```
*********************************************************
   40450 Fri Mar 18 09:33:47 2011
new/src/cpu/sparc/vm/methodHandles_sparc.cpp
*********************************************************
______unchanged_portion_omitted_

 393 //------------------------------------------------------------------------------
 394 // MethodHandles::generate_method_handle_stub
 395 //
 396 // Generate an "entry" field for a method handle.
 397 // This determines how the method handle will respond to calls.
 398 void MethodHandles::generate_method_handle_stub(MacroAssembler* _masm, MethodHan
 399   // Here is the register state during an interpreted call,
 400   // as set up by generate_method_handle_interpreter_entry():
 401   // - G5: garbage temp (was MethodHandle.invoke methodOop, unused)
 402   // - G3: receiver method handle
 403   // - O5_savedSP: sender SP (must preserve)

 405   const Register O0_argslot = O0;
 406   const Register O1_scratch = O1;
 407   const Register O2_scratch = O2;
 408   const Register O3_scratch = O3;
 409   const Register G5_index   = G5;

 411   // Argument registers for _raise_exception.
 412   const Register O0_code     = O0;
 413   const Register O1_actual   = O1;
 414   const Register O2_required = O2;

 416   guarantee(java_lang_invoke_MethodHandle::vmentry_offset_in_bytes() != 0, "must

 418   // Some handy addresses:
 419   Address G5_method_fie(    G5_method,        in_bytes(methodOopDesc::from_inter
 420   Address G5_method_fce(    G5_method,        in_bytes(methodOopDesc::from_compi

 422   Address G3_mh_vmtarget(   G3_method_handle, java_lang_invoke_MethodHandle::vmt

 424   Address G3_dmh_vmindex(   G3_method_handle, java_lang_invoke_DirectMethodHandl

 426   Address G3_bmh_vmargslot( G3_method_handle, java_lang_invoke_BoundMethodHandle
 427   Address G3_bmh_argument(  G3_method_handle, java_lang_invoke_BoundMethodHandle

 429   Address G3_amh_vmargslot( G3_method_handle, java_lang_invoke_AdapterMethodHand
 430   Address G3_amh_argument ( G3_method_handle, java_lang_invoke_AdapterMethodHand
 431   Address G3_amh_conversion(G3_method_handle, java_lang_invoke_AdapterMethodHand

 433   const int java_mirror_offset = klassOopDesc::klass_part_offset_in_bytes() + Kl

 435   if (have_entry(ek)) {
 436     __ nop();  // empty stubs make SG sick
 437     return;
 438   }

 440   address interp_entry = __ pc();

 442   trace_method_handle(_masm, entry_name(ek));

 444   switch ((int) ek) {
 445   case _raise_exception:
 446     {
 447       // Not a real MH entry, but rather shared code for raising an
 448       // exception.  Since we use the compiled entry, arguments are
 449       // expected in compiler argument registers.
 450       assert(raise_exception_method(), "must be set");
 451       assert(raise_exception_method()->from_compiled_entry(), "method must be li
```



```
 453       __ mov(O5_savedSP, SP);  // Cut the stack back to where the caller started

 455       Label L_no_method;
 456       // FIXME: fill in _raise_exception_method with a suitable java.lang.invoke
 457       __ set(AddressLiteral((address) &_raise_exception_method), G5_method);
 458       __ ld_ptr(Address(G5_method, 0), G5_method);
 459       __ tst(G5_method);
 460       __ brx(Assembler::zero, false, Assembler::pn, L_no_method);
 461       __ delayed()->nop();

 463       const int jobject_oop_offset = 0;
 464       __ ld_ptr(Address(G5_method, jobject_oop_offset), G5_method);
 465       __ tst(G5_method);
 466       __ brx(Assembler::zero, false, Assembler::pn, L_no_method);
 467       __ delayed()->nop();

 469       __ verify_oop(G5_method);
 470       __ jump_indirect_to(G5_method_fce, O3_scratch);  // jump to compiled entry
 471       __ delayed()->nop();

 473       // Do something that is at least causes a valid throw from the interpreter
 474       __ bind(L_no_method);
 475       __ unimplemented("call throw_WrongMethodType_entry");
 476     }
 477     break;

 479   case _invokestatic_mh:
 480   case _invokespecial_mh:
 481     {
 482       __ load_heap_oop(G3_mh_vmtarget, G5_method);  // target is a methodOop
 483       __ verify_oop(G5_method);
 484       // Same as TemplateTable::invokestatic or invokespecial,
 485       // minus the CP setup and profiling:
 486       if (ek == _invokespecial_mh) {
 487         // Must load & check the first argument before entering the target metho
 488         __ load_method_handle_vmslots(O0_argslot, G3_method_handle, O1_scratch);
 489         __ ld_ptr(__ argument_address(O0_argslot), G3_method_handle);
 490         __ null_check(G3_method_handle);
 491         __ verify_oop(G3_method_handle);
 492       }
 493       __ jump_indirect_to(G5_method_fie, O1_scratch);
 494       __ delayed()->nop();
 495     }
 496     break;

 498   case _invokevirtual_mh:
 499     {
 500       // Same as TemplateTable::invokevirtual,
 501       // minus the CP setup and profiling:

 503       // Pick out the vtable index and receiver offset from the MH,
 504       // and then we can discard it:
 505       __ load_method_handle_vmslots(O0_argslot, G3_method_handle, O1_scratch);
 506       __ ldsw(G3_dmh_vmindex, G5_index);
 507       // Note:  The verifier allows us to ignore G3_mh_vmtarget.
 508       __ ld_ptr(__ argument_address(O0_argslot, -1), G3_method_handle);
 509       __ null_check(G3_method_handle, oopDesc::klass_offset_in_bytes());

 511       // Get receiver klass:
 512       Register O0_klass = O0_argslot;
 513       __ load_klass(G3_method_handle, O0_klass);
 514       __ verify_oop(O0_klass);

 516       // Get target methodOop & entry point:
 517       const int base = instanceKlass::vtable_start_offset() * wordSize;
 518       assert(vtableEntry::size() * wordSize == wordSize, "adjust the scaling in
```

```
520       __ sll_ptr(G5_index, LogBytesPerWord, G5_index);
521       __ add(O0_klass, G5_index, O0_klass);
522       Address vtable_entry_addr(O0_klass, base + vtableEntry::method_offset_in_b
523       __ ld_ptr(vtable_entry_addr, G5_method);

525       __ verify_oop(G5_method);
526       __ jump_indirect_to(G5_method_fie, O1_scratch);
527       __ delayed()->nop();
528     }
529     break;

531   case _invokeinterface_mh:
532     {
533       // Same as TemplateTable::invokeinterface,
534       // minus the CP setup and profiling:
535       __ load_method_handle_vmslots(O0_argslot, G3_method_handle, O1_scratch);
536       Register O1_intf  = O1_scratch;
537       __ load_heap_oop(G3_mh_vmtarget, O1_intf);
538       __ ldsw(G3_dmh_vmindex, G5_index);
539       __ ld_ptr(__ argument_address(O0_argslot, -1), G3_method_handle);
540       __ null_check(G3_method_handle, oopDesc::klass_offset_in_bytes());

542       // Get receiver klass:
543       Register O0_klass = O0_argslot;
544       __ load_klass(G3_method_handle, O0_klass);
545       __ verify_oop(O0_klass);

547       // Get interface:
548       Label no_such_interface;
549       __ verify_oop(O1_intf);
550       __ lookup_interface_method(O0_klass, O1_intf,
551                                  // Note: next two args must be the same:
552                                  G5_index, G5_method,
553                                  O2_scratch,
554                                  O3_scratch,
555                                  no_such_interface);

557       __ verify_oop(G5_method);
558       __ jump_indirect_to(G5_method_fie, O1_scratch);
559       __ delayed()->nop();

561       __ bind(no_such_interface);
562       // Throw an exception.
563       // For historical reasons, it will be IncompatibleClassChangeError.
564       __ unimplemented("not tested yet");
565       __ ld_ptr(Address(O1_intf, java_mirror_offset), O2_required);  // required
566       __ mov(   O0_klass,                             O1_actual);    // bad rece
567       __ jump_to(AddressLiteral(from_interpreted_entry(_raise_exception)), O3_sc
568       __ delayed()->mov(Bytecodes::_invokeinterface,  O0_code);      // who is c
569     }
570     break;

572   case _bound_ref_mh:
573   case _bound_int_mh:
574   case _bound_long_mh:
575   case _bound_ref_direct_mh:
576   case _bound_int_direct_mh:
577   case _bound_long_direct_mh:
578     {
579       const bool direct_to_method = (ek >= _bound_ref_direct_mh);
580       BasicType arg_type  = T_ILLEGAL;
581       int       arg_mask  = _INSERT_NO_MASK;
582       int       arg_slots = -1;
583       get_ek_bound_mh_info(ek, arg_type, arg_mask, arg_slots);
```



```
585       // Make room for the new argument:
586       __ ldsw(G3_bmh_vmargslot, O0_argslot);
587       __ add(Gargs, __ argument_offset(O0_argslot), O0_argslot);

589       insert_arg_slots(_masm, arg_slots * stack_move_unit(), arg_mask, O0_argslo

591       // Store bound argument into the new stack slot:
592       __ load_heap_oop(G3_bmh_argument, O1_scratch);
593       if (arg_type == T_OBJECT) {
594         __ st_ptr(O1_scratch, Address(O0_argslot, 0));
595       } else {
596         Address prim_value_addr(O1_scratch, java_lang_boxing_object::value_offse
597         const int arg_size = type2aelembytes(arg_type);
598         __ load_sized_value(prim_value_addr, O2_scratch, arg_size, is_signed_sub
599         __ store_sized_value(O2_scratch, Address(O0_argslot, 0), arg_size);  //
600       }

602       if (direct_to_method) {
603         __ load_heap_oop(G3_mh_vmtarget, G5_method);  // target is a methodOop
604         __ verify_oop(G5_method);
605         __ jump_indirect_to(G5_method_fie, O1_scratch);
606         __ delayed()->nop();
607       } else {
608         __ load_heap_oop(G3_mh_vmtarget, G3_method_handle);  // target is a meth
609         __ verify_oop(G3_method_handle);
610         __ jump_to_method_handle_entry(G3_method_handle, O1_scratch);
611       }
612     }
613     break;

615   case _adapter_retype_only:
616   case _adapter_retype_raw:
617     // Immediately jump to the next MH layer:
618     __ load_heap_oop(G3_mh_vmtarget, G3_method_handle);
619     __ jump_to_method_handle_entry(G3_method_handle, O1_scratch);
620     // This is OK when all parameter types widen.
621     // It is also OK when a return type narrows.
622     break;

624   case _adapter_check_cast:
625     {
626       // Temps:
627       Register G5_klass = G5_index;  // Interesting AMH data.

629       // Check a reference argument before jumping to the next layer of MH:
630       __ ldsw(G3_amh_vmargslot, O0_argslot);
631       Address vmarg = __ argument_address(O0_argslot);

633       // What class are we casting to?
634       __ load_heap_oop(G3_amh_argument, G5_klass);  // This is a Class object!
635       __ load_heap_oop(Address(G5_klass, java_lang_Class::klass_offset_in_bytes(

637       Label done;
638       __ ld_ptr(vmarg, O1_scratch);
639       __ tst(O1_scratch);
640       __ brx(Assembler::zero, false, Assembler::pn, done);  // No cast if null.
641       __ delayed()->nop();
642       __ load_klass(O1_scratch, O1_scratch);

644       // Live at this point:
645       // - G5_klass        :  klass required by the target method
646       // - O1_scratch      :  argument klass to test
647       // - G3_method_handle:  adapter method handle
648       __ check_klass_subtype(O1_scratch, G5_klass, O0_argslot, O2_scratch, done)

650       // If we get here, the type check failed!
```

```
651       __ load_heap_oop(G3_amh_argument,        O2_required);  // required class
652       __ ld_ptr(       vmarg,                  O1_actual);    // bad object
653       __ jump_to(AddressLiteral(from_interpreted_entry(_raise_exception)), O3_sc
654       __ delayed()->mov(Bytecodes::_checkcast, O0_code);      // who is complain

656       __ bind(done);
657       // Get the new MH:
658       __ load_heap_oop(G3_mh_vmtarget, G3_method_handle);
659       __ jump_to_method_handle_entry(G3_method_handle, O1_scratch);
660     }
661     break;

663   case _adapter_prim_to_prim:
664   case _adapter_ref_to_prim:
665     // Handled completely by optimized cases.
666     __ stop("init_AdapterMethodHandle should not issue this");
667     break;

669   case _adapter_opt_i2i:        // optimized subcase of adapt_prim_to_prim
670 //case _adapter_opt_f2i:        // optimized subcase of adapt_prim_to_prim
671   case _adapter_opt_l2i:        // optimized subcase of adapt_prim_to_prim
672   case _adapter_opt_unboxi:     // optimized subcase of adapt_ref_to_prim
673     {
674       // Perform an in-place conversion to int or an int subword.
675       __ ldsw(G3_amh_vmargslot, O0_argslot);
676       Address value;
677       Address vmarg = __ argument_address(O0_argslot);
678       bool value_left_justified = false;

680       switch (ek) {
681       case _adapter_opt_i2i:
682         value = vmarg;
683         break;
684       case _adapter_opt_l2i:
685         {
686           // just delete the extra slot
687 #ifdef _LP64
688           // In V9, longs are given 2 64-bit slots in the interpreter, but the
689           // data is passed in only 1 slot.
690           // Keep the second slot.
691           __ add(Gargs, __ argument_offset(O0_argslot, -1), O0_argslot);
692           remove_arg_slots(_masm, -stack_move_unit(), O0_argslot, O1_scratch, O2
693           value = Address(O0_argslot, 4);  // Get least-significant 32-bit of 64
694           vmarg = Address(O0_argslot, Interpreter::stackElementSize);
695 #else
696           // Keep the first slot.
697           __ add(Gargs, __ argument_offset(O0_argslot), O0_argslot);
698           remove_arg_slots(_masm, -stack_move_unit(), O0_argslot, O1_scratch, O2
699           value = Address(O0_argslot, 0);
700           vmarg = value;
701 #endif
702         }
703         break;
704       case _adapter_opt_unboxi:
705         {
706           // Load the value up from the heap.
707           __ ld_ptr(vmarg, O1_scratch);
708           int value_offset = java_lang_boxing_object::value_offset_in_bytes(T_IN
709 #ifdef ASSERT
710           for (int bt = T_BOOLEAN; bt < T_INT; bt++) {
711             if (is_subword_type(BasicType(bt)))
712               assert(value_offset == java_lang_boxing_object::value_offset_in_by
713           }
714 #endif
715           __ null_check(O1_scratch, value_offset);
716           value = Address(O1_scratch, value_offset);
```



```
717 #ifdef _BIG_ENDIAN
718           // Values stored in objects are packed.
719           value_left_justified = true;
720 #endif
721         }
722         break;
723       default:
724         ShouldNotReachHere();
725       }

727       // This check is required on _BIG_ENDIAN
728       Register G5_vminfo = G5_index;
729       __ ldsw(G3_amh_conversion, G5_vminfo);
730       assert(CONV_VMINFO_SHIFT == 0, "preshifted");

732       // Original 32-bit vmdata word must be of this form:
733       // | MBZ:6 | signBitCount:8 | srcDstTypes:8 | conversionOp:8 |
734       __ lduw(value, O1_scratch);
735       if (!value_left_justified)
736         __ sll(O1_scratch, G5_vminfo, O1_scratch);
737       Label zero_extend, done;
738       __ btst(CONV_VMINFO_SIGN_FLAG, G5_vminfo);
739       __ br(Assembler::zero, false, Assembler::pn, zero_extend);
740       __ delayed()->nop();

742       // this path is taken for int->byte, int->short
743       __ sra(O1_scratch, G5_vminfo, O1_scratch);
744       __ ba(false, done);
745       __ delayed()->nop();

747       __ bind(zero_extend);
748       // this is taken for int->char
749       __ srl(O1_scratch, G5_vminfo, O1_scratch);

751       __ bind(done);
752       __ st(O1_scratch, vmarg);

754       // Get the new MH:
755       __ load_heap_oop(G3_mh_vmtarget, G3_method_handle);
756       __ jump_to_method_handle_entry(G3_method_handle, O1_scratch);
757     }
758     break;

760   case _adapter_opt_i2l:        // optimized subcase of adapt_prim_to_prim
761   case _adapter_opt_unboxl:     // optimized subcase of adapt_ref_to_prim
762     {
763       // Perform an in-place int-to-long or ref-to-long conversion.
764       __ ldsw(G3_amh_vmargslot, O0_argslot);

766       // On big-endian machine we duplicate the slot and store the MSW
767       // in the first slot.
768       __ add(Gargs, __ argument_offset(O0_argslot, 1), O0_argslot);

770       insert_arg_slots(_masm, stack_move_unit(), _INSERT_INT_MASK, O0_argslot, O

772       Address arg_lsw(O0_argslot, 0);
773       Address arg_msw(O0_argslot, -Interpreter::stackElementSize);

775       switch (ek) {
776       case _adapter_opt_i2l:
777         {
778 #ifdef _LP64
779           __ ldsw(arg_lsw, O2_scratch);                // Load LSW sign-extende
780 #else
781           __ ldsw(arg_lsw, O3_scratch);                // Load LSW sign-extende
782           __ srlx(O3_scratch, BitsPerInt, O2_scratch);  // Move MSW value to low
```

```
783 #endif
778           __ ldsw(arg_lsw, O2_scratch);                          // Load LSW
779           NOT_LP64(__ srlx(O2_scratch, BitsPerInt, O3_scratch));  // Move high b
784           __ st_long(O2_scratch, arg_msw);              // Uses O2/O3 on !_LP64
785         }
786         break;
787       case _adapter_opt_unboxl:
788         {
789           // Load the value up from the heap.
790           __ ld_ptr(arg_lsw, O1_scratch);
791           int value_offset = java_lang_boxing_object::value_offset_in_bytes(T_LO
792           assert(value_offset == java_lang_boxing_object::value_offset_in_bytes(
793           __ null_check(O1_scratch, value_offset);
794           __ ld_long(Address(O1_scratch, value_offset), O2_scratch);  // Uses O2
795           __ st_long(O2_scratch, arg_msw);
796         }
797         break;
798       default:
799         ShouldNotReachHere();
800       }

802       __ load_heap_oop(G3_mh_vmtarget, G3_method_handle);
803       __ jump_to_method_handle_entry(G3_method_handle, O1_scratch);
804     }
805     break;

807   case _adapter_opt_f2d:        // optimized subcase of adapt_prim_to_prim
808   case _adapter_opt_d2f:        // optimized subcase of adapt_prim_to_prim
809     {
810       // perform an in-place floating primitive conversion
811       __ unimplemented(entry_name(ek));
812     }
813     break;

815   case _adapter_prim_to_ref:
816     __ unimplemented(entry_name(ek)); // %%% FIXME: NYI
817     break;

819   case _adapter_swap_args:
820   case _adapter_rot_args:
821     // handled completely by optimized cases
822     __ stop("init_AdapterMethodHandle should not issue this");
823     break;

825   case _adapter_opt_swap_1:
826   case _adapter_opt_swap_2:
827   case _adapter_opt_rot_1_up:
828   case _adapter_opt_rot_1_down:
829   case _adapter_opt_rot_2_up:
830   case _adapter_opt_rot_2_down:
831     {
832       int swap_bytes = 0, rotate = 0;
833       get_ek_adapter_opt_swap_rot_info(ek, swap_bytes, rotate);

835       // 'argslot' is the position of the first argument to swap.
836       __ ldsw(G3_amh_vmargslot, O0_argslot);
837       __ add(Gargs, __ argument_offset(O0_argslot), O0_argslot);

839       // 'vminfo' is the second.
840       Register O1_destslot = O1_scratch;
841       __ ldsw(G3_amh_conversion, O1_destslot);
842       assert(CONV_VMINFO_SHIFT == 0, "preshifted");
843       __ and3(O1_destslot, CONV_VMINFO_MASK, O1_destslot);
844       __ add(Gargs, __ argument_offset(O1_destslot), O1_destslot);

846       if (!rotate) {
```



```
847         for (int i = 0; i < swap_bytes; i += wordSize) {
848           __ ld_ptr(Address(O0_argslot,  i), O2_scratch);
849           __ ld_ptr(Address(O1_destslot, i), O3_scratch);
850           __ st_ptr(O3_scratch, Address(O0_argslot,  i));
851           __ st_ptr(O2_scratch, Address(O1_destslot, i));
852         }
853       } else {
854         // Save the first chunk, which is going to get overwritten.
855         switch (swap_bytes) {
856         case 4 : __ lduw(Address(O0_argslot, 0), O2_scratch); break;
857         case 16: __ ldx( Address(O0_argslot, 8), O3_scratch); //fall-thru
858         case 8 : __ ldx( Address(O0_argslot, 0), O2_scratch); break;
859         default: ShouldNotReachHere();
860         }

862         if (rotate > 0) {
863           // Rorate upward.
864           __ sub(O0_argslot, swap_bytes, O0_argslot);
865 #if ASSERT
866           {
867             // Verify that argslot > destslot, by at least swap_bytes.
868             Label L_ok;
869             __ cmp(O0_argslot, O1_destslot);
870             __ brx(Assembler::greaterEqualUnsigned, false, Assembler::pt, L_ok);
871             __ delayed()->nop();
872             __ stop("source must be above destination (upward rotation)");
873             __ bind(L_ok);
874           }
875 #endif
876           // Work argslot down to destslot, copying contiguous data upwards.
877           // Pseudo-code:
878           //   argslot  = src_addr - swap_bytes
879           //   destslot = dest_addr
880           //   while (argslot >= destslot) {
881           //     *(argslot + swap_bytes) = *(argslot + 0);
882           //     argslot--;
883           //   }
884           Label loop;
885           __ bind(loop);
886           __ ld_ptr(Address(O0_argslot, 0), G5_index);
887           __ st_ptr(G5_index, Address(O0_argslot, swap_bytes));
888           __ sub(O0_argslot, wordSize, O0_argslot);
889           __ cmp(O0_argslot, O1_destslot);
890           __ brx(Assembler::greaterEqualUnsigned, false, Assembler::pt, loop);
891           __ delayed()->nop();  // FILLME
892         } else {
893           __ add(O0_argslot, swap_bytes, O0_argslot);
894 #if ASSERT
895           {
896             // Verify that argslot < destslot, by at least swap_bytes.
897             Label L_ok;
898             __ cmp(O0_argslot, O1_destslot);
899             __ brx(Assembler::lessEqualUnsigned, false, Assembler::pt, L_ok);
900             __ delayed()->nop();
901             __ stop("source must be above destination (upward rotation)");
902             __ bind(L_ok);
903           }
904 #endif
905           // Work argslot up to destslot, copying contiguous data downwards.
906           // Pseudo-code:
907           //   argslot  = src_addr + swap_bytes
908           //   destslot = dest_addr
909           //   while (argslot >= destslot) {
910           //     *(argslot - swap_bytes) = *(argslot + 0);
911           //     argslot++;
912           //   }
```

```
913           Label loop;
914           __ bind(loop);
915           __ ld_ptr(Address(O0_argslot, 0), G5_index);
916           __ st_ptr(G5_index, Address(O0_argslot, -swap_bytes));
917           __ add(O0_argslot, wordSize, O0_argslot);
918           __ cmp(O0_argslot, O1_destslot);
919           __ brx(Assembler::lessEqualUnsigned, false, Assembler::pt, loop);
920           __ delayed()->nop();  // FILLME
921         }

923         // Store the original first chunk into the destination slot, now free.
924         switch (swap_bytes) {
925         case 4 : __ stw(O2_scratch, Address(O1_destslot, 0)); break;
926         case 16: __ stx(O3_scratch, Address(O1_destslot, 8)); // fall-thru
927         case 8 : __ stx(O2_scratch, Address(O1_destslot, 0)); break;
928         default: ShouldNotReachHere();
929         }
930       }

932       __ load_heap_oop(G3_mh_vmtarget, G3_method_handle);
933       __ jump_to_method_handle_entry(G3_method_handle, O1_scratch);
934     }
935     break;

937   case _adapter_dup_args:
938     {
939       // 'argslot' is the position of the first argument to duplicate.
940       __ ldsw(G3_amh_vmargslot, O0_argslot);
941       __ add(Gargs, __ argument_offset(O0_argslot), O0_argslot);

943       // 'stack_move' is negative number of words to duplicate.
944       Register G5_stack_move = G5_index;
945       __ ldsw(G3_amh_conversion, G5_stack_move);
946       __ sra(G5_stack_move, CONV_STACK_MOVE_SHIFT, G5_stack_move);

948       // Remember the old Gargs (argslot[0]).
949       Register O1_oldarg = O1_scratch;
950       __ mov(Gargs, O1_oldarg);

952       // Move Gargs down to make room for dups.
953       __ sll_ptr(G5_stack_move, LogBytesPerWord, G5_stack_move);
954       __ add(Gargs, G5_stack_move, Gargs);

956       // Compute the new Gargs (argslot[0]).
957       Register O2_newarg = O2_scratch;
958       __ mov(Gargs, O2_newarg);

960       // Copy from oldarg[0...] down to newarg[0...]
961       // Pseude-code:
962       //   O1_oldarg  = old-Gargs
963       //   O2_newarg  = new-Gargs
964       //   O0_argslot = argslot
965       //   while (O2_newarg < O1_oldarg) *O2_newarg = *O0_argslot++
966       Label loop;
967       __ bind(loop);
968       __ ld_ptr(Address(O0_argslot, 0), O3_scratch);
969       __ st_ptr(O3_scratch, Address(O2_newarg, 0));
970       __ add(O0_argslot, wordSize, O0_argslot);
971       __ add(O2_newarg,  wordSize, O2_newarg);
972       __ cmp(O2_newarg, O1_oldarg);
973       __ brx(Assembler::less, false, Assembler::pt, loop);
974       __ delayed()->nop();  // FILLME

976       __ load_heap_oop(G3_mh_vmtarget, G3_method_handle);
977       __ jump_to_method_handle_entry(G3_method_handle, O1_scratch);
978     }
```



```
979     break;

981   case _adapter_drop_args:
982     {
983       // 'argslot' is the position of the first argument to nuke.
984       __ ldsw(G3_amh_vmargslot, O0_argslot);
985       __ add(Gargs, __ argument_offset(O0_argslot), O0_argslot);

987       // 'stack_move' is number of words to drop.
988       Register G5_stack_move = G5_index;
989       __ ldsw(G3_amh_conversion, G5_stack_move);
990       __ sra(G5_stack_move, CONV_STACK_MOVE_SHIFT, G5_stack_move);

992       remove_arg_slots(_masm, G5_stack_move, O0_argslot, O1_scratch, O2_scratch,

994       __ load_heap_oop(G3_mh_vmtarget, G3_method_handle);
995       __ jump_to_method_handle_entry(G3_method_handle, O1_scratch);
996     }
997     break;

999   case _adapter_collect_args:
1000    __ unimplemented(entry_name(ek)); // %%% FIXME: NYI
1001    break;

1003  case _adapter_spread_args:
1004    // Handled completely by optimized cases.
1005    __ stop("init_AdapterMethodHandle should not issue this");
1006    break;

1008  case _adapter_opt_spread_0:
1009  case _adapter_opt_spread_1:
1010  case _adapter_opt_spread_more:
1011    {
1012      // spread an array out into a group of arguments
1013      __ unimplemented(entry_name(ek));
1014    }
1015    break;

1017  case _adapter_flyby:
1018  case _adapter_ricochet:
1019    __ unimplemented(entry_name(ek)); // %%% FIXME: NYI
1020    break;

1022  default:
1023    ShouldNotReachHere();
1024  }

1026  address me_cookie = MethodHandleEntry::start_compiled_entry(_masm, interp_entr
1027  __ unimplemented(entry_name(ek)); // %%% FIXME: NYI

1029  init_entry(ek, MethodHandleEntry::finish_compiled_entry(_masm, me_cookie));
1030 }
```

_____***unchanged_portion_omitted_***