to the Java heap.)
1055       if (task() != NULL)  task()->set_code(nm);

1057       if (entry_bci == InvocationEntryBci) {
1058         if (TieredCompilation) {
1059           // If there is an old version we're done with it
```



```
1060           nmethod* old = method->code();
1061           if (TraceMethodReplacement && old != NULL) {
1062             ResourceMark rm;
1063             char *method_name = method->name_and_sig_as_C_string();
1064             tty->print_cr("Replacing method %s", method_name);
1065           }
1066           if (old != NULL ) {
1067             old->make_not_entrant();
1068           }
1069         }
1070         if (TraceNMethodInstalls ) {
1071           ResourceMark rm;
1072           char *method_name = method->name_and_sig_as_C_string();
1073           ttyLocker ttyl;
1074           tty->print_cr("Installing method (%d) %s ",
1075                         comp_level,
1076                         method_name);
1077         }
1078         // Allow the code to be executed
1079         method->set_code(method, nm);
1080       } else {
1081         if (TraceNMethodInstalls ) {
1082           ResourceMark rm;
1083           char *method_name = method->name_and_sig_as_C_string();
1084           ttyLocker ttyl;
1085           tty->print_cr("Installing osr method (%d) %s @ %d",
1086                         comp_level,
1087                         method_name,
1088                         entry_bci);
1089         }
1090         instanceKlass::cast(method->method_holder())->add_osr_nmethod(nm);

1092       }
1093     }
1094   }
1095   // JVMTI -- compiled method notification (must be done outside lock)
1096   if (nm != NULL) {
1097     nm->post_compiled_method_load_event();
1098   }

1100 }
```

_____***unchanged_portion_omitted_***

```
*********************************************************
   16572 Thu Aug 25 01:58:53 2011
new/src/share/vm/ci/ciEnv.hpp
*********************************************************
```

```
   1 /*
   2  * Copyright (c) 1999, 2011, Oracle and/or its affiliates. All rights reserved.
   3  * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
   4  *
   5  * This code is free software; you can redistribute it and/or modify it
   6  * under the terms of the GNU General Public License version 2 only, as
   7  * published by the Free Software Foundation.
   8  *
   9  * This code is distributed in the hope that it will be useful, but WITHOUT
  10  * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
  11  * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
  12  * version 2 for more details (a copy is included in the LICENSE file that
  13  * accompanied this code).
  14  *
  15  * You should have received a copy of the GNU General Public License version
  16  * 2 along with this work; if not, write to the Free Software Foundation,
  17  * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
  18  *
  19  * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
  20  * or visit www.oracle.com if you need additional information or have any
  21  * questions.
  22  *
  23  */

  25 #ifndef SHARE_VM_CI_CIENV_HPP
  26 #define SHARE_VM_CI_CIENV_HPP

  28 #include "ci/ciClassList.hpp"
  29 #include "ci/ciObjectFactory.hpp"
  30 #include "classfile/systemDictionary.hpp"
  31 #include "code/debugInfoRec.hpp"
  32 #include "code/dependencies.hpp"
  33 #include "code/exceptionHandlerTable.hpp"
  34 #include "compiler/oopMap.hpp"
  35 #include "runtime/thread.hpp"

  37 class CompileTask;

  39 // ciEnv
  40 //
  41 // This class is the top level broker for requests from the compiler
  42 // to the VM.
  43 class ciEnv : StackObj {
  44   CI_PACKAGE_ACCESS_TO

  46   friend class CompileBroker;
  47   friend class Dependencies;  // for get_object, during logging

  49 private:
  50   Arena*          _arena;       // Alias for _ciEnv_arena except in init_shared
  51   Arena           _ciEnv_arena;
  52   int             _system_dictionary_modification_counter;
  53   ciObjectFactory* _factory;
  54   OopRecorder*    _oop_recorder;
  55   DebugInformationRecorder* _debug_info;
  56   Dependencies*   _dependencies;
  57   const char*     _failure_reason;
  58   int             _compilable;
  59   bool            _break_at_compile;
  60   int             _num_inlined_bytecodes;
  61   CompileTask*    _task;          // faster access to CompilerThread::task
  62   CompileLog*     _log;           // faster access to CompilerThread::log
```



```
  63   void*           _compiler_data;  // compiler-specific stuff, if any

  65   char* _name_buffer;
  66   int   _name_buffer_len;

  68   // Cache Jvmti state
  69   bool  _jvmti_can_hotswap_or_post_breakpoint;
  70   bool  _jvmti_can_access_local_variables;
  71   bool  _jvmti_can_post_on_exceptions;

  73   // Cache DTrace flags
  74   bool  _dtrace_extended_probes;
  75   bool  _dtrace_monitor_probes;
  76   bool  _dtrace_method_probes;
  77   bool  _dtrace_alloc_probes;

  79   // Distinguished instances of certain ciObjects..
  80   static ciObject*              _null_object_instance;
  81   static ciMethodKlass*         _method_klass_instance;
  82   static ciKlassKlass*          _klass_klass_instance;
  83   static ciInstanceKlassKlass*  _instance_klass_klass_instance;
  84   static ciTypeArrayKlassKlass* _type_array_klass_klass_instance;
  85   static ciObjArrayKlassKlass*  _obj_array_klass_klass_instance;

  87 #define WK_KLASS_DECL(name, ignore_s, ignore_o) static ciInstanceKlass* _##name;
  88   WK_KLASSES_DO(WK_KLASS_DECL)
  89 #undef WK_KLASS_DECL

  91   static ciSymbol*        _unloaded_cisymbol;
  92   static ciInstanceKlass* _unloaded_ciinstance_klass;
  93   static ciObjArrayKlass* _unloaded_ciobjarrayklass;

  95   static jobject _ArrayIndexOutOfBoundsException_handle;
  96   static jobject _ArrayStoreException_handle;
  97   static jobject _ClassCastException_handle;

  99   ciInstance* _NullPointerException_instance;
 100   ciInstance* _ArithmeticException_instance;
 101   ciInstance* _ArrayIndexOutOfBoundsException_instance;
 102   ciInstance* _ArrayStoreException_instance;
 103   ciInstance* _ClassCastException_instance;

 105   ciInstance* _the_null_string;      // The Java string "null"
 106   ciInstance* _the_min_jint_string; // The Java string "-2147483648"

 108   // Look up a klass by name from a particular class loader (the accessor's).
 109   // If require_local, result must be defined in that class loader, or NULL.
 110   // If !require_local, a result from remote class loader may be reported,
 111   // if sufficient class loader constraints exist such that initiating
 112   // a class loading request from the given loader is bound to return
 113   // the class defined in the remote loader (or throw an error).
 114   //
 115   // Return an unloaded klass if !require_local and no class at all is found.
 116   //
 117   // The CI treats a klass as loaded if it is consistently defined in
 118   // another loader, even if it hasn't yet been loaded in all loaders
 119   // that could potentially see it via delegation.
 120   ciKlass* get_klass_by_name(ciKlass* accessing_klass,
 121                              ciSymbol* klass_name,
 122                              bool require_local);

 124   // Constant pool access.
 125   ciKlass*   get_klass_by_index(constantPoolHandle cpool,
 126                                 int klass_index,
 127                                 bool& is_accessible,
 128                                 ciInstanceKlass* loading_klass);
```

```
129   ciConstant get_constant_by_index(constantPoolHandle cpool,
130                                    int pool_index, int cache_index,
131                                    ciInstanceKlass* accessor);
132   ciField*   get_field_by_index(ciInstanceKlass* loading_klass,
133                                 int field_index);
134   ciMethod*  get_method_by_index(constantPoolHandle cpool,
135                                  int method_index, Bytecodes::Code bc,
136                                  ciInstanceKlass* loading_klass);
137
138   // Implementation methods for loading and constant pool access.
139   ciKlass* get_klass_by_name_impl(ciKlass* accessing_klass,
140                                   constantPoolHandle cpool,
141                                   ciSymbol* klass_name,
142                                   bool require_local);
143   ciKlass*   get_klass_by_index_impl(constantPoolHandle cpool,
144                                      int klass_index,
145                                      bool& is_accessible,
146                                      ciInstanceKlass* loading_klass);
147   ciConstant get_constant_by_index_impl(constantPoolHandle cpool,
148                                         int pool_index, int cache_index,
149                                         ciInstanceKlass* loading_klass);
150   ciField*   get_field_by_index_impl(ciInstanceKlass* loading_klass,
151                                      int field_index);
152   ciMethod*  get_method_by_index_impl(constantPoolHandle cpool,
153                                       int method_index, Bytecodes::Code bc,
154                                       ciInstanceKlass* loading_klass);
155   ciMethod*  get_fake_invokedynamic_method_impl(constantPoolHandle cpool,
156                                                 int index, Bytecodes::Code bc);
157
158   // Helper methods
159   bool       check_klass_accessibility(ciKlass* accessing_klass,
160                                        klassOop resolved_klassOop);
161   methodOop  lookup_method(instanceKlass*  accessor,
162                            instanceKlass*  holder,
163                            Symbol*         name,
164                            Symbol*         sig,
165                            Bytecodes::Code bc);
166
167   // Get a ciObject from the object factory.  Ensures uniqueness
168   // of ciObjects.
169   ciObject* get_object(oop o) {
170     if (o == NULL) {
171       return _null_object_instance;
172     } else {
173       return _factory->get(o);
174     }
175   }
176
177   ciSymbol* get_symbol(Symbol* o) {
178     if (o == NULL) {
179       ShouldNotReachHere();
180       return NULL;
181     } else {
182       return _factory->get_symbol(o);
183     }
184   }
185
186   ciMethod* get_method_from_handle(jobject method);
187
188   ciInstance* get_or_create_exception(jobject& handle, Symbol* name);
189
190   // Get a ciMethod representing either an unfound method or
191   // a method with an unloaded holder.  Ensures uniqueness of
192   // the result.
193   ciMethod* get_unloaded_method(ciInstanceKlass* holder,
194                                 ciSymbol*        name,
```



```
195                                 ciSymbol*        signature) {
196     return _factory->get_unloaded_method(holder, name, signature);
197   }
198
199   // Get a ciKlass representing an unloaded klass.
200   // Ensures uniqueness of the result.
201   ciKlass* get_unloaded_klass(ciKlass* accessing_klass,
202                               ciSymbol* name) {
203     return _factory->get_unloaded_klass(accessing_klass, name, true);
204   }
205
206   // Get a ciKlass representing an unloaded klass mirror.
207   // Result is not necessarily unique, but will be unloaded.
208   ciInstance* get_unloaded_klass_mirror(ciKlass* type) {
209     return _factory->get_unloaded_klass_mirror(type);
210   }
211
212   // Get a ciInstance representing an unresolved method handle constant.
213   ciInstance* get_unloaded_method_handle_constant(ciKlass*  holder,
214                                                   ciSymbol* name,
215                                                   ciSymbol* signature,
216                                                   int       ref_kind) {
217     return _factory->get_unloaded_method_handle_constant(holder, name, signature
218   }
219
220   // Get a ciInstance representing an unresolved method type constant.
221   ciInstance* get_unloaded_method_type_constant(ciSymbol* signature) {
222     return _factory->get_unloaded_method_type_constant(signature);
223   }
224
225   // See if we already have an unloaded klass for the given name
226   // or return NULL if not.
227   ciKlass *check_get_unloaded_klass(ciKlass* accessing_klass, ciSymbol* name) {
228     return _factory->get_unloaded_klass(accessing_klass, name, false);
229   }
230
231   // Get a ciReturnAddress corresponding to the given bci.
232   // Ensures uniqueness of the result.
233   ciReturnAddress* get_return_address(int bci) {
234     return _factory->get_return_address(bci);
235   }
236
237   // Get a ciMethodData representing the methodData for a method
238   // with none.
239   ciMethodData* get_empty_methodData() {
240     return _factory->get_empty_methodData();
241   }
242
243   // General utility : get a buffer of some required length.
244   // Used in symbol creation.
245   char* name_buffer(int req_len);
246
247   // Is this thread currently in the VM state?
248   static bool is_in_vm();
249
250   // Helper routine for determining the validity of a compilation with
251   // respect to method dependencies (e.g. concurrent class loading).
252   void validate_compile_task_dependencies(ciMethod* target);
250   // Helper routine for determining the validity of a compilation
251   // with respect to concurrent class loading.
252   void check_for_system_dictionary_modification(ciMethod* target);
253
254 public:
255   enum {
256     MethodCompilable,
257     MethodCompilable_not_at_tier,
```

```
258     MethodCompilable_never
259   };
260
261   ciEnv(CompileTask* task, int system_dictionary_modification_counter);
262   // Used only during initialization of the ci
263   ciEnv(Arena* arena);
264   ~ciEnv();
265
266   OopRecorder* oop_recorder() { return _oop_recorder; }
267   void set_oop_recorder(OopRecorder* r) { _oop_recorder = r; }
268
269   DebugInformationRecorder* debug_info() { return _debug_info; }
270   void set_debug_info(DebugInformationRecorder* i) { _debug_info = i; }
271
272   Dependencies* dependencies() { return _dependencies; }
273   void set_dependencies(Dependencies* d) { _dependencies = d; }
274
275   // This is true if the compilation is not going to produce code.
276   // (It is reasonable to retry failed compilations.)
277   bool failing() { return _failure_reason != NULL; }
278
279   // Reason this compilation is failing, such as "too many basic blocks".
280   const char* failure_reason() { return _failure_reason; }
281
282   // Return state of appropriate compilability
283   int compilable() { return _compilable; }
284
285   bool break_at_compile() { return _break_at_compile; }
286   void set_break_at_compile(bool z) { _break_at_compile = z; }
287
288   // Cache Jvmti state
289   void  cache_jvmti_state();
290   bool  jvmti_can_hotswap_or_post_breakpoint() const { return _jvmti_can_hotswap
291   bool  jvmti_can_access_local_variables()     const { return _jvmti_can_access_
292   bool  jvmti_can_post_on_exceptions()         const { return _jvmti_can_post_on
293
294   // Cache DTrace flags
295   void  cache_dtrace_flags();
296   bool  dtrace_extended_probes() const { return _dtrace_extended_probes; }
297   bool  dtrace_monitor_probes()  const { return _dtrace_monitor_probes; }
298   bool  dtrace_method_probes()   const { return _dtrace_method_probes; }
299   bool  dtrace_alloc_probes()    const { return _dtrace_alloc_probes; }
300
301   // The compiler task which has created this env.
302   // May be useful to find out compile_id, comp_level, etc.
303   CompileTask* task() { return _task; }
304   // Handy forwards to the task:
305   int comp_level();   // task()->comp_level()
306   uint compile_id();  // task()->compile_id()
307
308   // Register the result of a compilation.
309   void register_method(ciMethod*                target,
310                        int                      entry_bci,
311                        CodeOffsets*             offsets,
312                        int                      orig_pc_offset,
313                        CodeBuffer*              code_buffer,
314                        int                      frame_words,
315                        OopMapSet*               oop_map_set,
316                        ExceptionHandlerTable*   handler_table,
317                        ImplicitExceptionTable*  inc_table,
318                        AbstractCompiler*        compiler,
319                        int                      comp_level,
320                        bool                     has_debug_info = true,
321                        bool                     has_unsafe_access = false);
```



```
324   // Access to certain well known ciObjects.
325 #define WK_KLASS_FUNC(name, ignore_s, ignore_o) \
326   ciInstanceKlass* name() { \
327     return _##name;\
328   }
329   WK_KLASSES_DO(WK_KLASS_FUNC)
330 #undef WK_KLASS_FUNC
331
332   ciInstance* NullPointerException_instance() {
333     assert(_NullPointerException_instance != NULL, "initialization problem");
334     return _NullPointerException_instance;
335   }
336   ciInstance* ArithmeticException_instance() {
337     assert(_ArithmeticException_instance != NULL, "initialization problem");
338     return _ArithmeticException_instance;
339   }
340
341   // Lazy constructors:
342   ciInstance* ArrayIndexOutOfBoundsException_instance();
343   ciInstance* ArrayStoreException_instance();
344   ciInstance* ClassCastException_instance();
345
346   ciInstance* the_null_string();
347   ciInstance* the_min_jint_string();
348
349   static ciSymbol* unloaded_cisymbol() {
350     return _unloaded_cisymbol;
351   }
352   static ciObjArrayKlass* unloaded_ciobjarrayklass() {
353     return _unloaded_ciobjarrayklass;
354   }
355   static ciInstanceKlass* unloaded_ciinstance_klass() {
356     return _unloaded_ciinstance_klass;
357   }
358
359   ciKlass*  find_system_klass(ciSymbol* klass_name);
360   // Note:  To find a class from its name string, use ciSymbol::make,
361   // but consider adding to vmSymbols.hpp instead.
362
363   // Use this to make a holder for non-perm compile time constants.
364   // The resulting array is guaranteed to satisfy "can_be_constant".
365   ciArray*  make_system_array(GrowableArray<ciObject*>* objects);
366
367   // converts the ciKlass* representing the holder of a method into a
368   // ciInstanceKlass*.  This is needed since the holder of a method in
369   // the bytecodes could be an array type.  Basically this converts
370   // array types into java/lang/Object and other types stay as they are.
371   static ciInstanceKlass* get_instance_klass_for_declared_method_holder(ciKlass*
372
373   // Return the machine-level offset of o, which must be an element of a.
374   // This may be used to form constant-loading expressions in lieu of simpler en
375   int       array_element_offset_in_bytes(ciArray* a, ciObject* o);
376
377   // Access to the compile-lifetime allocation arena.
378   Arena*    arena() { return _arena; }
379
380   // What is the current compilation environment?
381   static ciEnv* current() { return CompilerThread::current()->env(); }
382
383   // Overload with current thread argument
384   static ciEnv* current(CompilerThread *thread) { return thread->env(); }
385
386   // Per-compiler data.  (Used by C2 to publish the Compile* pointer.)
387   void* compiler_data() { return _compiler_data; }
388   void set_compiler_data(void* x) { _compiler_data = x; }
```

```
 390   // Notice that a method has been inlined in the current compile;
 391   // used only for statistics.
 392   void notice_inlined_method(ciMethod* method);

 394   // Total number of bytecodes in inlined methods in this compile
 395   int num_inlined_bytecodes() const;

 397   // Output stream for logging compilation info.
 398   CompileLog* log() { return _log; }
 399   void set_log(CompileLog* log) { _log = log; }

 401   // Check for changes to the system dictionary during compilation
 402   bool system_dictionary_modification_counter_changed();

 404   void record_failure(const char* reason);
 405   void record_method_not_compilable(const char* reason, bool all_tiers = true);
 406   void record_out_of_memory_failure();
 407 };
```

*unchanged_portion_omitted_*

```
**********************************************************
   58619 Thu Aug 25 01:58:54 2011
new/src/share/vm/code/dependencies.cpp
**********************************************************
```

_____*unchanged_portion_omitted_*

```
 116 void Dependencies::assert_call_site_target_value(ciCallSite* call_site, ciMethod
 117   check_ctxk(call_site->klass());
 118   assert_common_2(call_site_target_value, call_site, method_handle);
 116 void Dependencies::assert_call_site_target_value(ciKlass* ctxk, ciCallSite* call
 117   check_ctxk(ctxk);
 118   assert_common_3(call_site_target_value, ctxk, call_site, method_handle);
 119 }
```

_____*unchanged_portion_omitted_*

```
 138 void Dependencies::assert_common_1(DepType dept, ciObject* x) {
 138 void Dependencies::assert_common_1(Dependencies::DepType dept, ciObject* x) {
 139   assert(dep_args(dept) == 1, "sanity");
 140   log_dependency(dept, x);
 141   GrowableArray<ciObject*>* deps = _deps[dept];

 143   // see if the same (or a similar) dep is already recorded
 144   if (note_dep_seen(dept, x)) {
 145     assert(deps->find(x) >= 0, "sanity");
 146   } else {
 147     deps->append(x);
 148   }
 149 }

 151 void Dependencies::assert_common_2(DepType dept,
 152                                    ciObject* x0, ciObject* x1) {
 151 void Dependencies::assert_common_2(Dependencies::DepType dept,
 152                                    ciKlass* ctxk, ciObject* x) {
 153   assert(dep_context_arg(dept) == 0, "sanity");
 153   assert(dep_args(dept) == 2, "sanity");
 154   log_dependency(dept, x0, x1);
 155   log_dependency(dept, ctxk, x);
 155   GrowableArray<ciObject*>* deps = _deps[dept];

 157   // see if the same (or a similar) dep is already recorded
 158   bool has_ctxk = has_explicit_context_arg(dept);
 159   if (has_ctxk) {
 160     assert(dep_context_arg(dept) == 0, "sanity");
 161     if (note_dep_seen(dept, x1)) {
 162       // look in this bucket for redundant assertions
 163       const int stride = 2;
 164       for (int i = deps->length(); (i -= stride) >= 0; ) {
 165         ciObject* y1 = deps->at(i+1);
 166         if (x1 == y1) {  // same subject; check the context
 167           if (maybe_merge_ctxk(deps, i+0, x0->as_klass())) {
 168             return;
 169           }
 170         }
 171       }
 172     }
 173   } else {
 174     assert(dep_implicit_context_arg(dept) == 0, "sanity");
 175     if (note_dep_seen(dept, x0) && note_dep_seen(dept, x1)) {
 159   if (note_dep_seen(dept, x)) {
 176       // look in this bucket for redundant assertions
 177       const int stride = 2;
 178       for (int i = deps->length(); (i -= stride) >= 0; ) {
 179         ciObject* y0 = deps->at(i+0);
 180         ciObject* y1 = deps->at(i+1);
 181         if (x0 == y0 && x1 == y1) {
 163       ciObject* x1 = deps->at(i+1);
```



```
 164       if (x == x1) {  // same subject; check the context
 165         if (maybe_merge_ctxk(deps, i+0, ctxk)) {
 182           return;
 183         }
 184       }
 185     }
 186   }

 188   // append the assertion in the correct bucket:
 189   deps->append(x0);
 190   deps->append(x1);
 173   deps->append(ctxk);
 174   deps->append(x);
 191 }

 193 void Dependencies::assert_common_3(DepType dept,
 177 void Dependencies::assert_common_3(Dependencies::DepType dept,
 194                                    ciKlass* ctxk, ciObject* x, ciObject* x2) {
 195   assert(dep_context_arg(dept) == 0, "sanity");
 196   assert(dep_args(dept) == 3, "sanity");
 197   log_dependency(dept, ctxk, x, x2);
 198   GrowableArray<ciObject*>* deps = _deps[dept];

 200   // try to normalize an unordered pair:
 201   bool swap = false;
 202   switch (dept) {
 203   case abstract_with_exclusive_concrete_subtypes_2:
 204     swap = (x->ident() > x2->ident() && x != ctxk);
 205     break;
 206   case exclusive_concrete_methods_2:
 207     swap = (x->ident() > x2->ident() && x->as_method()->holder() != ctxk);
 208     break;
 209   }
 210   if (swap) { ciObject* t = x; x = x2; x2 = t; }

 212   // see if the same (or a similar) dep is already recorded
 213   if (note_dep_seen(dept, x) && note_dep_seen(dept, x2)) {
 214     // look in this bucket for redundant assertions
 215     const int stride = 3;
 216     for (int i = deps->length(); (i -= stride) >= 0; ) {
 217       ciObject* y  = deps->at(i+1);
 218       ciObject* y2 = deps->at(i+2);
 219       if (x == y && x2 == y2) {  // same subjects; check the context
 220         if (maybe_merge_ctxk(deps, i+0, ctxk)) {
 221           return;
 222         }
 223       }
 224     }
 225   }
 226   // append the assertion in the correct bucket:
 227   deps->append(ctxk);
 228   deps->append(x);
 229   deps->append(x2);
 230 }
```

_____*unchanged_portion_omitted_*

```
 369 int Dependencies::_dep_args[TYPE_LIMIT] = {
 370   -1,// end_marker
 371   1, // evol_method m
 372   1, // leaf_type ctxk
 373   2, // abstract_with_unique_concrete_subtype ctxk, k
 374   1, // abstract_with_no_concrete_subtype ctxk
 375   1, // concrete_with_no_concrete_subtype ctxk
 376   2, // unique_concrete_method ctxk, m
 377   3, // unique_concrete_subtypes_2 ctxk, k1, k2
 378   3, // unique_concrete_methods_2 ctxk, m1, m2
```

```
 379   1, // no_finalizable_subclasses ctxk
 380   2  // call_site_target_value call_site, method_handle
 364   3  // call_site_target_value ctxk, call_site, method_handle
 381 };
_____unchanged_portion_omitted_

 393 void Dependencies::check_valid_dependency_type(DepType dept) {
 394   guarantee(FIRST_TYPE <= dept && dept < TYPE_LIMIT, err_msg("invalid dependency
 378   for (int deptv = (int) FIRST_TYPE; deptv < (int) TYPE_LIMIT; deptv++) {
 379     if (dept == ((DepType) deptv))  return;
 380   }
 381   ShouldNotReachHere();
 395 }
_____unchanged_portion_omitted_
 584 #endif //ASSERT

 586 bool Dependencies::DepStream::next() {
 587   assert(_type != end_marker, "already at end");
 588   if (_bytes.position() == 0 && _code != NULL
 589       && _code->dependencies_size() == 0) {
 590     // Method has no dependencies at all.
 591     return false;
 592   }
 593   int code_byte = (_bytes.read_byte() & 0xFF);
 594   if (code_byte == end_marker) {
 595     DEBUG_ONLY(_type = end_marker);
 596     return false;
 597   } else {
 598     int ctxk_bit = (code_byte & Dependencies::default_context_type_bit);
 599     code_byte -= ctxk_bit;
 600     DepType dept = (DepType)code_byte;
 601     _type = dept;
 602     Dependencies::check_valid_dependency_type(dept);
 589     guarantee((dept - FIRST_TYPE) < (TYPE_LIMIT - FIRST_TYPE),
 590               "bad dependency type tag");
 603     int stride = _dep_args[dept];
 604     assert(stride == dep_args(dept), "sanity");
 605     int skipj = -1;
 606     if (ctxk_bit != 0) {
 607       skipj = 0;  // currently the only context argument is at zero
 608       assert(skipj == dep_context_arg(dept), "zero arg always ctxk");
 609     }
 610     for (int j = 0; j < stride; j++) {
 611       _xi[j] = (j == skipj)? 0: _bytes.read_int();
 612     }
 613     DEBUG_ONLY(_xi[stride] = -1);   // help detect overruns
 614     return true;
 615   }
 616 }
_____unchanged_portion_omitted_

 628 klassOop Dependencies::DepStream::context_type() {
 629   assert(must_be_in_vm(), "raw oops here");

 631   // Most dependencies have an explicit context type argument.
 632   {
 633     int ctxkj = dep_context_arg(_type);  // -1 if no explicit context arg
 634     if (ctxkj >= 0) {
 635       oop k = argument(ctxkj);
 618   int ctxkj = dep_context_arg(_type);  // -1 if no context arg
 619   if (ctxkj < 0) {
 620     return NULL;          // for example, evol_method
 621   } else {
 622     oop k = recorded_oop_at(_xi[ctxkj]);
 636       if (k != NULL) {       // context type was not compressed away
 637         assert(k->is_klass(), "type check");
```



```
 638         return (klassOop) k;
 639       }
 640       // recompute "default" context type
 641       return ctxk_encoded_as_null(_type, argument(ctxkj+1));
 626     } else {               // recompute "default" context type
 627       return ctxk_encoded_as_null(_type, recorded_oop_at(_xi[ctxkj+1]));
 642     }
 643   }

 645   // Some dependencies are using the klass of the first object
 646   // argument as implicit context type (e.g. call_site_target_value).
 647   {
 648     int ctxkj = dep_implicit_context_arg(_type);
 649     if (ctxkj >= 0) {
 650       oop k = argument(ctxkj)->klass();
 651       assert(k->is_klass(), "type check");
 652       return (klassOop) k;
 653     }
 654   }

 656   // And some dependencies don't have a context type at all,
 657   // e.g. evol_method.
 658   return NULL;
 659 #endif /* ! codereview */
 660 }

 662 /// Checking dependencies:

 664 // This hierarchy walker inspects subtypes of a given type,
 665 // trying to find a "bad" class which breaks a dependency.
 666 // Such a class is called a "witness" to the broken dependency.
 667 // While searching around, we ignore "participants", which
 668 // are already known to the dependency.
 669 class ClassHierarchyWalker {
 670  public:
 671   enum { PARTICIPANT_LIMIT = 3 };

 673  private:
 674   // optional method descriptor to check for:
 675   Symbol* _name;
 676   Symbol* _signature;

 678   // special classes which are not allowed to be witnesses:
 679   klassOop  _participants[PARTICIPANT_LIMIT+1];
 680   int       _num_participants;

 682   // cache of method lookups
 683   methodOop _found_methods[PARTICIPANT_LIMIT+1];

 685   // if non-zero, tells how many witnesses to convert to participants
 686   int       _record_witnesses;

 688   void initialize(klassOop participant) {
 689     _record_witnesses = 0;
 690     _participants[0]  = participant;
 691     _found_methods[0] = NULL;
 692     _num_participants = 0;
 693     if (participant != NULL) {
 694       // Terminating NULL.
 695       _participants[1] = NULL;
 696       _found_methods[1] = NULL;
 697       _num_participants = 1;
 698     }
 699   }

 701   void initialize_from_method(methodOop m) {
```

```
702     assert(m != NULL && m->is_method(), "sanity");
703     _name      = m->name();
704     _signature = m->signature();
705   }
706
707   public:
708    // The walker is initialized to recognize certain methods and/or types
709    // as friendly participants.
710    ClassHierarchyWalker(klassOop participant, methodOop m) {
711      initialize_from_method(m);
712      initialize(participant);
713    }
714    ClassHierarchyWalker(methodOop m) {
715      initialize_from_method(m);
716      initialize(NULL);
717    }
718    ClassHierarchyWalker(klassOop participant = NULL) {
719      _name      = NULL;
720      _signature = NULL;
721      initialize(participant);
722    }
723
724    // This is common code for two searches:  One for concrete subtypes,
725    // the other for concrete method implementations and overrides.
726    bool doing_subtype_search() {
727      return _name == NULL;
728    }
729
730    int num_participants() { return _num_participants; }
731    klassOop participant(int n) {
732      assert((uint)n <= (uint)_num_participants, "oob");
733      return _participants[n];
734    }
735
736    // Note:  If n==num_participants, returns NULL.
737    methodOop found_method(int n) {
738      assert((uint)n <= (uint)_num_participants, "oob");
739      methodOop fm = _found_methods[n];
740      assert(n == _num_participants || fm != NULL, "proper usage");
741      assert(fm == NULL || fm->method_holder() == _participants[n], "sanity");
742      return fm;
743    }
744
745 #ifdef ASSERT
746    // Assert that m is inherited into ctxk, without intervening overrides.
747    // (May return true even if this is not true, in corner cases where we punt.)
748    bool check_method_context(klassOop ctxk, methodOop m) {
749      if (m->method_holder() == ctxk)
750        return true;  // Quick win.
751      if (m->is_private())
752        return false; // Quick lose.  Should not happen.
753      if (!(m->is_public() || m->is_protected()))
754        // The override story is complex when packages get involved.
755        return true;  // Must punt the assertion to true.
756      Klass* k = Klass::cast(ctxk);
757      methodOop lm = k->lookup_method(m->name(), m->signature());
758      if (lm == NULL && k->oop_is_instance()) {
759        // It might be an abstract interface method, devoid of mirandas.
760        lm = ((instanceKlass*)k)->lookup_method_in_all_interfaces(m->name(),
761                                                                  m->signature());
762      }
763      if (lm == m)
764        // Method m is inherited into ctxk.
765        return true;
766      if (lm != NULL) {
767        if (!(lm->is_public() || lm->is_protected()))
```



```
768         // Method is [package-]private, so the override story is complex.
769         return true;  // Must punt the assertion to true.
770       if (   !Dependencies::is_concrete_method(lm)
771           && !Dependencies::is_concrete_method(m)
772           && Klass::cast(lm->method_holder())->is_subtype_of(m->method_holder())
773         // Method m is overridden by lm, but both are non-concrete.
774         return true;
775      }
776      ResourceMark rm;
777      tty->print_cr("Dependency method not found in the associated context:");
778      tty->print_cr("  context = %s", Klass::cast(ctxk)->external_name());
779      tty->print(   "  method = "); m->print_short_name(tty); tty->cr();
780      if (lm != NULL) {
781        tty->print( "  found = "); lm->print_short_name(tty); tty->cr();
782      }
783      return false;
784    }
785 #endif
786
787    void add_participant(klassOop participant) {
788      assert(_num_participants + _record_witnesses < PARTICIPANT_LIMIT, "oob");
789      int np = _num_participants++;
790      _participants[np] = participant;
791      _participants[np+1] = NULL;
792      _found_methods[np+1] = NULL;
793    }
794
795    void record_witnesses(int add) {
796      if (add > PARTICIPANT_LIMIT)  add = PARTICIPANT_LIMIT;
797      assert(_num_participants + add < PARTICIPANT_LIMIT, "oob");
798      _record_witnesses = add;
799    }
800
801    bool is_witness(klassOop k) {
802      if (doing_subtype_search()) {
803        return Dependencies::is_concrete_klass(k);
804      } else {
805        methodOop m = instanceKlass::cast(k)->find_method(_name, _signature);
806        if (m == NULL || !Dependencies::is_concrete_method(m))  return false;
807        _found_methods[_num_participants] = m;
808        // Note:  If add_participant(k) is called,
809        // the method m will already be memoized for it.
810        return true;
811      }
812    }
813
814    bool is_participant(klassOop k) {
815      if (k == _participants[0]) {
816        return true;
817      } else if (_num_participants <= 1) {
818        return false;
819      } else {
820        return in_list(k, &_participants[1]);
821      }
822    }
823    bool ignore_witness(klassOop witness) {
824      if (_record_witnesses == 0) {
825        return false;
826      } else {
827        --_record_witnesses;
828        add_participant(witness);
829        return true;
830      }
831    }
832    static bool in_list(klassOop x, klassOop* list) {
833      for (int i = 0; ; i++) {
```

```
834       klassOop y = list[i];
835       if (y == NULL)  break;
836       if (y == x)  return true;
837     }
838     return false;  // not in list
839   }

841  private:
842   // the actual search method:
843   klassOop find_witness_anywhere(klassOop context_type,
844                                  bool participants_hide_witnesses,
845                                  bool top_level_call = true);
846   // the spot-checking version:
847   klassOop find_witness_in(KlassDepChange& changes,
848                            klassOop context_type,
849                            bool participants_hide_witnesses);
850  public:
851   klassOop find_witness_subtype(klassOop context_type, KlassDepChange* changes =
852     assert(doing_subtype_search(), "must set up a subtype search");
853     // When looking for unexpected concrete types,
854     // do not look beneath expected ones.
855     const bool participants_hide_witnesses = true;
856     // CX > CC > C' is OK, even if C' is new.
857     // CX > { CC,  C' } is not OK if C' is new, and C' is the witness.
858     if (changes != NULL) {
859       return find_witness_in(*changes, context_type, participants_hide_witnesses
860     } else {
861       return find_witness_anywhere(context_type, participants_hide_witnesses);
862     }
863   }
864   klassOop find_witness_definer(klassOop context_type, KlassDepChange* changes =
865     assert(!doing_subtype_search(), "must set up a method definer search");
866     // When looking for unexpected concrete methods,
867     // look beneath expected ones, to see if there are overrides.
868     const bool participants_hide_witnesses = true;
869     // CX.m > CC.m > C'.m is not OK, if C'.m is new, and C' is the witness.
870     if (changes != NULL) {
871       return find_witness_in(*changes, context_type, !participants_hide_witnesse
872     } else {
873       return find_witness_anywhere(context_type, !participants_hide_witnesses);
874     }
875   }
876 };

878 #ifndef PRODUCT
879 static int deps_find_witness_calls = 0;
880 static int deps_find_witness_steps = 0;
881 static int deps_find_witness_recursions = 0;
882 static int deps_find_witness_singles = 0;
883 static int deps_find_witness_print = 0; // set to -1 to force a final print
884 static bool count_find_witness_calls() {
885   if (TraceDependencies || LogCompilation) {
886     int pcount = deps_find_witness_print + 1;
887     bool final_stats      = (pcount == 0);
888     bool initial_call     = (pcount == 1);
889     bool occasional_print = ((pcount & ((1<<10) - 1)) == 0);
890     if (pcount < 0)  pcount = 1; // crude overflow protection
891     deps_find_witness_print = pcount;
892     if (VerifyDependencies && initial_call) {
893       tty->print_cr("Warning:  TraceDependencies results may be inflated by Veri
894     }
895     if (occasional_print || final_stats) {
896       // Every now and then dump a little info about dependency searching.
897       if (xtty != NULL) {
898         ttyLocker ttyl;
899         xtty->elem("deps_find_witness calls='%d' steps='%d' recursions='%d' singl
```



```
900                    deps_find_witness_calls,
901                    deps_find_witness_steps,
902                    deps_find_witness_recursions,
903                    deps_find_witness_singles);
904       }
905       if (final_stats || (TraceDependencies && WizardMode)) {
906         ttyLocker ttyl;
907         tty->print_cr("Dependency check (find_witness) "
908                       "calls=%d, steps=%d (avg=%.1f), recursions=%d, singles=%d"
909                       deps_find_witness_calls,
910                       deps_find_witness_steps,
911                       (double)deps_find_witness_steps / deps_find_witness_calls,
912                       deps_find_witness_recursions,
913                       deps_find_witness_singles);
914       }
915     }
916     return true;
917   }
918   return false;
919 }
920 #else
921 #define count_find_witness_calls() (0)
922 #endif //PRODUCT


925 klassOop ClassHierarchyWalker::find_witness_in(KlassDepChange& changes,
926                                                klassOop context_type,
927                                                bool participants_hide_witnesses)
928   assert(changes.involves_context(context_type), "irrelevant dependency");
929   klassOop new_type = changes.new_type();

931   count_find_witness_calls();
932   NOT_PRODUCT(deps_find_witness_singles++);

934   // Current thread must be in VM (not native mode, as in CI):
935   assert(must_be_in_vm(), "raw oops here");
936   // Must not move the class hierarchy during this check:
937   assert_locked_or_safepoint(Compile_lock);

939   int nof_impls = instanceKlass::cast(context_type)->nof_implementors();
940   if (nof_impls > 1) {
941     // Avoid this case: *I.m > { A.m, C }; B.m > C
942     // %%% Until this is fixed more systematically, bail out.
943     // See corresponding comment in find_witness_anywhere.
944     return context_type;
945   }

947   assert(!is_participant(new_type), "only old classes are participants");
948   if (participants_hide_witnesses) {
949     // If the new type is a subtype of a participant, we are done.
950     for (int i = 0; i < num_participants(); i++) {
951       klassOop part = participant(i);
952       if (part == NULL)  continue;
953       assert(changes.involves_context(part) == Klass::cast(new_type)->is_subtype
954              "correct marking of participants, b/c new_type is unique");
955       if (changes.involves_context(part)) {
956         // new guy is protected from this check by previous participant
957         return NULL;
958       }
959     }
960   }

962   if (is_witness(new_type) &&
963       !ignore_witness(new_type)) {
964     return new_type;
965   }
```

```
967   return NULL;
968 }


971 // Walk hierarchy under a context type, looking for unexpected types.
972 // Do not report participant types, and recursively walk beneath
973 // them only if participants_hide_witnesses is false.
974 // If top_level_call is false, skip testing the context type,
975 // because the caller has already considered it.
976 klassOop ClassHierarchyWalker::find_witness_anywhere(klassOop context_type,
977                                                      bool participants_hide_witn
978                                                      bool top_level_call) {
979   // Current thread must be in VM (not native mode, as in CI):
980   assert(must_be_in_vm(), "raw oops here");
981   // Must not move the class hierarchy during this check:
982   assert_locked_or_safepoint(Compile_lock);

984   bool do_counts = count_find_witness_calls();

986   // Check the root of the sub-hierarchy first.
987   if (top_level_call) {
988     if (do_counts) {
989       NOT_PRODUCT(deps_find_witness_calls++);
990       NOT_PRODUCT(deps_find_witness_steps++);
991     }
992     if (is_participant(context_type)) {
993       if (participants_hide_witnesses)  return NULL;
994       // else fall through to search loop...
995     } else if (is_witness(context_type) && !ignore_witness(context_type)) {
996       // The context is an abstract class or interface, to start with.
997       return context_type;
998     }
999   }

1001   // Now we must check each implementor and each subclass.
1002   // Use a short worklist to avoid blowing the stack.
1003   // Each worklist entry is a *chain* of subklass siblings to process.
1004   const int CHAINMAX = 100;  // >= 1 + instanceKlass::implementors_limit
1005   Klass* chains[CHAINMAX];
1006   int    chaini = 0;  // index into worklist
1007   Klass* chain;       // scratch variable
1008 #define ADD_SUBCLASS_CHAIN(k)                     {  \
1009     assert(chaini < CHAINMAX, "oob");                \
1010     chain = instanceKlass::cast(k)->subklass();      \
1011     if (chain != NULL)  chains[chaini++] = chain;    }

1013   // Look for non-abstract subclasses.
1014   // (Note:  Interfaces do not have subclasses.)
1015   ADD_SUBCLASS_CHAIN(context_type);

1017   // If it is an interface, search its direct implementors.
1018   // (Their subclasses are additional indirect implementors.
1019   // See instanceKlass::add_implementor.)
1020   // (Note:  nof_implementors is always zero for non-interfaces.)
1021   int nof_impls = instanceKlass::cast(context_type)->nof_implementors();
1022   if (nof_impls > 1) {
1023     // Avoid this case: *I.m > { A.m, C }; B.m > C
1024     // Here, I.m has 2 concrete implementations, but m appears unique
1025     // as A.m, because the search misses B.m when checking C.
1026     // The inherited method B.m was getting missed by the walker
1027     // when interface 'I' was the starting point.
1028     // %%% Until this is fixed more systematically, bail out.
1029     // (Old CHA had the same limitation.)
1030     return context_type;
1031   }
```



```
1032   for (int i = 0; i < nof_impls; i++) {
1033     klassOop impl = instanceKlass::cast(context_type)->implementor(i);
1034     if (impl == NULL) {
1035       // implementors array overflowed => no exact info.
1036       return context_type;  // report an inexact witness to this sad affair
1037     }
1038     if (do_counts)
1039       { NOT_PRODUCT(deps_find_witness_steps++); }
1040     if (is_participant(impl)) {
1041       if (participants_hide_witnesses)  continue;
1042       // else fall through to process this guy's subclasses
1043     } else if (is_witness(impl) && !ignore_witness(impl)) {
1044       return impl;
1045     }
1046     ADD_SUBCLASS_CHAIN(impl);
1047   }

1049   // Recursively process each non-trivial sibling chain.
1050   while (chaini > 0) {
1051     Klass* chain = chains[--chaini];
1052     for (Klass* subk = chain; subk != NULL; subk = subk->next_sibling()) {
1053       klassOop sub = subk->as_klassOop();
1054       if (do_counts) { NOT_PRODUCT(deps_find_witness_steps++); }
1055       if (is_participant(sub)) {
1056         if (participants_hide_witnesses)  continue;
1057         // else fall through to process this guy's subclasses
1058       } else if (is_witness(sub) && !ignore_witness(sub)) {
1059         return sub;
1060       }
1061       if (chaini < (VerifyDependencies? 2: CHAINMAX)) {
1062         // Fast path.  (Partially disabled if VerifyDependencies.)
1063         ADD_SUBCLASS_CHAIN(sub);
1064       } else {
1065         // Worklist overflow.  Do a recursive call.  Should be rare.
1066         // The recursive call will have its own worklist, of course.
1067         // (Note that sub has already been tested, so that there is
1068         // no need for the recursive call to re-test.  That's handy,
1069         // since the recursive call sees sub as the context_type.)
1070         if (do_counts) { NOT_PRODUCT(deps_find_witness_recursions++); }
1071         klassOop witness = find_witness_anywhere(sub,
1072                                                  participants_hide_witnesses,
1073                                                  /*top_level_call=*/ false);
1074         if (witness != NULL)  return witness;
1075       }
1076     }
1077   }

1079   // No witness found.  The dependency remains unbroken.
1080   return NULL;
1081 #undef ADD_SUBCLASS_CHAIN
1082 }


1085 bool Dependencies::is_concrete_klass(klassOop k) {
1086   if (Klass::cast(k)->is_abstract())  return false;
1087   // %%% We could treat classes which are concrete but
1088   // have not yet been instantiated as virtually abstract.
1089   // This would require a deoptimization barrier on first instantiation.
1090   //if (k->is_not_instantiated())  return false;
1091   return true;
1092 }

1094 bool Dependencies::is_concrete_method(methodOop m) {
1095   if (m->is_abstract())  return false;
1096   // %%% We could treat unexecuted methods as virtually abstract also.
1097   // This would require a deoptimization barrier on first execution.
```

```
1098   return !m->is_abstract();
1099 }


1102 Klass* Dependencies::find_finalizable_subclass(Klass* k) {
1103   if (k->is_interface())  return NULL;
1104   if (k->has_finalizer()) return k;
1105   k = k->subklass();
1106   while (k != NULL) {
1107     Klass* result = find_finalizable_subclass(k);
1108     if (result != NULL) return result;
1109     k = k->next_sibling();
1110   }
1111   return NULL;
1112 }


1115 bool Dependencies::is_concrete_klass(ciInstanceKlass* k) {
1116   if (k->is_abstract())  return false;
1117   // We could return also false if k does not yet appear to be
1118   // instantiated, if the VM version supports this distinction also.
1119   //if (k->is_not_instantiated())  return false;
1120   return true;
1121 }

1123 bool Dependencies::is_concrete_method(ciMethod* m) {
1124   // Statics are irrelevant to virtual call sites.
1125   if (m->is_static())  return false;

1127   // We could return also false if m does not yet appear to be
1128   // executed, if the VM version supports this distinction also.
1129   return !m->is_abstract();
1130 }


1133 bool Dependencies::has_finalizable_subclass(ciInstanceKlass* k) {
1134   return k->has_finalizable_subclass();
1135 }


1138 // Any use of the contents (bytecodes) of a method must be
1139 // marked by an "evol_method" dependency, if those contents
1140 // can change.  (Note: A method is always dependent on itself.)
1141 klassOop Dependencies::check_evol_method(methodOop m) {
1142   assert(must_be_in_vm(), "raw oops here");
1143   // Did somebody do a JVMTI RedefineClasses while our backs were turned?
1144   // Or is there a now a breakpoint?
1145   // (Assumes compiled code cannot handle bkpts; change if UseFastBreakpoints.)
1146   if (m->is_old()
1147       || m->number_of_breakpoints() > 0) {
1148     return m->method_holder();
1149   } else {
1150     return NULL;
1151   }
1152 }

1154 // This is a strong assertion:  It is that the given type
1155 // has no subtypes whatever.  It is most useful for
1156 // optimizing checks on reflected types or on array types.
1157 // (Checks on types which are derived from real instances
1158 // can be optimized more strongly than this, because we
1159 // know that the checked type comes from a concrete type,
1160 // and therefore we can disregard abstract types.)
1161 klassOop Dependencies::check_leaf_type(klassOop ctxk) {
1162   assert(must_be_in_vm(), "raw oops here");
1163   assert_locked_or_safepoint(Compile_lock);
```



```
1164   instanceKlass* ctx = instanceKlass::cast(ctxk);
1165   Klass* sub = ctx->subklass();
1166   if (sub != NULL) {
1167     return sub->as_klassOop();
1168   } else if (ctx->nof_implementors() != 0) {
1169     // if it is an interface, it must be unimplemented
1170     // (if it is not an interface, nof_implementors is always zero)
1171     klassOop impl = ctx->implementor(0);
1172     return (impl != NULL)? impl: ctxk;
1173   } else {
1174     return NULL;
1175   }
1176 }

1178 // Test the assertion that conck is the only concrete subtype* of ctxk.
1179 // The type conck itself is allowed to have have further concrete subtypes.
1180 // This allows the compiler to narrow occurrences of ctxk by conck,
1181 // when dealing with the types of actual instances.
1182 klassOop Dependencies::check_abstract_with_unique_concrete_subtype(klassOop ctxk
1183                                                                    klassOop conc
1184                                                                    KlassDepChang
1185   ClassHierarchyWalker wf(conck);
1186   return wf.find_witness_subtype(ctxk, changes);
1187 }

1189 // If a non-concrete class has no concrete subtypes, it is not (yet)
1190 // instantiatable.  This can allow the compiler to make some paths go
1191 // dead, if they are gated by a test of the type.
1192 klassOop Dependencies::check_abstract_with_no_concrete_subtype(klassOop ctxk,
1193                                                                KlassDepChange* c
1194   // Find any concrete subtype, with no participants:
1195   ClassHierarchyWalker wf;
1196   return wf.find_witness_subtype(ctxk, changes);
1197 }


1200 // If a concrete class has no concrete subtypes, it can always be
1201 // exactly typed.  This allows the use of a cheaper type test.
1202 klassOop Dependencies::check_concrete_with_no_concrete_subtype(klassOop ctxk,
1203                                                                KlassDepChange* c
1204   // Find any concrete subtype, with only the ctxk as participant:
1205   ClassHierarchyWalker wf(ctxk);
1206   return wf.find_witness_subtype(ctxk, changes);
1207 }


1210 // Find the unique concrete proper subtype of ctxk, or NULL if there
1211 // is more than one concrete proper subtype.  If there are no concrete
1212 // proper subtypes, return ctxk itself, whether it is concrete or not.
1213 // The returned subtype is allowed to have have further concrete subtypes.
1214 // That is, return CC1 for CX > CC1 > CC2, but NULL for CX > { CC1, CC2 }.
1215 klassOop Dependencies::find_unique_concrete_subtype(klassOop ctxk) {
1216   ClassHierarchyWalker wf(ctxk);   // Ignore ctxk when walking.
1217   wf.record_witnesses(1);          // Record one other witness when walking.
1218   klassOop wit = wf.find_witness_subtype(ctxk);
1219   if (wit != NULL)  return NULL;   // Too many witnesses.
1220   klassOop conck = wf.participant(0);
1221   if (conck == NULL) {
1222 #ifndef PRODUCT
1223     // Make sure the dependency mechanism will pass this discovery:
1224     if (VerifyDependencies) {
1225       // Turn off dependency tracing while actually testing deps.
1226       FlagSetting fs(TraceDependencies, false);
1227       if (!Dependencies::is_concrete_klass(ctxk)) {
1228         guarantee(NULL ==
1229                   (void *)check_abstract_with_no_concrete_subtype(ctxk),
```

```
1230                   "verify dep.");
1231       } else {
1232         guarantee(NULL ==
1233                   (void *)check_concrete_with_no_concrete_subtype(ctxk),
1234                   "verify dep.");
1235       }
1236     }
1237 #endif //PRODUCT
1238     return ctxk;                  // Return ctxk as a flag for "no subtypes".
1239   } else {
1240 #ifndef PRODUCT
1241     // Make sure the dependency mechanism will pass this discovery:
1242     if (VerifyDependencies) {
1243       // Turn off dependency tracing while actually testing deps.
1244       FlagSetting fs(TraceDependencies, false);
1245       if (!Dependencies::is_concrete_klass(ctxk)) {
1246         guarantee(NULL == (void *)
1247                   check_abstract_with_unique_concrete_subtype(ctxk, conck),
1248                   "verify dep.");
1249       }
1250     }
1251 #endif //PRODUCT
1252     return conck;
1253   }
1254 }

1256 // Test the assertion that the k[12] are the only concrete subtypes of ctxk,
1257 // except possibly for further subtypes of k[12] themselves.
1258 // The context type must be abstract.  The types k1 and k2 are themselves
1259 // allowed to have further concrete subtypes.
1260 klassOop Dependencies::check_abstract_with_exclusive_concrete_subtypes(
1261                                                 klassOop ctxk,
1262                                                 klassOop k1,
1263                                                 klassOop k2,
1264                                                 KlassDepChange* changes) {
1265   ClassHierarchyWalker wf;
1266   wf.add_participant(k1);
1267   wf.add_participant(k2);
1268   return wf.find_witness_subtype(ctxk, changes);
1269 }

1271 // Search ctxk for concrete implementations.  If there are klen or fewer,
1272 // pack them into the given array and return the number.
1273 // Otherwise, return -1, meaning the given array would overflow.
1274 // (Note that a return of 0 means there are exactly no concrete subtypes.)
1275 // In this search, if ctxk is concrete, it will be reported alone.
1276 // For any type CC reported, no proper subtypes of CC will be reported.
1277 int Dependencies::find_exclusive_concrete_subtypes(klassOop ctxk,
1278                                                    int klen,
1279                                                    klassOop karray[]) {
1280   ClassHierarchyWalker wf;
1281   wf.record_witnesses(klen);
1282   klassOop wit = wf.find_witness_subtype(ctxk);
1283   if (wit != NULL)  return -1;  // Too many witnesses.
1284   int num = wf.num_participants();
1285   assert(num <= klen, "oob");
1286   // Pack the result array with the good news.
1287   for (int i = 0; i < num; i++)
1288     karray[i] = wf.participant(i);
1289 #ifndef PRODUCT
1290   // Make sure the dependency mechanism will pass this discovery:
1291   if (VerifyDependencies) {
1292     // Turn off dependency tracing while actually testing deps.
1293     FlagSetting fs(TraceDependencies, false);
1294     switch (Dependencies::is_concrete_klass(ctxk)? -1: num) {
1295     case -1: // ctxk was itself concrete
```



```
1296       guarantee(num == 1 && karray[0] == ctxk, "verify dep.");
1297       break;
1298     case 0:
1299       guarantee(NULL == (void *)check_abstract_with_no_concrete_subtype(ctxk),
1300                 "verify dep.");
1301       break;
1302     case 1:
1303       guarantee(NULL == (void *)
1304                 check_abstract_with_unique_concrete_subtype(ctxk, karray[0]),
1305                 "verify dep.");
1306       break;
1307     case 2:
1308       guarantee(NULL == (void *)
1309                 check_abstract_with_exclusive_concrete_subtypes(ctxk,
1310                                                                 karray[0],
1311                                                                 karray[1]),
1312                 "verify dep.");
1313       break;
1314     default:
1315       ShouldNotReachHere();  // klen > 2 yet supported
1316     }
1317   }
1318 #endif //PRODUCT
1319   return num;
1320 }

1322 // If a class (or interface) has a unique concrete method uniqm, return NULL.
1323 // Otherwise, return a class that contains an interfering method.
1324 klassOop Dependencies::check_unique_concrete_method(klassOop ctxk, methodOop uni
1325                                                     KlassDepChange* changes) {
1326   // Here is a missing optimization:  If uniqm->is_final(),
1327   // we don't really need to search beneath it for overrides.
1328   // This is probably not important, since we don't use dependencies
1329   // to track final methods.  (They can't be "definalized".)
1330   ClassHierarchyWalker wf(uniqm->method_holder(), uniqm);
1331   return wf.find_witness_definer(ctxk, changes);
1332 }

1334 // Find the set of all non-abstract methods under ctxk that match m.
1335 // (The method m must be defined or inherited in ctxk.)
1336 // Include m itself in the set, unless it is abstract.
1337 // If this set has exactly one element, return that element.
1338 methodOop Dependencies::find_unique_concrete_method(klassOop ctxk, methodOop m)
1339   ClassHierarchyWalker wf(m);
1340   assert(wf.check_method_context(ctxk, m), "proper context");
1341   wf.record_witnesses(1);
1342   klassOop wit = wf.find_witness_definer(ctxk);
1343   if (wit != NULL)  return NULL;  // Too many witnesses.
1344   methodOop fm = wf.found_method(0);  // Will be NULL if num_parts == 0.
1345   if (Dependencies::is_concrete_method(m)) {
1346     if (fm == NULL) {
1347       // It turns out that m was always the only implementation.
1348       fm = m;
1349     } else if (fm != m) {
1350       // Two conflicting implementations after all.
1351       // (This can happen if m is inherited into ctxk and fm overrides it.)
1352       return NULL;
1353     }
1354   }
1355 #ifndef PRODUCT
1356   // Make sure the dependency mechanism will pass this discovery:
1357   if (VerifyDependencies && fm != NULL) {
1358     guarantee(NULL == (void *)check_unique_concrete_method(ctxk, fm),
1359               "verify dep.");
1360   }
1361 #endif //PRODUCT
```

```
1362   return fm;
1363 }

1365 klassOop Dependencies::check_exclusive_concrete_methods(klassOop ctxk,
1366                                                         methodOop m1,
1367                                                         methodOop m2,
1368                                                         KlassDepChange* changes)
1369   ClassHierarchyWalker wf(m1);
1370   wf.add_participant(m1->method_holder());
1371   wf.add_participant(m2->method_holder());
1372   return wf.find_witness_definer(ctxk, changes);
1373 }

1375 // Find the set of all non-abstract methods under ctxk that match m[0].
1376 // (The method m[0] must be defined or inherited in ctxk.)
1377 // Include m itself in the set, unless it is abstract.
1378 // Fill the given array m[0..(mlen-1)] with this set, and return the length.
1379 // (The length may be zero if no concrete methods are found anywhere.)
1380 // If there are too many concrete methods to fit in marray, return -1.
1381 int Dependencies::find_exclusive_concrete_methods(klassOop ctxk,
1382                                                   int mlen,
1383                                                   methodOop marray[]) {
1384   methodOop m0 = marray[0];
1385   ClassHierarchyWalker wf(m0);
1386   assert(wf.check_method_context(ctxk, m0), "proper context");
1387   wf.record_witnesses(mlen);
1388   bool participants_hide_witnesses = true;
1389   klassOop wit = wf.find_witness_definer(ctxk);
1390   if (wit != NULL)  return -1;  // Too many witnesses.
1391   int num = wf.num_participants();
1392   assert(num <= mlen, "oob");
1393   // Keep track of whether m is also part of the result set.
1394   int mfill = 0;
1395   assert(marray[mfill] == m0, "sanity");
1396   if (Dependencies::is_concrete_method(m0))
1397     mfill++;  // keep m0 as marray[0], the first result
1398   for (int i = 0; i < num; i++) {
1399     methodOop fm = wf.found_method(i);
1400     if (fm == m0)  continue;  // Already put this guy in the list.
1401     if (mfill == mlen) {
1402       return -1;              // Oops.  Too many methods after all!
1403     }
1404     marray[mfill++] = fm;
1405   }
1406 #ifndef PRODUCT
1407   // Make sure the dependency mechanism will pass this discovery:
1408   if (VerifyDependencies) {
1409     // Turn off dependency tracing while actually testing deps.
1410     FlagSetting fs(TraceDependencies, false);
1411     switch (mfill) {
1412     case 1:
1413       guarantee(NULL == (void *)check_unique_concrete_method(ctxk, marray[0]),
1414                 "verify dep.");
1415       break;
1416     case 2:
1417       guarantee(NULL == (void *)
1418                 check_exclusive_concrete_methods(ctxk, marray[0], marray[1]),
1419                 "verify dep.");
1420       break;
1421     default:
1422       ShouldNotReachHere();  // mlen > 2 yet supported
1423     }
1424   }
1425 #endif //PRODUCT
1426   return mfill;
1427 }
```



```
1430 klassOop Dependencies::check_has_no_finalizable_subclasses(klassOop ctxk, KlassD
1431   Klass* search_at = ctxk->klass_part();
1432   if (changes != NULL)
1433     search_at = changes->new_type()->klass_part(); // just look at the new bit
1434   Klass* result = find_finalizable_subclass(search_at);
1435   if (result == NULL) {
1436     return NULL;
1437   }
1438   return result->as_klassOop();
1439 }


1442 klassOop Dependencies::check_call_site_target_value(oop call_site, oop method_ha
 630 klassOop Dependencies::check_call_site_target_value(klassOop ctxk, oop call_site
1443   assert(call_site    ->is_a(SystemDictionary::CallSite_klass()),     "sanity");
1444   assert(method_handle->is_a(SystemDictionary::MethodHandle_klass()), "sanity");
1445   if (changes == NULL) {
1446     // Validate all CallSites
1447     if (java_lang_invoke_CallSite::target(call_site) != method_handle)
1448       return call_site->klass();  // assertion failed
 636       return ctxk;  // assertion failed
1449   } else {
1450     // Validate the given CallSite
1451     if (call_site == changes->call_site() && java_lang_invoke_CallSite::target(c
1452       assert(method_handle != changes->method_handle(), "must be");
1453       return call_site->klass();  // assertion failed
 641       return ctxk;  // assertion failed
1454     }
1455   }
 644   assert(java_lang_invoke_CallSite::target(call_site) == method_handle, "should
1456   return NULL;  // assertion still valid
1457 }
______unchanged_portion_omitted_


1513 klassOop Dependencies::DepStream::check_call_site_dependency(CallSiteDepChange*
1514   assert_locked_or_safepoint(Compile_lock);
1515   Dependencies::check_valid_dependency_type(type());

1517   klassOop witness = NULL;
1518   switch (type()) {
1519   case call_site_target_value:
1520     witness = check_call_site_target_value(argument(0), argument(1), changes);
 709     witness = check_call_site_target_value(context_type(), argument(1), argument
1521     break;
1522   default:
1523     witness = NULL;
1524     break;
1525   }
1526   trace_and_log_witness(witness);
1527   return witness;
1528 }
______unchanged_portion_omitted_
```

```
**********************************************************
   24801 Thu Aug 25 01:58:55 2011
new/src/share/vm/code/dependencies.hpp
**********************************************************
```

```
/*
 * Copyright (c) 2005, 2011, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 *
 */

#ifndef SHARE_VM_CODE_DEPENDENCIES_HPP
#define SHARE_VM_CODE_DEPENDENCIES_HPP

#include "ci/ciCallSite.hpp"
#include "ci/ciKlass.hpp"
#include "ci/ciMethodHandle.hpp"
#include "classfile/systemDictionary.hpp"
#include "code/compressedStream.hpp"
#include "code/nmethod.hpp"
#include "utilities/growableArray.hpp"

//** Dependencies represent assertions (approximate invariants) within
// the runtime system, e.g. class hierarchy changes.  An example is an
// assertion that a given method is not overridden; another example is
// that a type has only one concrete subtype.  Compiled code which
// relies on such assertions must be discarded if they are overturned
// by changes in the runtime system.  We can think of these assertions
// as approximate invariants, because we expect them to be overturned
// very infrequently.  We are willing to perform expensive recovery
// operations when they are overturned.  The benefit, of course, is
// performing optimistic optimizations (!) on the object code.
//
// Changes in the class hierarchy due to dynamic linking or
// class evolution can violate dependencies.  There is enough
// indexing between classes and nmethods to make dependency
// checking reasonably efficient.

class ciEnv;
class nmethod;
class OopRecorder;
class xmlStream;
class CompileLog;
class DepChange;
class   KlassDepChange;
class   CallSiteDepChange;
class No_Safepoint_Verifier;

class Dependencies: public ResourceObj {
```



```
public:
 // Note: In the comments on dependency types, most uses of the terms
 // subtype and supertype are used in a "non-strict" or "inclusive"
 // sense, and are starred to remind the reader of this fact.
 // Strict uses of the terms use the word "proper".
 //
 // Specifically, every class is its own subtype* and supertype*.
 // (This trick is easier than continually saying things like "Y is a
 // subtype of X or X itself".)
 //
 // Sometimes we write X > Y to mean X is a proper supertype of Y.
 // The notation X > {Y, Z} means X has proper subtypes Y, Z.
 // The notation X.m > Y means that Y inherits m from X, while
 // X.m > Y.m means Y overrides X.m.  A star denotes abstractness,
 // as *I > A, meaning (abstract) interface I is a super type of A,
 // or A.*m > B.m, meaning B.m implements abstract method A.m.
 //
 // In this module, the terms "subtype" and "supertype" refer to
 // Java-level reference type conversions, as detected by
 // "instanceof" and performed by "checkcast" operations.  The method
 // Klass::is_subtype_of tests these relations.  Note that "subtype"
 // is richer than "subclass" (as tested by Klass::is_subclass_of),
 // since it takes account of relations involving interface and array
 // types.
 //
 // To avoid needless complexity, dependencies involving array types
 // are not accepted.  If you need to make an assertion about an
 // array type, make the assertion about its corresponding element
 // types.  Any assertion that might change about an array type can
 // be converted to an assertion about its element type.
 //
 // Most dependencies are evaluated over a "context type" CX, which
 // stands for the set Subtypes(CX) of every Java type that is a subtype*
 // of CX.  When the system loads a new class or interface N, it is
 // responsible for re-evaluating changed dependencies whose context
 // type now includes N, that is, all super types of N.
 //
 enum DepType {
   end_marker = 0,

   // An 'evol' dependency simply notes that the contents of the
   // method were used.  If it evolves (is replaced), the nmethod
   // must be recompiled.  No other dependencies are implied.
   evol_method,
   FIRST_TYPE = evol_method,

   // A context type CX is a leaf it if has no proper subtype.
   leaf_type,

   // An abstract class CX has exactly one concrete subtype CC.
   abstract_with_unique_concrete_subtype,

   // The type CX is purely abstract, with no concrete subtype* at all.
   abstract_with_no_concrete_subtype,

   // The concrete CX is free of concrete proper subtypes.
   concrete_with_no_concrete_subtype,

   // Given a method M1 and a context class CX, the set MM(CX, M1) of
   // "concrete matching methods" in CX of M1 is the set of every
   // concrete M2 for which it is possible to create an invokevirtual
   // or invokeinterface call site that can reach either M1 or M2.
   // That is, M1 and M2 share a name, signature, and vtable index.
   // We wish to notice when the set MM(CX, M1) is just {M1}, or
   // perhaps a set of two {M1,M2}, and issue dependencies on this.
```

```
129     // The set MM(CX, M1) can be computed by starting with any matching
130     // concrete M2 that is inherited into CX, and then walking the
131     // subtypes* of CX looking for concrete definitions.
132
133     // The parameters to this dependency are the method M1 and the
134     // context class CX.  M1 must be either inherited in CX or defined
135     // in a subtype* of CX.  It asserts that MM(CX, M1) is no greater
136     // than {M1}.
137     unique_concrete_method,       // one unique concrete method under CX
138
139     // An "exclusive" assertion concerns two methods or subtypes, and
140     // declares that there are at most two (or perhaps later N>2)
141     // specific items that jointly satisfy the restriction.
142     // We list all items explicitly rather than just giving their
143     // count, for robustness in the face of complex schema changes.
144
145     // A context class CX (which may be either abstract or concrete)
146     // has two exclusive concrete subtypes* C1, C2 if every concrete
147     // subtype* of CX is either C1 or C2.  Note that if neither C1 or C2
148     // are equal to CX, then CX itself must be abstract.  But it is
149     // also possible (for example) that C1 is CX (a concrete class)
150     // and C2 is a proper subtype of C1.
151     abstract_with_exclusive_concrete_subtypes_2,
152
153     // This dependency asserts that MM(CX, M1) is no greater than {M1,M2}.
154     exclusive_concrete_methods_2,
155
156     // This dependency asserts that no instances of class or it's
157     // subclasses require finalization registration.
158     no_finalizable_subclasses,
159
160     // This dependency asserts when the CallSite.target value changed.
161     call_site_target_value,
162
163     TYPE_LIMIT
164   };
165   enum {
166     LG2_TYPE_LIMIT = 4,  // assert(TYPE_LIMIT <= (1<<LG2_TYPE_LIMIT))
167
168     // handy categorizations of dependency types:
169     all_types           = ((1 << TYPE_LIMIT) - 1) & ((-1) << FIRST_TYPE),
170
171     non_klass_types     = (1 << call_site_target_value),
172     klass_types         = all_types & ~non_klass_types,
173
174     non_ctxk_types      = (1 << evol_method),
175     implicit_ctxk_types = (1 << call_site_target_value),
176     explicit_ctxk_types = all_types & ~(non_ctxk_types | implicit_ctxk_types),
169     all_types      = ((1<<TYPE_LIMIT)-1) & ((-1)<<FIRST_TYPE),
170     non_ctxk_types = (1<<evol_method),
171     ctxk_types     = all_types & ~non_ctxk_types,
172
178     max_arg_count = 3,   // current maximum number of arguments (incl. ctxk)
179
180     // A "context type" is a class or interface that
181     // provides context for evaluating a dependency.
182     // When present, it is one of the arguments (dep_context_arg).
183     //
184     // If a dependency does not have a context type, there is a
185     // default context, depending on the type of the dependency.
186     // This bit signals that a default context has been compressed away.
187     default_context_type_bit = (1<<LG2_TYPE_LIMIT)
188   };
189
190   static const char* dep_name(DepType dept);
191   static int         dep_args(DepType dept);
```



```
193   static bool is_klass_type(           DepType dept) { return dept_in_mask(dept,
194
195   static bool has_explicit_context_arg(DepType dept) { return dept_in_mask(dept,
196   static bool has_implicit_context_arg(DepType dept) { return dept_in_mask(dept,
197
198   static int           dep_context_arg(DepType dept) { return has_explicit_conte
199   static int  dep_implicit_context_arg(DepType dept) { return has_implicit_conte
200
187   static int  dep_context_arg(DepType dept) {
188     return dept_in_mask(dept, ctxk_types)? 0: -1;
189   }
201   static void check_valid_dependency_type(DepType dept);
202
203 private:
204   // State for writing a new set of dependencies:
205   GrowableArray<int>*       _dep_seen;  // (seen[h->ident] & (1<<dept))
206   GrowableArray<ciObject*>* _deps[TYPE_LIMIT];
207
208   static const char* _dep_name[TYPE_LIMIT];
209   static int         _dep_args[TYPE_LIMIT];
210
211   static bool dept_in_mask(DepType dept, int mask) {
212     return (int)dept >= 0 && dept < TYPE_LIMIT && ((1<<dept) & mask) != 0;
213   }
214
215   bool note_dep_seen(int dept, ciObject* x) {
216     assert(dept < BitsPerInt, "oob");
217     int x_id = x->ident();
218     assert(_dep_seen != NULL, "deps must be writable");
219     int seen = _dep_seen->at_grow(x_id, 0);
220     _dep_seen->at_put(x_id, seen | (1<<dept));
221     // return true if we've already seen dept/x
222     return (seen & (1<<dept)) != 0;
223   }
224
225   bool maybe_merge_ctxk(GrowableArray<ciObject*>* deps,
226                         int ctxk_i, ciKlass* ctxk);
227
228   void sort_all_deps();
229   size_t estimate_size_in_bytes();
230
231   // Initialize _deps, etc.
232   void initialize(ciEnv* env);
233
234   // State for making a new set of dependencies:
235   OopRecorder* _oop_recorder;
236
237   // Logging support
238   CompileLog* _log;
239
240   address  _content_bytes;  // everything but the oop references, encoded
241   size_t   _size_in_bytes;
242
243 public:
244   // Make a new empty dependencies set.
245   Dependencies(ciEnv* env) {
246     initialize(env);
247   }
248
249 private:
250   // Check for a valid context type.
251   // Enforce the restriction against array types.
252   static void check_ctxk(ciKlass* ctxk) {
253     assert(ctxk->is_instance_klass(), "java types only");
254   }
```

```
255   static void check_ctxk_concrete(ciKlass* ctxk) {
256     assert(is_concrete_klass(ctxk->as_instance_klass()), "must be concrete");
257   }
258   static void check_ctxk_abstract(ciKlass* ctxk) {
259     check_ctxk(ctxk);
260     assert(!is_concrete_klass(ctxk->as_instance_klass()), "must be abstract");
261   }

263   void assert_common_1(DepType dept, ciObject* x);
264   void assert_common_2(DepType dept, ciObject* x0, ciObject* x1);
265   void assert_common_3(DepType dept, ciKlass* ctxk, ciObject* x1, ciObject* x2);
253   void assert_common_2(DepType dept, ciKlass* ctxk, ciObject* x);
254   void assert_common_3(DepType dept, ciKlass* ctxk, ciObject* x, ciObject* x2);

267  public:
268   // Adding assertions to a new dependency set at compile time:
269   void assert_evol_method(ciMethod* m);
270   void assert_leaf_type(ciKlass* ctxk);
271   void assert_abstract_with_unique_concrete_subtype(ciKlass* ctxk, ciKlass* conc
272   void assert_abstract_with_no_concrete_subtype(ciKlass* ctxk);
273   void assert_concrete_with_no_concrete_subtype(ciKlass* ctxk);
274   void assert_unique_concrete_method(ciKlass* ctxk, ciMethod* uniqm);
275   void assert_abstract_with_exclusive_concrete_subtypes(ciKlass* ctxk, ciKlass*
276   void assert_exclusive_concrete_methods(ciKlass* ctxk, ciMethod* m1, ciMethod*
277   void assert_has_no_finalizable_subclasses(ciKlass* ctxk);
278   void assert_call_site_target_value(ciCallSite* call_site, ciMethodHandle* meth
267   void assert_call_site_target_value(ciKlass* ctxk, ciCallSite* call_site, ciMet

280   // Define whether a given method or type is concrete.
281   // These methods define the term "concrete" as used in this module.
282   // For this module, an "abstract" class is one which is non-concrete.
283   //
284   // Future optimizations may allow some classes to remain
285   // non-concrete until their first instantiation, and allow some
286   // methods to remain non-concrete until their first invocation.
287   // In that case, there would be a middle ground between concrete
288   // and abstract (as defined by the Java language and VM).
289   static bool is_concrete_klass(klassOop k);    // k is instantiable
290   static bool is_concrete_method(methodOop m);  // m is invocable
291   static Klass* find_finalizable_subclass(Klass* k);

293   // These versions of the concreteness queries work through the CI.
294   // The CI versions are allowed to skew sometimes from the VM
295   // (oop-based) versions.  The cost of such a difference is a
296   // (safely) aborted compilation, or a deoptimization, or a missed
297   // optimization opportunity.
298   //
299   // In order to prevent spurious assertions, query results must
300   // remain stable within any single ciEnv instance.  (I.e., they must
301   // not go back into the VM to get their value; they must cache the
302   // bit in the CI, either eagerly or lazily.)
303   static bool is_concrete_klass(ciInstanceKlass* k); // k appears instantiable
304   static bool is_concrete_method(ciMethod* m);      // m appears invocable
305   static bool has_finalizable_subclass(ciInstanceKlass* k);

307   // As a general rule, it is OK to compile under the assumption that
308   // a given type or method is concrete, even if it at some future
309   // point becomes abstract.  So dependency checking is one-sided, in
310   // that it permits supposedly concrete classes or methods to turn up
311   // as really abstract.  (This shouldn't happen, except during class
312   // evolution, but that's the logic of the checking.)  However, if a
313   // supposedly abstract class or method suddenly becomes concrete, a
314   // dependency on it must fail.

316   // Checking old assertions at run-time (in the VM only):
317   static klassOop check_evol_method(methodOop m);
```



```
318   static klassOop check_leaf_type(klassOop ctxk);
319   static klassOop check_abstract_with_unique_concrete_subtype(klassOop ctxk, kla
320                                                               KlassDepChange* ch
321   static klassOop check_abstract_with_no_concrete_subtype(klassOop ctxk,
322                                                           KlassDepChange* change
323   static klassOop check_concrete_with_no_concrete_subtype(klassOop ctxk,
324                                                           KlassDepChange* change
325   static klassOop check_unique_concrete_method(klassOop ctxk, methodOop uniqm,
326                                                KlassDepChange* changes = NULL);
327   static klassOop check_abstract_with_exclusive_concrete_subtypes(klassOop ctxk,
328                                                                   KlassDepChange
329   static klassOop check_exclusive_concrete_methods(klassOop ctxk, methodOop m1,
330                                                    KlassDepChange* changes = NUL
331   static klassOop check_has_no_finalizable_subclasses(klassOop ctxk, KlassDepCha
332   static klassOop check_call_site_target_value(oop call_site, oop method_handle,
321   static klassOop check_call_site_target_value(klassOop ctxk, oop call_site, oop
333   // A returned klassOop is NULL if the dependency assertion is still
334   // valid.  A non-NULL klassOop is a 'witness' to the assertion
335   // failure, a point in the class hierarchy where the assertion has
336   // been proven false.  For example, if check_leaf_type returns
337   // non-NULL, the value is a subtype of the supposed leaf type.  This
338   // witness value may be useful for logging the dependency failure.
339   // Note that, when a dependency fails, there may be several possible
340   // witnesses to the failure.  The value returned from the check_foo
341   // method is chosen arbitrarily.

343   // The 'changes' value, if non-null, requests a limited spot-check
344   // near the indicated recent changes in the class hierarchy.
345   // It is used by DepStream::spot_check_dependency_at.

347   // Detecting possible new assertions:
348   static klassOop  find_unique_concrete_subtype(klassOop ctxk);
349   static methodOop find_unique_concrete_method(klassOop ctxk, methodOop m);
350   static int       find_exclusive_concrete_subtypes(klassOop ctxk, int klen, kla
351   static int       find_exclusive_concrete_methods(klassOop ctxk, int mlen, meth

353   // Create the encoding which will be stored in an nmethod.
354   void encode_content_bytes();

356   address content_bytes() {
357     assert(_content_bytes != NULL, "encode it first");
358     return _content_bytes;
359   }
360   size_t size_in_bytes() {
361     assert(_content_bytes != NULL, "encode it first");
362     return _size_in_bytes;
363   }

365   OopRecorder* oop_recorder() { return _oop_recorder; }
366   CompileLog*  log()          { return _log; }

368   void copy_to(nmethod* nm);

370   void log_all_dependencies();
371   void log_dependency(DepType dept, int nargs, ciObject* args[]) {
372     write_dependency_to(log(), dept, nargs, args);
373   }
374   void log_dependency(DepType dept,
375                       ciObject* x0,
376                       ciObject* x1 = NULL,
377                       ciObject* x2 = NULL) {
378     if (log() == NULL)  return;
379     ciObject* args[max_arg_count];
380     args[0] = x0;
381     args[1] = x1;
382     args[2] = x2;
```

```
383     assert(2 < max_arg_count, "");
384     log_dependency(dept, dep_args(dept), args);
385   }

387   static void write_dependency_to(CompileLog* log,
388                                   DepType dept,
389                                   int nargs, ciObject* args[],
390                                   klassOop witness = NULL);
391   static void write_dependency_to(CompileLog* log,
392                                   DepType dept,
393                                   int nargs, oop args[],
394                                   klassOop witness = NULL);
395   static void write_dependency_to(xmlStream* xtty,
396                                   DepType dept,
397                                   int nargs, oop args[],
398                                   klassOop witness = NULL);
399   static void print_dependency(DepType dept,
400                                int nargs, oop args[],
401                                klassOop witness = NULL);

403  private:
404   // helper for encoding common context types as zero:
405   static ciKlass* ctxk_encoded_as_null(DepType dept, ciObject* x);

407   static klassOop ctxk_encoded_as_null(DepType dept, oop x);

409  public:
410   // Use this to iterate over an nmethod's dependency set.
411   // Works on new and old dependency sets.
412   // Usage:
413   //
414   // ;
415   // Dependencies::DepType dept;
416   // for (Dependencies::DepStream deps(nm); deps.next(); ) {
417   //   ...
418   // }
419   //
420   // The caller must be in the VM, since oops are not wrapped in handles.
421   class DepStream {
422   private:
423     nmethod*              _code;   // null if in a compiler thread
424     Dependencies*         _deps;   // null if not in a compiler thread
425     CompressedReadStream  _bytes;
426 #ifdef ASSERT
427     size_t                _byte_limit;
428 #endif

430     // iteration variables:
431     DepType               _type;
432     int                   _xi[max_arg_count+1];

434     void initial_asserts(size_t byte_limit) NOT_DEBUG({});

436     inline oop recorded_oop_at(int i);
437         // => _code? _code->oop_at(i): *_deps->_oop_recorder->handle_at(i)

439     klassOop check_klass_dependency(KlassDepChange* changes);
440     klassOop check_call_site_dependency(CallSiteDepChange* changes);

442     void trace_and_log_witness(klassOop witness);

444   public:
445     DepStream(Dependencies* deps)
446       : _deps(deps),
447         _code(NULL),
448         _bytes(deps->content_bytes())
```



```
449     {
450       initial_asserts(deps->size_in_bytes());
451     }
452     DepStream(nmethod* code)
453       : _deps(NULL),
454         _code(code),
455         _bytes(code->dependencies_begin())
456     {
457       initial_asserts(code->dependencies_size());
458     }

460     bool next();

462     DepType type()               { return _type; }
463     int argument_count()         { return dep_args(type()); }
464     int argument_index(int i)    { assert(0 <= i && i < argument_count(), "oob")
465                                    return _xi[i]; }
466     oop argument(int i);         // => recorded_oop_at(argument_index(i))
467     klassOop context_type();

469     bool is_klass_type()         { return Dependencies::is_klass_type(type()); }

471 #endif /* ! codereview */
472     methodOop method_argument(int i) {
473       oop x = argument(i);
474       assert(x->is_method(), "type");
475       return (methodOop) x;
476     }
477     klassOop type_argument(int i) {
478       oop x = argument(i);
479       assert(x->is_klass(), "type");
480       return (klassOop) x;
481     }

483     // The point of the whole exercise:  Is this dep still OK?
484     klassOop check_dependency() {
485       klassOop result = check_klass_dependency(NULL);
486       if (result != NULL)  return result;
487       return check_call_site_dependency(NULL);
488     }

490     // A lighter version:  Checks only around recent changes in a class
491     // hierarchy.  (See Universe::flush_dependents_on.)
492     klassOop spot_check_dependency_at(DepChange& changes);

494     // Log the current dependency to xtty or compilation log.
495     void log_dependency(klassOop witness = NULL);

497     // Print the current dependency to tty.
498     void print_dependency(klassOop witness = NULL, bool verbose = false);
499   };
500   friend class Dependencies::DepStream;

502   static void print_statistics() PRODUCT_RETURN;
503 };


506 // Every particular DepChange is a sub-class of this class.
507 class DepChange : public StackObj {
508  public:
509   // What kind of DepChange is this?
510   virtual bool is_klass_change()     const { return false; }
511   virtual bool is_call_site_change() const { return false; }

513   // Subclass casting with assertions.
514   KlassDepChange*    as_klass_change() {
```

```
515     assert(is_klass_change(), "bad cast");
516     return (KlassDepChange*) this;
517   }
518   CallSiteDepChange* as_call_site_change() {
519     assert(is_call_site_change(), "bad cast");
520     return (CallSiteDepChange*) this;
521   }

523   void print();

525  public:
526   enum ChangeType {
527     NO_CHANGE = 0,              // an uninvolved klass
528     Change_new_type,            // a newly loaded type
529     Change_new_sub,             // a super with a new subtype
530     Change_new_impl,            // an interface with a new implementation
531     CHANGE_LIMIT,
532     Start_Klass = CHANGE_LIMIT  // internal indicator for ContextStream
533   };

535   // Usage:
536   // for (DepChange::ContextStream str(changes); str.next(); ) {
537   //   klassOop k = str.klass();
538   //   switch (str.change_type()) {
539   //     ...
540   //   }
541   // }
542   class ContextStream : public StackObj {
543    private:
544     DepChange&  _changes;
545     friend class DepChange;

547     // iteration variables:
548     ChangeType  _change_type;
549     klassOop    _klass;
550     objArrayOop _ti_base;    // i.e., transitive_interfaces
551     int         _ti_index;
552     int         _ti_limit;

554     // start at the beginning:
555     void start();

557    public:
558     ContextStream(DepChange& changes)
559       : _changes(changes)
560     { start(); }

562     ContextStream(DepChange& changes, No_Safepoint_Verifier& nsv)
563       : _changes(changes)
564       // the nsv argument makes it safe to hold oops like _klass
565     { start(); }

567     bool next();

569     ChangeType change_type()     { return _change_type; }
570     klassOop   klass()           { return _klass; }
571   };
572   friend class DepChange::ContextStream;
573 };


576 // A class hierarchy change coming through the VM (under the Compile_lock).
577 // The change is structured as a single new type with any number of supers
578 // and implemented interface types.  Other than the new type, any of the
579 // super types can be context types for a relevant dependency, which the
580 // new type could invalidate.
```



```
581 class KlassDepChange : public DepChange {
582  private:
583   // each change set is rooted in exactly one new type (at present):
584   KlassHandle _new_type;

586   void initialize();

588  public:
589   // notes the new type, marks it and all its super-types
590   KlassDepChange(KlassHandle new_type)
591     : _new_type(new_type)
592   {
593     initialize();
594   }

596   // cleans up the marks
597   ~KlassDepChange();

599   // What kind of DepChange is this?
600   virtual bool is_klass_change() const { return true; }

602   klassOop new_type() { return _new_type(); }

604   // involves_context(k) is true if k is new_type or any of the super types
605   bool involves_context(klassOop k);
606 };


609 // A CallSite has changed its target.
610 class CallSiteDepChange : public DepChange {
611  private:
612   Handle _call_site;
613   Handle _method_handle;

615  public:
616   CallSiteDepChange(Handle call_site, Handle method_handle)
617     : _call_site(call_site),
618       _method_handle(method_handle)
619   {
620     assert(_call_site()    ->is_a(SystemDictionary::CallSite_klass()),     "must
621     assert(_method_handle()->is_a(SystemDictionary::MethodHandle_klass()), "must
622   }

624   // What kind of DepChange is this?
625   virtual bool is_call_site_change() const { return true; }

627   oop call_site()     const { return _call_site();     }
628   oop method_handle() const { return _method_handle(); }
629 };

631 #endif // SHARE_VM_CODE_DEPENDENCIES_HPP
```

```
*********************************************************
   64150 Thu Aug 25 01:58:57 2011
new/src/share/vm/memory/universe.cpp
*********************************************************
______unchanged_portion_omitted_

1190 // Flushes compiled methods dependent on a particular CallSite
1191 // instance when its target is different than the given MethodHandle.
1192 void Universe::flush_dependents_on(Handle call_site, Handle method_handle) {
1193   assert_lock_strong(Compile_lock);

1195   if (CodeCache::number_of_nmethods_with_dependencies() == 0) return;

1197   // CodeCache can only be updated by a thread_in_VM and they will all be
1198   // stopped dring the safepoint so CodeCache will be safe to update without
1199   // holding the CodeCache_lock.

1201   CallSiteDepChange changes(call_site(), method_handle());

1203   // Compute the dependent nmethods that have a reference to a
1204   // CallSite object.  We use instanceKlass::mark_dependent_nmethod
1205   // directly instead of CodeCache::mark_for_deoptimization because we
1206   // want dependents on the call site class only not all classes in
1207   // the ContextStream.
1206   // want dependents on the class CallSite only not all classes in the
1207   // ContextStream.
1208   int marked = 0;
1209   {
1210     MutexLockerEx mu(CodeCache_lock, Mutex::_no_safepoint_check_flag);
1211     instanceKlass* call_site_klass = instanceKlass::cast(call_site->klass());
1211     instanceKlass* call_site_klass = instanceKlass::cast(SystemDictionary::CallS
1212     marked = call_site_klass->mark_dependent_nmethods(changes);
1213   }
1214   if (marked > 0) {
1215     // At least one nmethod has been marked for deoptimization
1216     VM_Deoptimize op;
1217     VMThread::execute(&op);
1218   }
1219 }
______unchanged_portion_omitted_
```

```
**********************************************************
   41167 Thu Aug 25 01:58:58 2011
new/src/share/vm/opto/callGenerator.cpp
**********************************************************
______unchanged_portion_omitted_

 338 CallGenerator* CallGenerator::for_dynamic_call(ciMethod* m) {
 339   assert(m->is_method_handle_invoke() || m->is_method_handle_adapter(), "for_dyn
 339   assert(m->is_method_handle_invoke(), "for_dynamic_call mismatch");
 340   return new DynamicCallGenerator(m);
 341 }
______unchanged_portion_omitted_


 702 CallGenerator* CallGenerator::for_method_handle_inline(Node* method_handle, JVMS
 703                                                         ciMethod* caller, ciMetho
 704   if (method_handle->Opcode() == Op_ConP) {
 705     const TypeOopPtr* oop_ptr = method_handle->bottom_type()->is_oopptr();
 706     ciObject* const_oop = oop_ptr->const_oop();
 707     ciMethodHandle* method_handle = const_oop->as_method_handle();

 709     // Set the callee to have access to the class and signature in
 710     // the MethodHandleCompiler.
 711     method_handle->set_callee(callee);
 712     method_handle->set_caller(caller);
 713     method_handle->set_call_profile(profile);

 715     // Get an adapter for the MethodHandle.
 716     ciMethod* target_method = method_handle->get_method_handle_adapter();
 717     if (target_method != NULL) {
 718       CallGenerator* cg = Compile::current()->call_generator(target_method, -1,
 719       if (cg != NULL && cg->is_inline())
 720         return cg;
 718       CallGenerator* hit_cg = Compile::current()->call_generator(target_method,
 719       if (hit_cg != NULL && hit_cg->is_inline())
 720         return hit_cg;
 721     }
 722   } else if (method_handle->Opcode() == Op_Phi && method_handle->req() == 3 &&
 723              method_handle->in(1)->Opcode() == Op_ConP && method_handle->in(2)->
 724     // selectAlternative idiom merging two constant MethodHandles.
 725     // Generate a guard so that each can be inlined.  We might want to
 726     // do more inputs at later point but this gets the most common
 727     // case.
 728     const TypeOopPtr* oop_ptr = method_handle->in(1)->bottom_type()->is_oopptr()
 729     ciObject* const_oop = oop_ptr->const_oop();
 730     ciMethodHandle* mh = const_oop->as_method_handle();

 732     CallGenerator* cg1 = for_method_handle_inline(method_handle->in(1), jvms, ca
 733     CallGenerator* cg2 = for_method_handle_inline(method_handle->in(2), jvms, ca
 734     if (cg1 != NULL && cg2 != NULL) {
 735       return new PredictedDynamicCallGenerator(mh, cg2, cg1, PROB_FAIR);
 736     }
 737   }
 738   return NULL;
 739 }


 742 CallGenerator* CallGenerator::for_invokedynamic_inline(ciCallSite* call_site, JV
 743                                                         ciMethod* caller, ciMetho
 744   assert(call_site->is_constant_call_site() || call_site->is_mutable_call_site()
 745   ciMethodHandle* method_handle = call_site->get_target();

 747   // Set the callee to have access to the class and signature in the
 748   // MethodHandleCompiler.
 749   method_handle->set_callee(callee);
 750   method_handle->set_caller(caller);
```



```
 751   method_handle->set_call_profile(profile);

 753   // Get an adapter for the MethodHandle.
 754   ciMethod* target_method = method_handle->get_invokedynamic_adapter();
 755   if (target_method != NULL) {
 756     Compile *C = Compile::current();
 757     CallGenerator* cg = C->call_generator(target_method, -1, false, jvms, true,
 758     if (cg != NULL && cg->is_inline()) {
 757     CallGenerator* hit_cg = C->call_generator(target_method, -1, false, jvms, tr
 758     if (hit_cg != NULL && hit_cg->is_inline()) {
 759       // Add a dependence for invalidation of the optimization.
 760       if (call_site->is_mutable_call_site()) {
 761         C->dependencies()->assert_call_site_target_value(call_site, method_handl
 761         C->dependencies()->assert_call_site_target_value(C->env()->CallSite_klas
 762       }
 763       return cg;
 763       return hit_cg;
 764     }
 765   }
 766   return NULL;
 767 }
______unchanged_portion_omitted_
```