```
**********************************************************
   53646 Fri Oct 21 04:46:39 2011
new/src/share/vm/c1/c1_Runtime1.cpp
**********************************************************
______unchanged_portion_omitted_


 597 JRT_ENTRY(void, Runtime1::throw_range_check_exception(JavaThread* thread, int in
 598   NOT_PRODUCT(_throw_range_check_exception_count++;)
 599   Events::log("throw_range_check");
 600   char message[jintAsStringSize];
 601   sprintf(message, "%d", index);
 602   SharedRuntime::throw_and_post_jvmti_exception(thread, vmSymbols::java_lang_Arr
 603 JRT_END


 606 JRT_ENTRY(void, Runtime1::throw_index_exception(JavaThread* thread, int index))
 607   NOT_PRODUCT(_throw_index_exception_count++;)
 608   Events::log("throw_index");
 609   char message[16];
 610   sprintf(message, "%d", index);
 611   SharedRuntime::throw_and_post_jvmti_exception(thread, vmSymbols::java_lang_Ind
 612 JRT_END


 615 JRT_ENTRY(void, Runtime1::throw_div0_exception(JavaThread* thread))
 616   NOT_PRODUCT(_throw_div0_exception_count++;)
 617   SharedRuntime::throw_and_post_jvmti_exception(thread, vmSymbols::java_lang_Ari
 618 JRT_END


 621 JRT_ENTRY(void, Runtime1::throw_null_pointer_exception(JavaThread* thread))
 622   NOT_PRODUCT(_throw_null_pointer_exception_count++;)
 623   SharedRuntime::throw_and_post_jvmti_exception(thread, vmSymbols::java_lang_Nul
 624 JRT_END


 627 JRT_ENTRY(void, Runtime1::throw_class_cast_exception(JavaThread* thread, oopDesc
 628   NOT_PRODUCT(_throw_class_cast_exception_count++;)
 629   ResourceMark rm(thread);
 630   char* message = SharedRuntime::generate_class_cast_message(
 631     thread, Klass::cast(object->klass())->external_name());
 632   SharedRuntime::throw_and_post_jvmti_exception(
 633     thread, vmSymbols::java_lang_ClassCastException(), message);
 634 JRT_END


 637 JRT_ENTRY(void, Runtime1::throw_incompatible_class_change_error(JavaThread* thre
 638   NOT_PRODUCT(_throw_incompatible_class_change_error_count++;)
 639   ResourceMark rm(thread);
 640   SharedRuntime::throw_and_post_jvmti_exception(thread, vmSymbols::java_lang_Inc
 641 JRT_END


 644 JRT_ENTRY_NO_ASYNC(void, Runtime1::monitorenter(JavaThread* thread, oopDesc* obj
 645   NOT_PRODUCT(_monitorenter_slowcase_cnt++;)
 646   if (PrintBiasedLockingStatistics) {
 647     Atomic::inc(BiasedLocking::slow_path_entry_count_addr());
 648   }
 649   Handle h_obj(thread, obj);
 650   assert(h_obj()->is_oop(), "must be NULL or an object");
 651   if (UseBiasedLocking) {
 652     // Retry fast entry if bias is revoked to avoid unnecessary inflation
 653     ObjectSynchronizer::fast_enter(h_obj, lock->lock(), true, CHECK);
 654   } else {
 655     if (UseFastLocking) {
```



```
 656       // When using fast locking, the compiled code has already tried the fast c
 657       assert(obj == lock->obj(), "must match");
 658       ObjectSynchronizer::slow_enter(h_obj, lock->lock(), THREAD);
 659     } else {
 660       lock->set_obj(obj);
 661       ObjectSynchronizer::fast_enter(h_obj, lock->lock(), false, THREAD);
 662     }
 663   }
 664 JRT_END


 667 JRT_LEAF(void, Runtime1::monitorexit(JavaThread* thread, BasicObjectLock* lock))
 668   NOT_PRODUCT(_monitorexit_slowcase_cnt++;)
 669   assert(thread == JavaThread::current(), "threads must correspond");
 670   assert(thread->last_Java_sp(), "last_Java_sp must be set");
 671   // monitorexit is non-blocking (leaf routine) => no exceptions can be thrown
 672   EXCEPTION_MARK;

 674   oop obj = lock->obj();
 675   assert(obj->is_oop(), "must be NULL or an object");
 676   if (UseFastLocking) {
 677     // When using fast locking, the compiled code has already tried the fast cas
 678     ObjectSynchronizer::slow_exit(obj, lock->lock(), THREAD);
 679   } else {
 680     ObjectSynchronizer::fast_exit(obj, lock->lock(), THREAD);
 681   }
 682 JRT_END

 684 // Cf. OptoRuntime::deoptimize_caller_frame
 685 JRT_ENTRY(void, Runtime1::deoptimize(JavaThread* thread))
 686   // Called from within the owner thread, so no need for safepoint
 687   RegisterMap reg_map(thread, false);
 688   frame stub_frame = thread->last_frame();
 689   assert(stub_frame.is_runtime_frame(), "sanity check");
 690   frame caller_frame = stub_frame.sender(&reg_map);

 692   // We are coming from a compiled method; check this is true.
 693   assert(CodeCache::find_nmethod(caller_frame.pc()) != NULL, "sanity");

 695   // Deoptimize the caller frame.
 696   Deoptimization::deoptimize_frame(thread, caller_frame.id());

 698   // Return to the now deoptimized frame.
 699 JRT_END

 701 #endif /* ! codereview */

 703 static klassOop resolve_field_return_klass(methodHandle caller, int bci, TRAPS)
 704   Bytecode_field field_access(caller, bci);
 705   // This can be static or non-static field access
 706   Bytecodes::Code code       = field_access.code();

 708   // We must load class, initialize class and resolvethe field
 709   FieldAccessInfo result; // initialize class if needed
 710   constantPoolHandle constants(THREAD, caller->constants());
 711   LinkResolver::resolve_field(result, constants, field_access.index(), Bytecodes
 712   return result.klass()();
 713 }


 716 //
 717 // This routine patches sites where a class wasn't loaded or
 718 // initialized at the time the code was generated.  It handles
 719 // references to classes, fields and forcing of initialization.  Most
 720 // of the cases are straightforward and involving simply forcing
 721 // resolution of a class, rewriting the instruction stream with the
```

```
722 // needed constant and replacing the call in this function with the
723 // patched code.  The case for static field is more complicated since
724 // the thread which is in the process of initializing a class can
725 // access it's static fields but other threads can't so the code
726 // either has to deoptimize when this case is detected or execute a
727 // check that the current thread is the initializing thread.  The
728 // current
729 //
730 // Patches basically look like this:
731 //
732 //
733 // patch_site: jmp patch stub     ;; will be patched
734 // continue:   ...
735 //             ...
736 //             ...
737 //             ...
738 //
739 // They have a stub which looks like this:
740 //
741 //             ;; patch body
742 //             movl <const>, reg           (for class constants)
743 //        <or> movl [reg1 + <const>], reg  (for field offsets)
744 //        <or> movl reg, [reg1 + <const>]  (for field offsets)
745 //             <being_init offset> <bytes to copy> <bytes to skip>
746 // patch_stub: call Runtime1::patch_code (through a runtime stub)
747 //             jmp patch_site
748 //
749 //
750 // A normal patch is done by rewriting the patch body, usually a move,
751 // and then copying it into place over top of the jmp instruction
752 // being careful to flush caches and doing it in an MP-safe way.  The
753 // constants following the patch body are used to find various pieces
754 // of the patch relative to the call site for Runtime1::patch_code.
755 // The case for getstatic and putstatic is more complicated because
756 // getstatic and putstatic have special semantics when executing while
757 // the class is being initialized.  getstatic/putstatic on a class
758 // which is being_initialized may be executed by the initializing
759 // thread but other threads have to block when they execute it.  This
760 // is accomplished in compiled code by executing a test of the current
761 // thread against the initializing thread of the class.  It's emitted
762 // as boilerplate in their stub which allows the patched code to be
763 // executed before it's copied back into the main body of the nmethod.
764 //
765 // being_init: get_thread(<tmp reg>
766 //             cmpl [reg1 + <init_thread_offset>], <tmp reg>
767 //             jne patch_stub
768 //             movl [reg1 + <const>], reg  (for field offsets)  <or>
769 //             movl reg, [reg1 + <const>]  (for field offsets)
770 //             jmp continue
771 //             <being_init offset> <bytes to copy> <bytes to skip>
772 // patch_stub: jmp Runtim1::patch_code (through a runtime stub)
773 //             jmp patch_site
774 //
775 // If the class is being initialized the patch body is rewritten and
776 // the patch site is rewritten to jump to being_init, instead of
777 // patch_stub.  Whenever this code is executed it checks the current
778 // thread against the intializing thread so other threads will enter
779 // the runtime and end up blocked waiting the class to finish
780 // initializing inside the calls to resolve_field below.  The
781 // initializing class will continue on it's way.  Once the class is
782 // fully_initialized, the intializing_thread of the class becomes
783 // NULL, so the next thread to execute this code will fail the test,
784 // call into patch_code and complete the patching process by copying
785 // the patch body back into the main part of the nmethod and resume
786 // executing.
787 //
```



```
788 //

790 JRT_ENTRY(void, Runtime1::patch_code(JavaThread* thread, Runtime1::StubID stub_i
791   NOT_PRODUCT(_patch_code_slowcase_cnt++;)

793   ResourceMark rm(thread);
794   RegisterMap reg_map(thread, false);
795   frame runtime_frame = thread->last_frame();
796   frame caller_frame = runtime_frame.sender(&reg_map);

798   // last java frame on stack
799   vframeStream vfst(thread, true);
800   assert(!vfst.at_end(), "Java frame must exist");

802   methodHandle caller_method(THREAD, vfst.method());
803   // Note that caller_method->code() may not be same as caller_code because of O
804   // Note also that in the presence of inlining it is not guaranteed
805   // that caller_method() == caller_code->method()


808   int bci = vfst.bci();

810   Events::log("patch_code @ " INTPTR_FORMAT , caller_frame.pc());

812   Bytecodes::Code code = caller_method()->java_code_at(bci);

814 #ifndef PRODUCT
815   // this is used by assertions in the access_field_patching_id
816   BasicType patch_field_type = T_ILLEGAL;
817 #endif // PRODUCT
818   bool deoptimize_for_volatile = false;
819   int patch_field_offset = -1;
820   KlassHandle init_klass(THREAD, klassOop(NULL)); // klass needed by access_fiel
821   Handle load_klass(THREAD, NULL);               // oop needed by load_klass_pa
822   if (stub_id == Runtime1::access_field_patching_id) {

824     Bytecode_field field_access(caller_method, bci);
825     FieldAccessInfo result; // initialize class if needed
826     Bytecodes::Code code = field_access.code();
827     constantPoolHandle constants(THREAD, caller_method->constants());
828     LinkResolver::resolve_field(result, constants, field_access.index(), Bytecod
829     patch_field_offset = result.field_offset();

831     // If we're patching a field which is volatile then at compile it
832     // must not have been know to be volatile, so the generated code
833     // isn't correct for a volatile reference.  The nmethod has to be
834     // deoptimized so that the code can be regenerated correctly.
835     // This check is only needed for access_field_patching since this
836     // is the path for patching field offsets.  load_klass is only
837     // used for patching references to oops which don't need special
838     // handling in the volatile case.
839     deoptimize_for_volatile = result.access_flags().is_volatile();

841 #ifndef PRODUCT
842     patch_field_type = result.field_type();
843 #endif
844   } else if (stub_id == Runtime1::load_klass_patching_id) {
845     oop k;
846     switch (code) {
847       case Bytecodes::_putstatic:
848       case Bytecodes::_getstatic:
849         { klassOop klass = resolve_field_return_klass(caller_method, bci, CHECK)
850           // Save a reference to the class that has to be checked for initializa
851           init_klass = KlassHandle(THREAD, klass);
852           k = klass->java_mirror();
853         }
```

```
854         break;
855       case Bytecodes::_new:
856         { Bytecode_new bnew(caller_method(), caller_method->bcp_from(bci));
857           k = caller_method->constants()->klass_at(bnew.index(), CHECK);
858         }
859         break;
860       case Bytecodes::_multianewarray:
861         { Bytecode_multianewarray mna(caller_method(), caller_method->bcp_from(b
862           k = caller_method->constants()->klass_at(mna.index(), CHECK);
863         }
864         break;
865       case Bytecodes::_instanceof:
866         { Bytecode_instanceof io(caller_method(), caller_method->bcp_from(bci));
867           k = caller_method->constants()->klass_at(io.index(), CHECK);
868         }
869         break;
870       case Bytecodes::_checkcast:
871         { Bytecode_checkcast cc(caller_method(), caller_method->bcp_from(bci));
872           k = caller_method->constants()->klass_at(cc.index(), CHECK);
873         }
874         break;
875       case Bytecodes::_anewarray:
876         { Bytecode_anewarray anew(caller_method(), caller_method->bcp_from(bci))
877           klassOop ek = caller_method->constants()->klass_at(anew.index(), CHECK
878           k = Klass::cast(ek)->array_klass(CHECK);
879         }
880         break;
881       case Bytecodes::_ldc:
882       case Bytecodes::_ldc_w:
883         {
884           Bytecode_loadconstant cc(caller_method, bci);
885           k = cc.resolve_constant(CHECK);
886           assert(k != NULL && !k->is_klass(), "must be class mirror or other Jav
887         }
888         break;
889       default: Unimplemented();
890     }
891     // convert to handle
892     load_klass = Handle(THREAD, k);
893   } else {
894     ShouldNotReachHere();
895   }
896
897   if (deoptimize_for_volatile) {
898     // At compile time we assumed the field wasn't volatile but after
899     // loading it turns out it was volatile so we have to throw the
900     // compiled code out and let it be regenerated.
901     if (TracePatching) {
902       tty->print_cr("Deoptimizing for patching volatile field reference");
903     }
904     // It's possible the nmethod was invalidated in the last
905     // safepoint, but if it's still alive then make it not_entrant.
906     nmethod* nm = CodeCache::find_nmethod(caller_frame.pc());
907     if (nm != NULL) {
908       nm->make_not_entrant();
909     }
910
911     Deoptimization::deoptimize_frame(thread, caller_frame.id());
912
913     // Return to the now deoptimized frame.
914   }
915
916   // If we are patching in a non-perm oop, make sure the nmethod
917   // is on the right list.
918   if (ScavengeRootsInCode && load_klass.not_null() && load_klass->is_scavengable
919     MutexLockerEx ml_code (CodeCache_lock, Mutex::_no_safepoint_check_flag);
```



```
920     nmethod* nm = CodeCache::find_nmethod(caller_frame.pc());
921     guarantee(nm != NULL, "only nmethods can contain non-perm oops");
922     if (!nm->on_scavenge_root_list())
923       CodeCache::add_scavenge_root_nmethod(nm);
924   }
925
926   // Now copy code back
927
928   {
929     MutexLockerEx ml_patch (Patching_lock, Mutex::_no_safepoint_check_flag);
930     //
931     // Deoptimization may have happened while we waited for the lock.
932     // In that case we don't bother to do any patching we just return
933     // and let the deopt happen
934     if (!caller_is_deopted()) {
935       NativeGeneralJump* jump = nativeGeneralJump_at(caller_frame.pc());
936       address instr_pc = jump->jump_destination();
937       NativeInstruction* ni = nativeInstruction_at(instr_pc);
938       if (ni->is_jump() ) {
939         // the jump has not been patched yet
940         // The jump destination is slow case and therefore not part of the stubs
941         // (stubs are only for StaticCalls)
942
943         // format of buffer
944         //    ....
945         //    instr byte 0     <-- copy_buff
946         //    instr byte 1
947         //    ..
948         //    instr byte n-1
949         //      n
950         //    ....             <-- call destination
951
952         address stub_location = caller_frame.pc() + PatchingStub::patch_info_off
953         unsigned char* byte_count = (unsigned char*) (stub_location - 1);
954         unsigned char* byte_skip = (unsigned char*) (stub_location - 2);
955         unsigned char* being_initialized_entry_offset = (unsigned char*) (stub_l
956         address copy_buff = stub_location - *byte_skip - *byte_count;
957         address being_initialized_entry = stub_location - *being_initialized_ent
958         if (TracePatching) {
959           tty->print_cr(" Patching %s at bci %d at address 0x%x  (%s)", Bytecode
960                         instr_pc, (stub_id == Runtime1::access_field_patching_id
961           nmethod* caller_code = CodeCache::find_nmethod(caller_frame.pc());
962           assert(caller_code != NULL, "nmethod not found");
963
964           // NOTE we use pc() not original_pc() because we already know they are
965           // identical otherwise we'd have never entered this block of code
966
967           OopMap* map = caller_code->oop_map_for_return_address(caller_frame.pc(
968           assert(map != NULL, "null check");
969           map->print();
970           tty->cr();
971
972           Disassembler::decode(copy_buff, copy_buff + *byte_count, tty);
973         }
974         // depending on the code below, do_patch says whether to copy the patch
975         bool do_patch = true;
976         if (stub_id == Runtime1::access_field_patching_id) {
977           // The offset may not be correct if the class was not loaded at code g
978           // Set it now.
979           NativeMovRegMem* n_move = nativeMovRegMem_at(copy_buff);
980           assert(n_move->offset() == 0 || (n_move->offset() == 4 && (patch_field
981           assert(patch_field_offset >= 0, "illegal offset");
982           n_move->add_offset_in_bytes(patch_field_offset);
983         } else if (stub_id == Runtime1::load_klass_patching_id) {
984           // If a getstatic or putstatic is referencing a klass which
985           // isn't fully initialized, the patch body isn't copied into
```

```
 986           // place until initialization is complete.  In this case the
 987           // patch site is setup so that any threads besides the
 988           // initializing thread are forced to come into the VM and
 989           // block.
 990           do_patch = (code != Bytecodes::_getstatic && code != Bytecodes::_putst
 991                       instanceKlass::cast(init_klass())->is_initialized();
 992           NativeGeneralJump* jump = nativeGeneralJump_at(instr_pc);
 993           if (jump->jump_destination() == being_initialized_entry) {
 994             assert(do_patch == true, "initialization must be complete at this po
 995           } else {
 996             // patch the instruction <move reg, klass>
 997             NativeMovConstReg* n_copy = nativeMovConstReg_at(copy_buff);

 999             assert(n_copy->data() == 0 ||
1000                    n_copy->data() == (intptr_t)Universe::non_oop_word(),
1001                    "illegal init value");
1002             assert(load_klass() != NULL, "klass not set");
1003             n_copy->set_data((intx) (load_klass()));

1005             if (TracePatching) {
1006               Disassembler::decode(copy_buff, copy_buff + *byte_count, tty);
1007             }

1009 #if defined(SPARC) || defined(PPC)
1010             // Update the oop location in the nmethod with the proper
1011             // oop.  When the code was generated, a NULL was stuffed
1012             // in the oop table and that table needs to be update to
1013             // have the right value.  On intel the value is kept
1014             // directly in the instruction instead of in the oop
1015             // table, so set_data above effectively updated the value.
1016             nmethod* nm = CodeCache::find_nmethod(instr_pc);
1017             assert(nm != NULL, "invalid nmethod_pc");
1018             RelocIterator oops(nm, copy_buff, copy_buff + 1);
1019             bool found = false;
1020             while (oops.next() && !found) {
1021               if (oops.type() == relocInfo::oop_type) {
1022                 oop_Relocation* r = oops.oop_reloc();
1023                 oop* oop_adr = r->oop_addr();
1024                 *oop_adr = load_klass();
1025                 r->fix_oop_relocation();
1026                 found = true;
1027               }
1028             }
1029             assert(found, "the oop must exist!");
1030 #endif

1032           }
1033         } else {
1034           ShouldNotReachHere();
1035         }
1036         if (do_patch) {
1037           // replace instructions
1038           // first replace the tail, then the call
1039 #ifdef ARM
1040           if(stub_id == Runtime1::load_klass_patching_id && !VM_Version::support
1041             nmethod* nm = CodeCache::find_nmethod(instr_pc);
1042             oop* oop_addr = NULL;
1043             assert(nm != NULL, "invalid nmethod_pc");
1044             RelocIterator oops(nm, copy_buff, copy_buff + 1);
1045             while (oops.next()) {
1046               if (oops.type() == relocInfo::oop_type) {
1047                 oop_Relocation* r = oops.oop_reloc();
1048                 oop_addr = r->oop_addr();
1049                 break;
1050               }
1051             }
```



```
1052             assert(oop_addr != NULL, "oop relocation must exist");
1053             copy_buff -= *byte_count;
1054             NativeMovConstReg* n_copy2 = nativeMovConstReg_at(copy_buff);
1055             n_copy2->set_pc_relative_offset((address)oop_addr, instr_pc);
1056           }
1057 #endif

1059           for (int i = NativeCall::instruction_size; i < *byte_count; i++) {
1060             address ptr = copy_buff + i;
1061             int a_byte = (*ptr) & 0xFF;
1062             address dst = instr_pc + i;
1063             *(unsigned char*)dst = (unsigned char) a_byte;
1064           }
1065           ICache::invalidate_range(instr_pc, *byte_count);
1066           NativeGeneralJump::replace_mt_safe(instr_pc, copy_buff);

1068           if (stub_id == Runtime1::load_klass_patching_id) {
1069             // update relocInfo to oop
1070             nmethod* nm = CodeCache::find_nmethod(instr_pc);
1071             assert(nm != NULL, "invalid nmethod_pc");

1073             // The old patch site is now a move instruction so update
1074             // the reloc info so that it will get updated during
1075             // future GCs.
1076             RelocIterator iter(nm, (address)instr_pc, (address)(instr_pc + 1));
1077             relocInfo::change_reloc_info_for_address(&iter, (address) instr_pc,
1078                                                      relocInfo::none, relocInfo:
1079 #ifdef SPARC
1080             // Sparc takes two relocations for an oop so update the second one.
1081             address instr_pc2 = instr_pc + NativeMovConstReg::add_offset;
1082             RelocIterator iter2(nm, instr_pc2, instr_pc2 + 1);
1083             relocInfo::change_reloc_info_for_address(&iter2, (address) instr_pc2
1084                                                      relocInfo::none, relocInfo:
1085 #endif
1086 #ifdef PPC
1087           { address instr_pc2 = instr_pc + NativeMovConstReg::lo_offset;
1088             RelocIterator iter2(nm, instr_pc2, instr_pc2 + 1);
1089             relocInfo::change_reloc_info_for_address(&iter2, (address) instr_pc2
1090           }
1091 #endif
1092           }

1094         } else {
1095           ICache::invalidate_range(copy_buff, *byte_count);
1096           NativeGeneralJump::insert_unconditional(instr_pc, being_initialized_en
1097         }
1098       }
1099     }
1100   }
1101 JRT_END

1103 //
1104 // Entry point for compiled code. We want to patch a nmethod.
1105 // We don't do a normal VM transition here because we want to
1106 // know after the patching is complete and any safepoint(s) are taken
1107 // if the calling nmethod was deoptimized. We do this by calling a
1108 // helper method which does the normal VM transition and when it
1109 // completes we can check for deoptimization. This simplifies the
1110 // assembly code in the cpu directories.
1111 //
1112 int Runtime1::move_klass_patching(JavaThread* thread) {
1113 //
1114 // NOTE: we are still in Java
1115 //
1116   Thread* THREAD = thread;
1117   debug_only(NoHandleMark nhm;)
```

```
1118   {
1119     // Enter VM mode

1121     ResetNoHandleMark rnhm;
1122     patch_code(thread, load_klass_patching_id);
1123   }
1124   // Back in JAVA, use no oops DON'T safepoint

1126   // Return true if calling code is deoptimized

1128   return caller_is_deopted();
1129 }

1131 //
1132 // Entry point for compiled code. We want to patch a nmethod.
1133 // We don't do a normal VM transition here because we want to
1134 // know after the patching is complete and any safepoint(s) are taken
1135 // if the calling nmethod was deoptimized. We do this by calling a
1136 // helper method which does the normal VM transition and when it
1137 // completes we can check for deoptimization. This simplifies the
1138 // assembly code in the cpu directories.
1139 //

1141 int Runtime1::access_field_patching(JavaThread* thread) {
1142 //
1143 // NOTE: we are still in Java
1144 //
1145   Thread* THREAD = thread;
1146   debug_only(NoHandleMark nhm;)
1147   {
1148     // Enter VM mode

1150     ResetNoHandleMark rnhm;
1151     patch_code(thread, access_field_patching_id);
1152   }
1153   // Back in JAVA, use no oops DON'T safepoint

1155   // Return true if calling code is deoptimized

1157   return caller_is_deopted();
1158 JRT_END


1161 JRT_LEAF(void, Runtime1::trace_block_entry(jint block_id))
1162   // for now we just print out the block id
1163   tty->print("%d ", block_id);
1164 JRT_END


1167 // Array copy return codes.
1168 enum {
1169   ac_failed = -1, // arraycopy failed
1170   ac_ok = 0       // arraycopy succeeded
1171 };


1174 // Below length is the # elements copied.
1175 template <class T> int obj_arraycopy_work(oopDesc* src, T* src_addr,
1176                                           oopDesc* dst, T* dst_addr,
1177                                           int length) {

1179   // For performance reasons, we assume we are using a card marking write
1180   // barrier. The assert will fail if this is not the case.
1181   // Note that we use the non-virtual inlineable variant of write_ref_array.
1182   BarrierSet* bs = Universe::heap()->barrier_set();
1183   assert(bs->has_write_ref_array_opt(), "Barrier set must have ref array opt");
```



```
1184   assert(bs->has_write_ref_array_pre_opt(), "For pre-barrier as well.");
1185   if (src == dst) {
1186     // same object, no check
1187     bs->write_ref_array_pre(dst_addr, length);
1188     Copy::conjoint_oops_atomic(src_addr, dst_addr, length);
1189     bs->write_ref_array((HeapWord*)dst_addr, length);
1190     return ac_ok;
1191   } else {
1192     klassOop bound = objArrayKlass::cast(dst->klass())->element_klass();
1193     klassOop stype = objArrayKlass::cast(src->klass())->element_klass();
1194     if (stype == bound || Klass::cast(stype)->is_subtype_of(bound)) {
1195       // Elements are guaranteed to be subtypes, so no check necessary
1196       bs->write_ref_array_pre(dst_addr, length);
1197       Copy::conjoint_oops_atomic(src_addr, dst_addr, length);
1198       bs->write_ref_array((HeapWord*)dst_addr, length);
1199       return ac_ok;
1200     }
1201   }
1202   return ac_failed;
1203 }

1205 // fast and direct copy of arrays; returning -1, means that an exception may be
1206 // and we did not copy anything
1207 JRT_LEAF(int, Runtime1::arraycopy(oopDesc* src, int src_pos, oopDesc* dst, int d
1208 #ifndef PRODUCT
1209   _generic_arraycopy_cnt++;        // Slow-path oop array copy
1210 #endif

1212   if (src == NULL || dst == NULL || src_pos < 0 || dst_pos < 0 || length < 0) re
1213   if (!dst->is_array() || !src->is_array()) return ac_failed;
1214   if ((unsigned int) arrayOop(src)->length() < (unsigned int)src_pos + (unsigned
1215   if ((unsigned int) arrayOop(dst)->length() < (unsigned int)dst_pos + (unsigned

1217   if (length == 0) return ac_ok;
1218   if (src->is_typeArray()) {
1219     const klassOop klass_oop = src->klass();
1220     if (klass_oop != dst->klass()) return ac_failed;
1221     typeArrayKlass* klass = typeArrayKlass::cast(klass_oop);
1222     const int l2es = klass->log2_element_size();
1223     const int ihs = klass->array_header_in_bytes() / wordSize;
1224     char* src_addr = (char*) ((oopDesc**)src + ihs) + (src_pos << l2es);
1225     char* dst_addr = (char*) ((oopDesc**)dst + ihs) + (dst_pos << l2es);
1226     // Potential problem: memmove is not guaranteed to be word atomic
1227     // Revisit in Merlin
1228     memmove(dst_addr, src_addr, length << l2es);
1229     return ac_ok;
1230   } else if (src->is_objArray() && dst->is_objArray()) {
1231     if (UseCompressedOops) {
1232       narrowOop *src_addr  = objArrayOop(src)->obj_at_addr<narrowOop>(src_pos);
1233       narrowOop *dst_addr  = objArrayOop(dst)->obj_at_addr<narrowOop>(dst_pos);
1234       return obj_arraycopy_work(src, src_addr, dst, dst_addr, length);
1235     } else {
1236       oop *src_addr  = objArrayOop(src)->obj_at_addr<oop>(src_pos);
1237       oop *dst_addr  = objArrayOop(dst)->obj_at_addr<oop>(dst_pos);
1238       return obj_arraycopy_work(src, src_addr, dst, dst_addr, length);
1239     }
1240   }
1241   return ac_failed;
1242 JRT_END


1245 JRT_LEAF(void, Runtime1::primitive_arraycopy(HeapWord* src, HeapWord* dst, int l
1246 #ifndef PRODUCT
1247   _primitive_arraycopy_cnt++;
1248 #endif
```

```
1250   if (length == 0) return;
1251   // Not guaranteed to be word atomic, but that doesn't matter
1252   // for anything but an oop array, which is covered by oop_arraycopy.
1253   Copy::conjoint_jbytes(src, dst, length);
1254 JRT_END

1256 JRT_LEAF(void, Runtime1::oop_arraycopy(HeapWord* src, HeapWord* dst, int num))
1257 #ifndef PRODUCT
1258   _oop_arraycopy_cnt++;
1259 #endif

1261   if (num == 0) return;
1262   BarrierSet* bs = Universe::heap()->barrier_set();
1263   assert(bs->has_write_ref_array_opt(), "Barrier set must have ref array opt");
1264   assert(bs->has_write_ref_array_pre_opt(), "For pre-barrier as well.");
1265   if (UseCompressedOops) {
1266     bs->write_ref_array_pre((narrowOop*)dst, num);
1267     Copy::conjoint_oops_atomic((narrowOop*) src, (narrowOop*) dst, num);
1268   } else {
1269     bs->write_ref_array_pre((oop*)dst, num);
1270     Copy::conjoint_oops_atomic((oop*) src, (oop*) dst, num);
1271   }
1272   bs->write_ref_array(dst, num);
1273 JRT_END


1276 #ifndef PRODUCT
1277 void Runtime1::print_statistics() {
1278   tty->print_cr("C1 Runtime statistics:");
1279   tty->print_cr(" _resolve_invoke_virtual_cnt:     %d", SharedRuntime::_resolve_
1280   tty->print_cr(" _resolve_invoke_opt_virtual_cnt: %d", SharedRuntime::_resolve_
1281   tty->print_cr(" _resolve_invoke_static_cnt:      %d", SharedRuntime::_resolve_
1282   tty->print_cr(" _handle_wrong_method_cnt:        %d", SharedRuntime::_wrong_me
1283   tty->print_cr(" _ic_miss_cnt:                    %d", SharedRuntime::_ic_miss_
1284   tty->print_cr(" _generic_arraycopy_cnt:          %d", _generic_arraycopy_cnt);
1285   tty->print_cr(" _generic_arraycopystub_cnt:      %d", _generic_arraycopystub_c
1286   tty->print_cr(" _byte_arraycopy_cnt:             %d", _byte_arraycopy_cnt);
1287   tty->print_cr(" _short_arraycopy_cnt:            %d", _short_arraycopy_cnt);
1288   tty->print_cr(" _int_arraycopy_cnt:              %d", _int_arraycopy_cnt);
1289   tty->print_cr(" _long_arraycopy_cnt:             %d", _long_arraycopy_cnt);
1290   tty->print_cr(" _primitive_arraycopy_cnt:        %d", _primitive_arraycopy_cnt
1291   tty->print_cr(" _oop_arraycopy_cnt (C):          %d", Runtime1::_oop_arraycopy
1292   tty->print_cr(" _oop_arraycopy_cnt (stub):       %d", _oop_arraycopy_cnt);
1293   tty->print_cr(" _arraycopy_slowcase_cnt:         %d", _arraycopy_slowcase_cnt)
1294   tty->print_cr(" _arraycopy_checkcast_cnt:        %d", _arraycopy_checkcast_cnt
1295   tty->print_cr(" _arraycopy_checkcast_attempt_cnt:%d", _arraycopy_checkcast_att

1297   tty->print_cr(" _new_type_array_slowcase_cnt:    %d", _new_type_array_slowcase
1298   tty->print_cr(" _new_object_array_slowcase_cnt:  %d", _new_object_array_slowca
1299   tty->print_cr(" _new_instance_slowcase_cnt:      %d", _new_instance_slowcase_c
1300   tty->print_cr(" _new_multi_array_slowcase_cnt:   %d", _new_multi_array_slowcas
1301   tty->print_cr(" _monitorenter_slowcase_cnt:      %d", _monitorenter_slowcase_c
1302   tty->print_cr(" _monitorexit_slowcase_cnt:       %d", _monitorexit_slowcase_cn
1303   tty->print_cr(" _patch_code_slowcase_cnt:        %d", _patch_code_slowcase_cnt

1305   tty->print_cr(" _throw_range_check_exception_count:            %d:", _throw_ra
1306   tty->print_cr(" _throw_index_exception_count:                  %d:", _throw_in
1307   tty->print_cr(" _throw_div0_exception_count:                   %d:", _throw_di
1308   tty->print_cr(" _throw_null_pointer_exception_count:           %d:", _throw_nu
1309   tty->print_cr(" _throw_class_cast_exception_count:             %d:", _throw_cl
1310   tty->print_cr(" _throw_incompatible_class_change_error_count:  %d:", _throw_in
1311   tty->print_cr(" _throw_array_store_exception_count:            %d:", _throw_ar
1312   tty->print_cr(" _throw_count:                                  %d:", _throw_co

1314   SharedRuntime::print_ic_miss_histogram();
1315   tty->cr();
```



```
1316 }
1317 #endif // PRODUCT
```

```
*********************************************************
    7807 Fri Oct 21 04:46:40 2011
new/src/share/vm/c1/c1_Runtime1.hpp
*********************************************************
```

```
/*
 * Copyright (c) 1999, 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_C1_C1_RUNTIME1_HPP
#define SHARE_VM_C1_C1_RUNTIME1_HPP

#include "c1/c1_FrameMap.hpp"
#include "code/stubs.hpp"
#include "interpreter/interpreter.hpp"
#include "memory/allocation.hpp"
#include "runtime/deoptimization.hpp"

class StubAssembler;

// The Runtime1 holds all assembly stubs and VM
// runtime routines needed by code code generated
// by the Compiler1.

#define RUNTIME1_STUBS(stub, last_entry) \
  stub(dtrace_object_alloc)          \
  stub(unwind_exception)             \
  stub(forward_exception)            \
  stub(throw_range_check_failed)       /* throws ArrayIndexOutOfBoundsException
  stub(throw_index_exception)          /* throws IndexOutOfBoundsException */ \
  stub(throw_div0_exception)         \
  stub(throw_null_pointer_exception) \
  stub(register_finalizer)           \
  stub(new_instance)                 \
  stub(fast_new_instance)            \
  stub(fast_new_instance_init_check) \
  stub(new_type_array)               \
  stub(new_object_array)             \
  stub(new_multi_array)              \
  stub(handle_exception_nofpu)         /* optimized version that does not preser
  stub(handle_exception)             \
  stub(handle_exception_from_callee) \
  stub(throw_array_store_exception)  \
  stub(throw_class_cast_exception)   \
  stub(throw_incompatible_class_change_error)   \
  stub(slow_subtype_check)           \
  stub(monitorenter)                 \
```



```
  stub(monitorenter_nofpu)             /* optimized version that does not preser
  stub(monitorexit)                  \
  stub(monitorexit_nofpu)              /* optimized version that does not preser
  stub(deoptimize)                   \
#endif /* ! codereview */
  stub(access_field_patching)        \
  stub(load_klass_patching)          \
  stub(g1_pre_barrier_slow)          \
  stub(g1_post_barrier_slow)         \
  stub(fpu2long_stub)                \
  stub(counter_overflow)             \
  last_entry(number_of_ids)

#define DECLARE_STUB_ID(x)       x ## _id ,
#define DECLARE_LAST_STUB_ID(x)  x
#define STUB_NAME(x)             #x " Runtime1 stub",
#define LAST_STUB_NAME(x)        #x " Runtime1 stub"

class Runtime1: public AllStatic {
  friend class VMStructs;
  friend class ArrayCopyStub;

 public:
  enum StubID {
    RUNTIME1_STUBS(DECLARE_STUB_ID, DECLARE_LAST_STUB_ID)
  };

  // statistics
#ifndef PRODUCT
  static int _resolve_invoke_cnt;
  static int _handle_wrong_method_cnt;
  static int _ic_miss_cnt;
  static int _generic_arraycopy_cnt;
  static int _primitive_arraycopy_cnt;
  static int _oop_arraycopy_cnt;
  static int _generic_arraycopystub_cnt;
  static int _arraycopy_slowcase_cnt;
  static int _arraycopy_checkcast_cnt;
  static int _arraycopy_checkcast_attempt_cnt;
  static int _new_type_array_slowcase_cnt;
  static int _new_object_array_slowcase_cnt;
  static int _new_instance_slowcase_cnt;
  static int _new_multi_array_slowcase_cnt;
  static int _monitorenter_slowcase_cnt;
  static int _monitorexit_slowcase_cnt;
  static int _patch_code_slowcase_cnt;
  static int _throw_range_check_exception_count;
  static int _throw_index_exception_count;
  static int _throw_div0_exception_count;
  static int _throw_null_pointer_exception_count;
  static int _throw_class_cast_exception_count;
  static int _throw_incompatible_class_change_error_count;
  static int _throw_array_store_exception_count;
  static int _throw_count;
#endif

 private:
  static CodeBlob* _blobs[number_of_ids];
  static const char* _blob_names[];

  // stub generation
  static void       generate_blob_for(BufferBlob* blob, StubID id);
  static OopMapSet* generate_code_for(StubID id, StubAssembler* sasm);
  static OopMapSet* generate_exception_throw(StubAssembler* sasm, address target
  static OopMapSet* generate_handle_exception(StubID id, StubAssembler* sasm);
  static void       generate_unwind_exception(StubAssembler *sasm);
```

```
  static OopMapSet* generate_patching(StubAssembler* sasm, address target);

  static OopMapSet* generate_stub_call(StubAssembler* sasm, Register result, add
                                       Register arg1 = noreg, Register arg2 = no

  // runtime entry points
  static void new_instance    (JavaThread* thread, klassOopDesc* klass);
  static void new_type_array  (JavaThread* thread, klassOopDesc* klass, jint len
  static void new_object_array(JavaThread* thread, klassOopDesc* klass, jint len
  static void new_multi_array (JavaThread* thread, klassOopDesc* klass, int rank

  static address counter_overflow(JavaThread* thread, int bci, methodOopDesc* me

  static void unimplemented_entry   (JavaThread* thread, StubID id);

  static address exception_handler_for_pc(JavaThread* thread);

  static void throw_range_check_exception(JavaThread* thread, int index);
  static void throw_index_exception(JavaThread* thread, int index);
  static void throw_div0_exception(JavaThread* thread);
  static void throw_null_pointer_exception(JavaThread* thread);
  static void throw_class_cast_exception(JavaThread* thread, oopDesc* object);
  static void throw_incompatible_class_change_error(JavaThread* thread);
  static void throw_array_store_exception(JavaThread* thread, oopDesc* object);

  static void monitorenter(JavaThread* thread, oopDesc* obj, BasicObjectLock* lo
  static void monitorexit (JavaThread* thread, BasicObjectLock* lock);

  static void deoptimize(JavaThread* thread);

#endif /* ! codereview */
  static int access_field_patching(JavaThread* thread);
  static int move_klass_patching(JavaThread* thread);

  static void patch_code(JavaThread* thread, StubID stub_id);

 public:
  // initialization
  static void initialize(BufferBlob* blob);
  static void initialize_pd();

  // stubs
  static CodeBlob* blob_for (StubID id);
  static address   entry_for(StubID id)          { return blob_for(id)->code_beg
  static const char* name_for (StubID id);
  static const char* name_for_address(address entry);

  // platform might add runtime names.
  static const char* pd_name_for_address(address entry);

  // method tracing
  static void trace_block_entry(jint block_id);

#ifndef PRODUCT
  static address throw_count_address()              { return (address)&_throw_c
  static address arraycopy_count_address(BasicType type);
#endif

  // directly accessible leaf routine
  static int  arraycopy(oopDesc* src, int src_pos, oopDesc* dst, int dst_pos, in
  static void primitive_arraycopy(HeapWord* src, HeapWord* dst, int length);
  static void oop_arraycopy(HeapWord* src, HeapWord* dst, int length);

  static void print_statistics()                PRODUCT_RETURN;
};
```



```
#endif // SHARE_VM_C1_C1_RUNTIME1_HPP
```

****
52224 Fri Oct 21 04:46:41 2011
new/src/share/vm/opto/runtime.cpp
****
_____***unchanged_portion_omitted_***

```
1124 void OptoRuntime::deoptimize_caller_frame(JavaThread *thread, bool doit) {
1125   // Deoptimize frame
1126   if (doit) {
1127     // Called from within the owner thread, so no need for safepoint
1128     RegisterMap reg_map(thread);
1129     frame stub_frame = thread->last_frame();
1130     assert(stub_frame.is_runtime_frame() || exception_blob()->contains(stub_fram
1131     frame caller_frame = stub_frame.sender(&reg_map);

1133     // Deoptimize the caller frame.
1133     // bypass VM_DeoptimizeFrame and deoptimize the frame directly
1134     Deoptimization::deoptimize_frame(thread, caller_frame.id());
1135   }
1136 }
```

_____***unchanged_portion_omitted_***

```
*********************************************************
   18471 Fri Oct 21 04:46:42 2011
new/src/cpu/sparc/vm/c1_CodeStubs_sparc.cpp
*********************************************************
______unchanged_portion_omitted_


 368 void DeoptimizeStub::emit_code(LIR_Assembler* ce) {
 369   __ bind(_entry);
 370   __ call(Runtime1::entry_for(Runtime1::deoptimize_id), relocInfo::runtime_call_
 370   __ call(SharedRuntime::deopt_blob()->unpack_with_reexecution());
 371   __ delayed()->nop();
 372   ce->add_call_info_here(_info);
 373   DEBUG_ONLY(__ should_not_reach_here());
 373   debug_only(__ should_not_reach_here());
 374 }
______unchanged_portion_omitted_
```

```
**********************************************************
   39661 Fri Oct 21 04:46:44 2011
new/src/cpu/sparc/vm/c1_Runtime1_sparc.cpp
**********************************************************
______unchanged_portion_omitted_

 355 OopMapSet* Runtime1::generate_code_for(StubID id, StubAssembler* sasm) {

 357   OopMapSet* oop_maps = NULL;
 358   // for better readability
 359   const bool must_gc_arguments = true;
 360   const bool dont_gc_arguments = false;

 362   // stub code & info for the different stubs
 363   switch (id) {
 364     case forward_exception_id:
 365       {
 366         oop_maps = generate_handle_exception(id, sasm);
 367       }
 368       break;

 370     case new_instance_id:
 371     case fast_new_instance_id:
 372     case fast_new_instance_init_check_id:
 373       {
 374         Register G5_klass = G5; // Incoming
 375         Register O0_obj   = O0; // Outgoing

 377         if (id == new_instance_id) {
 378           __ set_info("new_instance", dont_gc_arguments);
 379         } else if (id == fast_new_instance_id) {
 380           __ set_info("fast new_instance", dont_gc_arguments);
 381         } else {
 382           assert(id == fast_new_instance_init_check_id, "bad StubID");
 383           __ set_info("fast new_instance init check", dont_gc_arguments);
 384         }

 386         if ((id == fast_new_instance_id || id == fast_new_instance_init_check_id
 387             UseTLAB && FastTLABRefill) {
 388           Label slow_path;
 389           Register G1_obj_size = G1;
 390           Register G3_t1 = G3;
 391           Register G4_t2 = G4;
 392           assert_different_registers(G5_klass, G1_obj_size, G3_t1, G4_t2);

 394           // Push a frame since we may do dtrace notification for the
 395           // allocation which requires calling out and we don't want
 396           // to stomp the real return address.
 397           __ save_frame(0);

 399           if (id == fast_new_instance_init_check_id) {
 400             // make sure the klass is initialized
 401             __ ld(G5_klass, instanceKlass::init_state_offset_in_bytes() + sizeof
 402             __ cmp_and_br_short(G3_t1, instanceKlass::fully_initialized, Assembl
 403           }
 404 #ifdef ASSERT
 405           // assert object can be fast path allocated
 406           {
 407             Label ok, not_ok;
 408           __ ld(G5_klass, Klass::layout_helper_offset_in_bytes() + sizeof(oopDes
 409           // make sure it's an instance (LH > 0)
 410           __ cmp_and_br_short(G1_obj_size, 0, Assembler::lessEqual, Assembler::p
 411           __ btst(Klass::_lh_instance_slow_path_bit, G1_obj_size);
 412           __ br(Assembler::zero, false, Assembler::pn, ok);
 413           __ delayed()->nop();
 414           __ bind(not_ok);
```



```
 415           __ stop("assert(can be fast path allocated)");
 416           __ should_not_reach_here();
 417           __ bind(ok);
 418           }
 419 #endif // ASSERT
 420           // if we got here then the TLAB allocation failed, so try
 421           // refilling the TLAB or allocating directly from eden.
 422           Label retry_tlab, try_eden;
 423           __ tlab_refill(retry_tlab, try_eden, slow_path); // preserves G5_klass

 425           __ bind(retry_tlab);

 427           // get the instance size
 428           __ ld(G5_klass, klassOopDesc::header_size() * HeapWordSize + Klass::la

 430           __ tlab_allocate(O0_obj, G1_obj_size, 0, G3_t1, slow_path);

 432           __ initialize_object(O0_obj, G5_klass, G1_obj_size, 0, G3_t1, G4_t2);
 433           __ verify_oop(O0_obj);
 434           __ mov(O0, I0);
 435           __ ret();
 436           __ delayed()->restore();

 438           __ bind(try_eden);
 439           // get the instance size
 440           __ ld(G5_klass, klassOopDesc::header_size() * HeapWordSize + Klass::la
 441           __ eden_allocate(O0_obj, G1_obj_size, 0, G3_t1, G4_t2, slow_path);
 442           __ incr_allocated_bytes(G1_obj_size, G3_t1, G4_t2);

 444           __ initialize_object(O0_obj, G5_klass, G1_obj_size, 0, G3_t1, G4_t2);
 445           __ verify_oop(O0_obj);
 446           __ mov(O0, I0);
 447           __ ret();
 448           __ delayed()->restore();

 450           __ bind(slow_path);

 452           // pop this frame so generate_stub_call can push it's own
 453           __ restore();
 454         }

 456         oop_maps = generate_stub_call(sasm, I0, CAST_FROM_FN_PTR(address, new_in
 457         // I0->O0: new instance
 458       }

 460       break;

 462     case counter_overflow_id:
 463         // G4 contains bci, G5 contains method
 464       oop_maps = generate_stub_call(sasm, noreg, CAST_FROM_FN_PTR(address, count
 465       break;

 467     case new_type_array_id:
 468     case new_object_array_id:
 469       {
 470         Register G5_klass = G5; // Incoming
 471         Register G4_length = G4; // Incoming
 472         Register O0_obj   = O0; // Outgoing

 474         Address klass_lh(G5_klass, ((klassOopDesc::header_size() * HeapWordSize)
 475                                     + Klass::layout_helper_offset_in_bytes()));
 476         assert(Klass::_lh_header_size_shift % BitsPerByte == 0, "bytewise");
 477         assert(Klass::_lh_header_size_mask == 0xFF, "bytewise");
 478         // Use this offset to pick out an individual byte of the layout_helper:
 479         const int klass_lh_header_size_offset = ((BytesPerInt - 1)  // 3 - 2 sel
 480                                                  - Klass::_lh_header_size_shift
```

```
482         if (id == new_type_array_id) {
483           __ set_info("new_type_array", dont_gc_arguments);
484         } else {
485           __ set_info("new_object_array", dont_gc_arguments);
486         }

488 #ifdef ASSERT
489         // assert object type is really an array of the proper kind
490         {
491           Label ok;
492           Register G3_t1 = G3;
493           __ ld(klass_lh, G3_t1);
494           __ sra(G3_t1, Klass::_lh_array_tag_shift, G3_t1);
495           int tag = ((id == new_type_array_id)
496                      ? Klass::_lh_array_tag_type_value
497                      : Klass::_lh_array_tag_obj_value);
498           __ cmp_and_brx_short(G3_t1, tag, Assembler::equal, Assembler::pt, ok);
499           __ stop("assert(is an array klass)");
500           __ should_not_reach_here();
501           __ bind(ok);
502         }
503 #endif // ASSERT

505         if (UseTLAB && FastTLABRefill) {
506           Label slow_path;
507           Register G1_arr_size = G1;
508           Register G3_t1 = G3;
509           Register O1_t2 = O1;
510           assert_different_registers(G5_klass, G4_length, G1_arr_size, G3_t1, O1

512           // check that array length is small enough for fast path
513           __ set(C1_MacroAssembler::max_array_allocation_length, G3_t1);
514           __ cmp_and_br_short(G4_length, G3_t1, Assembler::greaterUnsigned, Asse

516           // if we got here then the TLAB allocation failed, so try
517           // refilling the TLAB or allocating directly from eden.
518           Label retry_tlab, try_eden;
519           __ tlab_refill(retry_tlab, try_eden, slow_path); // preserves G4_lengt

521           __ bind(retry_tlab);

523           // get the allocation size: (length << (layout_helper & 0x1F)) + heade
524           __ ld(klass_lh, G3_t1);
525           __ sll(G4_length, G3_t1, G1_arr_size);
526           __ srl(G3_t1, Klass::_lh_header_size_shift, G3_t1);
527           __ and3(G3_t1, Klass::_lh_header_size_mask, G3_t1);
528           __ add(G1_arr_size, G3_t1, G1_arr_size);
529           __ add(G1_arr_size, MinObjAlignmentInBytesMask, G1_arr_size);  // alig
530           __ and3(G1_arr_size, ~MinObjAlignmentInBytesMask, G1_arr_size);

532           __ tlab_allocate(O0_obj, G1_arr_size, 0, G3_t1, slow_path);  // preser

534           __ initialize_header(O0_obj, G5_klass, G4_length, G3_t1, O1_t2);
535           __ ldub(klass_lh, G3_t1, klass_lh_header_size_offset);
536           __ sub(G1_arr_size, G3_t1, O1_t2);  // body length
537           __ add(O0_obj, G3_t1, G3_t1);       // body start
538           __ initialize_body(G3_t1, O1_t2);
539           __ verify_oop(O0_obj);
540           __ retl();
541           __ delayed()->nop();

543           __ bind(try_eden);
544           // get the allocation size: (length << (layout_helper & 0x1F)) + heade
545           __ ld(klass_lh, G3_t1);
546           __ sll(G4_length, G3_t1, G1_arr_size);
```



```
547           __ srl(G3_t1, Klass::_lh_header_size_shift, G3_t1);
548           __ and3(G3_t1, Klass::_lh_header_size_mask, G3_t1);
549           __ add(G1_arr_size, G3_t1, G1_arr_size);
550           __ add(G1_arr_size, MinObjAlignmentInBytesMask, G1_arr_size);
551           __ and3(G1_arr_size, ~MinObjAlignmentInBytesMask, G1_arr_size);

553           __ eden_allocate(O0_obj, G1_arr_size, 0, G3_t1, O1_t2, slow_path);  //
554           __ incr_allocated_bytes(G1_arr_size, G3_t1, O1_t2);

556           __ initialize_header(O0_obj, G5_klass, G4_length, G3_t1, O1_t2);
557           __ ldub(klass_lh, G3_t1, klass_lh_header_size_offset);
558           __ sub(G1_arr_size, G3_t1, O1_t2);  // body length
559           __ add(O0_obj, G3_t1, G3_t1);       // body start
560           __ initialize_body(G3_t1, O1_t2);
561           __ verify_oop(O0_obj);
562           __ retl();
563           __ delayed()->nop();

565           __ bind(slow_path);
566         }

568         if (id == new_type_array_id) {
569           oop_maps = generate_stub_call(sasm, I0, CAST_FROM_FN_PTR(address, new_
570         } else {
571           oop_maps = generate_stub_call(sasm, I0, CAST_FROM_FN_PTR(address, new_
572         }
573         // I0 -> O0: new array
574       }
575       break;

577     case new_multi_array_id:
578       { // O0: klass
579         // O1: rank
580         // O2: address of 1st dimension
581         __ set_info("new_multi_array", dont_gc_arguments);
582         oop_maps = generate_stub_call(sasm, I0, CAST_FROM_FN_PTR(address, new_mu
583         // I0 -> O0: new multi array
584       }
585       break;

587     case register_finalizer_id:
588       {
589         __ set_info("register_finalizer", dont_gc_arguments);

591         // load the klass and check the has finalizer flag
592         Label register_finalizer;
593         Register t = O1;
594         __ load_klass(O0, t);
595         __ ld(t, Klass::access_flags_offset_in_bytes() + sizeof(oopDesc), t);
596         __ set(JVM_ACC_HAS_FINALIZER, G3);
597         __ andcc(G3, t, G0);
598         __ br(Assembler::notZero, false, Assembler::pt, register_finalizer);
599         __ delayed()->nop();

601         // do a leaf return
602         __ retl();
603         __ delayed()->nop();

605         __ bind(register_finalizer);
606         OopMap* oop_map = save_live_registers(sasm);
607         int call_offset = __ call_RT(noreg, noreg,
608                                      CAST_FROM_FN_PTR(address, SharedRuntime::re
609         oop_maps = new OopMapSet();
610         oop_maps->add_gc_map(call_offset, oop_map);

612         // Now restore all the live registers
```

```
613         restore_live_registers(sasm);

615         __ ret();
616         __ delayed()->restore();
617       }
618       break;

620     case throw_range_check_failed_id:
621       { __ set_info("range_check_failed", dont_gc_arguments); // arguments will
622         // G4: index
623         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
624       }
625       break;

627     case throw_index_exception_id:
628       { __ set_info("index_range_check_failed", dont_gc_arguments); // arguments
629         // G4: index
630         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
631       }
632       break;

634     case throw_div0_exception_id:
635       { __ set_info("throw_div0_exception", dont_gc_arguments);
636         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
637       }
638       break;

640     case throw_null_pointer_exception_id:
641       { __ set_info("throw_null_pointer_exception", dont_gc_arguments);
642         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
643       }
644       break;

646     case handle_exception_id:
647       { __ set_info("handle_exception", dont_gc_arguments);
648         oop_maps = generate_handle_exception(id, sasm);
649       }
650       break;

652     case handle_exception_from_callee_id:
653       { __ set_info("handle_exception_from_callee", dont_gc_arguments);
654         oop_maps = generate_handle_exception(id, sasm);
655       }
656       break;

658     case unwind_exception_id:
659       {
660         // O0: exception
661         // I7: address of call to this method

663         __ set_info("unwind_exception", dont_gc_arguments);
664         __ mov(Oexception, Oexception->after_save());
665         __ add(I7, frame::pc_return_offset, Oissuing_pc->after_save());

667         __ call_VM_leaf(L7_thread_cache, CAST_FROM_FN_PTR(address, SharedRuntime
668                         G2_thread, Oissuing_pc->after_save());
669         __ verify_not_null_oop(Oexception->after_save());

671         // Restore SP from L7 if the exception PC is a method handle call site.
672         __ mov(O0, G5);  // Save the target address.
673         __ lduw(Address(G2_thread, JavaThread::is_method_handle_return_offset())
674         __ tst(L0);  // Condition codes are preserved over the restore.
675         __ restore();

677         __ jmp(G5, 0);
678         __ delayed()->movcc(Assembler::notZero, false, Assembler::icc, L7_mh_SP_
```



```
679       }
680       break;

682     case throw_array_store_exception_id:
683       {
684         __ set_info("throw_array_store_exception", dont_gc_arguments);
685         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
686       }
687       break;

689     case throw_class_cast_exception_id:
690       {
691         // G4: object
692         __ set_info("throw_class_cast_exception", dont_gc_arguments);
693         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
694       }
695       break;

697     case throw_incompatible_class_change_error_id:
698       {
699         __ set_info("throw_incompatible_class_cast_exception", dont_gc_arguments
700         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
701       }
702       break;

704     case slow_subtype_check_id:
705       { // Support for uint StubRoutine::partial_subtype_check( Klass sub, Klass
706         // Arguments :
707         //
708         //      ret  : G3
709         //      sub  : G3, argument, destroyed
710         //      super: G1, argument, not changed
711         //      raddr: O7, blown by call
712         Label miss;

714         __ save_frame(0);               // Blow no registers!

716         __ check_klass_subtype_slow_path(G3, G1, L0, L1, L2, L4, NULL, &miss);

718         __ mov(1, G3);
719         __ ret();                       // Result in G5 is 'true'
720         __ delayed()->restore();        // free copy or add can go here

722         __ bind(miss);
723         __ mov(0, G3);
724         __ ret();                       // Result in G5 is 'false'
725         __ delayed()->restore();        // free copy or add can go here
726       }

728     case monitorenter_nofpu_id:
729     case monitorenter_id:
730       { // G4: object
731         // G5: lock address
732         __ set_info("monitorenter", dont_gc_arguments);

734         int save_fpu_registers = (id == monitorenter_id);
735         // make a frame and preserve the caller's caller-save registers
736         OopMap* oop_map = save_live_registers(sasm, save_fpu_registers);

738         int call_offset = __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, mon

740         oop_maps = new OopMapSet();
741         oop_maps->add_gc_map(call_offset, oop_map);
742         restore_live_registers(sasm, save_fpu_registers);

744         __ ret();
```

```
745         __ delayed()->restore();
746       }
747       break;

749     case monitorexit_nofpu_id:
750     case monitorexit_id:
751       { // G4: lock address
752         // note: really a leaf routine but must setup last java sp
753         //       => use call_RT for now (speed can be improved by
754         //       doing last java sp setup manually)
755         __ set_info("monitorexit", dont_gc_arguments);

757         int save_fpu_registers = (id == monitorexit_id);
758         // make a frame and preserve the caller's caller-save registers
759         OopMap* oop_map = save_live_registers(sasm, save_fpu_registers);

761         int call_offset = __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, mon

763         oop_maps = new OopMapSet();
764         oop_maps->add_gc_map(call_offset, oop_map);
765         restore_live_registers(sasm, save_fpu_registers);

767         __ ret();
768         __ delayed()->restore();
769       }
770       break;
771 #endif /* ! codereview */

773     case deoptimize_id:
774       {
775         __ set_info("deoptimize", dont_gc_arguments);
776         OopMap* oop_map = save_live_registers(sasm);
777         int call_offset = __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, deo
778         oop_maps = new OopMapSet();
779         oop_maps->add_gc_map(call_offset, oop_map);
780         restore_live_registers(sasm);
781         DeoptimizationBlob* deopt_blob = SharedRuntime::deopt_blob();
782         assert(deopt_blob != NULL, "deoptimization blob must have been created")
783         AddressLiteral dest(deopt_blob->unpack_with_reexecution());
784         __ jump_to(dest, O0);
785         __ delayed()->restore();
786 #endif /* ! codereview */
787       }
788       break;

790     case access_field_patching_id:
791       { __ set_info("access_field_patching", dont_gc_arguments);
792         oop_maps = generate_patching(sasm, CAST_FROM_FN_PTR(address, access_fiel
793       }
794       break;

796     case load_klass_patching_id:
797       { __ set_info("load_klass_patching", dont_gc_arguments);
798         oop_maps = generate_patching(sasm, CAST_FROM_FN_PTR(address, move_klass_
799       }
800       break;

802     case dtrace_object_alloc_id:
803       { // O0: object
804         __ set_info("dtrace_object_alloc", dont_gc_arguments);
805         // we can't gc here so skip the oopmap but make sure that all
806         // the live registers get saved.
807         save_live_registers(sasm);

809         __ save_thread(L7_thread_cache);
810         __ call(CAST_FROM_FN_PTR(address, SharedRuntime::dtrace_object_alloc),
```



```
811                 relocInfo::runtime_call_type);
812         __ delayed()->mov(I0, O0);
813         __ restore_thread(L7_thread_cache);

815         restore_live_registers(sasm);
816         __ ret();
817         __ delayed()->restore();
818       }
819       break;

821 #ifndef SERIALGC
822     case g1_pre_barrier_slow_id:
823       { // G4: previous value of memory
824         BarrierSet* bs = Universe::heap()->barrier_set();
825         if (bs->kind() != BarrierSet::G1SATBCTLogging) {
826           __ save_frame(0);
827           __ set((int)id, O1);
828           __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, unimplemented_entry
829           __ should_not_reach_here();
830           break;
831         }

833         __ set_info("g1_pre_barrier_slow_id", dont_gc_arguments);

835         Register pre_val = G4;
836         Register tmp  = G1_scratch;
837         Register tmp2 = G3_scratch;

839         Label refill, restart;
840         bool with_frame = false; // I don't know if we can do with-frame.
841         int satb_q_index_byte_offset =
842           in_bytes(JavaThread::satb_mark_queue_offset() +
843                    PtrQueue::byte_offset_of_index());
844         int satb_q_buf_byte_offset =
845           in_bytes(JavaThread::satb_mark_queue_offset() +
846                    PtrQueue::byte_offset_of_buf());

848         __ bind(restart);
849         // Load the index into the SATB buffer. PtrQueue::_index is a
850         // size_t so ld_ptr is appropriate
851         __ ld_ptr(G2_thread, satb_q_index_byte_offset, tmp);

853         // index == 0?
854         __ cmp_and_brx_short(tmp, G0, Assembler::equal, Assembler::pn, refill);

856         __ ld_ptr(G2_thread, satb_q_buf_byte_offset, tmp2);
857         __ sub(tmp, oopSize, tmp);

859         __ st_ptr(pre_val, tmp2, tmp);  // [_buf + index] := <address_of_card>
860         // Use return-from-leaf
861         __ retl();
862         __ delayed()->st_ptr(tmp, G2_thread, satb_q_index_byte_offset);

864         __ bind(refill);
865         __ save_frame(0);

867         __ mov(pre_val, L0);
868         __ mov(tmp,     L1);
869         __ mov(tmp2,    L2);

871         __ call_VM_leaf(L7_thread_cache,
872                         CAST_FROM_FN_PTR(address,
873                                          SATBMarkQueueSet::handle_zero_index_for
874                                          G2_thread);

876         __ mov(L0, pre_val);
```

```
877         __ mov(L1, tmp);
878         __ mov(L2, tmp2);
879
880         __ br(Assembler::always, /*annul*/false, Assembler::pt, restart);
881         __ delayed()->restore();
882       }
883       break;
884
885     case g1_post_barrier_slow_id:
886       {
887         BarrierSet* bs = Universe::heap()->barrier_set();
888         if (bs->kind() != BarrierSet::G1SATBCTLogging) {
889           __ save_frame(0);
890           __ set((int)id, O1);
891           __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, unimplemented_entry
892           __ should_not_reach_here();
893           break;
894         }
895
896         __ set_info("g1_post_barrier_slow_id", dont_gc_arguments);
897
898         Register addr = G4;
899         Register cardtable = G5;
900         Register tmp  = G1_scratch;
901         Register tmp2 = G3_scratch;
902         jbyte* byte_map_base = ((CardTableModRefBS*)bs)->byte_map_base;
903
904         Label not_already_dirty, restart, refill;
905
906 #ifdef _LP64
907         __ srlx(addr, CardTableModRefBS::card_shift, addr);
908 #else
909         __ srl(addr, CardTableModRefBS::card_shift, addr);
910 #endif
911
912         AddressLiteral rs(byte_map_base);
913         __ set(rs, cardtable);         // cardtable := <card table base>
914         __ ldub(addr, cardtable, tmp); // tmp := [addr + cardtable]
915
916         assert(CardTableModRefBS::dirty_card_val() == 0, "otherwise check this c
917         __ cmp_and_br_short(tmp, G0, Assembler::notEqual, Assembler::pt, not_alr
918
919         // We didn't take the branch, so we're already dirty: return.
920         // Use return-from-leaf
921         __ retl();
922         __ delayed()->nop();
923
924         // Not dirty.
925         __ bind(not_already_dirty);
926
927         // Get cardtable + tmp into a reg by itself
928         __ add(addr, cardtable, tmp2);
929
930         // First, dirty it.
931         __ stb(G0, tmp2, 0);  // [cardPtr] := 0  (i.e., dirty).
932
933         Register tmp3 = cardtable;
934         Register tmp4 = tmp;
935
936         // these registers are now dead
937         addr = cardtable = tmp = noreg;
938
939         int dirty_card_q_index_byte_offset =
940           in_bytes(JavaThread::dirty_card_queue_offset() +
941                    PtrQueue::byte_offset_of_index());
942         int dirty_card_q_buf_byte_offset =
```



```
943           in_bytes(JavaThread::dirty_card_queue_offset() +
944                    PtrQueue::byte_offset_of_buf());
945
946         __ bind(restart);
947
948         // Get the index into the update buffer. PtrQueue::_index is
949         // a size_t so ld_ptr is appropriate here.
950         __ ld_ptr(G2_thread, dirty_card_q_index_byte_offset, tmp3);
951
952         // index == 0?
953         __ cmp_and_brx_short(tmp3, G0, Assembler::equal,  Assembler::pn, refill)
954
955         __ ld_ptr(G2_thread, dirty_card_q_buf_byte_offset, tmp4);
956         __ sub(tmp3, oopSize, tmp3);
957
958         __ st_ptr(tmp2, tmp4, tmp3);  // [_buf + index] := <address_of_card>
959         // Use return-from-leaf
960         __ retl();
961         __ delayed()->st_ptr(tmp3, G2_thread, dirty_card_q_index_byte_offset);
962
963         __ bind(refill);
964         __ save_frame(0);
965
966         __ mov(tmp2, L0);
967         __ mov(tmp3, L1);
968         __ mov(tmp4, L2);
969
970         __ call_VM_leaf(L7_thread_cache,
971                         CAST_FROM_FN_PTR(address,
972                                          DirtyCardQueueSet::handle_zero_index_fo
973                                          G2_thread);
974
975         __ mov(L0, tmp2);
976         __ mov(L1, tmp3);
977         __ mov(L2, tmp4);
978
979         __ br(Assembler::always, /*annul*/false, Assembler::pt, restart);
980         __ delayed()->restore();
981       }
982       break;
983 #endif // !SERIALGC
984
985     default:
986       { __ set_info("unimplemented entry", dont_gc_arguments);
987         __ save_frame(0);
988         __ set((int)id, O1);
989         __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, unimplemented_entry),
990         __ should_not_reach_here();
991       }
992       break;
993   }
994   return oop_maps;
995 }
996
997
998 OopMapSet* Runtime1::generate_handle_exception(StubID id, StubAssembler* sasm) {
999   __ block_comment("generate_handle_exception");
1000
1001   // Save registers, if required.
1002   OopMapSet* oop_maps = new OopMapSet();
1003   OopMap* oop_map = NULL;
1004   switch (id) {
1005   case forward_exception_id:
1006     // We're handling an exception in the context of a compiled frame.
1007     // The registers have been saved in the standard places.  Perform
1008     // an exception lookup in the caller and dispatch to the handler
```

```
1009     // if found.  Otherwise unwind and dispatch to the callers
1010     // exception handler.
1011      oop_map = generate_oop_map(sasm, true);

1013      // transfer the pending exception to the exception_oop
1014      __ ld_ptr(G2_thread, in_bytes(JavaThread::pending_exception_offset()), Oexc
1015      __ ld_ptr(Oexception, 0, G0);
1016      __ st_ptr(G0, G2_thread, in_bytes(JavaThread::pending_exception_offset()));
1017      __ add(I7, frame::pc_return_offset, Oissuing_pc);
1018     break;
1019   case handle_exception_id:
1020     // At this point all registers MAY be live.
1021     oop_map = save_live_registers(sasm);
1022     __ mov(Oexception->after_save(),  Oexception);
1023     __ mov(Oissuing_pc->after_save(), Oissuing_pc);
1024     break;
1025   case handle_exception_from_callee_id:
1026     // At this point all registers except exception oop (Oexception)
1027     // and exception pc (Oissuing_pc) are dead.
1028     oop_map = new OopMap(frame_size_in_bytes / sizeof(jint), 0);
1029     sasm->set_frame_size(frame_size_in_bytes / BytesPerWord);
1030     __ save_frame_c1(frame_size_in_bytes);
1031     __ mov(Oexception->after_save(),  Oexception);
1032     __ mov(Oissuing_pc->after_save(), Oissuing_pc);
1033     break;
1034   default:  ShouldNotReachHere();
1035   }

1037   __ verify_not_null_oop(Oexception);

1039   // save the exception and issuing pc in the thread
1040   __ st_ptr(Oexception,  G2_thread, in_bytes(JavaThread::exception_oop_offset())
1041   __ st_ptr(Oissuing_pc, G2_thread, in_bytes(JavaThread::exception_pc_offset()))

1043   // use the throwing pc as the return address to lookup (has bci & oop map)
1044   __ mov(Oissuing_pc, I7);
1045   __ sub(I7, frame::pc_return_offset, I7);
1046   int call_offset = __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, exception
1047   oop_maps->add_gc_map(call_offset, oop_map);

1049   // Note: if nmethod has been deoptimized then regardless of
1050   // whether it had a handler or not we will deoptimize
1051   // by entering the deopt blob with a pending exception.

1053   // Restore the registers that were saved at the beginning, remove
1054   // the frame and jump to the exception handler.
1055   switch (id) {
1056   case forward_exception_id:
1057   case handle_exception_id:
1058     restore_live_registers(sasm);
1059     __ jmp(O0, 0);
1060     __ delayed()->restore();
1061     break;
1062   case handle_exception_from_callee_id:
1063     // Restore SP from L7 if the exception PC is a method handle call site.
1064     __ mov(O0, G5);  // Save the target address.
1065     __ lduw(Address(G2_thread, JavaThread::is_method_handle_return_offset()), L0
1066     __ tst(L0);  // Condition codes are preserved over the restore.
1067     __ restore();

1069     __ jmp(G5, 0);  // jump to the exception handler
1070     __ delayed()->movcc(Assembler::notZero, false, Assembler::icc, L7_mh_SP_save
1071     break;
1072   default:  ShouldNotReachHere();
1073   }
```



```
1075   return oop_maps;
1076 }


1079 #undef __

1081 const char *Runtime1::pd_name_for_address(address entry) {
1082   return "<unknown function>";
1083 }
```

****
19507 Fri Oct 21 04:46:45 2011
new/src/cpu/x86/vm/c1_CodeStubs_x86.cpp
****
_____***unchanged_portion_omitted_***

```
 388 void DeoptimizeStub::emit_code(LIR_Assembler* ce) {
 389   __ bind(_entry);
 390   __ call(RuntimeAddress(Runtime1::entry_for(Runtime1::deoptimize_id)));
 390   __ call(RuntimeAddress(SharedRuntime::deopt_blob()->unpack_with_reexecution())
 391   ce->add_call_info_here(_info);
 392   DEBUG_ONLY(__ should_not_reach_here());
 392   debug_only(__ should_not_reach_here());
 393 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   68504 Fri Oct 21 04:46:46 2011
new/src/cpu/x86/vm/c1_Runtime1_x86.cpp
**********************************************************
______unchanged_portion_omitted_


 965 OopMapSet* Runtime1::generate_code_for(StubID id, StubAssembler* sasm) {

 967   // for better readability
 968   const bool must_gc_arguments = true;
 969   const bool dont_gc_arguments = false;

 971   // default value; overwritten for some optimized stubs that are called from me
 972   bool save_fpu_registers = true;

 974   // stub code & info for the different stubs
 975   OopMapSet* oop_maps = NULL;
 976   switch (id) {
 977     case forward_exception_id:
 978       {
 979         oop_maps = generate_handle_exception(id, sasm);
 980         __ leave();
 981         __ ret(0);
 982       }
 983       break;

 985     case new_instance_id:
 986     case fast_new_instance_id:
 987     case fast_new_instance_init_check_id:
 988       {
 989         Register klass = rdx; // Incoming
 990         Register obj   = rax; // Result

 992         if (id == new_instance_id) {
 993           __ set_info("new_instance", dont_gc_arguments);
 994         } else if (id == fast_new_instance_id) {
 995           __ set_info("fast new_instance", dont_gc_arguments);
 996         } else {
 997           assert(id == fast_new_instance_init_check_id, "bad StubID");
 998           __ set_info("fast new_instance init check", dont_gc_arguments);
 999         }

1001         if ((id == fast_new_instance_id || id == fast_new_instance_init_check_id
1002             UseTLAB && FastTLABRefill) {
1003           Label slow_path;
1004           Register obj_size = rcx;
1005           Register t1       = rbx;
1006           Register t2       = rsi;
1007           assert_different_registers(klass, obj, obj_size, t1, t2);

1009           __ push(rdi);
1010           __ push(rbx);

1012           if (id == fast_new_instance_init_check_id) {
1013             // make sure the klass is initialized
1014             __ cmpl(Address(klass, instanceKlass::init_state_offset_in_bytes() +
1015             __ jcc(Assembler::notEqual, slow_path);
1016           }

1018 #ifdef ASSERT
1019           // assert object can be fast path allocated
1020           {
1021             Label ok, not_ok;
1022             __ movl(obj_size, Address(klass, Klass::layout_helper_offset_in_byte
1023             __ cmpl(obj_size, 0);  // make sure it's an instance (LH > 0)
```



```
1024             __ jcc(Assembler::lessEqual, not_ok);
1025             __ testl(obj_size, Klass::_lh_instance_slow_path_bit);
1026             __ jcc(Assembler::zero, ok);
1027             __ bind(not_ok);
1028             __ stop("assert(can be fast path allocated)");
1029             __ should_not_reach_here();
1030             __ bind(ok);
1031           }
1032 #endif // ASSERT

1034           // if we got here then the TLAB allocation failed, so try
1035           // refilling the TLAB or allocating directly from eden.
1036           Label retry_tlab, try_eden;
1037           const Register thread =
1038             __ tlab_refill(retry_tlab, try_eden, slow_path); // does not destroy

1040           __ bind(retry_tlab);

1042           // get the instance size (size is postive so movl is fine for 64bit)
1043           __ movl(obj_size, Address(klass, klassOopDesc::header_size() * HeapWor

1045           __ tlab_allocate(obj, obj_size, 0, t1, t2, slow_path);

1047           __ initialize_object(obj, klass, obj_size, 0, t1, t2);
1048           __ verify_oop(obj);
1049           __ pop(rbx);
1050           __ pop(rdi);
1051           __ ret(0);

1053           __ bind(try_eden);
1054           // get the instance size (size is postive so movl is fine for 64bit)
1055           __ movl(obj_size, Address(klass, klassOopDesc::header_size() * HeapWor

1057           __ eden_allocate(obj, obj_size, 0, t1, slow_path);
1058           __ incr_allocated_bytes(thread, obj_size, 0);

1060           __ initialize_object(obj, klass, obj_size, 0, t1, t2);
1061           __ verify_oop(obj);
1062           __ pop(rbx);
1063           __ pop(rdi);
1064           __ ret(0);

1066           __ bind(slow_path);
1067           __ pop(rbx);
1068           __ pop(rdi);
1069         }

1071         __ enter();
1072         OopMap* map = save_live_registers(sasm, 2);
1073         int call_offset = __ call_RT(obj, noreg, CAST_FROM_FN_PTR(address, new_i
1074         oop_maps = new OopMapSet();
1075         oop_maps->add_gc_map(call_offset, map);
1076         restore_live_registers_except_rax(sasm);
1077         __ verify_oop(obj);
1078         __ leave();
1079         __ ret(0);

1081         // rax,: new instance
1082       }

1084       break;

1086     case counter_overflow_id:
1087       {
1088         Register bci = rax, method = rbx;
1089         __ enter();
```

```
1090         OopMap* map = save_live_registers(sasm, 3);
1091         // Retrieve bci
1092         __ movl(bci, Address(rbp, 2*BytesPerWord));
1093         // And a pointer to the methodOop
1094         __ movptr(method, Address(rbp, 3*BytesPerWord));
1095         int call_offset = __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, cou
1096         oop_maps = new OopMapSet();
1097         oop_maps->add_gc_map(call_offset, map);
1098         restore_live_registers(sasm);
1099         __ leave();
1100         __ ret(0);
1101       }
1102       break;

1104     case new_type_array_id:
1105     case new_object_array_id:
1106       {
1107         Register length   = rbx; // Incoming
1108         Register klass    = rdx; // Incoming
1109         Register obj      = rax; // Result

1111         if (id == new_type_array_id) {
1112           __ set_info("new_type_array", dont_gc_arguments);
1113         } else {
1114           __ set_info("new_object_array", dont_gc_arguments);
1115         }

1117 #ifdef ASSERT
1118         // assert object type is really an array of the proper kind
1119         {
1120           Label ok;
1121           Register t0 = obj;
1122           __ movl(t0, Address(klass, Klass::layout_helper_offset_in_bytes() + si
1123           __ sarl(t0, Klass::_lh_array_tag_shift);
1124           int tag = ((id == new_type_array_id)
1125                      ? Klass::_lh_array_tag_type_value
1126                      : Klass::_lh_array_tag_obj_value);
1127           __ cmpl(t0, tag);
1128           __ jcc(Assembler::equal, ok);
1129           __ stop("assert(is an array klass)");
1130           __ should_not_reach_here();
1131           __ bind(ok);
1132         }
1133 #endif // ASSERT

1135         if (UseTLAB && FastTLABRefill) {
1136           Register arr_size = rsi;
1137           Register t1       = rcx;  // must be rcx for use as shift count
1138           Register t2       = rdi;
1139           Label slow_path;
1140           assert_different_registers(length, klass, obj, arr_size, t1, t2);

1142           // check that array length is small enough for fast path.
1143           __ cmpl(length, C1_MacroAssembler::max_array_allocation_length);
1144           __ jcc(Assembler::above, slow_path);

1146           // if we got here then the TLAB allocation failed, so try
1147           // refilling the TLAB or allocating directly from eden.
1148           Label retry_tlab, try_eden;
1149           const Register thread =
1150             __ tlab_refill(retry_tlab, try_eden, slow_path); // preserves rbx &

1152           __ bind(retry_tlab);

1154           // get the allocation size: round_up(hdr + length << (layout_helper &
1155           // since size is positive movl does right thing on 64bit
```



```
1156           __ movl(t1, Address(klass, klassOopDesc::header_size() * HeapWordSize
1157           // since size is postive movl does right thing on 64bit
1158           __ movl(arr_size, length);
1159           assert(t1 == rcx, "fixed register usage");
1160           __ shlptr(arr_size /* by t1=rcx, mod 32 */);
1161           __ shrptr(t1, Klass::_lh_header_size_shift);
1162           __ andptr(t1, Klass::_lh_header_size_mask);
1163           __ addptr(arr_size, t1);
1164           __ addptr(arr_size, MinObjAlignmentInBytesMask); // align up
1165           __ andptr(arr_size, ~MinObjAlignmentInBytesMask);

1167           __ tlab_allocate(obj, arr_size, 0, t1, t2, slow_path);  // preserves a

1169           __ initialize_header(obj, klass, length, t1, t2);
1170           __ movb(t1, Address(klass, klassOopDesc::header_size() * HeapWordSize
1171           assert(Klass::_lh_header_size_shift % BitsPerByte == 0, "bytewise");
1172           assert(Klass::_lh_header_size_mask <= 0xFF, "bytewise");
1173           __ andptr(t1, Klass::_lh_header_size_mask);
1174           __ subptr(arr_size, t1);  // body length
1175           __ addptr(t1, obj);       // body start
1176           __ initialize_body(t1, arr_size, 0, t2);
1177           __ verify_oop(obj);
1178           __ ret(0);

1180           __ bind(try_eden);
1181           // get the allocation size: round_up(hdr + length << (layout_helper &
1182           // since size is positive movl does right thing on 64bit
1183           __ movl(t1, Address(klass, klassOopDesc::header_size() * HeapWordSize
1184           // since size is postive movl does right thing on 64bit
1185           __ movl(arr_size, length);
1186           assert(t1 == rcx, "fixed register usage");
1187           __ shlptr(arr_size /* by t1=rcx, mod 32 */);
1188           __ shrptr(t1, Klass::_lh_header_size_shift);
1189           __ andptr(t1, Klass::_lh_header_size_mask);
1190           __ addptr(arr_size, t1);
1191           __ addptr(arr_size, MinObjAlignmentInBytesMask); // align up
1192           __ andptr(arr_size, ~MinObjAlignmentInBytesMask);

1194           __ eden_allocate(obj, arr_size, 0, t1, slow_path);  // preserves arr_s
1195           __ incr_allocated_bytes(thread, arr_size, 0);

1197           __ initialize_header(obj, klass, length, t1, t2);
1198           __ movb(t1, Address(klass, klassOopDesc::header_size() * HeapWordSize
1199           assert(Klass::_lh_header_size_shift % BitsPerByte == 0, "bytewise");
1200           assert(Klass::_lh_header_size_mask <= 0xFF, "bytewise");
1201           __ andptr(t1, Klass::_lh_header_size_mask);
1202           __ subptr(arr_size, t1);  // body length
1203           __ addptr(t1, obj);       // body start
1204           __ initialize_body(t1, arr_size, 0, t2);
1205           __ verify_oop(obj);
1206           __ ret(0);

1208           __ bind(slow_path);
1209         }

1211         __ enter();
1212         OopMap* map = save_live_registers(sasm, 3);
1213         int call_offset;
1214         if (id == new_type_array_id) {
1215           call_offset = __ call_RT(obj, noreg, CAST_FROM_FN_PTR(address, new_typ
1216         } else {
1217           call_offset = __ call_RT(obj, noreg, CAST_FROM_FN_PTR(address, new_obj
1218         }

1220         oop_maps = new OopMapSet();
1221         oop_maps->add_gc_map(call_offset, map);
```

```
1222         restore_live_registers_except_rax(sasm);

1224         __ verify_oop(obj);
1225         __ leave();
1226         __ ret(0);

1228         // rax,: new array
1229       }
1230       break;

1232     case new_multi_array_id:
1233       { StubFrame f(sasm, "new_multi_array", dont_gc_arguments);
1234         // rax,: klass
1235         // rbx,: rank
1236         // rcx: address of 1st dimension
1237         OopMap* map = save_live_registers(sasm, 4);
1238         int call_offset = __ call_RT(rax, noreg, CAST_FROM_FN_PTR(address, new_m

1240         oop_maps = new OopMapSet();
1241         oop_maps->add_gc_map(call_offset, map);
1242         restore_live_registers_except_rax(sasm);

1244         // rax,: new multi array
1245         __ verify_oop(rax);
1246       }
1247       break;

1249     case register_finalizer_id:
1250       {
1251         __ set_info("register_finalizer", dont_gc_arguments);

1253         // This is called via call_runtime so the arguments
1254         // will be place in C abi locations

1256 #ifdef _LP64
1257         __ verify_oop(c_rarg0);
1258         __ mov(rax, c_rarg0);
1259 #else
1260         // The object is passed on the stack and we haven't pushed a
1261         // frame yet so it's one work away from top of stack.
1262         __ movptr(rax, Address(rsp, 1 * BytesPerWord));
1263         __ verify_oop(rax);
1264 #endif // _LP64

1266         // load the klass and check the has finalizer flag
1267         Label register_finalizer;
1268         Register t = rsi;
1269         __ load_klass(t, rax);
1270         __ movl(t, Address(t, Klass::access_flags_offset_in_bytes() + sizeof(oop
1271         __ testl(t, JVM_ACC_HAS_FINALIZER);
1272         __ jcc(Assembler::notZero, register_finalizer);
1273         __ ret(0);

1275         __ bind(register_finalizer);
1276         __ enter();
1277         OopMap* oop_map = save_live_registers(sasm, 2 /*num_rt_args */);
1278         int call_offset = __ call_RT(noreg, noreg,
1279                                      CAST_FROM_FN_PTR(address, SharedRuntime::re
1280         oop_maps = new OopMapSet();
1281         oop_maps->add_gc_map(call_offset, oop_map);

1283         // Now restore all the live registers
1284         restore_live_registers(sasm);

1286         __ leave();
1287         __ ret(0);
```



```
1288       }
1289       break;

1291     case throw_range_check_failed_id:
1292       { StubFrame f(sasm, "range_check_failed", dont_gc_arguments);
1293         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
1294       }
1295       break;

1297     case throw_index_exception_id:
1298       { StubFrame f(sasm, "index_range_check_failed", dont_gc_arguments);
1299         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
1300       }
1301       break;

1303     case throw_div0_exception_id:
1304       { StubFrame f(sasm, "throw_div0_exception", dont_gc_arguments);
1305         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
1306       }
1307       break;

1309     case throw_null_pointer_exception_id:
1310       { StubFrame f(sasm, "throw_null_pointer_exception", dont_gc_arguments);
1311         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
1312       }
1313       break;

1315     case handle_exception_nofpu_id:
1316     case handle_exception_id:
1317       { StubFrame f(sasm, "handle_exception", dont_gc_arguments);
1318         oop_maps = generate_handle_exception(id, sasm);
1319       }
1320       break;

1322     case handle_exception_from_callee_id:
1323       { StubFrame f(sasm, "handle_exception_from_callee", dont_gc_arguments);
1324         oop_maps = generate_handle_exception(id, sasm);
1325       }
1326       break;

1328     case unwind_exception_id:
1329       { __ set_info("unwind_exception", dont_gc_arguments);
1330         // note: no stubframe since we are about to leave the current
1331         //       activation and we are calling a leaf VM function only.
1332         generate_unwind_exception(sasm);
1333       }
1334       break;

1336     case throw_array_store_exception_id:
1337       { StubFrame f(sasm, "throw_array_store_exception", dont_gc_arguments);
1338         // tos + 0: link
1339         //     + 1: return address
1340         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
1341       }
1342       break;

1344     case throw_class_cast_exception_id:
1345       { StubFrame f(sasm, "throw_class_cast_exception", dont_gc_arguments);
1346         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
1347       }
1348       break;

1350     case throw_incompatible_class_change_error_id:
1351       { StubFrame f(sasm, "throw_incompatible_class_cast_exception", dont_gc_arg
1352         oop_maps = generate_exception_throw(sasm, CAST_FROM_FN_PTR(address, thro
1353       }
```

```
1354       break;
1356     case slow_subtype_check_id:
1357       {
1358         // Typical calling sequence:
1359         // __ push(klass_RInfo);  // object klass or other subclass
1360         // __ push(sup_k_RInfo);  // array element klass or other superclass
1361         // __ call(slow_subtype_check);
1362         // Note that the subclass is pushed first, and is therefore deepest.
1363         // Previous versions of this code reversed the names 'sub' and 'super'.
1364         // This was operationally harmless but made the code unreadable.
1365         enum layout {
1366           rax_off, SLOT2(raxH_off)
1367           rcx_off, SLOT2(rcxH_off)
1368           rsi_off, SLOT2(rsiH_off)
1369           rdi_off, SLOT2(rdiH_off)
1370           // saved_rbp_off, SLOT2(saved_rbpH_off)
1371           return_off, SLOT2(returnH_off)
1372           sup_k_off, SLOT2(sup_kH_off)
1373           klass_off, SLOT2(superH_off)
1374           framesize,
1375           result_off = klass_off  // deepest argument is also the return value
1376         };
1378         __ set_info("slow_subtype_check", dont_gc_arguments);
1379         __ push(rdi);
1380         __ push(rsi);
1381         __ push(rcx);
1382         __ push(rax);
1384         // This is called by pushing args and not with C abi
1385         __ movptr(rsi, Address(rsp, (klass_off) * VMRegImpl::stack_slot_size));
1386         __ movptr(rax, Address(rsp, (sup_k_off) * VMRegImpl::stack_slot_size));
1388         Label miss;
1389         __ check_klass_subtype_slow_path(rsi, rax, rcx, rdi, NULL, &miss);
1391         // fallthrough on success:
1392         __ movptr(Address(rsp, (result_off) * VMRegImpl::stack_slot_size), 1); /
1393         __ pop(rax);
1394         __ pop(rcx);
1395         __ pop(rsi);
1396         __ pop(rdi);
1397         __ ret(0);
1399         __ bind(miss);
1400         __ movptr(Address(rsp, (result_off) * VMRegImpl::stack_slot_size), NULL_
1401         __ pop(rax);
1402         __ pop(rcx);
1403         __ pop(rsi);
1404         __ pop(rdi);
1405         __ ret(0);
1406       }
1407       break;
1409     case monitorenter_nofpu_id:
1410       save_fpu_registers = false;
1411       // fall through
1412     case monitorenter_id:
1413       {
1414         StubFrame f(sasm, "monitorenter", dont_gc_arguments);
1415         OopMap* map = save_live_registers(sasm, 3, save_fpu_registers);
1417         // Called with store_parameter and not C abi
1419         f.load_argument(1, rax); // rax,: object
```



```
1420         f.load_argument(0, rbx); // rbx,: lock address
1422         int call_offset = __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, mon
1424         oop_maps = new OopMapSet();
1425         oop_maps->add_gc_map(call_offset, map);
1426         restore_live_registers(sasm, save_fpu_registers);
1427       }
1428       break;
1430     case monitorexit_nofpu_id:
1431       save_fpu_registers = false;
1432       // fall through
1433     case monitorexit_id:
1434       {
1435         StubFrame f(sasm, "monitorexit", dont_gc_arguments);
1436         OopMap* map = save_live_registers(sasm, 2, save_fpu_registers);
1438         // Called with store_parameter and not C abi
1440         f.load_argument(0, rax); // rax,: lock address
1442         // note: really a leaf routine but must setup last java sp
1443         //       => use call_RT for now (speed can be improved by
1444         //       doing last java sp setup manually)
1445         int call_offset = __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, mon
1447         oop_maps = new OopMapSet();
1448         oop_maps->add_gc_map(call_offset, map);
1449         restore_live_registers(sasm, save_fpu_registers);
1450       }
1451       break;
1452 #endif /* ! codereview */
1454     case deoptimize_id:
1455       {
1456         StubFrame f(sasm, "deoptimize", dont_gc_arguments);
1457         const int num_rt_args = 1;  // thread
1458         OopMap* oop_map = save_live_registers(sasm, num_rt_args);
1459         int call_offset = __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, deo
1460         oop_maps = new OopMapSet();
1461         oop_maps->add_gc_map(call_offset, oop_map);
1462         restore_live_registers(sasm);
1463         DeoptimizationBlob* deopt_blob = SharedRuntime::deopt_blob();
1464         assert(deopt_blob != NULL, "deoptimization blob must have been created")
1465         __ leave();
1466         __ jump(RuntimeAddress(deopt_blob->unpack_with_reexecution()));
1467 #endif /* ! codereview */
1468       }
1469       break;
1471     case access_field_patching_id:
1472       { StubFrame f(sasm, "access_field_patching", dont_gc_arguments);
1473         // we should set up register map
1474         oop_maps = generate_patching(sasm, CAST_FROM_FN_PTR(address, access_fiel
1475       }
1476       break;
1478     case load_klass_patching_id:
1479       { StubFrame f(sasm, "load_klass_patching", dont_gc_arguments);
1480         // we should set up register map
1481         oop_maps = generate_patching(sasm, CAST_FROM_FN_PTR(address, move_klass_
1482       }
1483       break;
1485     case dtrace_object_alloc_id:
```

```
1486       { // rax,: object
1487         StubFrame f(sasm, "dtrace_object_alloc", dont_gc_arguments);
1488         // we can't gc here so skip the oopmap but make sure that all
1489         // the live registers get saved.
1490         save_live_registers(sasm, 1);

1492         __ NOT_LP64(push(rax)) LP64_ONLY(mov(c_rarg0, rax));
1493         __ call(RuntimeAddress(CAST_FROM_FN_PTR(address, SharedRuntime::dtrace_o
1494         NOT_LP64(__ pop(rax));

1496         restore_live_registers(sasm);
1497       }
1498       break;

1500     case fpu2long_stub_id:
1501       {
1502         // rax, and rdx are destroyed, but should be free since the result is re
1503         // preserve rsi,ecx
1504         __ push(rsi);
1505         __ push(rcx);
1506         LP64_ONLY(__ push(rdx);)

1508         // check for NaN
1509         Label return0, do_return, return_min_jlong, do_convert;

1511         Address value_high_word(rsp, wordSize + 4);
1512         Address value_low_word(rsp, wordSize);
1513         Address result_high_word(rsp, 3*wordSize + 4);
1514         Address result_low_word(rsp, 3*wordSize);

1516         __ subptr(rsp, 32);                   // more than enough on 32bit
1517         __ fst_d(value_low_word);
1518         __ movl(rax, value_high_word);
1519         __ andl(rax, 0x7ff00000);
1520         __ cmpl(rax, 0x7ff00000);
1521         __ jcc(Assembler::notEqual, do_convert);
1522         __ movl(rax, value_high_word);
1523         __ andl(rax, 0xfffff);
1524         __ orl(rax, value_low_word);
1525         __ jcc(Assembler::notZero, return0);

1527         __ bind(do_convert);
1528         __ fnstcw(Address(rsp, 0));
1529         __ movzwl(rax, Address(rsp, 0));
1530         __ orl(rax, 0xc00);
1531         __ movw(Address(rsp, 2), rax);
1532         __ fldcw(Address(rsp, 2));
1533         __ fwait();
1534         __ fistp_d(result_low_word);
1535         __ fldcw(Address(rsp, 0));
1536         __ fwait();
1537         // This gets the entire long in rax on 64bit
1538         __ movptr(rax, result_low_word);
1539         // testing of high bits
1540         __ movl(rdx, result_high_word);
1541         __ mov(rcx, rax);
1542         // What the heck is the point of the next instruction???
1543         __ xorl(rcx, 0x0);
1544         __ movl(rsi, 0x80000000);
1545         __ xorl(rsi, rdx);
1546         __ orl(rcx, rsi);
1547         __ jcc(Assembler::notEqual, do_return);
1548         __ fldz();
1549         __ fcomp_d(value_low_word);
1550         __ fnstsw_ax();
1551 #ifdef _LP64
```



```
1552         __ testl(rax, 0x4100);  // ZF & CF == 0
1553         __ jcc(Assembler::equal, return_min_jlong);
1554 #else
1555         __ sahf();
1556         __ jcc(Assembler::above, return_min_jlong);
1557 #endif // _LP64
1558         // return max_jlong
1559 #ifndef _LP64
1560         __ movl(rdx, 0x7fffffff);
1561         __ movl(rax, 0xffffffff);
1562 #else
1563         __ mov64(rax, CONST64(0x7fffffffffffffff));
1564 #endif // _LP64
1565         __ jmp(do_return);

1567         __ bind(return_min_jlong);
1568 #ifndef _LP64
1569         __ movl(rdx, 0x80000000);
1570         __ xorl(rax, rax);
1571 #else
1572         __ mov64(rax, CONST64(0x8000000000000000));
1573 #endif // _LP64
1574         __ jmp(do_return);

1576         __ bind(return0);
1577         __ fpop();
1578 #ifndef _LP64
1579         __ xorptr(rdx,rdx);
1580         __ xorptr(rax,rax);
1581 #else
1582         __ xorptr(rax, rax);
1583 #endif // _LP64

1585         __ bind(do_return);
1586         __ addptr(rsp, 32);
1587         LP64_ONLY(__ pop(rdx);)
1588         __ pop(rcx);
1589         __ pop(rsi);
1590         __ ret(0);
1591       }
1592       break;

1594 #ifndef SERIALGC
1595     case g1_pre_barrier_slow_id:
1596       {
1597         StubFrame f(sasm, "g1_pre_barrier", dont_gc_arguments);
1598         // arg0 : previous value of memory

1600         BarrierSet* bs = Universe::heap()->barrier_set();
1601         if (bs->kind() != BarrierSet::G1SATBCTLogging) {
1602           __ movptr(rax, (int)id);
1603           __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, unimplemented_entry
1604           __ should_not_reach_here();
1605           break;
1606         }
1607         __ push(rax);
1608         __ push(rdx);

1610         const Register pre_val = rax;
1611         const Register thread = NOT_LP64(rax) LP64_ONLY(r15_thread);
1612         const Register tmp = rdx;

1614         NOT_LP64(__ get_thread(thread);)

1616         Address in_progress(thread, in_bytes(JavaThread::satb_mark_queue_offset(
1617                                              PtrQueue::byte_offset_of_active()))
```

```
1619         Address queue_index(thread, in_bytes(JavaThread::satb_mark_queue_offset(
1620                                              PtrQueue::byte_offset_of_index()));
1621         Address buffer(thread, in_bytes(JavaThread::satb_mark_queue_offset() +
1622                                         PtrQueue::byte_offset_of_buf()));


1625         Label done;
1626         Label runtime;

1628         // Can we store original value in the thread's buffer?

1630 #ifdef _LP64
1631         __ movslq(tmp, queue_index);
1632         __ cmpq(tmp, 0);
1633 #else
1634         __ cmpl(queue_index, 0);
1635 #endif
1636         __ jcc(Assembler::equal, runtime);
1637 #ifdef _LP64
1638         __ subq(tmp, wordSize);
1639         __ movl(queue_index, tmp);
1640         __ addq(tmp, buffer);
1641 #else
1642         __ subl(queue_index, wordSize);
1643         __ movl(tmp, buffer);
1644         __ addl(tmp, queue_index);
1645 #endif

1647         // prev_val (rax)
1648         f.load_argument(0, pre_val);
1649         __ movptr(Address(tmp, 0), pre_val);
1650         __ jmp(done);

1652         __ bind(runtime);
1653         __ push(rcx);
1654 #ifdef _LP64
1655         __ push(r8);
1656         __ push(r9);
1657         __ push(r10);
1658         __ push(r11);
1659 #  ifndef _WIN64
1660         __ push(rdi);
1661         __ push(rsi);
1662 #  endif
1663 #endif
1664         // load the pre-value
1665         f.load_argument(0, rcx);
1666         __ call_VM_leaf(CAST_FROM_FN_PTR(address, SharedRuntime::g1_wb_pre), rcx
1667 #ifdef _LP64
1668 #  ifndef _WIN64
1669         __ pop(rsi);
1670         __ pop(rdi);
1671 #  endif
1672         __ pop(r11);
1673         __ pop(r10);
1674         __ pop(r9);
1675         __ pop(r8);
1676 #endif
1677         __ pop(rcx);
1678         __ bind(done);

1680         __ pop(rdx);
1681         __ pop(rax);
1682       }
1683       break;
```



```
1685     case g1_post_barrier_slow_id:
1686       {
1687         StubFrame f(sasm, "g1_post_barrier", dont_gc_arguments);


1690         // arg0: store_address
1691         Address store_addr(rbp, 2*BytesPerWord);

1693         BarrierSet* bs = Universe::heap()->barrier_set();
1694         CardTableModRefBS* ct = (CardTableModRefBS*)bs;
1695         Label done;
1696         Label runtime;

1698         // At this point we know new_value is non-NULL and the new_value crosses
1699         // Must check to see if card is already dirty

1701         const Register thread = NOT_LP64(rax) LP64_ONLY(r15_thread);

1703         Address queue_index(thread, in_bytes(JavaThread::dirty_card_queue_offset
1704                                              PtrQueue::byte_offset_of_index()));
1705         Address buffer(thread, in_bytes(JavaThread::dirty_card_queue_offset() +
1706                                         PtrQueue::byte_offset_of_buf()));

1708         __ push(rax);
1709         __ push(rcx);

1711         NOT_LP64(__ get_thread(thread);)
1712         ExternalAddress cardtable((address)ct->byte_map_base);
1713         assert(sizeof(*ct->byte_map_base) == sizeof(jbyte), "adjust this code");

1715         const Register card_addr = rcx;
1716 #ifdef _LP64
1717         const Register tmp = rscratch1;
1718         f.load_argument(0, card_addr);
1719         __ shrq(card_addr, CardTableModRefBS::card_shift);
1720         __ lea(tmp, cardtable);
1721         // get the address of the card
1722         __ addq(card_addr, tmp);
1723 #else
1724         const Register card_index = rcx;
1725         f.load_argument(0, card_index);
1726         __ shrl(card_index, CardTableModRefBS::card_shift);

1728         Address index(noreg, card_index, Address::times_1);
1729         __ leal(card_addr, __ as_Address(ArrayAddress(cardtable, index)));
1730 #endif

1732         __ cmpb(Address(card_addr, 0), 0);
1733         __ jcc(Assembler::equal, done);

1735         // storing region crossing non-NULL, card is clean.
1736         // dirty card and log.

1738         __ movb(Address(card_addr, 0), 0);

1740         __ cmpl(queue_index, 0);
1741         __ jcc(Assembler::equal, runtime);
1742         __ subl(queue_index, wordSize);

1744         const Register buffer_addr = rbx;
1745         __ push(rbx);

1747         __ movptr(buffer_addr, buffer);

1749 #ifdef _LP64
```

```
1750         __ movslq(rscratch1, queue_index);
1751         __ addptr(buffer_addr, rscratch1);
1752 #else
1753         __ addptr(buffer_addr, queue_index);
1754 #endif
1755         __ movptr(Address(buffer_addr, 0), card_addr);

1757         __ pop(rbx);
1758         __ jmp(done);

1760         __ bind(runtime);
1761         __ push(rdx);
1762 #ifdef _LP64
1763         __ push(r8);
1764         __ push(r9);
1765         __ push(r10);
1766         __ push(r11);
1767 #  ifndef _WIN64
1768         __ push(rdi);
1769         __ push(rsi);
1770 #  endif
1771 #endif
1772         __ call_VM_leaf(CAST_FROM_FN_PTR(address, SharedRuntime::g1_wb_post), ca
1773 #ifdef _LP64
1774 #  ifndef _WIN64
1775         __ pop(rsi);
1776         __ pop(rdi);
1777 #  endif
1778         __ pop(r11);
1779         __ pop(r10);
1780         __ pop(r9);
1781         __ pop(r8);
1782 #endif
1783         __ pop(rdx);
1784         __ bind(done);

1786         __ pop(rcx);
1787         __ pop(rax);

1789       }
1790       break;
1791 #endif // !SERIALGC

1793     default:
1794       { StubFrame f(sasm, "unimplemented entry", dont_gc_arguments);
1795         __ movptr(rax, (int)id);
1796         __ call_RT(noreg, noreg, CAST_FROM_FN_PTR(address, unimplemented_entry),
1797         __ should_not_reach_here();
1798       }
1799       break;
1800   }
1801   return oop_maps;
1802 }

1804 #undef __

1806 const char *Runtime1::pd_name_for_address(address entry) {
1807   return "<unknown function>";
1808 }
```